



Manual for Version 2.10-CVS

Manual for Version 2.10-CVS

```
\begin{ti
  egin(likzpicture)
/coordinate (front) at (0,0);
/coordinate (horizon) at (0,.31\paperheight);
/coordinate (bottom) at (0,-.6\paperheight);
/coordinate (sky) at (0,.57\paperheight);
/coordinate (left) at (-.51\paperwidth,0);
/coordinate (right) at (.51\paperwidth,0);
  \coordinate (right) at (.51\paperwidth,0);
  \shade [bottom color=white,
              top color=blue!30!black!50]
                    ([yshift=-5mm]horizon - | left)
     rectangle (sky -| right);
  \shade [bottom color=black!70!green!25,
     top color=black!70!green!10]
(front -| left) -- (horizon -| left)
decorate [decoration=random steps] {
     -- (horizon -| right) }
-- (front -| right) -- cycle;
  \shade [top color=black!70!green!25,
             bottom color=black!25]
                   ([yshift=-5mm-1pt]front - | left)
      rectangle ([yshift=1pt]front -| right);
  \fill [black!25]
     (bottom - | left)
rectangle ([yshift=-5mm]front - | right);
  \def \nodeshadowed[#1]#2;{
          de[scale=2,above,#1]{
         \global\setbox\mbox=\hbox{$\{\#2$}
         \copy\mybox};
          ode[scale=2,above,#1,yscale=-1,
              scope fading=south, opacity=0.4] {\box\mybox};
```

```
\nodeshadowed [at={(-5,8)},yslant=0.05]
     {\Huge Ti\textcolor{orange}{\emph{k}}Z};
  \nodeshadowed [at={(0,8.3)}]
    {\huge \textcolor{green!50!black!50}{\&}};
  \nodeshadowed [at={(5,8)},yslant=-0.05]
     {\Huge \textsc{PGF}};
  \nodeshadowed [at={(0,5)}]
{Manual for Version \pgftypesetversion};
  \foreach \where in {-9cm, 9cm} {
     \nodeshadowed [at={(\where,5cm)}] { \tikz
       \draw [green!20!black, rotate=90,
              1-system={rule set={F -> FF-[-F+F]+[+F-F]},
                 axiom=F, order=4, step=2pt,
randomize step percent=50, angle=30,
              randomize angle percent=5}] 1-system; }}
  \foreach \i in {0.5,0.6,...,2}
       [white, opacity=\i/2,
       decoration=Koch snowflake,
shift=(horizon),shift={(rand*11,rnd*7)},
        scale=\i,double copy shadow={
          opacity=0.2, shadow xshift=0pt,
          shadow yshift=3*\i pt, fill=white, draw=none}]
      decorate
         decorate {
           decorate {
             (0,0)- ++(60:1) -- ++(-60:1) -- cycle
           } } };
   \node (left text) ...
\node (right text) ...
   \fill [decorate, decoration={footprints, foot of=gnome},
     opacity=.5,brown] (rand*8,-rnd*10)
to [out=rand*180,in=rand*180] (rand*8,-rnd*10);
\end{tikzpicture}
```

Für meinen Vater, damit er noch viele schöne TeX-Graphiken erschaffen kann. $Till$
Copyright 2007 by Till Tantau
Permission is granted to copy, distribute and/or modify the documentation under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.
Permission is granted to copy, distribute and/or modify the code of the package under the terms of the GNU Public License, Version 2 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled GNU Public License.
Permission is also granted to distribute and/or modify both the documentation and the code under the conditions of the LaTeX Project Public License, either version 1.3 of this license or (at your option) any later version. A copy of the license is included in the section entitled LATeX Project Public License.

The TikZ and PGF Packages Manual for version 2.10-cvs

http://sourceforge.net/projects/pgf

Till Tantau*

Institut für Theoretische Informatik Universität zu Lübeck

November 4, 2012

Contents

1		duction	4
	1.1	Structure of the System	4
	1.2	Comparison with Other Graphics Packages	5
	1.3	Utility Packages	5
	1.4	How to Read This Manual	6
	1.5	Authors and Acknowledgements	6
	1.6	Getting Help	6
Ι	Tutor	rials and Guidelines	7
2	Tuto	rial: A Picture for Karl's Students	8
	2.1	Problem Statement	8
	2.2	Setting up the Environment	8
		2.2.1 Setting up the Environment in LATEX	8
		2.2.2 Setting up the Environment in Plain T _E X	9
		2.2.3 Setting up the Environment in ConTEXt	9
	2.3	Straight Path Construction	10
	2.4	Curved Path Construction	10
	2.5	Circle Path Construction	11
	2.6	Rectangle Path Construction	11
	2.7	Grid Path Construction	12
	2.8	Adding a Touch of Style	12
	2.9	Drawing Options	13
	2.10	Arc Path Construction	13
	2.11	Clipping a Path	14
	2.12	Parabola and Sine Path Construction	15
	2.13	Filling and Drawing	15
	2.14	Shading	16
	2.15	Specifying Coordinates	16
	2.16	Intersecting Paths	17
	2.17	Adding Arrow Tips	18
	2.18		19
	2.19	Transformations	19
	2.20	Repeating Things: For-Loops	20
	2.21		21

^{*}Editor of this documentation. Parts of this documentation have been written by other authors as indicated in these parts or chapters and in Section 1.5.

3	Tuto	rial: A Petri												2
	3.1	Problem Sta	atement						 		 	 		2
	3.2	Setting up t	he Environme	ent					 		 	 		2
		3.2.1 Se	tting up the I	Environn	nent in l	$AT_{F}X$			 		 	 		2
			tting up the I											
			tting up the I											
	3.3		n to Nodes			_	•							
	3.4		les Using the											
	3.5	-	3											2
	3.6	0 0												2
	3.7		des											2
	3.8		les Using Rela											
	3.9	9	els Next to N											
	3.10	_	Nodes											
	3.10 3.11	_	els Next to L											
		_												
	3.12	_	Snaked Line a											
	3.13		s: The Backg		_									
	3.14	The Comple	ete Code						 	• •	 	 		3
4	Tuto	rial: Euclid's	a Amaham Wa	al	f the E	1								3
4														
	4.1		position I											3
			tting up the I											
			ne Line AB .											
			ne Circle Arou											
			ne Intersection											4
			ne Complete (4
	4.2		position II											4
			sing Partway											4
			tersecting a L											4
		4.2.3 Th	ne Complete (Code .					 		 	 		4
_	m 4	. I D:	G. I	C 1	1									4
5		rial: Diagrai												4
	5.1		Nodes											4
	5.2		e Nodes Using											4
	5.3	~ ~	e Nodes Using											
	5.4		m as a Graph											
			onnecting Alre											
		5.4.2 Cr	reating Nodes	Using the	ne Grap	h Com	nmand		 		 	 		5
6	Tuto	mial. A Taats	una Man fan	Iohoni	nog									54
U	6.1	rial: A Lecti	atement											5
	6.1													5
	6.2		n to Trees											5 5
		_	e Lecture Maj											-
	6.4	_	Lecture Anno											6
	6.5		Background.											6
	0.0	Adding the	Calendar											6
	6.6	_												6
	$6.6 \\ 6.7$	_	ete Code											
7	6.7	The Comple	ete Code										• •	
7	6.7 Gui d	The Comple	ete Code $\mathbf{caphics}$						 		 	 		6
7	6.7 Gui d 7.1	The Completelines on Granning the	ete Code caphics e Time Neede	ed for the	e Creati	on of \mathbf{c}	 Graphi	ics.	 		 	 		6
7	6.7 Gui d 7.1 7.2	The Complete Interest on Graph Planning the Workflow for	ete Code caphics e Time Neede or Creating a	ed for the	e Creati	 on of (Graphi	ics .	 		 	 		6 6
7	6.7 Guid 7.1 7.2 7.3	The Complete Planning the Workflow for Linking Grant	ete Code caphics e Time Neede or Creating a caphics With the	ed for the Graphic ne Main	e Creation	on of (Graphi	ics .	 		 	 	 	6 6 7
7	6.7 Guid 7.1 7.2 7.3 7.4	The Completelines on Graphanning the Workflow for Linking Graph Consistency	ete Code caphics e Time Neede or Creating a caphics With the Between Gra	ed for the Graphic ne Main	e Creation	on of (Graphi	ics .	 		 	 		6 6 6 7 7
7	6.7 Guid 7.1 7.2 7.3 7.4 7.5	The Completelines on Graphanning the Workflow for Linking Graphan Consistency Labels in Graphan Completelines and Comple	raphics e Time Needer Creating a caphics With the Between Graraphics	ed for the Graphic ne Main aphics an	e Creation of the Creation of Text of Text of Text of Text of the Creation of Text of	on of (Graphi	ics	 		 	 		6 6 6 7 7
7	6.7 Guid 7.1 7.2 7.3 7.4	The Complete elines on Granning the Workflow for Linking Granning Consistency Labels in Granning Plots and Complete eliminates	raphics e Time Needer Creating a caphics With the Between Graraphics	ed for the Graphic ne Main phics an	e Creation of the Creation of Text and	on of (Graphi	ics	 		 	 		69 69 70 70 70 70

Installation and Configuration

8	Instal	lation		77
	8.1	Package	e and Driver Versions	77
	8.2	Installin	ng Prebundled Packages	77
		8.2.1	Debian	77
		8.2.2	MiKTeX	78
	8.3		tion in a texmf Tree	78
		8.3.1	Installation that Keeps Everything Together	78
		8.3.2	Installation that is TDS-Compliant	78
	8.4	Updatin	ng the Installation	78
9	Licens	ses and	Copyright	79
	9.1		License Applies?	79
	9.2		IU Public License, Version 2	79
		9.2.1	Preamble	79
		9.2.2	Terms and Conditions For Copying, Distribution and Modification	80
		9.2.3	No Warranty	82
	9.3	The LAT	EX Project Public License, Version 1.3c 2006-05-20	82
		9.3.1	Preamble	82
		9.3.2	Definitions	82
		9.3.3	Conditions on Distribution and Modification	83
		9.3.4	No Warranty	84
		9.3.5	Maintenance of The Work	85
		9.3.6	Whether and How to Distribute Works under This License	85
		9.3.7	Choosing This License or Another License	85
		9.3.8	A Recommendation on Modification Without Distribution	86
		9.3.9	How to Use This License	86
		9.3.10	Derived Works That Are Not Replacements	86
		9.3.11	Important Recommendations	86
	9.4		ree Documentation License, Version 1.2, November 2002	87
		9.4.1	Preamble	87
		9.4.2	Applicability and definitions	87
		9.4.3	Verbatim Copying	88
		9.4.4	Copying in Quantity	88
		9.4.5	Modifications	88
		9.4.6	Combining Documents	90
		9.4.7	Collection of Documents	90
		9.4.8	Aggregating with independent Works	90
		9.4.9	Translation	90
		9.4.10 9.4.11	Termination	90
		9.4.11 $9.4.12$	Future Revisions of this License	91 91
		9.4.12	Addendum. How to use this License for your documents	91
10	Input		tput Formats	92
	10.1	Support	sed Input Formats	92
		10.1.1	Using the LATEX Format	92
		10.1.2	Using the Plain T _E X Format	92
		10.1.3	Using the ConTEXt Format	92
	10.2	Support	ged Output Formats	93
		10.2.1	Selecting the Backend Driver	93
		10.2.2	Producing PDF Output	93
		10.2.3	Producing PostScript Output	94
		10.2.4	Producing HTML / SVG Output	95
		10.2.5	Producing Perfectly Portable DVI Output	96
III	$\mathrm{Ti} k^2$	${f Z}$ ist $k\epsilon$	ein Zeichenprogramm	97

11	Desig	n Principles 98
	11.1	Special Syntax For Specifying Points
	11.2	Special Syntax For Path Specifications
	11.3	Actions on Paths
	11.4	Key-Value Syntax for Graphic Parameters
	11.5	Special Syntax for Specifying Nodes
	11.6	Special Syntax for Specifying Trees
	11.7	Grouping of Graphic Parameters
	11.8	Coordinate Transformation System
12		rchical Structures: Package, Environments, Scopes, and Styles 101
	12.1	Loading the Package and the Libraries
	12.2	Creating a Picture
		12.2.1 Creating a Picture Using an Environment
		12.2.2 Creating a Picture Using a Command
	400	12.2.3 Adding a Background
	12.3	Using Scopes to Structure a Picture
		12.3.1 The Scope Environment
		12.3.2 Shorthand for Scope Environments
		12.3.3 Single Command Scopes
	10.1	12.3.4 Using Scopes Inside Paths
	12.4	Using Graphic Options
		12.4.1 How Graphic Options Are Processed
		12.4.2 Using Styles to Manage How Pictures Look
13	Speci	fying Coordinates 108
10	13.1	Overview
	13.2	Coordinate Systems
	10.2	13.2.1 Canvas, XYZ, and Polar Coordinate Systems
		13.2.2 Barycentric Systems
		13.2.3 Node Coordinate System
		13.2.4 Tangent Coordinate Systems
		13.2.5 Defining New Coordinate Systems
	13.3	Coordinates at Intersections
	10.0	13.3.1 Intersections of Perpendicular Lines
		13.3.2 Intersections of Arbitrary Paths
	13.4	Relative and Incremental Coordinates
	10.1	13.4.1 Specifying Relative Coordinates
		13.4.2 Relative Coordinates and Scopes
	13.5	Coordinate Calculations
	10.0	13.5.1 The General Syntax
		13.5.2 The Syntax of Factors
		13.5.3 The Syntax of Partway Modifiers
		13.5.4 The Syntax of Distance Modifiers
		13.5.5 The Syntax of Projection Modifiers
		1900 110 Systems 01 1 10 Joseph 110 minutes 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
14	Synta	x for Path Specifications 123
	14.1	The Move-To Operation
	14.2	The Line-To Operation
		14.2.1 Straight Lines
		14.2.2 Horizontal and Vertical Lines
	14.3	The Curve-To Operation
	14.4	The Cycle Operation
	14.5	The Rectangle Operation
	14.6	Rounding Corners
	14.7	The Circle and Ellipse Operations
	14.8	The Arc Operation
	14.9	The Grid Operation
	14.10	The Parabola Operation 131

	14.11	The Sine and Cosine Operation
	14.12	The SVG Operation
	14.13	The Plot Operation
	14.14	The To Path Operation
	14.15	The Let Operation
	14.16	The Scoping Operation
	14.17	The Node and Edge Operations
	14.18	The Graph Operation
	14.19	The PGF-Extra Operation
	14.19	The FGF-Extra Operation
15	Actio	ns on Paths
	15.1	Overview
	15.2	Specifying a Color
	15.2 15.3	Drawing a Path
	10.0	15.3.1 Graphic Parameters: Line Width, Line Cap, and Line Join
		15.3.2 Graphic Parameters: Dash Pattern
		1
		1 0
		15.3.4 Graphic Parameters: Arrow Tips
		15.3.5 Graphic Parameters: Double Lines and Bordered Lines
	15.4	Filling a Path
		15.4.1 Graphic Parameters: Fill Pattern
		15.4.2 Graphic Parameters: Interior Rules
		15.4.3 Graphic Parameters: Fill Opacity
	15.5	Generalized Filling: Using Arbitrary Pictures to Fill a Path
	15.6	Shading a Path
	15.7	Establishing a Bounding Box
	15.8	Clipping and Fading (Soft Clipping)
	15.9	Doing Multiple Actions on a Path
	15.10	Decorating and Morphing a Path
16		s and Edges 158
16	16.1	Overview
16		Overview
16	16.1	OverviewNodes and Their Shapes16.2.1 Predefined Shapes
16	16.1	OverviewNodes and Their Shapes16.2.1 Predefined Shapes16.2.2 Common Options: Separations, Margins, Padding and Border Rotation
16	16.1	OverviewNodes and Their Shapes16.2.1 Predefined Shapes
16	16.1 16.2	OverviewNodes and Their Shapes16.2.1 Predefined Shapes16.2.2 Common Options: Separations, Margins, Padding and Border Rotation
16	16.1 16.2	Overview
16	16.1 16.2	Overview156Nodes and Their Shapes15616.2.1 Predefined Shapes16616.2.2 Common Options: Separations, Margins, Padding and Border Rotation166Multi-Part Nodes166The Node Text16616.4.1 Text Parameters: Color and Opacity166
16	16.1 16.2	Overview156Nodes and Their Shapes15616.2.1 Predefined Shapes16616.2.2 Common Options: Separations, Margins, Padding and Border Rotation166Multi-Part Nodes166The Node Text16616.4.1 Text Parameters: Color and Opacity16616.4.2 Text Parameters: Font166
16	16.1 16.2	Overview156Nodes and Their Shapes15616.2.1 Predefined Shapes16616.2.2 Common Options: Separations, Margins, Padding and Border Rotation166Multi-Part NodesThe Node Text16616.4.1 Text Parameters: Color and Opacity16616.4.2 Text Parameters: Font16616.4.3 Text Parameters: Alignment and Width for Multi-Line Text166
16	16.1 16.2 16.3 16.4	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160
16	16.1 16.2	Overview 156 Nodes and Their Shapes 156 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 166 The Node Text 166 16.4.1 Text Parameters: Color and Opacity 166 16.4.2 Text Parameters: Font 166 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 166 16.4.4 Text Parameters: Height and Depth of Text 166 Positioning Nodes 166
16	16.1 16.2 16.3 16.4	Overview 156 Nodes and Their Shapes 156 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 166 The Node Text 166 16.4.1 Text Parameters: Color and Opacity 166 16.4.2 Text Parameters: Font 166 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 166 16.4.4 Text Parameters: Height and Depth of Text 166 Positioning Nodes 166 16.5.1 Positioning Nodes Using Anchors 166
16	16.1 16.2 16.3 16.4	Overview 156 Nodes and Their Shapes 156 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 166 The Node Text 166 16.4.1 Text Parameters: Color and Opacity 166 16.4.2 Text Parameters: Font 166 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 166 16.4.4 Text Parameters: Height and Depth of Text 166 Positioning Nodes 166 16.5.1 Positioning Nodes Using Anchors 166 16.5.2 Basic Placement Options 166
16	16.1 16.2 16.3 16.4	Overview 158 Nodes and Their Shapes 156 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 166 The Node Text 166 16.4.1 Text Parameters: Color and Opacity 166 16.4.2 Text Parameters: Font 166 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 166 16.4.4 Text Parameters: Height and Depth of Text 166 Positioning Nodes 166 16.5.1 Positioning Nodes Using Anchors 166 16.5.2 Basic Placement Options 169 16.5.3 Advanced Placement Options 170
16	16.1 16.2 16.3 16.4	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 Positioning Nodes 160 16.5.1 Positioning Nodes Using Anchors 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 170 16.5.4 Arranging Nodes Using Chains and Matrices 170
16	16.1 16.2 16.3 16.4	Overview 158 Nodes and Their Shapes 158 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 16.5.1 Positioning Nodes 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 170 16.5.4 Arranging Nodes Using Chains and Matrices 170 Fitting Nodes to a Set of Coordinates 170
16	16.1 16.2 16.3 16.4 16.5	Overview 158 Nodes and Their Shapes 158 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 16.5.1 Positioning Nodes 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 170 16.5.4 Arranging Nodes Using Chains and Matrices 170 Fitting Nodes to a Set of Coordinates 170 Transformations 170
16	16.1 16.2 16.3 16.4 16.5	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 16.5.1 Positioning Nodes 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 170 16.5.4 Arranging Nodes Using Chains and Matrices 170 Fitting Nodes to a Set of Coordinates 170 Transformations 170 Placing Nodes on a Line or Curve Explicitly 170
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 16.5.1 Positioning Nodes 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 170 16.5.4 Arranging Nodes Using Chains and Matrices 170 Fitting Nodes to a Set of Coordinates 170 Transformations 170 Placing Nodes on a Line or Curve Explicitly 170 Placing Nodes on a Line or Curve Implicitly 170 Placing Nodes on a Line or Curve Implicitly 170
16	16.1 16.2 16.3 16.4 16.5	Overview 156 Nodes and Their Shapes 156 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 166 The Node Text 166 16.4.1 Text Parameters: Color and Opacity 166 16.4.2 Text Parameters: Font 166 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 166 16.4.4 Text Parameters: Height and Depth of Text 166 16.5.1 Positioning Nodes 166 16.5.2 Basic Placement Options 166 16.5.3 Advanced Placement Options 176 16.5.4 Arranging Nodes Using Chains and Matrices 177 Fitting Nodes to a Set of Coordinates 177 Transformations 178 Placing Nodes on a Line or Curve Explicitly 179 Placing Nodes on a Line or Curve Implicitly 179 The Label and Pin Options 179
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9	Overview 156 Nodes and Their Shapes 156 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 166 The Node Text 166 16.4.1 Text Parameters: Color and Opacity 166 16.4.2 Text Parameters: Font 166 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 166 16.4.4 Text Parameters: Height and Depth of Text 166 Positioning Nodes 166 16.5.1 Positioning Nodes Using Anchors 166 16.5.2 Basic Placement Options 166 16.5.3 Advanced Placement Options 176 16.5.4 Arranging Nodes Using Chains and Matrices 176 Fitting Nodes to a Set of Coordinates 177 Transformations 178 Placing Nodes on a Line or Curve Explicitly 178 Placing Nodes on a Line or Curve Implicitly 178 The Label and Pin Options 179 16.10.1 Overview 179
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9	Overview 156 Nodes and Their Shapes 156 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 16 The Node Text 16 16.4.1 Text Parameters: Color and Opacity 16 16.4.2 Text Parameters: Font 16 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 16 16.4.4 Text Parameters: Height and Depth of Text 16 Positioning Nodes 16 16.5.1 Positioning Nodes Using Anchors 16 16.5.2 Basic Placement Options 16 16.5.3 Advanced Placement Options 17 16.5.4 Arranging Nodes Using Chains and Matrices 17 Fitting Nodes to a Set of Coordinates 17 Transformations 17 Placing Nodes on a Line or Curve Explicitly 17 Placing Nodes on a Line or Curve Implicitly 17 The Label and Pin Options 17 16.10.1 Overview 17 16.10.2 The Label Option 18
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 16.5.1 Positioning Nodes 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 160 16.5.4 Arranging Nodes Using Chains and Matrices 17 Fitting Nodes to a Set of Coordinates 17 Transformations 17 Placing Nodes on a Line or Curve Explicitly 17 Placing Nodes on a Line or Curve Implicitly 17 The Label and Pin Options 17 16.10.1 Overview 17 16.10.2 The Label Option 18 16.10.3 The Pin Option 18
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 16.5.1 Positioning Nodes 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 160 16.5.4 Arranging Nodes Using Chains and Matrices 170 Fitting Nodes to a Set of Coordinates 170 Transformations 170 Placing Nodes on a Line or Curve Explicitly 170 Placing Nodes on a Line or Curve Implicitly 170 The Label and Pin Options 170 16.10.1 Overview 170 16.10.2 The Label Option 180 16.10.4 The Quotes Syntax 180
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 160 Multi-Part Nodes 160 The Node Text 160 16.4.1 Text Parameters: Color and Opacity 160 16.4.2 Text Parameters: Font 160 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 160 16.4.4 Text Parameters: Height and Depth of Text 160 16.5.1 Positioning Nodes 160 16.5.2 Basic Placement Options 160 16.5.3 Advanced Placement Options 160 16.5.4 Arranging Nodes Using Chains and Matrices 17 Fitting Nodes to a Set of Coordinates 17 Transformations 17 Placing Nodes on a Line or Curve Explicitly 17 Placing Nodes on a Line or Curve Implicitly 17 The Label and Pin Options 17 16.10.1 Overview 17 16.10.2 The Label Option 18 16.10.3 The Pin Option 18
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9 16.10	Overview 150 Nodes and Their Shapes 155 16.2.1 Predefined Shapes 166 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 166 Multi-Part Nodes 166 The Node Text 16 16.4.1 Text Parameters: Color and Opacity 16 16.4.2 Text Parameters: Font 16 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 16 16.4.4 Text Parameters: Height and Depth of Text 16 16.5.1 Positioning Nodes 16 16.5.2 Basic Placement Options 16 16.5.3 Advanced Placement Options 16 16.5.4 Arranging Nodes Using Chains and Matrices 17 Fitting Nodes to a Set of Coordinates 17 Transformations 17 Placing Nodes on a Line or Curve Explicitly 17 Placing Nodes on a Line or Curve Implicitly 17 The Label and Pin Options 17 16.10.1 Overview 17 16.10.2 The Label Option 18 16.10.4 The Quotes Syntax 18 Connecting Nodes: Using Nodes as Coordinates 18 Connecting Nodes: Using the Edge Operation
16	16.1 16.2 16.3 16.4 16.5 16.6 16.7 16.8 16.9 16.10	Overview 150 Nodes and Their Shapes 150 16.2.1 Predefined Shapes 160 16.2.2 Common Options: Separations, Margins, Padding and Border Rotation 161 Multi-Part Nodes 163 The Node Text 164 16.4.1 Text Parameters: Color and Opacity 166 16.4.2 Text Parameters: Font 166 16.4.3 Text Parameters: Alignment and Width for Multi-Line Text 166 16.4.4 Text Parameters: Height and Depth of Text 166 16.5.1 Positioning Nodes 166 16.5.2 Basic Placement Options 166 16.5.3 Advanced Placement Options 166 16.5.4 Arranging Nodes Using Chains and Matrices 17 Fitting Nodes to a Set of Coordinates 17 Transformations 17 Placing Nodes on a Line or Curve Explicitly 17 Placing Nodes on a Line or Curve Implicitly 17 The Label and Pin Option 18 16.10.2 The Label Option 18 16.10.3 The Pin Option 18 16.10.4 The Quotes Syntax 18 16.10.4 The Quotes Syntax 18

	16.13	Referencing Nodes Outside the Current Picture
		16.13.1 Referencing a Node in a Different Picture
		16.13.2 Referencing the Current Page Node – Absolute Positioning 189
	16.14	Late Code and Late Options
	~ .	
17	_	fying Graphs 192
	17.1	Overview
	17.2	Concepts
		17.2.1 Concept: Node Chains
		17.2.2 Concept: Chain Groups
		17.2.3 Concept: Edge Labels and Styles
		17.2.4 Concept: Node Sets
		17.2.5 Concept: Graph Macros
		17.2.6 Concept: Graph Expressions and Color Classes
	17.3	Syntax of the Graph Path Command
		17.3.1 The Graph Command
		17.3.2 Syntax of Group Specifications
		17.3.3 Syntax of Chain Specifications
		17.3.4 Syntax of Node Specifications
		17.3.5 Specifying Tries
	17.4	Simple Versus Multi-Graphs
	17.5	Graph Edges: Labeling and Styling
		17.5.1 Options For All Edges Between Two Groups
		17.5.2 Changing Options For Certain Edges
		17.5.3 Options For Incoming and Outgoing Edges
		17.5.4 Special Syntax for Options For Incoming and Outgoing Edges
		17.5.5 Placing Node Texts on Incoming Edges
	17.6	Graph Operators, Color Classes, and Graph Expressions
		17.6.1 Color Classes
		17.6.2 Graph Operators on Groups of Nodes
		17.6.3 Graph Operators for Joining Groups
	17.7	Graph Macros
	17.8	Online Placement Strategies
		17.8.1 Manual Placement
		17.8.2 Placement on a Grid
		17.8.3 Placement Taking Node Sizes Into Account
		17.8.4 Placement On a Circle
		17.8.5 Levels and Level Styles
		17.8.6 Defining New Online Placement Strategies
	17.9	Reference: Predefined Elements
	_,,,	17.9.1 Graph Macros
		17.9.2 Group Operators
		17.9.3 Joining Operators
18	Matri	ices and Alignment 238
	18.1	Overview
	18.2	Matrices are Nodes
	18.3	Cell Pictures
		18.3.1 Alignment of Cell Pictures
		18.3.2 Setting and Adjusting Column and Row Spacing
		18.3.3 Cell Styles and Options
	18.4	Anchoring a Matrix
	18.5	Considerations Concerning Active Characters
	18.6	Examples

19	Making Trees G		249
		on to the Child Operation	
		s and Child Nodes	
		nild Nodes	
		Options for Trees and Children	
	9	ild Nodes	
		Sasic Idea	
	19.5.2 D	Default Growth Function	253
	19.5.3 M	Issing Children	255
	19.5.4 C	Sustom Growth Functions	256
	19.6 Edges From	n the Parent Node	257
2 0	Plots of Function		260
		ıld One Use Ti k Z for Generating Plots?	
		ath Operation	
	<u> </u>	pints Given Inline	
	<u> </u>	pints Read From an External File	
	0	Function	
		Function Using Gnuplot	
		arks on the Plot	
	20.8 Smooth Plo	ots, Sharp Plots, Jump Plots, Comb Plots and Bar Plots	266
2 1	Transparency		271
		a Uniform Opacity	271
		Streating Fadings	
		ading a Path	275
		ading a Scope	
	21.4 Transparen	cy Groups	277
22	Decorated Paths	5	27 9
	22.1 Overview.		279
	22.2 Decorating	a Subpath Using the Decorate Path Command	281
	22.3 Decorating	a Complete Path	282
		Decorations	
	22.4.1 P	ositioning Decorations Relative to the To-Be-Decorate Path	283
	22.4.2 St	tarting and Ending Decorations Early or Late	284
23	Transformations	:	286
	23.1 The Differe	ent Coordinate Systems	286
	23.2 The XY- ar	nd XYZ-Coordinate Systems	286
	23.3 Coordinate	Transformations	287
	23.4 Canvas Tra	ansformations	290
	a		
IV	Graph Drawi	ng 2	292
T 7	T:lamaniaa	ก	ഹെ
\mathbf{V}	Libraries	2	292
24	Arrow Tip Libra		294
		The state of the s	294
	_	•	294
		T	295
		1	295
		1	295
		r ···	296
	24.7 Partial Arr	row Tips	296

	$24.8 \\ 24.9$	Line Caps	
25	Auto	nata Drawing Library 29	7
	25.1	Drawing Automata	7
	25.2	States With and Without Output	
	25.3	Initial and Accepting States	
	25.4	Examples	0
26	Back	round Library 30	2
27	Calc	ibrary 30	5
28	Caler	lar Library 30	6
	28.1	Calendar Command	6
		28.1.1 Creating a Simple List of Days	3
		28.1.2 Adding a Month Label	3
		28.1.3 Creating a Week List Arrangement	3
		28.1.4 Creating a Month List Arrangement	
	28.2	Arrangements	
	28.3	Month Labels	
	28.4	Examples	9
29	Chair	32	3
	29.1	Overview	3
	29.2	Starting and Continuing a Chain	
	29.3	Nodes on a Chain	
	29.4	Joining Nodes on a Chain	
	29.5	Branches	7
30	Circu	t Libraries 32	
	30.1	Introduction $\dots \dots \dots$	
		30.1.1 A First Example	
		30.1.2 Symbols	
		30.1.3 Symbol Graphics	
		30.1.4 Annotations	
	30.2	The Base Circuit Library	
		30.2.1 Symbol Size	
		30.2.2 Declaring New Symbols	
		30.2.3 Pointing Symbols in the Right Direction	
		30.2.4 Info Labels	
		30.2.5 Declaring and Using Annotations	
	20.2	30.2.6 Theming Symbols	
	30.3	Logical Circuits	
		30.3.2 Symbols: The Gates	
		30.3.3 Implementation: The Logic Gates Shape Library	
		30.3.4 Implementation: The US-Style Logic Gates Shape Library	
		30.3.5 Implementation: The IEC-Style Logic Gates Shape Library	
	30.4	Electrical Engineering Circuits	
	30.4	30.4.1 Overview	
		30.4.2 Symbols: Indicating Current Directions	
		30.4.3 Symbols: Basic Elements	
		30.4.4 Symbols: Diodes	
		30.4.5 Symbols: Contacts	
		30.4.6 Units	
		30.4.7 Annotations	
		30.4.8 Implementation: The EE-Symbols Shape Library	
		30 4 9 Implementation: The IEC-Style EE-Symbols Shape Library 35	

31	\mathbf{Deco}	ration Library 359
	31.1	Overview and Common Options
	31.2	Path Morphing Decorations
		31.2.1 Decorations Producing Straight Line Paths
		31.2.2 Decorations Producing Curved Line Paths
	31.3	Path Replacing Decorations
	31.4	Marking Decorations
		31.4.1 Overview
	31.5	Arbitrary Markings
	01.0	31.5.1 Arrow Tip Markings
		31.5.2 Footprint Markings
		31.5.3 Shape Background Markings
	31.6	Text Decorations
	31.7	Fractal Decorations
	91.7	Tractal Decorations
32	Entit	y-Relationship Diagram Drawing Library 380
_	32.1	Entities
	32.1	Relationships
	32.2	Attributes
	32.3	Titolibutes
33	Exter	rnalization Library 385
00	33.1	Overview
	33.2	Requirements
	33.3	A Word About ConT _E Xt And Plain T _E X
	33.4	Externalizing Graphics
	99.4	33.4.1 Support for Labels and References In External Files
		33.4.2 Customizing the Generated File Names
		· ·
		33.4.3 Remaking Figures or Skipping Figures
		33.4.4 Customizing the Externalization
	99.5	33.4.5 Details About The Process
	33.5	Using External Graphics Without PGF Installed
	33.6	eps Graphics Export
	33.7	Bitmap Graphics Export
	33.8	Compatibility Issues
		33.8.1 References In External Pictures
		33.8.2 Compatibility With Other Libraries or Packages
		33.8.3 Compatibility With Bounding Box Restrictions
		33.8.4 Interoperability With The Basic Layer Externalization
34	Fadir	ag Library 396
35	Fitti	ng Library 39'
36	Fixed	l Point Arithmetic Library 400
	36.1	Overview
	36.2	Using Fixed Point Arithmetic in PGF and TikZ
	30. <u>-</u>	
37	Float	ing Point Unit Library 403
	37.1	Overview
	37.2	Usage
	37.2	Comparison to the fixed point arithmetics library
	37.4	Command Reference and Programmer's Manual
	91.I	37.4.1 Creating and Converting Floats
		37.4.2 Symbolic Rounding Operations
		37.4.3 Math Operations Commands
		37.4.4 Accessing the Original Math Routines for Programmers
		- original moocoome one Original Maon Rougines for Figurial and Figure 7 400

38		enmayer System Drawing Library 410
	38.1	Overview
		38.1.1 Declaring L-systems
	38.2	Using Lindenmayer Systems
		38.2.1 Using L-Systems in PGF
		38.2.2 Using L-Systems in $TikZ$
39	Matr	ix Library 415
J	39.1	Matrices of Nodes
	39.2	End-of-Lines and End-of-Row Characters in Matrices of Nodes
	39.2	Delimiters
	30.3	
40	Mind	map Drawing Library 419
	40.1	Overview
	40.2	The Mindmap Style
	40.3	Concepts Nodes
		40.3.1 Isolated Concepts
		40.3.2 Concepts in Trees
	40.4	Connecting Concepts
		40.4.1 Simple Connections
		40.4.2 The Circle Connection Bar Decoration
		40.4.3 The Circle Connection Bar To-Path
		40.4.4 Tree Edges
	40.5	Adding Annotations
41	Pano	r Folding Diagrams Library 429
41	_	
42		ern Library 433
	42.1	Form-Only Patterns
	42.2	Inherently Colored Patterns
43	Petri	-Net Drawing Library 434
	43.1	Places
	43.2	Transitions
	43.3	Tokens
	43.4	Examples
		" "
44		Handler Library 439
		Curve Plot Handlers
	44.2	Constant Plot Handlers
	44.3	Comb Plot Handlers
	44.4	Bar Plot Handlers
	44.5	Gapped Plot Handlers
	44.6	Mark Plot Handler
45	Plot	Mark Library 448
46	Drofi	ler Library 450
40	46.1	Overview
		Requirements
	46.2	<u>.</u>
	46.3	Defining Profiler Entries
47	Shad	ings Library 453
48	Shade	ows Library 456
-	48.1	Overview
	48.2	The General Shadow Option
	48.3	Shadows for Arbitrary Paths and Shapes
		48.3.1 Drop Shadows
		48.3.2 Copy Shadows

	48.4	Shadows for Special Paths and Nodes
49	Shap	e Library 460
	49.1	Overview
	49.2	Predefined Shapes
	49.3	Geometric Shapes
	49.4	Symbol Shapes
	49.5	Arrow Shapes
	49.6	Shapes with Multiple Text Parts
	49.7	Callout Shapes
	49.8	Miscellaneous Shapes
50	C T	Library: Magnifying Parts of Pictures 503
3 0		Library: Magnifying Parts of Pictures 503 Magnifying a Part of a Picture
	$50.1 \\ 50.2$	
	50.2 50.3	Spy Scopes 504 The Spy Command 504
	50.3	Predefined Spy Styles
	$50.4 \\ 50.5$	Examples
	50.5	Examples
51	SVG-	-Path Library 509
52	To Pa	ath Library 510
	52.1	Straight Lines
	52.2	Move-Tos
	52.3	Curves
	52.4	Loops
53	Thro	ugh Library 515
54		Library 516
	54.1	Growth Functions
	54.2	Edges From Parent
55	Turtl	e Graphics Library 519
VI	Dat	a Visualization 521
56	Intro	duction to Data Visualization 522
•	56.1	Concept: Data Points
	56.2	Concept: Visualization Pipeline
		•
57		ting Data Visualizations 524
	57.1	Overview
	57.2	Concept: Data Points and Data Formats
	57.3	Concept: Axes, Ticks, and Grids
	57.4	Concept: Visualizers
	57.5	Concept: Style Sheets and Legends
	57.6	Usage
	$57.7 \\ 57.8$	Advanced: Executing User Code During a Data Visualization
58		iding Data for a Data Visualization 534
	58.1	Overview
	58.2	Concepts
	58.3	Reference: Build-In Formats
	58.4	Reference: Advanced Formats
	58.5	Advanced: The Data Parsing Process
	58.6	Advanced: Defining New Formats

59	\mathbf{Axes}		542
	59.1	Overviev	$_{\scriptscriptstyle W}$
	59.2	Basic Co	onfiguration of Axes
		59.2.1	Usage
		59.2.2	The Axis Attribute
		59.2.3	The Axis Attribute Range Interval
		59.2.4	Scaling: The General Mechanism
		59.2.5	Scaling: Logarithmic Axes
		59.2.6	Scaling: Setting the Length or Unit Length
		59.2.7	Axis Label
		59.2.8	Reference: Axis Types
	59.3		stems
	50.5	59.3.1	Usage
		59.3.2	Reference: Scientific Axis Systems
		59.3.3	Reference: School Book Axis Systems
		59.3.4	Advanced Reference: Underlying Cartesian Axis Systems
	59.4		ad Grids
	55.4	59.4.1	Concepts
		59.4.1 $59.4.2$	The Main Options: Tick and Grid
		59.4.2	Semi-Automatic Computation of Tick and Grid Line Positions
		59.4.3	Automatic Computation of Tick and Grid Line Positions
		59.4.4 $59.4.5$	Manual Specification of Tick and Grid Line Positions
		59.4.6	Styling Ticks and Grid Lines: Introduction
			v o
		59.4.7	Styling Ticks and Grid Lines: The Style and Node Style Keys
		59.4.8	Styling Ticks and Grid Lines: Styling Grid Lines
		59.4.9	Styling Ticks and Grid Lines: Styling Ticks and Tick Labels
		59.4.10	Styling Ticks and Grid Lines: Exceptional Ticks
		59.4.11	Styling Ticks and Grid Lines: Styling and Typesetting a Value
		59.4.12	Stacked Ticks
		59.4.13	Reference: Basic Strategies
	F0 F	59.4.14	Advanced: Defining New Placement Strategies
	59.5		ed: Creating New Axis Systems
		59.5.1	Creating the Axes
		59.5.2	Visualizing the Axes
		59.5.3	Visualizing Grid Lines
		59.5.4	Visualizing the Ticks and Tick Labels
		59.5.5	Visualizing the Axis Labels
		59.5.6	The Complete Axis System
		59.5.7	Using the New Axis System Key
60	Visua	lia ona	207
oo	60.1		587 w
	60.1		· · · · · · · · · · · · · · · · · · ·
	00.2	0	
		60.2.1	Using a Single Visualizer
		60.2.2	Using Multiple Visualizers
	co o	60.2.3	Styling a Visualizer
	60.3		ee: Basic Visualizers
		60.3.1	Visualizing Data Points Using Lines
	00.4	60.3.2	Visualizing Data Points Using Marks
	60.4	Advance	ed: Creating New Visualizers
61	Style	Sheets	and Legends 597
OΙ	61.1		und Legends 597
	61.1		s: Style Sheets
	61.2		s: Style Sheets
	61.4		Style Sheets
	01.4	61.4.1	Picking a Style Sheet
		61.4.1 $61.4.2$	Creating a New Style Sheet
		61.4.2	Creating a New Color Style Sheet
		(11.4 ·)	NANOTHER OF INC. VALUE OF CONTROL

	61.5	Reference: Style Sheets for Lines	1
	61.6	Reference: Style Sheets for Scatter Plots	
	61.7	Reference: Color Style Sheets	,
	61.8	Usage: Labeling Data Sets Inside the Visualization	į
		61.8.1 Placing a Label Next to a Data Set	
		61.8.2 Connecting a Label to a Data Set via a Pin 609	
	61.9	Usage: Labeling Data Sets Inside a Legend	
		61.9.1 Creating Legends and Legend Entries 611	
		61.9.2 Rows and Columns of Legend Entries 613	
		61.9.3 Legend Placement: The General Mechanism 616	
		61.9.4 Legend Placement: Outside to the Data Visualization 617	
		61.9.5 Legend Placement: Inside to the Data Visualization 619	
		61.9.6 Legend Entries: General Styling	
		61.9.7 Legend Entries: Styling the Text Node	
		61.9.8 Legend Entries: Text Placement	
		61.9.9 Advanced: Labels in Legends and Their Visualizers 622	
		61.9.10 Reference: Label in Legend Visualizers for Lines and Scatter Plots 624	:
32	The	Data Visualization Backend 629	
) 4	62.1	Overview	
	62.1	The Rendering Pipeline	
	62.2	Usage	
	62.4	The Mathematical Micro-Kernel	
	02.4	The Mathematical Micro-Relation	
VII	I Ut	ilities 630	į
33	Kow	Management 631	
,,	63.1	Introduction	
	00.1	63.1.1 Comparison to Other Packages	
		63.1.2 Quick Guide to Using the Key Mechanism	
	63.2	The Key Tree	
	63.3	Setting Keys	
	00.0	63.3.1 First Char Syntax Detection	
		63.3.2 Default Arguments	
		63.3.3 Keys That Execute Commands	
		63.3.4 Keys That Store Values	
		63.3.5 Keys That Are Handled	
		63.3.6 Keys That Are Unknown	
		63.3.7 Search Paths And Handled Keys	
	63.4	Key Handlers	
		63.4.1 Handlers for Path Management	
		63.4.2 Setting Defaults	
		63.4.3 Defining Key Codes	
		63.4.4 Defining Styles	;
		63.4.5 Defining Value-, Macro-, If- and Choice-Keys 644	
		63.4.6 Expanded and Multiple Values	;
		63.4.7 Handlers for Forwarding	;
		63.4.8 Handlers for Testing Keys	,
		63.4.9 Handlers for Key Inspection)
	63.5	Error Keys)
	63.6	Key Filtering)
		63.6.1 Starting With An Example)
		63.6.2 Setting Filters)
		63.6.3 Handlers For Unprocessed Keys	,
		63.6.4 Family Support	,
		63.6.5 Other Key Filters	:
		63.6.6 Programmer Interface	
		63.6.7 Defining Own Filters Or Filter Handlers	,

64	Repeating Things: The Foreach Statement	657
65	Date and Calendar Utility Macros 65.1 Handling Dates 65.1.1 Conversions Between Date Types 65.1.2 Checking Dates 65.1.3 Typesetting Dates 65.1.4 Localization 65.2 Typesetting Calendars	. 662 . 663 . 664 . 665
66	Page Management	668
	66.1 Basic Usage	. 669 . 671
67	Extended Color Support	675
68	Parser Module	676
VII	II Mathematical and Object-Oriented Engines	677
69	Design Principles 69.1 Loading the Mathematical Engine	678
	69.2 Layers of the Mathematical Engine	. 678
70	Evaluating Mathematical Expressions 70.1 Commands for Parsing Expressions	679 . 679
71	Syntax for mathematical expressions 71.1 Operators 71.2 Functions 71.2.1 Basic arithmetic functions 71.2.2 Rounding functions 71.2.3 Integer arithmetics functions 71.2.4 Trigonometric functions 71.2.5 Comparison and logical functions 71.2.6 Pseudo-random functions 71.2.7 Base conversion functions 71.2.8 Miscellaneous functions	. 682 . 683 . 685 . 686 . 686 . 688 . 689
72	Additional Mathematical Commands72.1 Basic arithmetic functions72.2 Comparison and logical functions72.3 Pseudo-Random Numbers72.4 Base Conversion	. 693 . 693
73	Customizing the Mathematical Engine	695
74	Number Printing 74.1 Changing display styles	697 . 702
7 5	Object-Oriented Programming 75.1 Overview 75.2 A Running Example: The Stamp Class 75.3 Classes 75.4 Objects 75.5 Methods	. 707 . 707

	75.6	Attributes
	75.7	Identities
	75.8	The Signal Class
	75.9	Implementation Notes
IX	The	Basic Layer 715
76	Desig	n Principles 716
	76.1	Core and Modules
	76.2	Communicating with the Basic Layer via Macros
	76.3	Path-Centered Approach
	76.4	Coordinate Versus Canvas Transformations
77	Hiera	rchical Structures: Package, Environments, Scopes, and Text 718
	77.1	Overview
		77.1.1 The Hierarchical Structure of the Package
		77.1.2 The Hierarchical Structure of Graphics
	77.2	The Hierarchical Structure of the Package
	2	77.2.1 The Core Package
		77.2.2 The Modules
		77.2.3 The Library Packages
	77.3	The Hierarchical Structure of the Graphics
	11.0	77.3.1 The Main Environment
		77.3.2 Graphic Scope Environments
		77.3.3 Inserting Text and Images
		11.5.5 Inscring Text and Images
78	_	fying Coordinates 727
	78.1	Overview
	78.2	Basic Coordinate Commands
	78.3	Coordinates in the XY-Coordinate System
	78.4	Three Dimensional Coordinates
	78.5	Building Coordinates From Other Coordinates
		78.5.1 Basic Manipulations of Coordinates
		78.5.2 Points Traveling along Lines and Curves
		78.5.3 Points on Borders of Objects
		78.5.4 Points on the Intersection of Lines
		78.5.5 Points on the Intersection of Two Circles
		78.5.6 Points on the Intersection of Two Paths
	78.6	Extracting Coordinates
	78.7	Internals of How Point Commands Work
7 9	Const	ructing Paths 736
	79.1	Overview
	79.2	The Move-To Path Operation
	79.3	The Line-To Path Operation
	79.4	The Curve-To Path Operations
	79.5	The Close Path Operation
	79.6	Arc, Ellipse and Circle Path Operations
	79.7	Rectangle Path Operations
	79.8	The Grid Path Operation
	79.9	The Parabola Path Operation
	79.10	Sine and Cosine Path Operations
	79.10	Plot Path Operations
	79.11	Rounded Corners
	79.12	Internal Tracking of Bounding Boxes for Paths and Pictures
	10.10	THE PROPERTY OF THE PROPERTY O

80	Decor	rations 74
	80.1	Overview
	80.2	Decoration Automata
		80.2.1 The Different Paths
		80.2.2 Segments and States
	80.3	Declaring Decorations
		80.3.1 Predefined Decorations
	80.4	Using Decorations
	80.5	Meta-Decorations
		80.5.1 Declaring Meta-Decorations
		80.5.2 Predefined Meta-decorations
		80.5.3 Using Meta-Decorations
81	Using	Paths 76
	81.1	Overview
	81.2	Stroking a Path
		81.2.1 Graphic Parameter: Line Width
		81.2.2 Graphic Parameter: Caps and Joins
		81.2.3 Graphic Parameter: Dashing
		81.2.4 Graphic Parameter: Stroke Color
		81.2.5 Graphic Parameter: Stroke Opacity
		81.2.6 Graphic Parameter: Arrows
		81.2.7 Inner Lines
	81.3	Filling a Path
	0_10	81.3.1 Graphic Parameter: Interior Rule
		81.3.2 Graphic Parameter: Filling Color
		81.3.3 Graphic Parameter: Fill Opacity
	81.4	Clipping a Path
	81.5	Using a Path as a Bounding Box
82	Arrov	v Tips 76
	82.1	Overview
		82.1.1 When Does PGF Draw Arrow Tips?
		82.1.2 Meta-Arrow Tips
	82.2	Declaring an Arrow Tip Kind
	82.3	Declaring a Derived Arrow Tip Kind
	82.4	Using an Arrow Tip Kind
	82.5	Predefined Arrow Tip Kinds
83	Nodo	s and Shapes 77
00	83.1	Overview
	00.1	83.1.1 Creating and Referencing Nodes
		83.1.2 Anchors
		83.1.3 Layers of a Shape
		v i
	02.0	
	83.2	9
		83.2.1 Creating Simple Nodes
		83.2.2 Creating Multi-Part Nodes
	00.0	83.2.3 Deferred Node Positioning
	83.3	Using Anchors
		83.3.1 Referencing Anchors of Nodes in the Same Picture
	09 4	83.3.2 Referencing Anchors of Nodes in Different Pictures
	83.4	Special Nodes
	83.5	Declaring New Shapes
		83.5.1 What Must Be Defined For a Shape?
		83.5.2 Normal Anchors Versus Saved Anchors
		83.5.3 Command for Declaring New Shapes

84	\mathbf{Matr}	ices 789
	84.1	Overview
	84.2	Cell Pictures and Their Alignment
	84.3	The Matrix Command
	84.4	Row and Column Spacing
	84.5	Callbacks
85		dinate and Canvas Transformations 794
	85.1	Overview
	85.2	Coordinate Transformations
		85.2.1 How PGF Keeps Track of the Coordinate Transformation Matrix 794
		85.2.2 Commands for Relative Coordinate Transformations
		85.2.3 Commands for Absolute Coordinate Transformations
		85.2.4 Saving and Restoring the Coordinate Transformation Matrix 799
	85.3	Canvas Transformations
86	Patte	$_{ m erns}$
00	86.1	Overview
	86.2	Declaring a Pattern
	86.3	Setting a Pattern
	00.0	Setting a fattern
87	Decla	aring and Using Images 805
	87.1	Overview
	87.2	Declaring an Image
	87.3	Using an Image
	87.4	Masking an Image
88	Evto	rnalizing Graphics 809
00	88.1	Overview
	88.2	Workflow Step 1: Naming Graphics
	88.3	Workflow Step 2: Generating the External Graphics
	88.4	Workflow Step 3: Including the External Graphics
	88.5	A Complete Example
	00.0	The Complete Example
89	Creat	Fing Plots 816
	89.1	Overview
	89.2	Generating Plot Streams
		89.2.1 Basic Building Blocks of Plot Streams
		89.2.2 Commands That Generate Plot Streams
	89.3	Plot Handlers
90	Lovo	red Graphics 820
90	90.1	Overview
	90.1	
	90.2 90.3	Declaring Layers 820 Using Layers 820
	00.0	Coming English Transfer Transf
91	Shad	
	91.1	Overview
	91.2	Declaring Shadings
		91.2.1 Horizontal and Vertical Shadings
		91.2.2 Radial Shadings
		91.2.3 General (Functional) Shadings
	91.3	Using Shadings
92	Trans	sparency 829
	92.1	Specifying a Uniform Opacity
	92.1	Specifying a Fading
	92.2	Transparency Groups
	<i>52.</i> 0	Transparator Oroups
93	Addi	ng libraries to pgf: temporary registers 833

94	Quick	c Commands	835		
	94.1	Quick Coordinate Commands	. 835		
	94.2	Quick Path Construction Commands	. 835		
	94.3	Quick Path Usage Commands	. 836		
	94.4	Quick Text Box Commands	. 836		
\mathbf{X}	The	System Layer	838		
95	Desig	gn of the System Layer	839		
	95.1	Driver Files	. 839		
	95.2	Common Definition Files	. 839		
96	Comr	mands of the System Layer	840		
	96.1	Beginning and Ending a Stream of System Commands			
	96.2	Path Construction System Commands			
	96.3	Canvas Transformation System Commands			
	96.4	Stroking, Filling, and Clipping System Commands			
	96.5	Graphic State Option System Commands			
	96.6	Color System Commands			
	96.7	Pattern System Commands			
	96.8	Scoping System Commands			
	96.9	Image System Commands			
	96.10	Shading System Commands			
	96.11	Transparency System Commands			
	96.12	Reusable Objects System Commands			
	96.13	Invisibility System Commands			
	96.14	Position Tracking Commands			
	96.15	Internal Conversion Commands	. 850		
97	The Soft Path Subsystem 85				
	97.1	Path Creation Process			
	97.2	Starting and Ending a Soft Path			
	97.3	Soft Path Creation Commands			
	97.4	The Soft Path Data Structure	. 852		
98	The I	Protocol Subsystem	854		
ΧI	Refe	erences and Index	855		
Inde	Index 8				

1 Introduction

The PGF package, where "PGF" is supposed to mean "portable graphics format" (or "pretty, good, functional" if you prefer...), is a package for creating graphics in an "inline" manner. It defines a number of TEX commands that draw graphics. For example, the code \tikz \draw (Opt,Opt) -- (2Opt,6pt); yields the line — and the code \tikz \fill[orange] (1ex,1ex) circle (1ex); yields •.

In a sense, when you use PGF you "program" your graphics, just as you "program" your document when you use TEX. You get all the advantages of the "TEX-approach to typesetting" for your graphics: quick creation of simple graphics, precise positioning, the use of macros, often superior typography. You also inherit all the disadvantages: steep learning curve, no WYSIWYG, small changes require a long recompilation time, and the code does not really "show" how things will look like.

1.1 Structure of the System

The PGF system consists of different layers:

System layer: This layer provides a complete abstraction of what is going on "in the driver." The driver is a program like dvips or dvipdfm that takes a .dvi file as input and generates a .ps or a .pdf file. (The pdftex program also counts as a driver, even though it does not take a .dvi file as input. Never mind.) Each driver has its own syntax for the generation of graphics, causing headaches to everyone who wants to create graphics in a portable way. PGF's system layer "abstracts away" these differences. For example, the system command \pgfsys@lineto{10pt}{10pt} extends the current path to the coordinate (10pt, 10pt) of the current {pgfpicture}. Depending on whether dvips, dvipdfm, or pdftex is used to process the document, the system command will be converted to different \special commands. The system layer is as "minimalistic" as possible since each additional command makes it more work to port PGF to a new driver.

As a user, you will not use the system layer directly.

Basic layer: The basic layer provides a set of basic commands that allow you to produce complex graphics in a much easier manner than by using the system layer directly. For example, the system layer provides no commands for creating circles since circles can be composed from the more basic Bézier curves (well, almost). However, as a user you will want to have a simple command to create circles (at least I do) instead of having to write down half a page of Bézier curve support coordinates. Thus, the basic layer provides a command \pgfpathcircle that generates the necessary curve coordinates for you.

The basic layer is consists of a *core*, which consists of several interdependent packages that can only be loaded *en bloc*, and additional *modules* that extend the core by more special-purpose commands like node management or a plotting interface. For instance, the BEAMER package uses only the core and not, say, the **shapes** modules.

Frontend layer: A frontend (of which there can be several) is a set of commands or a special syntax that makes using the basic layer easier. A problem with directly using the basic layer is that code written for this layer is often too "verbose." For example, to draw a simple triangle, you may need as many as five commands when using the basic layer: One for beginning a path at the first corner of the triangle, one for extending the path to the second corner, one for going to the third, one for closing the path, and one for actually painting the triangle (as opposed to filling it). With the tikz frontend all this boils down to a single simple METAFONT-like command:

```
\draw (0,0) -- (1,0) -- (1,1) -- cycle;
```

There are different frontends:

- The TikZ frontend is the "natural" frontend for PGF. It gives you access to all features of PGF, but it is intended to be easy to use. The syntax is a mixture of METAFONT and PSTRICKS and some ideas of myself. This frontend is *neither* a complete METAFONT compatibility layer nor a PSTRICKS compatibility layer and it is not intended to become either.
- The pgfpict2e frontend reimplements the standard IATEX {picture} environment and commands like \line or \vector using the PGF basic layer. This layer is not really "necessary" since the pict2e.sty package does at least as good a job at reimplementing the {picture} environment. Rather, the idea behind this package is to have a simple demonstration of how a frontend can be implemented.

It would be possible to implement a pgftricks frontend that maps PSTRICKS commands to PGF commands. However, I have not done this and even if fully implemented, many things that work in PSTRICKS will not work, namely whenever some PSTRICKS command relies too heavily on PostScript trickery. Nevertheless, such a package might be useful in some situations.

As a user of PGF you will use the commands of a frontend plus perhaps some commands of the basic layer. For this reason, this manual explains the frontends first, then the basic layer, and finally the system layer.

1.2 Comparison with Other Graphics Packages

PGF is not the only graphics package for T_EX. In the following, I try to give a reasonably fair comparison of the PGF-system and other packages.

- 1. The standard IATEX {picture} environment allows you to create simple graphics, but little more. This is certainly not due to a lack of knowledge or imagination on the part of IATEX's designer(s). Rather, this is the price paid for the {picture} environment's portability: It works together with all backend drivers.
- 2. The pstricks package is certainly powerful enough to create any conceivable kind of graphic, but it is not portable at all. Most importantly, it does not work with pdftex nor with any other driver that produces anything but PostScript code.
 - Compared to PGF, pstricks has a broader support base. There are many nice extra packages for special purpose situations that have been contributed by users over the last decade.
 - The TikZ syntax is more consistent than the pstricks syntax as TikZ was developed "in a more centralized manner" and also "with the shortcomings on pstricks in mind."
 - Note that a number of neat tricks that pstricks can do are impossible in PGF. In particular, pstricks has access to the powerful PostScript programming language, which allows trickery such as inline function plotting.
- 3. The xypic package is an older package for creating graphics. However, it is more difficult to use and to learn because the syntax and the documentation are a bit cryptic.
- 4. The dratex package is a small graphic package for creating a graphics. Compared to the other package, including PGF, it is very small, which may or may not be an advantage.
- 5. The metapost program is a very powerful alternative to PGF. However, it is an external program, which entails a bunch of problems. The time needed both to create a small graphic and also to compile it is much greater than in PGF. The main problem with metapost, however, is the inclusion of labels. This is much easier to achieve using PGF.
- 6. The xfig program is an important alternative to TikZ for users who do not wish to "program" their graphics as is necessary with TikZ and the other packages above. Their is a conversion program that will convert xfig graphics to both TikZ and for PGF, but it is still under construction.

1.3 Utility Packages

The PGF package comes along with a number of utility package that are not really about creating graphics and which can be used independently of PGF. However, they are bundled with PGF, partly out of convenience, partly because their functionality is closely intertwined with PGF. These utility packages are:

- 1. The pgfkeys package defines a powerful key management facility. It can be used completely independently of PGF.
- 2. The pgffor package defines a useful \foreach statement.
- 3. The pgfcalendar package defines macros for creating calendars. Typically, these calendars will be rendered using PGF's graphic engine, but you can use pgfcalendar also typeset calendars using normal text. The package also defines commands for "working" with dates.

4. The pgfpages package is used to assemble several pages into a single page. It provides commands for assembling several "virtual pages" into a single "physical page." The idea is that whenever TeX has a page ready for "shipout," pgfpages interrupts this shipout and instead stores the page to be shipped out in a special box. When enough "virtual pages" have been accumulated in this way, they are scaled down and arranged on a "physical page," which then really shipped out. This mechanism allows you to create "two page on one page" versions of a document directly inside LATEX without the use of any external programs. However, pgfpages can do quite a lot more than that. You can use it to put logos and watermark on pages, print up to 16 pages on one page, add borders to pages, and more.

1.4 How to Read This Manual

This manual describes both the design of the PGF system and its usage. The organization is very roughly according to "user-friendliness." The commands and subpackages that are easiest and most frequently used are described first, more low-level and esoteric features are discussed later.

If you have not yet installed PGF, please read the installation first. Second, it might be a good idea to read the tutorial. Finally, you might wish to skim through the description of TikZ. Typically, you will not need to read the sections on the basic layer. You will only need to read the part on the system layer if you intend to write your own frontend or if you wish to port PGF to a new driver.

The "public" commands and environments provided by the pgf package are described throughout the text. In each such description, the described command, environment or option is printed in red. Text shown in green is optional and can be left out.

1.5 Authors and Acknowledgements

The bulk of the PGF system and its documentation was written by Till Tantau. A further member of the main team is Mark Wibrow, who is responsible, for example, for the PGF mathematical engine, many shapes, the decoration engine, and matrices. The third member is Christian Feuersänger who contributed the floating point library, image externalization, extended key processing, and automatic hyperlinks in the manual.

Furthermore, occasional contributions have been made by Christophe Jorssen, Jin-Hwan Cho, Olivier Binda, Matthias Schulz, Renée Ahrens, Stephan Schuster, and Thomas Neumann.

Additionally, numerous people have contributed to the PGF system by writing emails, spotting bugs, or sending libraries and patches. Many thanks to all these people, who are too numerous to name them all!

1.6 Getting Help

When you need help with PGF and TikZ, please do the following:

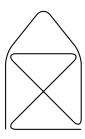
- 1. Read the manual, at least the part that has to do with your problem.
- 2. If that does not solve the problem, try having a look at the sourceforge development page for PGF and TikZ (see the title of this document). Perhaps someone has already reported a similar problem and someone has found a solution.
- 3. On the website you will find numerous forums for getting help. There, you can write to help forums, file bug reports, join mailing lists, and so on.
- 4. Before you file a bug report, especially a bug report concerning the installation, make sure that this is really a bug. In particular, have a look at the .log file that results when you TeX your files. This .log file should show that all the right files are loaded from the right directories. Nearly all installation problems can be resolved by looking at the .log file.
- 5. As a last resort you can try to email me (Till Tantau) or, if the problem concerns the mathematical engine, Mark Wibrow. I do not mind getting emails, I simply get way too many of them. Because of this, I cannot guarantee that your emails will be answered timely or even at all. Your chances that your problem will be fixed are somewhat higher if you mail to the PGF mailing list (naturally, I read this list and answer questions when I have the time).
- 6. Please, do not phone me in my office (unless, of course, you attend one of my lectures).

Part I

Tutorials and Guidelines

$by\ Till\ Tantau$

To help you get started with TikZ, instead of a long installation and configuration section, this manual starts with tutorials. They explain all the basic and some of the more advanced features of the system, without going into all the details. This part also contains some guidelines on how you should proceed when creating graphics using TikZ.



```
\tikz \draw[thick,rounded corners=8pt]
(0,0) -- (0,2) -- (1,3.25) -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
```

2 Tutorial: A Picture for Karl's Students

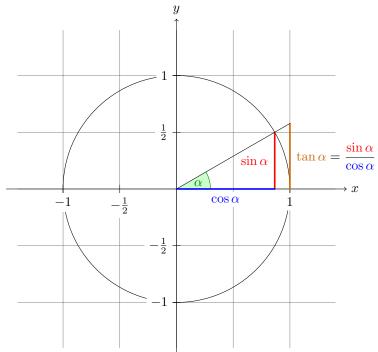
This tutorial is intended for new users of PGF and TikZ. It does not give an exhaustive account of all the features of TikZ or PGF, just of those that you are likely to use right away.

Karl is a math and chemistry high-school teacher. He used to create the graphics in his worksheets and exams using LaTeX's {picture} environment. While the results were acceptable, creating the graphics often turned out to be a lengthy process. Also, there tended to be problems with lines having slightly wrong angles and circles also seemed to be hard to get right. Naturally, his students could not care less whether the lines had the exact right angles and they find Karl's exams too difficult no matter how nicely they were drawn. But Karl was never entirely satisfied with the result.

Karl's son, who was even less satisfied with the results (he did not have to take the exams, after all), told Karl that he might wish to try out a new package for creating graphics. A bit confusingly, this package seems to have two names: First, Karl had to download and install a package called PGF. Then it turns out that inside this package there is another package called TikZ, which is supposed to stand for "TikZ ist kein Zeichenprogramm." Karl finds this all a bit strange and TikZ seems to indicate that the package does not do what he needs. However, having used GNU software for quite some time and "GNU not being Unix," there seems to be hope yet. His son assures him that TikZ's name is intended to warn people that TikZ is not a program that you can use to draw graphics with your mouse or tablet. Rather, it is more like a "graphics language."

2.1 Problem Statement

Karl wants to put a graphic on the next worksheet for his students. He is currently teaching his students about sine and cosine. What he would like to have is something that looks like this (ideally):



The angle α is 30° in the example $(\pi/6 \text{ in radians})$. The sine of α , which is the height of the red line, is

$$\sin \alpha = 1/2$$
.

By the Theorem of Pythagoras we have $\cos^2 \alpha + \sin^2 \alpha = 1$. Thus the length of the blue line, which is the cosine of α , must be

$$\cos \alpha = \sqrt{1 - 1/4} = \frac{1}{2}\sqrt{3}.$$

This shows that $\tan \alpha$, which is the height of the orange line, is

$$\tan \alpha = \frac{\sin \alpha}{\cos \alpha} = 1/\sqrt{3}.$$

2.2 Setting up the Environment

In TikZ, to draw a picture, at the start of the picture you need to tell TEX or LATEX that you want to start a picture. In LATEX this is done using the environment $\{tikzpicture\}$, in plain TEX you just use $\{tikzpicture\}$ to start the picture and $\{tikzpicture\}$ to end it.

2.2.1 Setting up the Environment in LATEX

Karl, being a LATEX user, thus sets up his file as follows:

```
\documentclass{article} % say
\usepackage{tikz}
\begin{document}

We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
\end{document}
```

When executed, that is, run via pdflatex or via latex followed by dvips, the resulting will contain something that looks like this:

```
We are working on \begin{tikzpicture} \draw (-1.5,0) -- (1.5,0); \draw (0,-1.5) -- (0,1.5); \end{tikzpicture}.

We are working on
```

Admittedly, not quite the whole picture, yet, but we do have the axes established. Well, not quite, but we have the lines that make up the axes drawn. Karl suddenly has a sinking feeling that the picture is still some way off.

Let's have a more detailed look at the code. First, the package tikz is loaded. This package is a so-called "frontend" to the basic PGF system. The basic layer, which is also described in this manual, is somewhat more, well, basic and thus harder to use. The frontend makes things easier by providing a simpler syntax.

Inside the environment there are two \draw commands. They mean: "The path, which is specified following the command up to the semicolon, should be drawn." The first path is specified as (-1.5,0) -- (0,1.5), which means "a straight line from the point at position (-1.5,0) to the point at position (0,1.5)." Here, the positions are specified within a special coordinate system in which, initially, one unit is 1cm.

Karl is quite pleased to note that the environment automatically reserves enough space to encompass the picture.

2.2.2 Setting up the Environment in Plain TEX

Karl's wife Gerda, who also happens to be a math teacher, is not a LATEX user, but uses plain TEX since she prefers to do things "the old way." She can also use TikZ. Instead of \usepackage{tikz} she has to write \input tikz.tex and instead of \begin{tikzpicture} she writes \tikzpicture and instead of \end{tikzpicture} she writes \endtikzpicture.

Thus, she would use:

```
%% Plain TeX file
\input tikz.tex
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
We are working on
\tikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\endtikzpicture.
\bye
```

Gerda can typeset this file using either pdftex or tex together with dvips. TikZ will automatically discern which driver she is using. If she wishes to use dvipdfm together with tex, she either needs to modify the file pgf.cfg or can write \def\pgfsysdriver{pgfsys-dvipdfm.def} somewhere before she inputs tikz.tex or pgf.tex.

2.2.3 Setting up the Environment in ConTeXt

Karl's uncle Hans uses ConTEXt. Like Gerda, Hans can also use TikZ. Instead of \usepackage{tikz} he says \usemodule[tikz]. Instead of \begin{tikzpicture} he writes \starttikzpicture and instead of \end{tikzpicture} he writes \stoptikzpicture.

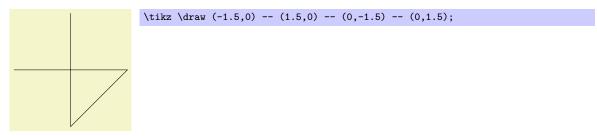
His version of the example looks like this:

```
%% ConTeXt file
\usemodule[tikz]
\starttext
We are working on
\starttikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\stoptikzpicture.
\stoptext
```

Hans will now typeset this file in the usual way using texexec¹.

2.3 Straight Path Construction

The basic building block of all pictures in TikZ is the path. A path is a series of straight lines and curves that are connected (that is not the whole picture, but let us ignore the complications for the moment). You start a path by specifying the coordinates of the start position as a point in round brackets, as in (0,0). This is followed by a series of "path extension operations." The simplest is --, which we used already. It must be followed by another coordinate and it extends the path in a straight line to this new position. For example, if we were to turn the two paths of the axes into one path, the following would result:



Karl is a bit confused by the fact that there is no {tikzpicture} environment, here. Instead, the little command \tikz is used. This command either takes one argument (starting with an opening brace as in \tikz{\draw (0,0) -- (1.5,0)}, which yields ______) or collects everything up to the next semicolon and puts it inside a {tikzpicture} environment. As a rule of thumb, all TikZ graphic drawing commands must occur as an argument of \tikz or inside a {tikzpicture} environment. Fortunately, the command \draw will only be defined inside this environment, so there is little chance that you will accidentally do something wrong here.

2.4 Curved Path Construction

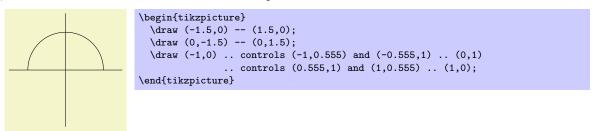
The next thing Karl wants to do is to draw the circle. For this, straight lines obviously will not do. Instead, we need some way to draw curves. For this, TikZ provides a special syntax. One or two "control points" are needed. The math behind them is not quite trivial, but here is the basic idea: Suppose you are at point x and the first control point is y. Then the curve will start "going in the direction of y at x," that is, the tangent of the curve at x will point toward y. Next, suppose the curve should end at z and the second support point is w. Then the curve will, indeed, end at z and the tangent of the curve at point z will go through w.

Here is an example (the control points have been added for clarity):

The general syntax for extending a path in a "curved" way is .. controls $\langle first\ control\ point \rangle$ and $\langle second\ control\ point \rangle$.. $\langle end\ point \rangle$. You can leave out the and $\langle second\ control\ point \rangle$, which causes the first one to be used twice.

¹Note that PGF/TikZ is not supported by recent ConTEXt versions (like mark IV, the LuaTEX-aware part of ConTEXt).

So, Karl can now add the first half circle to the picture:



Karl is happy with the result, but finds specifying circles in this way to be extremely awkward. Fortunately, there is a much simpler way.

2.5 Circle Path Construction

In order to draw a circle, the path construction operation circle can be used. This operation is followed by a radius in round brackets as in the following example: (Note that the previous position is used as the center of the circle.)

```
\tikz \draw (0,0) circle (10pt);
```

You can also append an ellipse to the path using the ellipse operation. Instead of a single radius you can specify two of them, one for the x-direction and one for the y-direction, separated by and:

```
\tikz \draw (0,0) ellipse (20pt and 10pt);
```

To draw an ellipse whose axes are not horizontal and vertical, but point in an arbitrary direction (a "turned ellipse" like \bigcirc) you can use transformations, which are explained later. The code for the little ellipse is \tikz \draw[rotate=30] (0,0) ellipse (6pt and 3pt);, by the way.

So, returning to Karl's problem, he can write \draw (0,0) circle (1cm); to draw the circle:

```
\begin{tikzpicture}
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle (1cm);
\end{tikzpicture}
```

At this point, Karl is a bit alarmed that the circle is so small when he wants the final picture to be much bigger. He is pleased to learn that TikZ has powerful transformation options and scaling everything by a factor of three is very easy. But let us leave the size as it is for the moment to save some space.

2.6 Rectangle Path Construction

The next things we would like to have is the grid in the background. There are several ways to produce it. For example, one might draw lots of rectangles. Since rectangles are so common, there is a special syntax for them: To add a rectangle to the current path, use the rectangle path construction operation. This operation should be followed by another coordinate and will append a rectangle to the path such that the previous coordinate and the next coordinates are corners of the rectangle. So, let us add two rectangles to the picture:

```
\beg
\dd \dd \dd \dd \dd \dd \end
```

```
\begin{tikzpicture}
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle (1cm);
\draw (0,0) rectangle (0.5,0.5);
\draw (-0.5,-0.5) rectangle (-1,-1);
\end{tikzpicture}
```

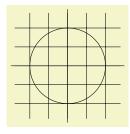
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed."

So, Karl is about to resort to simply drawing four vertical and four horizontal lines using the nice \draw command, when he learns that there is a grid path construction operation.

2.7 Grid Path Construction

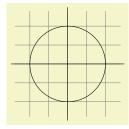
The grid path operation adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the grid operation. For example, the code \tikz \draw[step=2pt] (0,0) grid (10pt,10pt); produces . Note how the optional argument for \draw can be used to specify a grid width (there are also xstep and ystep to define the steppings independently). As Karl will learn soon, there are lots of things that can be influenced using such options.

For Karl, the following code could be used:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw[step=.5cm] (-1.4,-1.4) grid (1.4,1.4);
  \end{tikzpicture}
```

Having another look at the desired picture, Karl notices that it would be nice for the grid to be more subdued. (His son told him that grids tend to be distracting if they are not subdued.) To subdue the grid, Karl adds two more options to the \draw command that draws the grid. First, he uses the color gray for the grid lines. Second, he reduces the line width to very thin. Finally, he swaps the ordering of the commands so that the grid is drawn first and everything else on top.



```
\begin{tikzpicture}
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle (1cm);
\end{tikzpicture}
```

2.8 Adding a Touch of Style

Instead of the options gray, very thin Karl could also have said help lines. Styles are predefined sets of options that can be used to organize how a graphic is drawn. By saying help lines you say "use the style that I (or someone else) has set for drawing help lines." If Karl decides, at some later point, that grids should be drawn, say, using the color blue!50 instead of gray, he could provide the following option somewhere:

```
help lines/.style={color=blue!50,very thin}
```

The effect of this "style setter" is that in the current scope or environment the help lines option has the same effect as color=blue!50, very thin.

Using styles makes your graphics code more flexible. You can change the way things look easily in a consistent manner. Normally, styles are defined at the beginning of a picture. However, you may sometimes wish to define a style globally, so that all pictures of your document can use this style. Then you can easily change the way all graphics look by changing this one style. In this situation you can use the **\tikzset** command at the beginning of the document as in

```
\tikzset{help lines/.style=very thin}
```

To build a hierarchy of styles you can have one style use another. So in order to define a style Karl's grid that is based on the grid style Karl could say

```
\tikzset{Karl's grid/.style={help lines,color=blue!50}}
...
\draw[Karl's grid] (0,0) grid (5,5);
```

Styles are made even more powerful by parametrization. This means that, like other options, styles can also be used with a parameter. For instance, Karl could parameterize his grid so that, by default, it is blue, but he could also use another color.

```
\begin{tikzpicture}
  [Karl's grid/.style ={help lines,color=#1!50},
  Karl's grid/.default=blue]

\draw[Karl's grid] (0,0) grid (1.5,2);
  \draw[Karl's grid=red] (2,0) grid (3.5,2);
\end{tikzpicture}
```

2.9 Drawing Options

Karl wonders what other options there are that influence how a path is drawn. He saw already that the $color=\langle color \rangle$ option can be used to set the line's color. The option $draw=\langle color \rangle$ does nearly the same, only it sets the color for the lines only and a different color can be used for filling (Karl will need this when he fills the arc for the angle).

He saw that the style very thin yields very thin lines. Karl is not really surprised by this and neither is he surprised to learn that thin yields thin lines, thick yields thick lines, very thick yields very thick lines, ultra thick yields really, really thick lines and ultra thin yields lines that are so thin that low-resolution printers and displays will have trouble showing them. He wonders what gives lines of "normal" thickness. It turns out that thin is the correct choice. This seems strange to Karl, but his son explains him that LATEX has two commands called \thinlines and \thicklines and that \thinlines gives the line width of "normal" lines, more precisely, of the thickness that, say, the stem of a letter like "T" or "i" has. Nevertheless, Karl would like to know whether there is anything "in the middle" between thin and thick. There is: semithick.

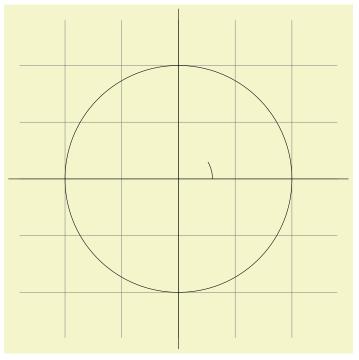
Another useful thing one can do with lines is to dash or dot them. For this, the two styles dashed and dotted can be used, yielding __ and Both options also exist in a loose and a dense version, called loosely dashed, densely dashed, loosely dotted, and densely dotted. If he really, really needs to, Karl can also define much more complex dashing patterns with the dash pattern option, but his son insists that dashing is to be used with utmost care and mostly distracts. Karl's son claims that complicated dashing patterns are evil. Karl's students do not care about dashing patterns.

2.10 Arc Path Construction

Our next obstacle is to draw the arc for the angle. For this, the arc path construction operation is useful, which draws part of a circle or ellipse. This arc operation must be followed by a triple in rounded brackets, where the components of the triple are separated by colons. The first two components are angles, the last one is a radius. An example would be (10:80:10pt), which means "an arc from 10 degrees to 80 degrees on a circle of radius 10pt." Karl obviously needs an arc from 0° to 30°. The radius should be something relatively small, perhaps around one third of the circle's radius. This gives: (0:30:3mm).

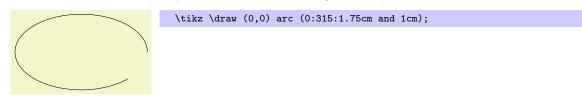
When one uses the arc path construction operation, the specified arc will be added with its starting point at the current position. So, we first have to "get there."

Karl thinks this is really a bit small and he cannot continue unless he learns how to do scaling. For this, he can add the [scale=3] option. He could add this option to each \draw command, but that would be awkward. Instead, he adds it to the whole environment, which causes this option to apply to everything within.



```
\begin{tikzpicture}[scale=3]
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
  \end{tikzpicture}
```

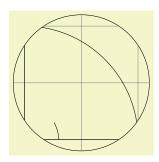
As for circles, you can specify "two" radii in order to get an elliptical arc.



2.11 Clipping a Path

In order to save space in this manual, it would be nice to clip Karl's graphics a bit so that we can focus on the "interesting" parts. Clipping is pretty easy in TikZ. You can use the \clip command clip all subsequent drawing. It works like \draw , only it does not draw anything, but uses the given path to clip everything subsequently.

You can also do both at the same time: Draw and clip a path. For this, use the \draw command and add the clip option. (This is not the whole picture: You can also use the \clip command and add the draw option. Well, that is also not the whole picture: In reality, \draw is just a shorthand for \path[draw] and \clip is a shorthand for \path[clip] and you could also say \path[draw,clip].) Here is an example:



```
\begin{tikzpicture}[scale=3]
  \clip[draw] (0.5,0.5) circle (.6cm);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
  \end{tikzpicture}
```

2.12 Parabola and Sine Path Construction

Although Karl does not need them for his picture, he is pleased to learn that there are parabola and sin and cos path operations for adding parabolas and sine and cosine curves to the current path. For the parabola operation, the current point will lie on the parabola as well as the point given after the parabola operation. Consider the following example:



```
\tikz \draw (0,0) rectangle (1,1) (0,0) parabola (1,1);
```

It is also possible to place the bend somewhere else:

```
\tikz \draw[x=1pt,y=1pt] (0,0) parabola bend (4,16) (6,12);
```

The operations sin and cos add a sine or cosine curve in the interval $[0, \pi/2]$ such that the previous current point is at the start of the curve and the curve ends at the given end point. Here are two examples:

```
A sine / curve.

A sine \tikz \draw[x=1ex,y=1ex] (0,0) sin (1.57,1); curve.

\tikz \draw[x=1.57ex,y=1ex] (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0) (0,1) cos (1,0) sin (2,-1) cos (3,0) sin (4,1);
```

2.13 Filling and Drawing

Returning to the picture, Karl now wants the angle to be "filled" with a very light green. For this he uses \fill instead of \draw. Here is what Karl does:

```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,0.75);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle (1cm);
\fill[green!20!white] (0,0) -- (3mm,0mm) arc (0:30:3mm) -- (0,0);
\end{tikzpicture}
```

The color green!20!white means 20% green and 80% white mixed together. Such color expression are possible since PGF uses Uwe Kern's xcolor package, see the documentation of that package for details on color expressions.

What would have happened, if Karl had not "closed" the path using --(0,0) at the end? In this case, the path is closed automatically, so this could have been omitted. Indeed, it would even have been better to write the following, instead:

```
\fill[green!20!white] (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
```

The --cycle causes the current path to be closed (actually the current part of the current path) by smoothly joining the first and last point. To appreciate the difference, consider the following example:



```
\begin{tikzpicture}[line width=5pt]
\draw (0,0) -- (1,0) -- (1,1) -- (0,0);
\draw (2,0) -- (3,0) -- (3,1) -- cycle;
\useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

You can also fill and draw a path at the same time using the \filldraw command. This will first draw the path, then fill it. This may not seem too useful, but you can specify different colors to be used for filling and for stroking. These are specified as optional arguments like this:

2.14 Shading

\end{tikzpicture}

Karl briefly considers the possibility of making the angle "more fancy" by *shading* it. Instead of filling the with a uniform color, a smooth transition between different colors is used. For this, \shade and \shadedraw, for shading and drawing at the same time, can be used:

```
\tikz \shade (0,0) rectangle (2,1) (3,0.5) circle (.5cm);
```

The default shading is a smooth transition from gray to white. To specify different colors, you can use options:

```
\begin{tikzpicture}[rounded corners,ultra thick]
\shade[top color=yellow,bottom color=black] (0,0) rectangle +(2,1);
\shade[left color=yellow,right color=black] (3,0) rectangle +(2,1);
\shadedraw[inner color=yellow,outer color=black,draw=yellow] (6,0) rectangle +(2,1);
\shade[ball color=green] (9,.5) circle (.5cm);
```

For Karl, the following might be appropriate:

```
begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \shadedraw[left color=gray,right color=green, draw=green!50!black]
  (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \end{tikzpicture}
```

However, he wisely decides that shadings usually only distract without adding anything to the picture.

2.15 Specifying Coordinates

Karl now wants to add the sine and cosine lines. He knows already that he can use the color= option to set the lines's colors. So, what is the best way to specify the coordinates?

There are different ways of specifying coordinates. The easiest way is to say something like (10pt,2cm). This means 10pt in x-direction and 2cm in y-directions. Alternatively, you can also leave out the units as in (1,2), which means "one times the current x-vector plus twice the current y-vector." These vectors default to 1cm in the x-direction and 1cm in the y-direction, respectively.

In order to specify points in polar coordinates, use the notation (30:1cm), which means 1cm in direction 30 degree. This is obviously quite useful to "get to the point (cos 30°, sin 30°) on the circle."

You can add a single + sign in front of a coordinate or two of them as in +(1cm,0cm) or ++(0cm,2cm). Such coordinates are interpreted differently: The first form means "1cm upwards from the previous specified position" and the second means "2cm to the right of the previous specified position, making this the new specified position." For example, we can draw the sine line as follows:

Karl used the fact $\sin 30^\circ = 1/2$. However, he very much doubts that his students know this, so it would be nice to have a way of specifying "the point straight down from (30:1cm) that lies on the x-axis." This is, indeed, possible using a special syntax: Karl can write (30:1cm |- 0,0). In general, the meaning of $(\langle p \rangle \mid - \langle q \rangle)$ is "the intersection of a vertical line through p and a horizontal line through q."

Next, let us draw the cosine line. One way would be to say (30:1cm |- 0,0) -- (0,0). Another way is the following: we "continue" from where the sine ends:

Note that there is no -- between (30:1cm) and ++(0,-0.5). In detail, this path is interpreted as follows: "First, the (30:1cm) tells me to move by pen to $(\cos 30^{\circ}, 1/2)$. Next, there comes another coordinate specification, so I move my pen there without drawing anything. This new point is half a unit down from the last position, thus it is at $(\cos 30^{\circ}, 0)$. Finally, I move the pen to the origin, but this time drawing something (because of the --)."

To appreciate the difference between + and ++ consider the following example:

```
begin{tikzpicture}
    \def\rectanglepath{-- ++(1cm,0cm) -- ++(0cm,1cm) -- ++(-1cm,0cm) -- cycle}
    \draw (0,0) \rectanglepath;
    \draw (1.5,0) \rectanglepath;
    \end{tikzpicture}
```

By comparison, when using a single +, the coordinates are different:

```
\begin{tikzpicture}
  \def\rectanglepath{-- +(1cm,0cm) -- +(1cm,1cm) -- +(0cm,1cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
  \end{tikzpicture}
```

Naturally, all of this could have been written more clearly and more economically like this (either with a single of a double +):

```
\tikz \draw (0,0) rectangle +(1,1) (1.5,0) rectangle +(1,1);
```

2.16 Intersecting Paths

Karl is left with the line for $\tan \alpha$, which seems difficult to specify using transformations and polar coordinates. For this he needs another way of specifying coordinates: Karl can specify intersections of paths as coordinates. The line for $\tan \alpha$ starts at (1,0) and goes upward to a point that is at the intersection of a line going "up" and a line going from the origin through (30:1cm). Such computations are made available by the intersections library.

What Karl must do is to create two "invisible" paths that intersect at the position of interest. Creating paths that are not otherwise seen can be done using the \path command without any options like draw or

fill. Then, Karl can add the name path option to the path for later reference. Once the paths have been constructed, Karl can use the name intersections to assign names to the coordinate for later reference.

```
\path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm); % a bit longer, so that there is an intersection
\draw [name intersections={of=upward line and sloped line, by=x}]
[very thick,orange] (1,0) -- (x);
```

2.17 Adding Arrow Tips

Karl now wants to add the little arrow tips at the end of the axes. He has noticed that in many plots, even in scientific journals, these arrow tips seem to missing, presumably because the generating programs cannot produce them. Karl thinks arrow tips belong at the end of axes. His son agrees. His students do not care about arrow tips.

It turns out that adding arrow tips is pretty easy: Karl adds the option -> to the drawing commands for the axes:

```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
                           (30:1cm) -- +(0,-0.5);
  \draw[red, very thick]
                           (30:1cm) ++(0,-0.5) -- (0,0);
  \draw[blue, very thick]
  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=x}]
        [very thick, orange] (1,0) -- (x);
\end{tikzpicture}
```

If Karl had used the option <- instead of ->, arrow tips would have been put at the beginning of the path. The option <-> puts arrow tips at both ends of the path.

There are certain restrictions to the kind of paths to which arrow tips can be added. As a rule of thumb, you can add arrow tips only to a single open "line." For example, you should not try to add tips to, say, a rectangle or a circle. (You can try, but no guarantees as to what will happen now or in future versions.) However, you can add arrow tips to curved paths and to paths that have several segments, as in the following examples:

```
\begin{tikzpicture}
\draw [<->] (0,0) arc (180:30:10pt);
\draw [<->] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl has a more detailed look at the arrow that TikZ puts at the end. It looks like this when he zooms it: \rightarrow . The shape seems vaguely familiar and, indeed, this is exactly the end of T_EX 's standard arrow used in something like $f: A \rightarrow B$.

Karl likes the arrow, especially since it is not "as thick" as the arrows offered by many other packages. However, he expects that, sometimes, he might need to use some other kinds of arrow. To do so, Karl can say $>=\langle right\ arrow\ tip\ kind\rangle$, where $\langle right\ arrow\ tip\ kind\rangle$ is a special arrow tip specification. For example, if Karl says >=stealth, then he tells TikZ that he would like "stealth-fighter-like" arrow tips:

```
\text{\login{tikzpicture}[>=stealth]}
\draw [->] (0,0) arc (180:30:10pt);
\draw [<<-,very thick] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}</pre>
```

Karl wonders whether such a military name for the arrow type is really necessary. He is not really mollified when his son tells him that Microsoft's PowerPoint uses the same name. He decides to have his students discuss this at some point.

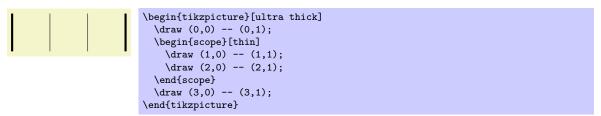
In addition to **stealth** there are several other predefined arrow tip kinds Karl can choose from, see Section 24. Furthermore, he can define arrows types himself, if he needs new ones.

2.18 Scoping

Karl saw already that there are numerous graphic options that affect how paths are rendered. Often, he would like to apply certain options to a whole set of graphic commands. For example, Karl might wish to draw three paths using a thick pen, but would like everything else to be drawn "normally."

If Karl wishes to set a certain graphic option for the whole picture, he can simply pass this option to the \tikz command or to the {tikzpicture} environment (Gerda would pass the options to \tikzpicture and Hans passes them to \starttikzpicture). However, if Karl wants to apply graphic options to a local group, he put these commands inside a {scope} environment (Gerda uses \scope and \endscope, Hans uses \startscope and \stopscope). This environment takes graphic options as an optional argument and these options apply to everything inside the scope, but not to anything outside.

Here is an example:



Scoping has another interesting effect: Any changes to the clipping area are local to the scope. Thus, if you say \clip somewhere inside a scope, the effect of the \clip command ends at the end of the scope. This is useful since there is no other way of "enlarging" the clipping area.

Karl has also already seen that giving options to commands like \draw apply only to that command. In turns out that the situation is slightly more complex. First, options to a command like \draw are not really options to the command, but they are "path options" and can be given anywhere on the path. So, instead of \draw[thin] (0,0) -- (1,0); one can also write \draw (0,0) [thin] -- (1,0); or \draw (0,0) -- (1,0) [thin];; all of these have the same effect. This might seem strange since in the last case, it would appear that the thin should take effect only "after" the line from (0,0) to (1,0) has been drawn. However, most graphic options only apply to the whole path. Indeed, if you say both thin and thick on the same path, the last option given will "win."

When reading the above, Karl notices that only "most" graphic options apply to the whole path. Indeed, all transformation options do *not* apply to the whole path, but only to "everything following them on the path." We will have a more detailed look at this in a moment. Nevertheless, all options given during a path construction apply only to this path.

2.19 Transformations

When you specify a coordinate like (1cm,1cm), where is that coordinate placed on the page? To determine the position, TikZ, TeX, and PDF or PostScript all apply certain transformations to the given coordinate in order to determine the final position on the page.

TikZ provides numerous options that allow you to transform coordinates in PGF's private coordinate system. For example, the xshift option allows you to shift all subsequent points by a certain amount:

```
\tikz \draw (0,0) -- (0,0.5) [xshift=2pt] (0,0) -- (0,0.5);
```

It is important to note that you can change transformation "in the middle of a path," a feature that is not supported by PDF or PostScript. The reason is that PGF keeps track of its own transformation matrix. Here is a more complicated example:

The most useful transformations are xshift and yshift for shifting, shift for shifting to a given point as in shift={(1,0)} or shift={+(0,0)} (the braces are necessary so that TEX does not mistake the comma for separating options), rotate for rotating by a certain angle (there is also a rotate around for rotating around a given point), scale for scaling by a certain factor, xscale and yscale for scaling only in the x-or y-direction (xscale=-1 is a flip), and xslant and yslant for slanting. If these transformation and those

that I have not mentioned are not sufficient, the cm option allows you to apply an arbitrary transformation matrix. Karl's students, by the way, do not know what a transformation matrix is.

2.20 Repeating Things: For-Loops

Karl's next aim is to add little ticks on the axes at positions -1, -1/2, 1/2, and 1. For this, it would be nice to use some kind of "loop," especially since he wishes to do the same thing at each of these positions. There are different packages for doing this. LATEX has its own internal command for this, pstricks comes along with the powerful \multido command. All of these can be used together with PGF and TikZ, so if you are familiar with them, feel free to use them. PGF introduces yet another command, called \foreach, which I introduced since I could never remember the syntax of the other packages. \foreach is defined in the package pgffor and can be used independently of PGF. TikZ includes it automatically.

In its basic form, the \foreach command is easy to use:

```
x = 1, x = 2, x = 3, \foreach \x in \{1,2,3\} \{\$x = \x\$, \}
```

The general syntax is $\langle variable \rangle$ in $\{\langle list\ of\ values \rangle\}$ $\langle commands \rangle$. Inside the $\langle commands \rangle$, the $\langle variable \rangle$ will be assigned to the different values. If the $\langle commands \rangle$ do not start with a brace, everything up to the next semicolon is used as $\langle commands \rangle$.

For Karl and the ticks on the axes, he could use the following code:

```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,1.51);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
(0:30:3mm) -- cycle;
\draw[->] (-1.5,0) -- (1.5,0);
\draw[->] (0,-1.5) -- (0,1.5);
\draw (0,0) circle (1cm);

\foreach \x in {-1cm,-0.5cm,1cm}
\draw (\x,-1pt) -- (\x,1pt);
\foreach \y in {-1cm,-0.5cm,0.5cm,1cm}
\draw (-1pt,\y) -- (1pt,\y);
\end{tikzpicture}
```

As a matter of fact, there are many different ways of creating the ticks. For example, Karl could have put the \draw ...; inside curly braces. He could also have used, say,

```
\foreach \x in {-1,-0.5,1}
\draw[xshift=\x cm] (0pt,-1pt) -- (0pt,1pt);
```

Karl is curious what would happen in a more complicated situation where there are, say, 20 ticks. It seems bothersome to explicitly mention all these numbers in the set for \foreach. Indeed, it is possible to use ... inside the \foreach statement to iterate over a large number of values (which must, however, be dimensionless real numbers) as in the following example:

```
000000000
```

```
\tikz \foreach \x in {1,...,10}
\draw (\x,0) circle (0.4cm);
```

If you provide *two* numbers before the ..., the \foreach statement will use their difference for the stepping:

```
\tikz \foreach \x in {-1,-0.5,...,1}
\draw (\x cm,-1pt) -- (\x cm,1pt);
```

We can also nest loops to create interesting effects:

1,5	2,5	3,5	4,5	5,5	7,5	8,5	9,5	10,5	11,5	12,5
1,4	2,4	3,4	4,4	5,4	7,4	8,4	9,4	10,4	11,4	12,4
1,3	2,3	3,3	4,3	5,3	7,3	8,3	9,3	10,3	11,3	12,3
1,2	2,2	3,2	4,2	5,2	7,2	8,2	9,2	10,2	11,2	12,2
1,1	2,1	3,1	4,1	5,1	7,1	8,1	9,1	10,1	11,1	12,1

```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
   \foreach \y in {1,...,5}
   {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
   }
}
```

The \foreach statement can do even trickier stuff, but the above gives the idea.

2.21 Adding Text

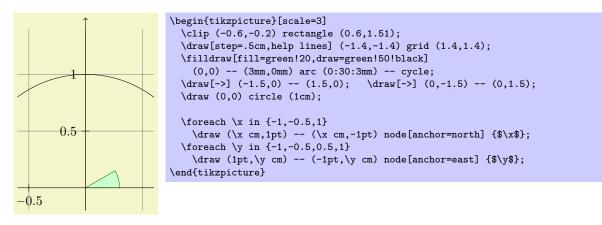
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

TikZ offers an easy-to-use and powerful system for adding text and, more generally, complex shapes to a picture at specific positions. The basic idea is the following: When TikZ is constructing a path and encounters the keyword node in the middle of a path, it reads a node specification. The keyword node is typically followed by some options and then some text between curly braces. This text is put inside a normal TeX box (if the node specification directly follows a coordinate, which is usually the case, TikZ is able to perform some magic so that it is even possible to use verbatim text inside the boxes) and then placed at the current position, that is, at the last specified position (possibly shifted a bit, according to the given options). However, all nodes are drawn only after the path has been completely drawn/filled/shaded/clipped/whatever.

```
Text at node 2

| Text at node 2 | \draw (0,0) rectangle (2,2); \draw (0.5,0.5) node [fill=examplefill] | \text{Text at \verb!node 1!} | \text{-- (1.5,1.5) node \text{Text at \verb!node 2!}; \end{\text{tikzpicture}}
```

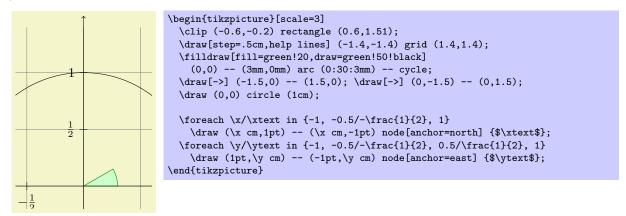
Obviously, Karl would not only like to place nodes on the last specified position, but also to the left or the right of these positions. For this, every node object that you put in your picture is equipped with several anchors. For example, the north anchor is in the middle at the upper end of the shape, the south anchor is at the bottom and the north east anchor is in the upper right corner. When you given the option anchor=north, the text will be placed such that this northern anchor will lie on the current position and the text is, thus, below the current position. Karl uses this to draw the ticks as follows:



This is quite nice, already. Using these anchors, Karl can now add most of the other text elements. However, Karl thinks that, though "correct," it is quite counter-intuitive that in order to place something below a given point, he has to use the north anchor. For this reason, there is an option called below, which does the same as anchor=north. Similarly, above right does the same as anchor=south east. In addition, below takes an optional dimension argument. If given, the shape will additionally be shifted downwards by the given amount. So, below=1pt can be used to put a text label below some point and, additionally shift it 1pt downwards.

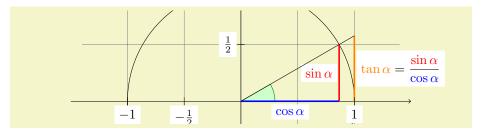
Karl is not quite satisfied with the ticks. He would like to have 1/2 or $\frac{1}{2}$ shown instead of 0.5, partly to show off the nice capabilities of T_EX and TikZ, partly because for positions like 1/3 or π it is certainly very much preferable to have the "mathematical" tick there instead of just the "numeric" tick. His students, on the other hand, prefer 0.5 over 1/2 since they are not too fond of fractions in general.

Karl now faces a problem: For the \foreach statement, the position \x should still be given as 0.5 since TikZ will not know where \frac{1}{2} is supposed to be. On the other hand, the typeset text should really be \frac{1}{2}. To solve this problem, \foreach offers a special syntax: Instead of having one variable \x, Karl can specify two (or even more) variables separated by a slash as in \x / \xtext. Then, the elements in the set over which \foreach iterates must also be of the form $\langle first \rangle / \langle second \rangle$. In each iteration, \x will be set to $\langle first \rangle$ and \xtext will be set to $\langle second \rangle$. If no $\langle second \rangle$ is given, the $\langle first \rangle$ will be used again. So, here is the new code for the ticks:



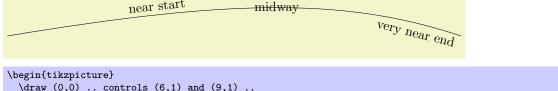
Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy solution: Karl can add the [fill=white] option to fill out the background of the text shape with a white color.

The next thing Karl wants to do is to add the labels like $\sin \alpha$. For this, he would like to place a label "in the middle of line." To do so, instead of specifying the label node {\$\sin\alpha\$} directly after one of the endpoints of the line (which would place the label at that endpoint), Karl can give the label directly after the --, before the coordinate. By default, this places the label in the middle of the line, but the poseptions can be used to modify this. Also, options like near start and near end can be used to modify this position:



```
\begin{tikzpicture}[scale=3]
  \clip (-2,-0.2) rectangle (2,0.8);
 \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
 \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
 (0:30:3mm) -- cycle;
 \draw[->] (-1.5,0) -- (1.5,0) coordinate (x axis);
 \draw[->] (0,-1.5) -- (0,1.5) coordinate (y axis);
 \draw (0,0) circle (1cm);
 \draw[very thick,red]
   (30:1cm) -- node[left=1pt,fill=white] {\$\sin \alpha\$\} (30:1cm |- x axis);
 \draw[very thick,blue]
   (30:1cm \mid -x axis) -- node[below=2pt,fill=white] {$\cos \alphapha$} (0,0);
 \path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm);
 \draw [name intersections={of=upward line and sloped line, by=t}]
   [very thick,orange] (1,0) -- node [right=1pt,fill=white]
   {$\displaystyle \tan \alpha \color{black}=
     \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} (t);
 draw (0,0) -- (t);
 \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north,fill=white] {\$\xtext\$};
 \int \frac{1}{2}, 0.5/\frac{1}{2}, 1
   \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east,fill=white] {\$\ytext$};
\end{tikzpicture}
```

You can also position labels on curves and, by adding the sloped option, have them rotated such that they match the line's slope. Here is an example:



```
\draw (0,0) .. controls (6,1) and (9,1) ..

node[near start,sloped,above] {near start}

node {midway}

node[very near end,sloped,below] {very near end} (12,0);

\end{tikzpicture}
```

It remains to draw the explanatory text at the right of the picture. The main difficulty here lies in limiting the width of the text "label," which is quite long, so that line breaking is used. Fortunately, Karl can use the option text width=6cm to get the desired effect. So, here is the full code:

```
\begin{tikzpicture}
 [scale=3,line cap=round,
 % Styles
 axes/.style=,
 important line/.style={very thick},
 information text/.style={rounded corners,fill=red!10,inner sep=1ex}]
 % Local definitions
 \def\costhirty{0.8660256}
 % Colors
 \colorlet{anglecolor}{green!50!black}
 \colorlet{sincolor}{red}
 \colorlet{tancolor}{orange!80!black}
 \colorlet{coscolor}{blue}
 % The graphic
 \draw[help lines, step=0.5cm] (-1.4,-1.4) grid (1.4,1.4);
 \draw (0,0) circle (1cm);
 \begin{scope}[axes]
   \draw[->] (-1.5,0) -- (1.5,0) node[right] {\$x\$} coordinate(x axis);
   \draw[->] (0,-1.5) -- (0,1.5) node[above] {$y$} coordinate(y axis);
   \draw[xshift=\x cm] (Opt,1pt) -- (Opt,-1pt) node[below,fill=white] {\$\xtext\};
   \draw[yshift=\y cm] (1pt,0pt) -- (-1pt,0pt) node[left,fill=white] {$\ytext$};
 \end{scope}
 \filldraw[fill=green!20,draw=anglecolor] (0,0) -- (3mm,0pt) arc(0:30:3mm);
 \draw (15:2mm) node[anglecolor] {\$\alpha\$};
 \draw[important line, sincolor]
   (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha} (30:1cm |- x axis);
 \draw[important line,coscolor]
   (30:1cm \mid -x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);
 \path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm);
 \draw [name intersections={of=upward line and sloped line, by=t}]
   [very thick,orange] (1,0) -- node [right=1pt,fill=white]
   {$\displaystyle \tan \alpha \color{black}=
     \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} (t);
 \frac{0,0}{-}
 \draw[xshift=1.85cm]
   node[right,text width=6cm,information text]
     The {\color{anglecolor} angle $\alpha$} is $30^\circ$ in the
     example ($\pi/6$ in radians). The {\color{sincolor}sine of
       $\alpha$}, which is the height of the red line, is
     \ [
     { \color{sincolor} \sin \alpha} = 1/2.
     \1
     By the Theorem of Pythagoras ...
\end{tikzpicture}
```

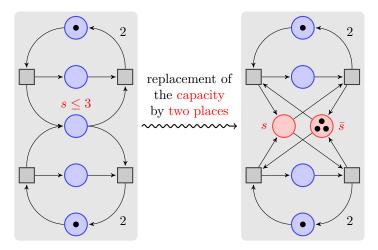
3 Tutorial: A Petri-Net for Hagen

In this second tutorial we explore the node mechanism of TikZ and PGF.

Hagen must give a talk tomorrow about his favorite formalism for distributed systems: Petri nets! Hagen used to give his talks using a blackboard and everyone seemed to be perfectly content with this. Unfortunately, his audience has been spoiled recently with fancy projector-based presentations and there seems to be a certain amount of peer pressure that this Petri nets should also be drawn using a graphic program. One of the professors at his institutes recommends TikZ for this and Hagen decides to give it a try.

3.1 Problem Statement

For his talk, Hagen wishes to create a graphic that demonstrates how a net with place capacities can be simulated by a net without capacities. The graphic should look like this, ideally:



3.2 Setting up the Environment

For the picture Hagen will need to load the TikZ package as did Karl in the previous tutorial. However, Hagen will also need to load some additional *library packages* that Karl did not need. These library packages contain additional definitions like extra arrow tips that are typically not needed in a picture and that need to be loaded explicitly.

Hagen will need to load several libraries: The arrows library for the special arrow tip used in the graphic, the decoration.pathmorphing library for the "snaking line" in the middle, the backgrounds library for the two rectangular areas that are behind the two main parts of the picture, the fit library to easily compute the sizes of these rectangles, and the positioning library for placing nodes relative to other nodes.

3.2.1 Setting up the Environment in LATEX

When using LATEX use:

```
\documentclass{article} % say
\usepackage{tikz}
\usetikzlibrary{arrows,decorations.pathmorphing,backgrounds,positioning,fit,petri}
\begin{document}
\begin{tikzpicture}
\draw (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```

3.2.2 Setting up the Environment in Plain TEX

When using plain TFX use:

```
%% Plain TeX file
\input tikz.tex
\usetikzlibrary{arrows,decorations.pathmorphing,backgrounds,positioning,fit,petri}
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
\tikzpicture
\draw (0,0) -- (1,1);
\endtikzpicture
\bye
```

3.2.3 Setting up the Environment in ConTeXt

When using $ConT_FX$ use²:

```
%% ConTeXt file
\usemodule[tikz]
\usetikzlibrary[arrows,decorations.pathmorphing,backgrounds,positioning,fit,petri]
\starttext
\starttikzpicture
\draw (0,0) -- (1,1);
\stoptikzpicture
\stoptext
```

3.3 Introduction to Nodes

In principle, we already know how to create the graphics that Hagen desires (except perhaps for the snaked line, we will come to that): We start with big light gray rectangle and then add lots of circles and small rectangle, plus some arrows.

However, this approach has numerous disadvantages: First, it is hard to change anything at a later stage. For example, if we decide to add more places to the Petri nets (the circles are called places in Petri net theory), all of the coordinates change and we need to recalculate everything. Second, it is hard to read the code for the Petri net as it is just a long and complicated list of coordinates and drawing commands – the underlying structure of the Petri net is lost.

Fortunately, TikZ offers a powerful mechanism for avoiding the above problems: nodes. We already came across nodes in the previous tutorial, where we used them to add labels to Karl's graphic. In the present tutorial we will see that nodes are much more powerful.

A node is a small part of a picture. When a node is created, you provide a position where the node should be drawn and a *shape*. A node of shape circle will be drawn as a circle, a node of shape rectangle as a rectangle, and so on. A node may also contain some text, which is why Karl used nodes to show text. Finally, a node can get a *name* for later reference.

In Hagen's picture we will use nodes for the places and for the transitions of the Petri net (the places are the circles, the transitions are the rectangles). Let us start with the upper half of the left Petri net. In this upper half we have three places and two transitions. Instead of drawing three circles and two rectangles, we use three nodes of shape circle and two nodes of shape rectangle.

Hagen notes that this does not quite look like the final picture, but it seems like a good first step.

Let us have a more detailed look at the code. The whole picture consists of a single path. Ignoring the node operations there is not much going on in this path: It is just a sequence of coordinates with nothing "happening" between them. Indeed, even if something were to happen like a line-to or a curve-to, the \path command would not "do" anything with the resulting path. So, all the magic must be in the node commands.

In the previous tutorial we learned that a node will add a piece of text at the last coordinate. Thus, each of the five nodes is added at a different position. In the above code, this text is empty (because of the

²Note that PGF/TikZ is not supported by recent ConTEXt versions (like mark IV, the LuaTEX-aware part of ConTEXt).

empty {}). So, why do we see anything at all? The answer is the draw option for the node operation: It causes the "shape around the text" to be drawn.

So, the code (0,2) node [shape=circle,draw] {} means the following: "In the main path, add a move-to to the coordinate (0,2). Then, temporarily suspend the construction of the main path while the node is built. This node will be a circle around an empty text. This circle is to be drawn, but not filled or otherwise used. Once this whole node is constructed, it is saved until after the main path is finished. Then, it is drawn." The following (0,1) node [shape=circle,draw] {} then has the following effect: "Continue the main path with a move-to to (0,1). Then construct a node at this position also. This node is also shown after the main path is finished." And so on.

3.4 Placing Nodes Using the At Syntax

Hagen now understands how the node operation adds nodes to the path, but it seems a bit silly to create a path using the \path operation, consisting of numerous superfluous move-to operations, only to place nodes. He is pleased to learn that there are ways to add nodes in a more sensible manner.

First, the node operation allows one to add at ($\langle coordinate \rangle$) in order to directly specify where the node should be placed, sidestepping the rule that nodes are placed on the last coordinate. Hagen can then write the following:

```
\begin{tikzpicture}
\path node at (0,2) [shape=circle,draw] {}

node at (0,1) [shape=circle,draw] {}

node at (0,0) [shape=circle,draw] {}

node at (1,1) [shape=rectangle,draw] {}

node at (-1,1) [shape=rectangle,draw] {};

\end{tikzpicture}
```

Now Hagen is still left with a single empty path, but at least the path no longer contains strange movetos. It turns out that this can be improved further: The \node command is an abbreviation for \path node, which allows Hagen to write:

Hagen likes this syntax much better than the previous one. Note that Hagen has also omitted the shape= since, like color=, TikZ allows you to omit the shape= if there is no confusion.

3.5 Using Styles

Feeling adventurous, Hagen tries to make the nodes look nicer. In the final picture, the circles and rectangle should be filled with different colors, resulting in the following code:

While this looks nicer in the picture, the code starts to get a bit ugly. Ideally, we would like our code to transport the message "there are three places and two transitions" and not so much which filling colors should be used.

To solve this problem, Hagen uses styles. He defines a style for places and another style for transitions:

3.6 Node Size

Before Hagen starts naming and connecting the nodes, let us first make sure that the nodes get their final appearance. They are still too small. Indeed, Hagen wonders why they have any size at all, after all, the text is empty. The reason is than TikZ automatically adds some space around the text. The amount is set using the option inner sep. So, to increase the size of the nodes, Hagen could write:

However, this is not really the best way to achieve the desired effect. It is much better to use the minimum size option instead. This option allows Hagen to specify a minimum size that the node should have. If the nodes actually needs to be bigger because of a longer text, it will be larger, but if the text is empty, then the node will have minimum size. This option is also useful to ensure that several nodes containing different amounts of text have the same size. The options minimum height and minimum width allow you to specify the minimum height and width independently.

So, what Hagen needs to do is to provide minimum size for the nodes. To be on the safe side, he also sets inner sep=0pt. This ensures that the nodes will really have size minimum size and not, for very small minimum sizes, the minimal size necessary to encompass the automatically added space.

3.7 Naming Nodes

Hagen's next aim is to connect the nodes using arrows. This seems like a tricky business since the arrows should not start in the middle of the nodes, but somewhere on the border and Hagen would very much like to avoid computing these positions by hand.

Fortunately, PGF will perform all the necessary calculations for him. However, he first has to assign names to the nodes so that he can reference them later on.

There are two ways to name a node. The first is the use the name= option. The second method is to write the desired name in parentheses after the node operation. Hagen thinks that this second method seems strange, but he will soon change his opinion.

Hagen is pleased to note that the names help in understanding the code. Names for nodes can be pretty arbitrary, but they should not contain commas, periods, parentheses, colons, and some other special characters. However, they can contain underscores and hyphens.

The syntax for the node operation is quite liberal with respect to the order in which node names, the at specifier, and the options must come. Indeed, you can even have multiple option blocks between the node and the text in curly braces, they accumulate. You can rearrange them arbitrarily and perhaps the following might be preferable:

3.8 Placing Nodes Using Relative Placement

Although Hagen still wishes to connect the nodes, he first wishes to address another problem again: The placement of the nodes. Although he likes the at syntax, in this particular case he would prefer placing the nodes "relative to each other." So, Hagen would like to say that the critical 1 node should be below the waiting 1 node, wherever the waiting 1 node might be. There are different ways of achieving this, but the nicest one in Hagen's case is the below option:

```
\begin{tikzpicture}
  \node[place] (waiting) {};
  \node[place] (critical) [below=of waiting] {};
  \node[place] (semaphore) [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical] {};
  \end{tikzpicture}
```

With the positioning library loaded, when an option like below is followed by of, then the position of the node is shifted in such a manner that it is placed at the distance node distance in the specified direction of the given direction. The node distance is either the distance between the centers of the nodes (when the on grid option is set to true) or the distance between the borders (when the on grid option is set to false, which is the default).

Even though the above code has the same effect as the earlier code, Hagen can pass it to his colleagues who will be able to just read and understand it, perhaps without even having to see the picture.

3.9 Adding Labels Next to Nodes

Before we have a look at how Hagen can connect the nodes, let us add the capacity " $s \le 3$ " to the bottom node. For this, two approaches are possible:

1. Hagen can just add a new node above the north anchor of the semaphore node.

This is a general approach that will "always work."

2. Hagen can use the special label option. This option is given to a node and it causes another node to be added next to the node where the option is given. Here is the idea: When we construct the semaphore node, we wish to indicate that we want another node with the capacity above it. For this, we use the option label=above:\$s\le 3\$. This option is interpreted as follows: We want a node above the semaphore node and this node should read "s ≤ 3." Instead of above we could also use things like below left before the colon or a number like 60.

```
\begin{tikzpicture} \\ \node[place] & (waiting) & \{\}; \\ \node[place] & (critical) & [below=of waiting] & \{\}; \\ \node[place] & (semaphore) & [below=of critical, \\ \node[transition] & (leave critical) & [right=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & \{\}; \\ \node[transition] & (enter critical) & [left=of critical] & (enter critical) & [left=of critical] & (enter critical) & [left=of critical] & (enter critical) & [left=o
```

It is also possible to give multiple label options, this causes multiple labels to be drawn.

```
\tikz \node [circle,draw,label=60:$60^\circ$,label=below:$-90^\circ$] {my circle};

my circle

-90°
```

Hagen is not fully satisfied with the label option since the label is not red. To achieve this, he has two options: First, he can redefine the every label style. Second, he can add options to the label's node. These options are given following the label=, so he would write label=[red]above:\$s\le3\$. However, this does not quite work since TEX thinks that the] closes the whole option list of the semaphore node. So, Hagen has to add braces and writes label={[red]above:\$s\le3\$}. Since this looks a bit ugly, Hagen decides to redefine the every label style.

```
\begin{tikzpicture}[every label/.style={red}]
                     (waiting)
  \node[place]
                                                           {}:
  \node[place]
                    (critical)
                                      [below=of waiting]
                                                           {};
                                      [below=of critical,
  \node[place]
                    (semaphore)
                                       label=above:$s\le3$] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition]
                    (enter critical) [left=of critical]
\end{tikzpicture}
```

3.10 Connecting Nodes

It is now high time to connect the nodes. Let us start with something simple, namely with the straight line from enter critical to critical. We want this line to start at the right side of enter critical and to end at the left side of critical. For this, we can use the *anchors* of the nodes. Every node defines a whole bunch of anchors that lie on its border or inside it. For example, the center anchor is at the center of the node, the west anchor is on the left of the node, and so on. To access the coordinate of a node, we use a coordinate that contains the node's name followed by a dot, followed by the anchor's name:

```
\begin{tikzpicture}
                    (waiting)
  \node[place]
                                                           {};
  \node[place]
                    (critical)
                                      [below=of waiting]
                                                           {};
  \node[place]
                                      [below=of critical] {};
                    (semaphore)
  \node[transition] (leave critical)
                                      [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]
  \draw [->] (critical.west) -- (enter critical.east);
\end{tikzpicture}
```

Next, let us tackle the curve from waiting to enter critical. This can be specified using curves and controls:

```
\begin{tikzpicture}
  \node[place]
                    (waiting)
                                                           {};
  \node[place]
                    (critical)
                                      [below=of waiting]
                                                           {};
                                      [below=of critical] {};
  \node[place]
                    (semaphore)
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]
  \draw [->] (enter critical.east) -- (critical.west);
  \draw [->] (waiting.west) .. controls +(left:5mm) and +(up:5mm)
                            .. (enter critical.north);
\end{tikzpicture}
```

Hagen sees how he can now add all his edges, but the whole process seems a but awkward and not very flexible. Again, the code seems to obscure the structure of the graphic rather than showing it.

So, let us start improving the code for the edges. First, Hagen can leave out the anchors:

```
\begin{tikzpicture}
  \node[place]
                     (waiting)
  \node[place]
                                      [below=of waiting]
                    (critical)
                                                           {}:
  \node[place]
                     (semaphore)
                                      [below=of critical] {};
  \node[transition]
                    (leave critical)
                                      [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]
  \draw [->] (enter critical) -- (critical);
  \draw [->] (waiting) .. controls +(left:8mm) and +(up:8mm)
                        .. (enter critical);
\end{tikzpicture}
```

Hagen is a bit surprised that this works. After all, how did TikZ know that the line from enter critical to critical should actually start on the borders? Whenever TikZ encounters a whole node name as a "coordinate," it tries to "be smart" about the anchor that it should choose for this node. Depending on what happens next, TikZ will choose an anchor that lies on the border of the node on a line to the next coordinate or control point. The exact rules are a bit complex, but the chosen point will usually be correct – and when it is not, Hagen can still specify the desired anchor by hand.

Hagen would now like to simplify the curve operation somehow. It turns out that this can be accomplished using a special path operation: the to operation. This operation takes many options (you can even define new ones yourself). One pair of options is useful for Hagen: The pair in and out. These options take angles at which a curve should leave or reach the start or target coordinates. Without these options, a straight line is drawn:

```
\begin{tikzpicture}
  \node[place]
                    (waiting)
                                                           {}:
  \node[place]
                    (critical)
                                      [below=of waiting]
                                                           {};
  \node[place]
                    (semaphore)
                                      [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical] {};
  \draw [->] (enter critical) to
                                                   (critical);
  \draw [->] (waiting)
                               to [out=180,in=90] (enter critical);
\end{tikzpicture}
```

There is another option for the to operation, that is even better suited to Hagen's problem: The bend right option. This option also takes an angle, but this angle only specifies the angle by which the curve is bent to the right:

```
\begin{tikzpicture}
  \node[place]
                    (waiting)
                                                          {};
  \node[place]
                                      [below=of waiting]
                    (critical)
  \node[place]
                                      [below=of critical] {}:
                    (semaphore)
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical] {};
  \draw [->] (enter critical) to
                                                  (critical);
  \draw [->] (waiting)
                              to [bend right=45] (enter critical);
  \draw [->] (enter critical) to [bend right=45] (semaphore);
\end{tikzpicture}
```

It is now time for Hagen to learn about yet another way of specifying edges: Using the edge path operation. This operation is very similar to the to operation, but there is one important difference: Like a node the edge generated by the edge operation is not part of the main path, but is added only later. This may not seem very important, but it has some nice consequences. For example, every edge can have its own arrow tips and its own color and so on and, still, all the edges can be given on the same path. This allows Hagen to write the following:

```
\begin{tikzpicture}
  \node[place]
                    (waiting)
                                                           {};
  \node[place]
                    (critical)
                                      [below=of waiting]
                                                           {};
  \node[place]
                                      [below=of critical] {};
                    (semaphore)
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]
    edge [->]
                             (critical)
    edge [<-,bend left=45]
                             (waiting)
    edge [->,bend right=45] (semaphore);
\end{tikzpicture}
```

Each edge caused a new path to be constructed, consisting of a to between the node enter critical and the node following the edge command.

The finishing touch is to introduce two styles pre and post and to use the bend angle=45 option to set the bend angle once and for all:

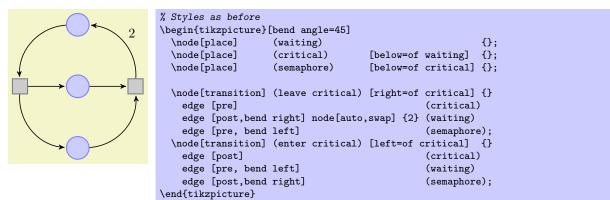
```
% Styles place and transition as before
\begin{tikzpicture}
  [bend angle=45,
  pre/.style={<-,shorten <=1pt,>=stealth',semithick},
  post/.style={->,shorten >=1pt,>=stealth',semithick}]
  \node[place]
                    (waiting)
                                                           {};
  \node[place]
                    (critical)
                                      [below=of waiting]
                                                           {};
  \node[place]
                     (semaphore)
                                      [below=of critical] {}:
  \node[transition] (leave critical) [right=of critical] {}
    edge [pre]
                            (critical)
    edge [post,bend right] (waiting)
    edge [pre, bend left]
                           (semaphore);
  \node[transition] (enter critical) [left=of critical] {}
    edge [post]
                            (critical)
    edge [pre, bend left]
                            (waiting)
    edge [post,bend right] (semaphore);
\end{tikzpicture}
```

3.11 Adding Labels Next to Lines

The next thing that Hagen needs to add is the "2" at the arcs. For this Hagen can use TikZ's automatic node placement: By adding the option auto, TikZ will position nodes on curves and lines in such a way that they are not on the curve but next to it. Adding swap will mirror the label with respect to the line. Here is a general example:

What is happening here? The nodes are given somehow inside the to operation! When this is done, the node is placed on the middle of the curve or line created by the to operation. The auto option then causes the node to be moved in such a way that it does not lie on the curve, but next to it. In the example we provide even two nodes on each to operation.

For Hagen that auto option is not really necessary since the two "2" labels could also easily be placed "by hand." However, in a complicated plot with numerous edges automatic placement can be a blessing.



3.12 Adding the Snaked Line and Multi-Line Text

With the node mechanism Hagen can now easily create the two Petri nets. What he is unsure of is how he can create the snaked line between the nets.

For this he can use a *decoration*. To draw the snaked line, Hagen only needs to set the two options decoration=snake and decorate on the path. This causes all lines of the path to be replaced by snakes. It is also possible to use snakes only in certain parts of a path, but Hagen will not need this.

```
\text{\text{begin}{tikzpicture}}
\draw [->,decorate,decoration=snake] (0,0) -- (2,0);
\end{tikzpicture}
```

Well, that does not look quite right, yet. The problem is that the snake happens to end exactly at the position where the arrow begins. Fortunately, there is an option that helps here. Also, the snake should be a bit smaller, which can be influenced by even more options.

```
\tikzpicture}
\draw [->,decorate,
    decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0);
\end{tikzpicture}
```

Now Hagen needs to add the text above the snake. This text is a bit challenging since it is a multi-line text. Hagen has two options for this: First, he can specify an align=center and then use the \\ command to enforce the line breaks at the desired positions.

Instead of specifying the line breaks "by hand," Hagen can also specify a width for the text and let TEX perform the line breaking for him:

```
replacement of
the capacity
by two places

(0,0) -- (3,0)
node [above,text width=3cm,align=center,midway]

{
    replacement of the \textcolor{red}{capacity} by
    \textcolor{red}{two places}
```

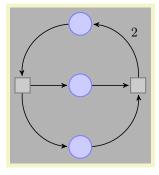
3.13 Using Layers: The Background Rectangles

\end{tikzpicture}

Hagen still needs to add the background rectangles. These are a bit tricky: Hagen would like to draw the rectangles *after* the Petri nets are finished. The reason is that only then can he conveniently refer to the coordinates that make up the corners of the rectangle. If Hagen draws the rectangle first, then he needs to know the exact size of the Petri net – which he does not.

The solution is to use *layers*. When the background library is loaded, Hagen can put parts of his picture inside a scope with the **on background layer** option. Then this part of the picture becomes part of the layer that is given as an argument to this environment. When the {tikzpicture} environment ends, the layers are put on top of each other, starting with the background layer. This causes everything drawn on the background layer to be behind the main text.

The next tricky question is, how big should the rectangle be? Naturally, Hagen can compute the size "by hand" or using some clever observations concerning the x- and y-coordinates of the nodes, but it would be nicer to just have TikZ compute a rectangle into which all the nodes "fit." For this, the fit library can be used. It defines the fit options, which, when given to a node, causes the node to be resized and shifted such that it exactly covers all the nodes and coordinates given as parameters to the fit option.



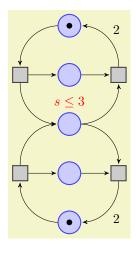
```
% Styles as before
\begin{tikzpicture}[bend angle=45]
  \node[place]
                    (waiting)
                                      [below=of waiting]
  \node[place]
                     (critical)
                                                           {};
                                      [below=of critical] {};
  \node[place]
                    (semaphore)
  \node[transition] (leave critical) [right=of critical] {}
    edge [pre]
                                                 (critical)
    edge [post,bend right] node[auto,swap] {2} (waiting)
    edge [pre, bend left]
                                                 (semaphore);
  \node[transition] (enter critical) [left=of critical] {}
                                                 (critical)
    edge [post]
    edge [pre, bend left]
                                                 (waiting)
    edge [post,bend right]
                                                 (semaphore);
  \begin{scope}[on background layer]
    \node [fill=black!30,fit=(waiting) (critical) (semaphore)
             (leave critical) (enter critical)] {};
  \end{scope}
\end{tikzpicture}
```

3.14 The Complete Code

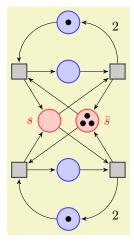
Hagen has now finally put everything together. Only then does he learn that there is already a library for drawing Petri nets! It turns out that this library mainly provides the same definitions as Hagen did. For example, it defines a place style in a similar way as Hagen did. Adjusting the code so that it uses the library shortens Hagen code a bit, as shown in the following.

First, Hagen needs less style definitions, but he still needs to specify the colors of places and transitions.

Now comes the code for the nets:



```
\node [place,tokens=1] (w1)
                                                                  {};
\node [place]
                        (c1)
                            [below=of w1]
                                                                  {}:
\node [place]
                        (s)
                             [below=of c1,label=above:$s\le 3$]
                                                                  {};
\node [place]
                        (c2) [below=of s]
                                                                  {};
\node [place, tokens=1] (w2) [below=of c2]
                                                                  {};
\node [transition] (e1) [left=of c1] {}
                                          (w1)
 edge [pre,bend left]
 edge [post,bend right]
                                          (s)
 edge [post]
                                          (c1):
\node [transition] (e2) [left=of c2] {}
                                          (w2)
 edge [pre,bend right]
 edge [post,bend left]
                                          (s)
                                          (c2);
 edge [post]
\node [transition] (11) [right=of c1] {}
  edge [pre]
                                          (c1)
  edge [pre,bend left]
                                          (s)
 edge [post,bend right] node[swap] {2} (w1);
\node [transition] (12) [right=of c2] {}
 edge [pre]
                                          (c2)
 edge [pre,bend right]
                                          (s)
  edge [post,bend left]
                                          (w2);
                         node {2}
```



```
\begin{scope}[xshift=6cm]
  \node [place,tokens=1]
                              (w1')
                                                                {};
                              (c1') [below=of w1']
  \node [place]
                                                                {};
  \node [red place]
                              (s1') [below=of c1',xshift=-5mm]
          [label=left:$s$]
                                                                {};
  \node [red place,tokens=3] (s2') [below=of c1',xshift=5mm]
          [label=right:$\bar s$]
                                                                {};
                              (c2') [below=of s1',xshift=5mm]
  \node [place]
                                                                {};
  \node [place,tokens=1]
                              (w2') [below=of c2']
                                                                {};
  \node [transition] (e1') [left=of c1'] {}
    edge [pre,bend left]
                                            (w1')
                                            (s1')
    edge [post]
    edge [pre]
                                            (s2')
    edge [post]
                                            (c1');
  \node [transition] (e2') [left=of c2'] {}
    edge [pre,bend right]
                                            (w2')
    edge [post]
                                            (s1')
                                            (s2')
    edge [pre]
    edge [post]
                                            (c2');
  \node [transition] (11') [right=of c1'] {}
    edge [pre]
                                            (c1')
    edge [pre]
                                            (s1')
                                            (s2')
    edge [post]
    edge [post,bend right] node[swap] {2} (w1');
  \node [transition] (12') [right=of c2'] {}
                                            (c2')
    edge [pre]
    edge [pre]
                                            (s1')
    edge [post]
                                           (s2')
    edge [post,bend left] node {2}
                                           (w2');
\end{scope}
```

The code for the background and the snake is the following:

4 Tutorial: Euclid's Amber Version of the *Elements*

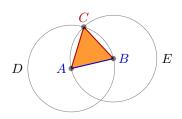
In this third tutorial we have a look at how TikZ can be used to draw geometric constructions.

Euclid is currently quite busy writing his new book series, whose working title is "Elements" (Euclid is not quite sure whether this title will convey the message of the series to future generations correctly, but he intends to change the title before it goes to the publisher). Up to know, he wrote down his text and graphics on papyrus, but his publisher suddenly insists that he must submit in electronic form. Euclid tries to argue with the publisher that electronics will only be discovered thousands of years later, but the publisher informs him that the use of papyrus is no longer cutting edge technology and Euclid will just have to keep up with modern tools.

Slightly disgruntled, Euclid starts converting his papyrus entitled "Book I, Proposition I" to an amber version.

4.1 Book I, Proposition I

The drawing on his papyrus looks like this:³



Proposition I

To construct an equilateral triangle on a given finite straight line.

Let AB be the given finite straight line. It is required to construct an equilateral triangle on the straight line AB.

Describe the circle BCD with center A and radius AB. Again describe the circle ACE with center B and radius BA. Join the straight lines CA and CB from the point C at which the circles cut one another to the points A and B.

Now, since the point A is the center of the circle CDB, therefore AC equals AB. Again, since the point B is the center of the circle CAE, therefore BC equals BA. But AC was proved equal to AB, therefore each of the straight lines AC and BC equals AB. And things which equal the same thing also equal one another, therefore AC also equals BC. Therefore the three straight lines AC, AB, and BC equal one another. Therefore the triangle ABC is equilateral, and it has been constructed on the given finite straight line AB.

Let us have a look at how Euclid can turn this into TikZ code.

4.1.1 Setting up the Environment

As in the previous tutorials, Euclid needs to load TikZ, together with some libraries. These libraries are calc, intersections, through, and backgrounds. Depending on which format⁴ he uses, Euclid would use one of the following in the preamble:

```
% For LaTeX:
\usepackage{tikz}
\usetikzlibrary{calc,intersections,through,backgrounds}
```

```
% For plain TeX:
\input tikz.tex
\usetikzlibrary{calc,intersections,through,backgrounds}
```

```
% For ConTeXt:
\usemodule[tikz]
\usetikzlibrary[calc,intersections,through,backgrounds]
```

³The text is taken from the wonderful interactive version of Euclid's Elements by David E. Joyce, to be found on his website at Clark University.

⁴Note that PGF/TikZ is not supported by recent ConTEXt versions (like mark IV, the LuaTEX-aware part of ConTEXt).

4.1.2 The Line AB

The first part of the picture that Euclid wishes to draw is the line AB. That is easy enough, something like \d raw (0,0) -- (2,1); might do. However, Euclid does not wish to reference the two points A and B as (0,0) and (2,1) subsequently. Rather, he wishes to just write A and B. Indeed, the whole point of his book is that the points A and B can be arbitrary and all other points (like C) are constructed in terms of their positions. It would not do if Euclid were to write down the coordinates of C explicitly.

So, Euclid starts with defining two coordinates using the \coordinate command:

```
\begin{tikzpicture}
  \coordinate (A) at (0,0);
  \coordinate (B) at (1.25,0.25);

  \draw[blue] (A) -- (B);
  \end{tikzpicture}
```

That was easy enough. What is missing at this point are the labels for the coordinates. Euclid does not want them *on* the points, but next to them. He decides to use the label option:

```
A begin{tikzpicture}
\text{color{blue}{$A$}} (A) at (0,0);
\text{coordinate [label=right:\textcolor{blue}{$B$}] (B) at (1.25,0.25);
\text{draw[blue] (A) -- (B);}
\text{end{tikzpicture}}
```

At this point, Euclid decides that it would be even nicer if the points A and B were in some sense "random." Then, neither Euclid nor the reader can make the mistake of taking "anything for granted" concerning these position of these points. Euclid is pleased to learn that there is a rand function in TikZ that does exactly what he needs: It produces a number between -1 and 1. Since TikZ can do a bit of math, Euclid can change the coordinates of the points as follows:

```
\coordinate [...] (A) at (0+0.1*rand,0+0.1*rand);
\coordinate [...] (B) at (1.25+0.1*rand,0.25+0.1*rand);
```

This works fine. However, Euclid is not quite satisfied since he would prefer that the "main coordinates" (0,0) and (1.25,0.25) are "kept separate" from the perturbation 0.1(rand, rand). This means, he would like to specify that coordinate A as "The point that is at (0,0) plus one tenth of the vector (rand, rand)."

It turns out that the calc library allows him to do exactly this kind of computation. When this library is loaded, you can use special coordinates that start with (\$ and end with \$) rather than just (and). Inside these special coordinates you can give a linear combination of coordinates. (Note that the dollar signs are only intended to signal that a "computation" is going on; no mathematical typesetting is done.)

The new code for the coordinates is the following:

```
\coordinate [...] (A) at ($ (0,0) + .1*(rand,rand) $);
\coordinate [...] (B) at ($ (1.25,0.25) + .1*(rand,rand) $);
```

Note that if a coordinate in such a computation has a factor (like .1), you must place a * directly before the opening parenthesis of the coordinate. You can nest such computations.

4.1.3 The Circle Around A

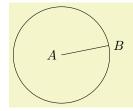
The first tricky construction is the circle around A. We will see later how to do this in a very simple manner, but first let us do it the "hard" way.

The idea is the following: We draw a circle around the point A whose radius is given by the length of the line AB. The difficulty lies in computing the length of this line.

Two ideas "nearly" solve this problem: First, we can write (\$ (A) - (B) \$) for the vector that is the difference between A and B. All we need is the length of this vector. Second, given two numbers x and y, one can write veclen(x,y) inside a mathematical expression. This gives the value $\sqrt{x^2 + y^2}$, which is exactly the desired length.

The only remaining problem is to access the x- and y-coordinate of the vector AB. For this, we need a new concept: the *let operation*. A let operation can be given anywhere on a path where a normal path operation like a line-to or a move-to is expected. The effect of a let operation is to evaluate some coordinates and to assign the results to special macros. These macros make it easy to access the x- and y-coordinates of the coordinates.

Euclid would write the following:



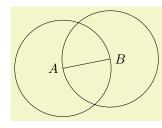
Each assignment in a let operation starts with p, usually followed by a $\langle digit \rangle$. Then comes an equal sign and a coordinate. The coordinate is evaluated and the result is stored internally. From then on you can use the following expressions:

- 1. $\langle x \langle digit \rangle$ yields the x-coordinate of the resulting point.
- 2. $\forall y \langle digit \rangle$ yields the y-coordinate of the resulting point.
- 3. $\p\langle digit \rangle$ yields the same as $\x\langle digit \rangle$, $\y\langle digit \rangle$.

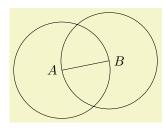
You can have multiple assignments in a let operation, just separate them with commas. In later assignments you can already use the results of earlier assignments.

Note that \p1 is not a coordinate in the usual sense. Rather, it just expands to a string like 10pt,20pt. So, you cannot write, for instance, (\p1.center) since this would just expand to (10pt,20pt.center), which makes no sense.

Next, we want to draw both circles at the same time. Each time the radius is veclen(x1,y1). It seems natural to compute this radius only once. For this, we can also use a let operation: Instead of writing $p1 = \ldots$, we write $n2 = \ldots$ Here, "n" stands for "number" (while "p" stands for "point"). The assignment of a number should be followed by a number in curly braces.

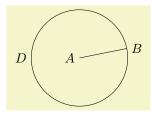


In the above example, you may wonder, what $\n1$ would yield? The answer is that it would be undefined – the \p , \x , and \y macros refer to the same logical point, while the \n macro has "its own namespace." We could even have replaced $\n2$ in the example by $\n1$ and it would still work. Indeed, the digits following these macros are just normal TeX parameters. We could also use a longer name, but then we have to use curly braces:



At the beginning of this section it was promised that there is an easier way to create the desired circle. The trick is to use the through library. As the name suggests, it contains code for creating shapes that go through a given point.

The option that we are looking for is circle through. This option is given to a *node* and has the following effects: First, it causes the node's inner and outer separations to be set to zero. Then it sets the shape of the node to circle. Finally, it sets the radius of the node such that it goes through the parameter given to circle through. This radius is computed in essentially the same way as above.



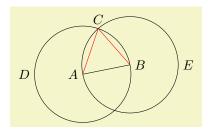
```
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \node [draw,circle through=(B),label=left:$D$] at (A) {};
  \end{tikzpicture}
```

4.1.4 The Intersection of the Circles

Euclid can now draw the line and the circles. The final problem is to compute the intersection of the two circles. This computation is a bit involved if you want to do it "by hand." Fortunately, the intersection library allows us to compute the intersection of arbitrary paths.

The idea is simple: First, you "name" two paths using the name path option. Then, at some later point, you can use the option name intersections, which creates coordinates called intersection-1, intersection-2, and so on at all intersections of the paths. Euclid assigns the names D and E to the paths of the two circles (which happen to be the same names as the nodes themselves, but nodes and their paths live in different "namespaces").



```
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

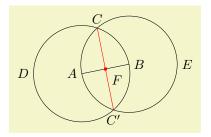
\node (D) [name path=D,draw,circle through=(B),label=left:$D$] at (A) {};
  \node (E) [name path=E,draw,circle through=(A),label=right:$E$] at (B) {};

% Name the coordinates, but do not draw anything:
  \path [name intersections={of=D and E}];

\coordinate [label=above:$C$] (C) at (intersection-1);

\draw [red] (A) -- (C);
  \draw [red] (B) -- (C);
\end{tikzpicture}
```

It turns out that this can be further shortened: The name intersections takes an optional argument by, which lets you specify names for the coordinates and options for them. This creates more compact code. Although Euclid does not need it for the current picture, it is just a small step to computing the bisection of the line AB:



```
\begin{tikzpicture}
\coordinate [label=left:$A$] (A) at (0,0);
\coordinate [label=right:$B$] (B) at (1.25,0.25);
\draw [name path=A--B] (A) -- (B);

\node (D) [name path=D,draw,circle through=(B),label=left:$D$] at (A) {};
\node (E) [name path=E,draw,circle through=(A),label=right:$E$] at (B) {};

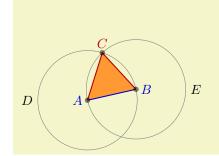
\path [name intersections={of=D and E, by={[label=above:$C$]C, [label=below:$C'$]C'}}];

\draw [name path=C--C',red] (C) -- (C');

\path [name intersections={of=A--B and C--C',by=F}];
\node [fill=red,inner sep=1pt,label=-45:$F$] at (F) {};
\end{tikzpicture}
```

4.1.5 The Complete Code

Back to Euclid's code. He introduces a few macros to make life simpler, like a A macro for typesetting a blue A. He also uses the background layer for drawing the triangle behind everything at the end.



Proposition I

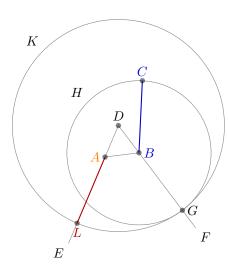
To construct an equilateral triangle on a given finite straight line.

Let AB be the given finite straight line. ...

```
\begin{tikzpicture}[thick,help lines/.style={thin,draw=black!50}]
        \def\A{\textcolor{input}{$A$}}
                                                                                                                                                 \def\B{\textcolor{input}{$B$}}
       \def\C{\textcolor{output}{$C$}}
                                                                                                                                                 \left( \int D{D}\right)
       \def\E{\$E\$}
       \colorlet{input}{blue!80!black}
                                                                                                                                                 \colorlet{output}{red!70!black}
       \colorlet{triangle}{orange}
       \coordinate [label=left:\A] (A) at (\$ (0,0) + .1*(rand,rand) \$);
       \coordinate [label=right:\B] (B) at ($ (1.25,0.25) + .1*(rand,rand) $);
       \draw [input] (A) -- (B);
       \node [name path=D,help lines,draw,label=left:\D]
                                                                                                                                                                                                                     (D) at (A) [circle through=(B)] {};
       \node [name path=E,help lines,draw,label=right:\E] (E) at (B) [circle through=(A)] {};
       \path [name intersections={of=D and E,by={[label=above:\C]C}}];
       \draw [output] (A) -- (C) -- (B);
       \foreach \point in {A,B,C}
               \fill [black,opacity=.5] (\point) circle (2pt);
       \begin{pgfonlayer}{background}
               \fill[triangle!80] (A) -- (C) -- (B) -- cycle;
       \end{pgfonlayer}
       \node [below right, text width=10cm,align=justify] at (4,3) {
               \mbox{\mbox{\mbox{$\sim$}}} \operatorname{\mbox{\mbox{$\sim$}}} \operatorname{\
                \emph{To construct an \textcolor{triangle}{equilateral triangle}
                       on a given \textcolor{input}{finite straight line}.}
                \par\vskip1em
              Let A\B be the given \text{input}\{finite straight line}. \dots
\end{tikzpicture}
```

4.2 Book I, Proposition II

The second proposition in the Elements is the following:



Proposition II

To place a straight line equal to a given straight line with one end at a given point.

Let A be the given point, and BC the given straight line. It is required to place a straight line equal to the given straight line BC with one end at the point A.

Join the straight line AB from the point A to the point B, and construct the equilateral triangle DAB on it.

Produce the straight lines AE and BF in a straight line with DA and DB. Describe the circle CGH with center B and radius BC, and again, describe the circle GKL with center D and radius DG.

Since the point B is the center of the circle CGH, therefore BC equals BG. Again, since the point D is the center of the circle GKL, therefore DL equals DG. And in these DA equals DB, therefore the remainder AL equals the remainder BG. But BC was also proved equal to BG, therefore each of the straight lines AL and BC equals BG. And things which equal the same thing also equal one another, therefore AL also equals BC.

Therefore the straight line AL equal to the given straight line BC has been placed with one end at the given point A.

4.2.1 Using Partway Calculations for the Construction of D

Euclid's construction starts with "referencing" Proposition I for the construction of the point D. Now, while we could simply repeat the construction, it seems a bit bothersome that one has to draw all these circles and do all these complicated constructions.

For this reason, TikZ supports some simplifications. First, there is a simple syntax for computing a point that is "partway" on a line from p to q: You place these two points in a coordinate calculation – remember, they start with (\$ and end with \$) – and then combine them using $!\langle part \rangle !$. A $\langle part \rangle$ of 0 refers to the first coordinate, a $\langle part \rangle$ of 1 refers to the second coordinate, and a value in between refers to a point on the line from p to q. Thus, the syntax is similar to the xcolor syntax for mixing colors.

Here is the computation of the point in the middle of the line AB:

```
\text{\lambda} \text{
```

The computation of the point D in Euclid's second proposition is a bit more complicated. It can be expressed as follows: Consider the line from X to B. Suppose we rotate this line around X for 90° and then stretch it by a factor of $\sin(60^{\circ}) \cdot 2$. This yields the desired point D. We can do the stretching using the partway modifier above, for the rotation we need a new modifier: the rotation modifier. The idea is that the second coordinate in a partway computation can be prefixed by an angle. Then the partway point is computed normally (as if no angle were given), but the resulting point is rotated by this angle around the first point.

```
begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);
  \node [fill=red,inner sep=1pt,label=below:$X$] (X) at ($ (A)!.5!(B) $) {};
  \node [fill=red,inner sep=1pt,label=above:$D$] (D) at
  ($ (X) ! {sin(60)*2} ! 90:(B) $) {};
  \draw (A) -- (D) -- (B);
  \end{tikzpicture}
```

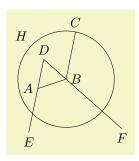
Finally, it is not necessary to explicitly name the point X. Rather, again like in the **xcolor** package, it is possible to chain partway modifiers:



```
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);
  \node [fill=red,inner sep=1pt,label=above:$D$] (D) at
  ($ (A) ! .5 ! (B) ! {sin(60)*2} ! 90:(B) $) {};
  \draw (A) -- (D) -- (B);
  \end{tikzpicture}
```

4.2.2 Intersecting a Line and a Circle

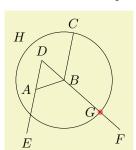
The next step in the construction is to draw a circle around B through C, which is easy enough to do using the circle through option. Extending the lines DA and DB can be done using partway calculations, but this time with a part value outside the range [0,1]:



```
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (0.75,0.25);
  \coordinate [label=above:$C$] (C) at (1,1.5);
  \draw (A) -- (B) -- (C);
  \coordinate [label=above:$D$] (D) at
    ($ (A) ! .5 ! (B) ! {\sin(60)*2} ! 90:(B) $) {};
  \node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
  \draw (D) -- ($ (D) ! 3.5 ! (B) $) coordinate [label=below:$F$] (F);
  \draw (D) -- ($ (D) ! 2.5 ! (A) $) coordinate [label=below:$E$] (E);
  \end{tikzpicture}
```

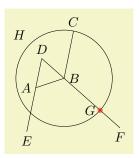
We now face the problem of finding the point G, which is the intersection of the line BF and the circle H. One way is to use yet another variant of the partway computation: Normally, a partway computation has the form $\langle p \rangle ! \langle factor \rangle ! \langle q \rangle$, resulting in the point $(1 - \langle factor \rangle) \langle p \rangle + \langle factor \rangle \langle q \rangle$. Alternatively, instead of $\langle factor \rangle$ you can also use a $\langle dimension \rangle$ between the points. In this case, you get the point that is $\langle dimension \rangle$ away from $\langle p \rangle$ on the straight line to $\langle q \rangle$.

We know that the point G is on the way from B to F. The distance is given by the radius of the circle H. Here is the code for computing H:



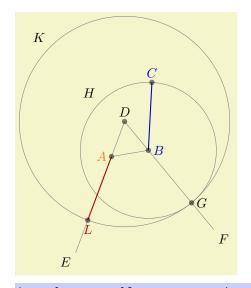
```
\node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
\path let \p1 = ($ (B) - (C) $) in
  coordinate [label=left:$G$] (G) at ($ (B) ! veclen(\x1,\y1) ! (F) $);
\fill[red,opacity=.5] (G) circle (2pt);
```

However, there is a simpler way: We can simply name the path of the circle and of the line in question and then use name intersections to compute the intersections.



```
\node (H) [name path=H,label=135:$H$,draw,circle through=(C)] at (B) {};
\path [name path=B--F] (B) -- (F);
\path [name intersections={of=H and B--F,by={[label=left:$G$]G}}];
\fill[red,opacity=.5] (G) circle (2pt);
```

4.2.3 The Complete Code

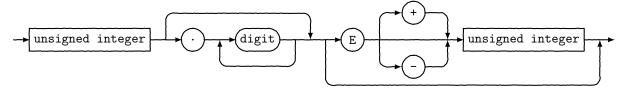


```
\begin{tikzpicture}[thick,help lines/.style={thin,draw=black!50}]
                                    \def\B{\textcolor{input}{$B$}}
 \def\A{\textcolor{orange}{$A$}}
 \def\C{\textcolor{input}{$C$}}
                                    \def\D{$D$}
 \def\E{\$E\$}
                                    \left\{ F^{F}\right\}
 \def\G\{\$G\$\}
                                    \left( \frac{H}{H}\right)
 \def\K{$K$}
                                    \def\L{\textcolor{output}{$L$}}
 \colorlet{input}{blue!80!black}
                                     \colorlet{output}{red!70!black}
 \coordinate [label=left:\A] (A) at ((0,0) + .1*(rand,rand));
 \coordinate [label=right:\B] (B) at ((1,0.2) + .1*(rand,rand));
 \coordinate [label=above:\C] (C) at ($ (1,2) + .1*(rand,rand) $);
 \draw [input] (B) -- (C);
 \draw [help lines] (A) -- (B);
 \coordinate [label=above:\D] (D) at ($ (A)!.5!(B) ! {\sin(60)*2} ! 90:(B) $);
 \draw [help lines] (D) -- ($ (D)!3.75!(A) $) coordinate [label=-135:\E] (E);
 \draw [help lines] (D) -- ($ (D)!3.75!(B) $) coordinate [label=-45:\F] (F);
 \node (H) at (B) [name path=H,help lines,circle through=(C),draw,label=135:\H] {};
 \path [name path=B--F] (B) -- (F);
 \path [name intersections={of=H and B--F,by={[label=right:\G]G}}];
 \node (K) at (D) [name path=K,help lines,circle through=(G),draw,label=135:\K] {};
 \beta = A - E (A) -- (E);
 \path [name intersections={of=K and A--E,by={[label=below:\L]L}}];
 \draw [output] (A) -- (L);
 \foreach \point in {A,B,C,D,G,L}
   \fill [black,opacity=.5] (\point) circle (2pt);
 % \node ...
\end{tikzpicture}
```

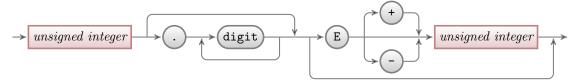
5 Tutorial: Diagrams as Simple Graphs

In this tutorial we have a look at how graphs and matrices can be used to typeset a diagram.

Ilka, who just got tenure for her professorship on Old and Lovable Programming Languages, has recently dug up a technical report entitled *The Programming Language Pascal* in the dusty cellar of the library of her university. Having been created in the good old times using pens and rules, it looks like this⁵:



For her next lecture, Ilka decides to redo this diagram, but this time perhaps a bit cleaner and perhaps also bit "cooler."



Having read the previous tutorials, Ilka knows already how to set up the environment for her diagram, namely using a tikzpicture environment. She wonders which libraries she will need. She decides that she will postpone the decision and add the necessary libraries as needed as she constructs the picture.

5.1 Styling the Nodes

The bulk of this tutorial will be about arranging the nodes and connecting them using chains, but let us start with setting up styles for the nodes.

There are two kinds of nodes in the diagram, namely what theoreticians like to call *terminals* and *nonterminals*. For the terminals, Ilka decides to use a black color, which visually shows that "nothing needs to be done about them." The nonterminals, which still need to be "processed" further, get a bit of red mixed in.

Ilka starts with the simpler nonterminals, as there are no rounded corners involved. Naturally, she sets up a style:

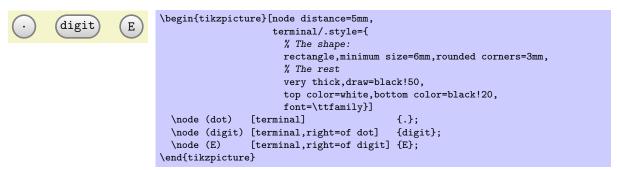
```
\begin{tikzpicture}[
unsigned integer
                          nonterminal/.style={
                             % The shape:
                             rectangle,
                             % The size:
                             minimum size=6mm.
                             % The border:
                             very thick,
                                                              % 50% red and 50% black,
                             draw=red!50!black!50,
                                                              % and that mixed with 50% white
                             % The filling:
                             top color=white,
                                                              \mbox{\ensuremath{\textit{\%}}} a shading that is white at the top...
                             bottom color=red!50!black!20, % and something else at the bottom
                             % Font
                             font=\itshape
                        \node [nonterminal] {unsigned integer};
                      \end{tikzpicture}
```

Ilka is pretty proud of the use of the minimum size option: As the name suggests, this option ensures that the node is at least 6mm by 6mm, but it will expand in size as necessary to accommodate longer text. By giving this option to all nodes, they will all have the same height of 6mm.

Styling the terminals is a bit more difficult because of the round corners. Ilka has several options how she can achieve them. One way is to use the rounded corners option. It gets a dimension as parameter and causes all corners to be replaced by little arcs with the given dimension as radius. By setting the radius

 $^{^{5}}$ The shown diagram was not scanned, but rather typeset using TikZ. The jittering lines were created using the random steps decoration.

to 3mm, she will get exactly what she needs: circles, when the shapes are, indeed, exactly 6mm by 6mm and otherwise half circles on the sides:



Another possibility is to use a shape that is specially made for typesetting rectangles with arcs on the sides (she has to use the **shapes.misc** library to use it). This shape gives Ilka much more control over the appearance. For instance, she could have an arc only on the left side, but she will not need this.

```
\begin{tikzpicture}[node distance=5mm,
digit
            Ε
                                       terminal/.style={
                                          % The shape:
                                          rounded rectangle,
                                          minimum size=6mm,
                                          % The rest
                                          very thick, draw=black!50,
                                          top color=white, bottom color=black!20,
                                          font=\ttfamily}]
                     \node (dot)
                                    [terminal]
                     \node (digit)
                                   [terminal, right=of dot]
                                                               {digit};
                     \node (E)
                                    [terminal,right=of digit] {E};
                  \end{tikzpicture}
```

At this point, she notices a problem. The baseline of the text in the nodes is not aligned:

```
\begin{tikzpicture}[node distance=5mm]
digit
                      \node (dot)
                                     [terminal]
                                                                {.};
                      \node (digit)
                                    [terminal, right=of dot]
                                                                {digit};
                                     [terminal,right=of digit] {E};
                      \draw [help lines] let \p1 = (dot.base),
                                              \p2 = (digit.base),
                                              p3 = (E.base)
                                          in (-.5,\y1) -- (3.5,\y1)
                                             (-.5, y2) -- (3.5, y2)
                                             (-.5,\y3) -- (3.5,\y3);
                    \end{tikzpicture}
```

(Ilka has moved the style definition to the preamble by saying \tikzset{terminal/.style=...}, so that she can use it in all pictures.)

For the digit and the E the difference in the baselines is almost imperceptible, but for the dot the problem is quite severe: It looks more like a multiplication dot than a period.

Ilka toys with the idea of using the base right=of... option rather than right=of... to align the nodes in such a way that the baselines are all on the same line (the base right option places a node right of something so that the baseline is right of the baseline of the other object). However, this does not have the desired effect:

The nodes suddenly "dance around"! There is no hope of changing the position of text inside a node using anchors. Instead, Ilka must use a trick: The problem of mismatching baselines is caused by the fact that . and digit and E all have different heights and depth. If they all had the same, they would all be positioned vertically in the same manner. So, all Ilka needs to do is to use the text height and text depth options to explicitly specify a height and depth for the nodes.

```
\\digit \( \begin\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}\{\texture}
```

5.2 Aligning the Nodes Using Positioning Options

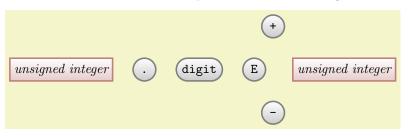
Ilka now has the "styling" of the nodes ready. The next problem is to place them in the right places. There are several ways to do this. The most straightforward is to simply explicitly place the nodes at certain coordinates "calculated by hand." For very simple graphics this is perfectly alright, but it has several disadvantages:

- 1. For more difficult graphics, the calculation may become complicated.
- 2. Changing the text of the nodes may make it necessary to recalculate the coordinates.
- 3. The source code of the graphic is not very clear since the relationships between the positions of the nodes are not made explicit.

For these reasons, Ilka decides to try out different ways of arranging the nodes on the page.

The first method is the use of *positioning options*. To use them, you need to load the **positioning** library. This gives you access to advanced implementations of options like above or left, since you can now say above=of some node in order to place a node above of some node, with the borders separated by node distance.

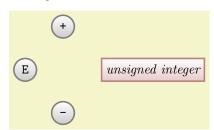
Ilka can use this to draw the place the nodes in a long row:



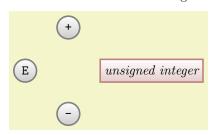
```
\begin{tikzpicture}[node distance=5mm and 5mm]
                                                     {unsigned integer};
  \node (ui1)
                 [nonterminal]
  \node (dot)
                 [terminal, right=of ui1]
                                                     {.};
  \node (digit)
                [terminal,right=of dot]
                                                     {digit};
  \node (E)
                 [terminal, right=of digit]
                                                     {E};
                                                     {+};
                 [terminal,above right=of E]
  \node (plus)
  \node (minus)
                [terminal,below right=of E]
                                                     {-};
  \node (ui2)
                 [nonterminal,below right=of plus] {unsigned integer};
\end{tikzpicture}
```

For the plus and minus nodes, Ilka is a bit startled by their placements. Shouldn't they be more to the right? The reason they are placed in that manner is the following: The north east anchor of the E node lies at the "upper start of the right arc," which, a bit unfortunately in this case, happens to be the top of the node. Likewise, the south west anchor of the + node is actually at its bottom and, indeed, the horizontal and vertical distances between the top of the E node and the bottom of the + node are both 5mm.

There are several ways of fixing this problem. The easiest way is to simply add a little bit of horizontal shift by hand:



A second way is to revert back to the idea of using a normal rectangle for the terminals, but with rounded corners. Since corner rounding does not affect anchors, she gets the following result:



A third way is to use matrices, which we will do later.

Now that the nodes have been placed, Ilka needs to add connections. Here, some connections are more difficult than others. Consider for instance the "repeat" line around the digit. One way of describing this line is to say "it starts a little to the right of digit than goes down and then goes to the left and finally ends at a point a little to the left of digit." Ilka can put this into code as follows:

```
\begin{tikzpicture}[node distance=5mm and 5mm]
digit
                    \node (dot)
                                   [terminal]
                                                                      {.};
                    \node (digit)
                                  [terminal, right=of dot]
                                                                      {digit};
                    \node (E)
                                   [terminal, right=of digit]
                                                                      {E};
                    \path (dot)
                                   edge[->] (digit) % simple edges
                          (digit) edge[->] (E);
                    \draw [->]
                       % start right of digit.east, that is, at the point that is the
                       \% linear combination of digit.east and the vector (2mm,0pt). We
                       % use the ($ ... $) notation for computing linear combinations
                       ($ (digit.east) + (2mm,0) $)
                       % Now go down
                          ++(0,-.5)
                       % And back to the left of digit.west
                       -| ($ (digit.west) - (2mm,0) $);
                  \end{tikzpicture}
```

Since Ilka needs this "go up/down then horizontally and then up/down to a target" several times, it seems sensible to define a special *to-path* for this. Whenever the **edge** command is used, it simply adds the current value of **to path** to the path. So, Ilka can set up a style that contains the correct path:

```
\begin{tikzpicture}[node distance=5mm and 5mm,
digit
                      skip loop/.style={to path={-- ++(0,-.5) -| (\tikztotarget)}}]
                    \node (dot)
                                   [terminal]
                                                                      {.};
                    \node (digit)
                                  [terminal, right=of dot]
                                                                      {digit};
                    \node (E)
                                   [terminal,right=of digit]
                                                                      {E};
                    \path (dot)
                                                      (digit) % simple edges
                                  edge[->]
                           (digit) edge[->]
                                                      (E)
                           ($ (digit.east) + (2mm,0) $)
                                   edge[->,skip loop] ($ (digit.west) - (2mm,0) $);
                  \end{tikzpicture}
```

Ilka can even go a step further and make her skip look style parameterized. For this, the skip loop's vertical offset is passed as parameter #1. Also, in the following code Ilka specifies the start and targets differently, namely as the positions that are "in the middle between the nodes."

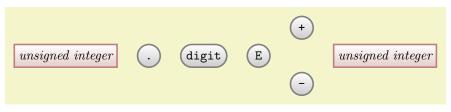
```
\begin{tikzpicture}[node distance=5mm and 5mm,
    skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}}]
  \node (dot)
                [terminal]
                                                    {.};
  \node (digit) [terminal,right=of dot]
                                                    {digit};
  \node (E)
                 [terminal, right=of digit]
                                                    {E};
  \path (dot)
                edge[->]
                                          (digit)
                                                  % simple edges
        (digit) edge[->]
                                          (E)
        ($ (digit.east)!.5!(E.west) $)
                edge[->,skip loop=-5mm] ($ (digit.west)!.5!(dot.east) $);
\end{tikzpicture}
```

5.3 Aligning the Nodes Using Matrices

Ilka is still bothered a bit by the placement of the plus and minus nodes. Somehow, having to add an explicit xshift seems too much like cheating.

A perhaps better way of positioning the nodes is to use a matrix. In TikZ matrices can be used to align quite arbitrary graphical objects in rows and columns. The syntax is very similar to the use of arrays and tables in T_FX (indeed, internally T_FX tables are used, but a lot of stuff is going on additionally).

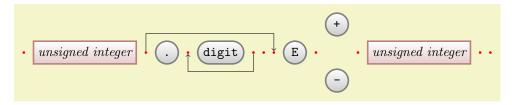
In Ilka's graphic, there will be three rows: One row containing only the plus node, one row containing the main nodes and one row containing only the minus node.



```
\begin{tikzpicture}
  \matrix[row sep=1mm,column sep=5mm] {
   % First row:
      & & & & \node [terminal] {+}; & \\
    % Second row:
    \node [nonterminal] {unsigned integer}; &
    \node [terminal]
                         {.};
    \node [terminal]
                         {digit};
    \node [terminal]
                                              Вr.
                         \{E\}:
    \node [nonterminal] {unsigned integer}; \\
    % Third row:
      & & & & \node [terminal] {-}; & \\
 };
\end{tikzpicture}
```

That was easy! By toying around with the row and columns separations, Ilka can achieve all sorts of pleasing arrangements of the nodes.

Ilka now faces the same connecting problem as before. This time, she has an idea: She adds small nodes (they will be turned into coordinates later on and be invisible) at all the places where she would like connections to start and end.



```
\begin{tikzpicture}[point/.style={circle,inner sep=0pt,minimum size=2pt,fill=red},
                   skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}}]
  \matrix[row sep=1mm,column sep=2mm] {
    % First row:
    & & & & & & & & & \node (plus) [terminal] {+};\\
    % Second row:
    \node (p1) [point]
                        {}; &
                                  \node (ui1)
                                                [nonterminal] {unsigned integer}; &
    \node (p2) [point]
                        {}; &
                                  \node (dot)
                                                [terminal]
                                                               {.};
    \node (p3) [point]
                        {}; &
                                  \node (digit)
                                                [terminal]
                                                               {digit};
                                                [point] {};
                                  \node (p5)
    \node (p4) [point]
                        {}; &
                                                                                   &
    \node (p6)
               [point]
                        {}; &
                                  \node (e)
                                                [terminal]
                                                               {E}:
                                                                                   Вr.
    \node (p7) [point]
                                  \node (ui2)
                                                [nonterminal] {unsigned integer}; &
    \node (p8) [point]
                        {}; &
    \node (p9) [point]
                        {}; &
                                  \node (p10)
    % Third row:
    & & & & & & & & & \node (minus)[terminal] {-};\\
  \path (p4) edge [->,skip loop=-5mm] (p3)
        (p2) edge [->,skip loop=5mm]
\end{tikzpicture}
```

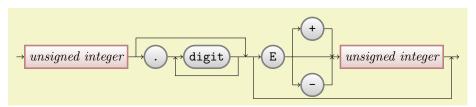
Now, it's only a small step to add all the missing edges.

5.4 The Diagram as a Graph

Matrices allow Ilka to align the nodes nicely, but the connections are not quite perfect. The problem is that the code does not really reflect the paths that underlie the diagram. For this, it seems natural enough to Ilka to use the graph library since, after all, connecting nodes by edges is exactly what happens in a graph. The graph library can both be used to connect nodes that have already been created, but it can also be used to create nodes "on the fly" and these processes can also be mixed.

5.4.1 Connecting Already Positioned Nodes

Ilka has already a fine method for positioning her nodes (using a matrix), so all that she needs is an easy way of specifying the edges. For this, she uses the \graph command (which is actually just a shorthand for \path graph). It allows her to write down edges between them in a simple way (the macro \matrixcontent contains exactly the matrix content from the previous example; no need to repeat it here):



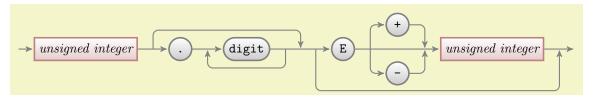
```
\begin{tikzpicture}[skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}},
                     hv path/.style={to path={-| (\tikztotarget)}},
                     vh path/.style={to path={|- (\tikztotarget)}}]
  \matrix[row sep=1mm,column sep=2mm] { \matrixcontent };
  \graph {
    (p1) \rightarrow (ui1) -- (p2) \rightarrow (dot) -- (p3) \rightarrow (digit) -- (p4)
          -- (p5) -- (p6) -> (e) -- (p7) -- (p8) -> (ui2) -- (p9) -> (p10);
    (p4) ->[skip loop=-5mm]
                              (p3);
    (p2) ->[skip loop=5mm]
                               (p5);
    (p6) ->[skip loop=-11mm] (p9);
    (p7) ->[vh path]
                             (plus) -> [hv path] (p8);
    (p7) ->[vh path]
                             (minus) -> [hv path] (p8);
 }:
\end{tikzpicture}
```

This is already pretty near to the desired result, just a few "finishing touches" are needed to style the edges more nicely.

However, Ilka does not have the feeling that the graph command is all that hot in the example. It certainly does cut down on the number of characters she has to write, but the overall graph structure is not that much clear – it is still mainly a list of paths through the graph. It would be nice to specify that, say,

there the path from (p7) sort of splits to (plus) and (minus) and then merges once more at (p8). Also, all these parentheses are bit hard to type.

It turns out that edges from a node to a whole group of nodes are quite easy to specify, as shown in the next example. Additionally, by using the use existing nodes option, Ilka can also leave out all the parentheses (again, some options have been moved outside to keep the examples shorter):



5.4.2 Creating Nodes Using the Graph Command

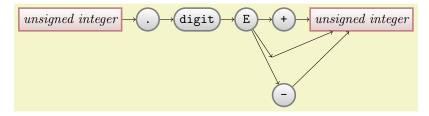
Ilka has heard that the graph command is also supposed to make it easy to create nodes, not only to connect them. This is, indeed, correct: When the use existing node option is not used and when a node name is not surrounded by parentheses, then TikZ will actually create a node whose name and text is the node name:

Not quite perfect, but we are getting somewhere. First, let us change the positioning algorithm by saying grow right sep, which causes new nodes to be placed to the right of the previous nodes with a certain fixed separation (1em by default). Second, we add some options to make the node "look nice". Third, note the funny d node above: Ilka tried writing just . there first, but got some error messages. The reason is that a node cannot be called . in TikZ, so she had to choose a different name – which is not good, since she wants a dot to be shown! The trick is to put the dot in quotation marks, this allows you to use "quite arbitrary text" as a node name:

```
unsigned integer → . → digit → E

\tikz \graph [grow right sep] {
  unsigned integer[nonterminal] -> "."[terminal] -> digit[terminal] -> E[terminal]
};
```

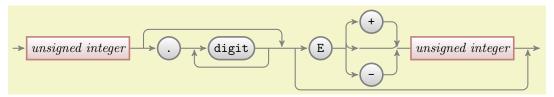
Now comes the fork to the plus and minus signs. Here, Ilka can use the grouping mechanism of the graph command to create a split:



Let us see, what is happening here. We want two unsigned integer nodes, but if we just were to use this text twice, then TikZ would have noticed that the same name was used already in the current graph and, being smart (actually too smart in this case), would have created an edge back to the already-created node. Thus, a fresh name is needed here. However, Ilka also cannot just write unsigned integer2, because she wants the original text to be shown, after all! The trick is to use a slash inside the node name: In order to "render" the node, the text following the slash is used instead of the node name, which is the text before the slash. Alternatively, the as option can be used, which also allows you to specify how a node should be rendered.

It turns out that Ilka does not need to invent a name like ui2 for a node that she will not reference again anyway. In this case, she can just leave out the name (write nothing before /), which always stands for a "fresh, anonymous" node name.

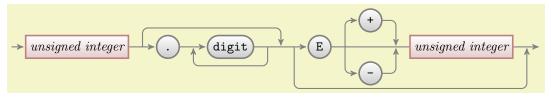
Next, Ilka needs to add some coordinates in between of some nodes where the back-loops should got and she needs to shift the nodes a bit:



```
\begin{tikzpicture}[>=stealth', thick, black!50, text=black,
                    every new ->/.style={shorten >=1pt},
                    graphs/every graph/.style={edges=rounded corners}]
  \graph [grow right sep, branch down=7mm] {
                        [coordinate] ->
    unsigned integer
                        [nonterminal] -
                        [coordinate] ->
   p1
"."
                        [terminal]
   p2
                        [coordinate] ->
   digit
                        [terminal]
    рЗ
                        [coordinate] --
   p4
                        [coordinate]
   p5
                        [coordinate] ->
    E.
                        [terminal] -
                        [coordinate] ->[vh path]
    q1
     [nodes={yshift=7mm}]
                          [terminal],
     q2/
                          [coordinate],
                          [terminal]
   } -> [hv path]
   q3
                        [coordinate] --
    /unsigned integer
                        [nonterminal] --
   p6
                        [coordinate] ->
                        [coordinate];
   p1 ->[skip loop=5mm]
   p3 ->[skip loop=-5mm]
   p5 ->[skip loop=-11mm] p6;
\end{tikzpicture}
```

All that remains to be done is to somehow get rid of the strange curves between the E and the unsigned integer. They are caused by TikZ's attempt at creating an edge that first goes vertical and then horizontal but is actually just horizontal. Additionally, the edge should not really be pointed; but it seems difficult to get rid of this since the *other* edges from q1, namely to plus and minus should be pointed.

It turns out that there is a nice way of solving this problem: You can specify that a graph is **simple**. This means that there can be at most one edge between any two nodes. Now, if you specify an edge twice, the options of the second specification "win." Thus, by adding two more lines that "correct" these edges, we get the final diagram with its complete code:



```
\tikz [>=stealth', black!50, text=black, thick,
                                   = {shorten >=1pt},
      every new ->/.style
      graphs/every graph/.style
                                   = {edges=rounded corners},
                                   = {to path={-- ++(0,#1) -| (\tikztotarget)}},
      skip loop/.style
                                   = {to path={-| (\tikztotarget)}},
      hv path/.style
      vh path/.style
                                   = {to path={|- (\tikztotarget)}},
      nonterminal/.style
        rectangle, minimum size=6mm, very thick, draw=red!50!black!50, top color=white,
        bottom color=red!50!black!20, font=\itshape, text height=1.5ex,text depth=.25ex},
      terminal/.style
                                   = {
        rounded rectangle, minimum size=6mm, very thick, draw=black!50, top color=white,
        bottom color=black!20, font=\ttfamily, text height=1.5ex, text depth=.25ex},
      shape
                                   = coordinate
 \graph [grow right sep, branch down=7mm, simple] {
   / -> unsigned integer[nonterminal] -- p1 -> "." [terminal] -- p2 -> digit[terminal] --
   p3 -- p4 -- p5 -> E[terminal] -- q1 ->[vh path]
   {[nodes={yshift=7mm}]
     "+"[terminal], q2, "-"[terminal]
   } -> [hv path]
   q3 -- /unsigned integer [nonterminal] -- p6 -> /;
                         p4;
   p1 ->[skip loop=5mm]
   p3 ->[skip loop=-5mm] p2;
   p5 ->[skip loop=-11mm] p6;
   q1 -- q2 -- q3; % make these edges plain
```

6 Tutorial: A Lecture Map for Johannes

In this tutorial we explore the tree and mind map mechanisms of TikZ.

Johannes is quite excited: For the first time he will be teaching a course all by himself during the upcoming semester! Unfortunately, the course is not on his favorite subject, which is of course Theoretical Immunology, but on Complexity Theory, but as a young academic Johannes is not likely to complain too loudly. In order to help the students get a general overview of what is going to happen during the course as a whole, he intends to draw some kind of tree or graph containing the basic concepts. He got this idea from his old professor who seems to be using these "lecture maps" with some success. Independently of the success of these maps, Johannes thinks they look quite neat.

6.1 Problem Statement

Johannes wishes to create a lecture map with the following features:

- 1. It should contain a tree or graph depicting the main concepts.
- 2. It should somehow visualize the different lectures that will be taught. Note that the lectures are not necessarily the same as the concepts since the graph may contain more concepts than will be addressed in lectures and some concepts may be addressed during more than one lecture.
- 3. The map should also contain a calendar showing when the individual lectures will be given.
- 4. The aesthetical reasons, the whole map should have a visually nice and information-rich background.

As always, Johannes will have to include the right libraries and set up the environment. Johannes is going to use the mindmap library and since he wishes to show a calendar, he will also need the calendar library. In order to put something on a background layer, it seems like a good idea to also include the backgrounds library.

6.2 Introduction to Trees

The first choice Johannes must make is whether he will organize the concepts as a tree, with root concepts and concept branches and leaf concepts, or as a general graph. The tree implicitly organizes the concepts, while a graph is more flexible. Johannes decides to compromise: Basically, the concepts will be organized as a tree. However, he will selectively add connections between concepts that are related, but which appear on different levels or branches of the tree.

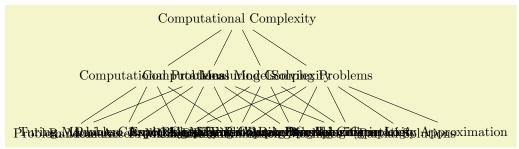
Johannes starts with a tree-like list of concepts that he feels are important in Computational Complexity:

- Computational Problems
 - Problem Measures
 - Problem Aspects
 - Problem Domains
 - Key Problems
- Computational Models
 - Turing Machines
 - Random-Access Machines
 - Circuits
 - Binary Decision Diagrams
 - Oracle Machines
 - Programming in Logic
- Measuring Complexity
 - Complexity Measures
 - Classifying Complexity
 - Comparing Complexity
 - Describing Complexity
- Solving Problems

- Exact Algorithms
- Randomization
- Fixed-Parameter Algorithms
- Parallel Computation
- Partial Solutions
- Approximation

Johannes will surely need to modify this list later on, but it looks good as a first approximation. He will also need to add a number of subtopics (like *lots* of complexity classes under the topic "classifying complexity"), but he will do this as he constructs the map.

Turning the list of topics into a TikZ-tree is easy, in principle. The basic idea is that a node can have *children*, which in turn can have children of their own, and so on. To add a child to a node, Johannes can simply write $child \{\langle node \rangle\}$ right after a node. The $\langle node \rangle$ should, in turn, be the code for creating a node. To add another node, Johannes can use child once more, and so on. Johannes is eager to try out this construct and writes down the following:



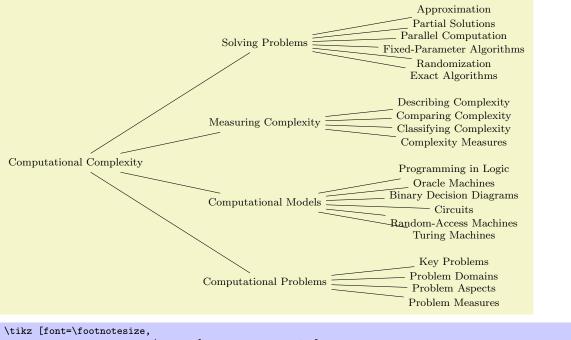
```
\tikz
 \node {Computational Complexity} % root
   child { node {Computational Problems}
     child { node {Problem Measures} }
     child { node {Problem Aspects} }
     child { node {Problem Domains} }
     child { node {Key Problems} }
   child { node {Computational Models}
     child { node {Turing Machines} }
     child { node {Random-Access Machines} }
     child { node {Circuits} }
     child { node {Binary Decision Diagrams} }
     child { node {Oracle Machines} }
     child { node {Programming in Logic} }
   child { node {Measuring Complexity}
     child { node {Complexity Measures} }
     child { node {Classifying Complexity} }
     child { node {Comparing Complexity} }
     child { node {Describing Complexity} }
   child { node {Solving Problems}
     child { node {Exact Algorithms} }
     child { node {Randomization} }
     child { node {Fixed-Parameter Algorithms} }
     child { node {Parallel Computation} }
     child { node {Partial Solutions} }
     child { node {Approximation} }
```

Well, that did not quite work out as expected (although, what, exactly, did one expect?). There are two problems:

1. The overlap of the nodes is due to the fact that TikZ is not particularly smart when it comes to placing child nodes. Even though it is possible to configure TikZ to use rather clever placement methods, TikZ has no way of taking the actual size of the child nodes into account. This may seem strange but the reason is that the child nodes are rendered and placed one at a time, so the size of the last node is not known when the first node is being processed. In essence, you have to specify appropriate level and sibling node spacings "by hand."

2. The standard computer-science-top-down rendering of a tree is rather ill-suited to visualizing the concepts. It would be better to either rotate the map by ninety degrees or, even better, to use some sort of circular arrangement.

Johannes redraws the tree, but this time with some more appropriate options set, which he found more or less by trial-and-error:

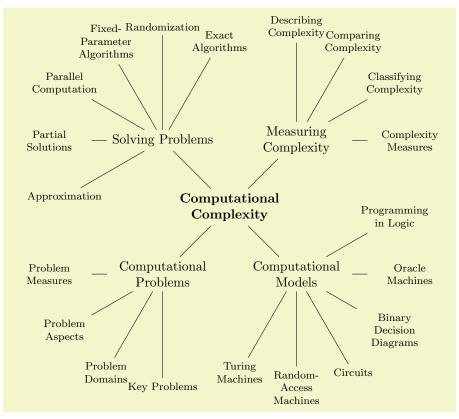


Still not quite what Johannes had in mind, but he is getting somewhere.

For configuring the tree, two parameters are of particular importance: The level distance tells TikZ the distance between (the centers of) the nodes on adjacent levels or layers of a tree. The sibling distance is, as the name suggests, the distance between (the centers of) siblings of the tree.

You can globally set these parameters for a tree by simply setting them somewhere before the tree starts, but you will typically wish them to be different for different levels of the tree. In this case, you should set styles like level 1 or level 2. For the first level of the tree, the level 1 style is used, for the second level the level 2 style, and so on. You can also set the sibling and level distances only for certain nodes by passing these options to the child command as options. (Note that the options of a node command are local to the node and have no effect on the children. Also note that it is possible to specify options that do have an effect on the children. Finally note that specifying options for children "at the right place" is an arcane art and you should peruse Section 19.4 on a rainy Sunday afternoon, if you are really interested.)

The grow key is used to configure the direction in which a tree grows. You can change growth direction "in the middle of a tree" simply by changing this key for a single child or a whole level. By including the trees library you also get access to additional growth strategies such as a "circular" growth:



Johannes is pleased to learn that he can access and manipulate the nodes of the tree like any normal node. In particular, he can name them using the name= option or the $(\langle name \rangle)$ notation and he can use any available shape or style for the trees nodes. He can connect trees later on using the normal \draw (some node) -- (another node); syntax. In essence, the child command just computes an appropriate position for a node and adds a line from the child to the parent node.

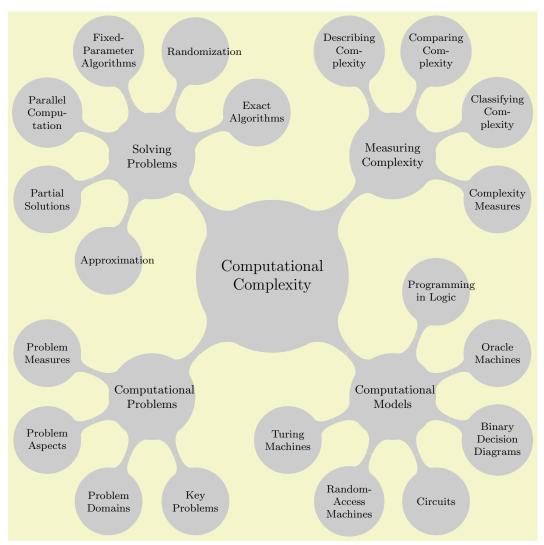
6.3 Creating the Lecture Map

Johannes now has a first possible layout for his lecture map. The next step is to make it "look nicer." For this, the mindmap library is helpful since it makes a number of styles available that will make a tree look like a nice "mind map" or "concept map."

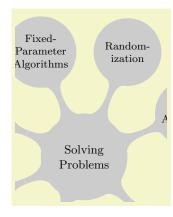
The first step is to include the mindmap library, which Johannes already did. Next, he must add one of the following options to a scope that will contain the lecture map: mindmap or large mindmap or huge mindmap. These options all have the same effect, except that for a large mindmap the predefined font size and node sizes are somewhat larger than for a standard mindmap and for a huge mindmap they are even larger. So, a large mindmap does not necessarily need to have a lot of concepts, but it will need a lot of paper.

The second step is to add the concept option to every node that will, indeed, be a concept of the mindmap. The idea is that some nodes of a tree will be real concepts, while other nodes might just be "simple children." Typically, this is not the case, so you might consider saying every node/.style=concept.

The third step is to set up the sibling angle (rather than a sibling distance) to specify the angle between sibling concepts.



When Johannes typesets the above map, TeX (rightfully) starts complaining about several overfull boxes and, indeed, words like "Randomization" stretch out beyond the circle of the concept. This seems a bit mysterious at first sight: Why does TeX not hyphenate the word? The reason is that TeX will never hyphenate the first word of a paragraph because it starts looking for "hyphenatable" letters only after a so-called glue. In order to have TeX hyphenate these single words, Johannes must use a bit of evil trickery: He inserts a hskipOpt before the word. This has no effect except for inserting an (invisible) glue before the word and, thereby, allowing TeX to hyphenate the first word also. Since Johannes does not want to add hskipOpt inside each node, he uses the execute at begin node option to make TikZ insert this text with every node.



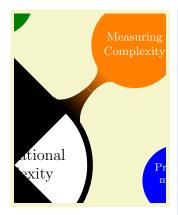
```
\begin{tikzpicture}
[mindmap,
    every node/.style={concept, execute at begin node=\hskipOpt},
    concept color=black!20,
    grow cyclic,
    level 1/.append style={level distance=4.5cm,sibling angle=90},
    level 2/.append style={level distance=3cm,sibling angle=45}]
    \clip (-1,2) rectangle ++ (-4,5);
    \node [root concept] {Computational Complexity} % root
    child { node {Computational Problems}
        child { node {Problem Measures} }
        child { node {Problem Aspects} }
        ... % as before
\end{tikzpicture}
```

In the above example a clipping was used to show only part of the lecture map, in order to save space. The same will be done in the following examples, we return to the complete lecture map at the end of this tutorial.

Johannes is now eager to colorize the map. The idea is to use different colors for different parts of the map. He can then, during his lectures, talk about the "green" or the "red" topics. This will make it easier for his students to locate the topic he is talking about on the map. Since "computational problems" somehow sounds "problematic," Johannes chooses red for them, while he picks green for the "solving problems." The topics "measuring complexity" and "computational models" get more neutral colors; Johannes picks orange and blue

To set the colors, Johannes must use the concept color option, rather than just, say, node [fill=red]. Setting just the fill color to red would, indeed, make the node red, but it would just make the node red and not the bar connecting the concept to its parent and also not its children. By comparison, the special concept color option will not only set the color of the node and its children, but it will also (magically) create appropriate shadings so that the color of a parent concept smoothly changes to the color of a child concept.

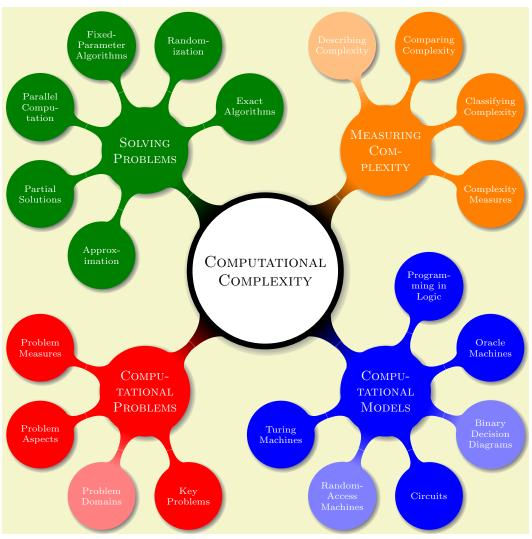
For the root concept Johannes decides to do something special: He sets the concept color to black, sets the line width to a large value, and sets the fill color to white. The effect of this is that the root concept will be encircled with a thick black line and the children are connected to the central concept via bars.



```
\begin{tikzpicture}
   every node/.style={concept, execute at begin node=\hskipOpt},
   root concept/.append style={
    concept color=black, fill=white, line width=1ex, text=black},
   text=white.
   grow cyclic,
   level 1/.append style={level distance=4.5cm, sibling angle=90},
   level 2/.append style={level distance=3cm,sibling angle=45}]
   \clip (0,-1) rectangle ++(4,5);
  \node [root concept] {Computational Complexity} % root
    child [concept color=red] { node {Computational Problems}
      child { node {Problem Measures} }
      ... % as before
   child [concept color=blue] { node {Computational Models}
     child { node {Turing Machines} }
      ... % as before
    child [concept color=orange] { node {Measuring Complexity}
      child { node {Complexity Measures} }
      ... % as before
    child [concept color=green!50!black] { node {Solving Problems}
      child { node {Exact Algorithms} }
         % as before
   };
\end{tikzpicture}
```

Johannes adds three finishing touches: First, he changes the font of the main concepts to small caps. Second, he decides that some concepts should be "faded," namely those that are important in principle and belong on the map, but which he will not talk about in his lecture. To achieve this, Johannes defines four styles, one for each of the four main branches. These styles (a) set up the correct concept

color for the whole branch and (b) define the faded style appropriately for this branch. Third, he adds a circular drop shadow, defined in the shadows library, to the concepts, just to make things look a bit more fancy.



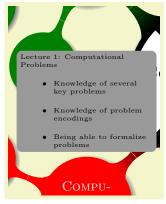
```
\begin{tikzpicture}[mindmap]
 \begin{scope}[
   every node/.style={concept, circular drop shadow,execute at begin node=\hskipOpt},
   root concept/.append style={
     concept color=black, fill=white, line width=1ex, text=black, font=\large\scshape},
   text=white,
   computational problems/.style={concept color=red,faded/.style={concept color=red!50}},
   computational models/.style={concept color=blue,faded/.style={concept color=blue!50}},
   measuring complexity/.style={concept color=orange,faded/.style={concept color=orange!50}},
   solving problems/.style={concept color=green!50!black,faded/.style={concept color=green!50!black!50}},
   grow cyclic,
   level 1/.append style={level distance=4.5cm, sibling angle=90, font=\scshape},
   level 2/.append style={level distance=3cm,sibling angle=45,font=\scriptsize}]
   \node [root concept] {Computational Complexity} % root
     child [computational problems] { node {Computational Problems}
       child
                     { node {Problem Measures} }
                      { node {Problem Aspects} }
       child
       child [faded] { node {Problem Domains} }
                     { node {Key Problems} }
       child
     child [computational models] { node {Computational Models}
       child
                     { node {Turing Machines} }
       child [faded] { node {Random-Access Machines} }
 \end{scope}
\end{tikzpicture}
```

6.4 Adding the Lecture Annotations

Johannes will give about a dozen lectures during the course "computational complexity." For each lecture he has compiled a (short) list of learning targets that state what knowledge and qualifications his students should acquire during this particular lecture (note that learning targets are not the same as the contents of a lecture). For each lecture he intends to put a little rectangle on the map containing these learning targets and the name of the lecture, each time somewhere near the topic of the lecture. Such "little rectangles" are called "annotations" by the mindmap library.

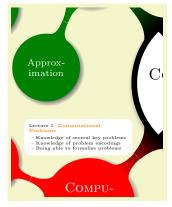
In order to place the annotations next to the concepts, Johannes must assign names to the nodes of the concepts. He could rely on TikZ's automatic naming of the nodes in a tree, where the children of a node named root are named root-1, root-2, root-3, and so on. However, since Johannes is not sure about the final order of the concepts in the tree, it seems better to explicitly name all concepts of the tree in the following manner:

The annotation style of the mind map library mainly sets up a rectangular shape of appropriate size. Johannes configures the style by defining every annotation appropriately.



```
\begin{tikzpicture}[mindmap]
  \clip (-5,-5) rectangle ++ (4,5);
  \begin{scope}[
     every node/.style={concept, circular drop shadow, ...}] % as before
    \node [root concept] (Computational Complexity)
                                                              % as before
  \end{scope}
  \begin{scope}[every annotation/.style={fill=black!40}]
    \node [annotation, above] at (Computational Problems.north) {
      Lecture 1: Computational Problems
      \begin{itemize}
      \item Knowledge of several key problems
      \item Knowledge of problem encodings
      \item Being able to formalize problems
      \end{itemize}
   }:
  \end{scope}
\end{tikzpicture}
```

Well, that does not yet look quite perfect. The spacing or the {itemize} is not really appropriate and the node is too large. Johannes can configure these things "by hand," but it seems like a good idea to define a macro that will take care of these things for him. The "right" way to do this is to define a \lecture macro that takes a list of key-value pairs as argument and produces the desired annotation. However, to keep things simple, Johannes' \lecture macro simply takes a fixed number of arguments having the following meaning: The first argument is the number of the lecture, the second is the name of the lecture, the third are positioning options like above, the fourth is the position where the node is placed, the fifth is the list of items to be shown, and the sixth is a date when the lecture will be held (this parameter is not yet needed, we will, however, need it later on).



```
\begin{tikzpicture}[mindmap,every annotation/.style={fill=white}]
\clip (-5,-5) rectangle ++ (4,5);
\begin{scope}[
    every node/.style={concept, circular drop shadow, ... % as before
    \node [root concept] (Computational Complexity) ... % as before
\end{scope}

\lecture{1}{Computational Problems}{above,xshift=-3mm}
{Computational Problems.north}{
    \item Knowledge of several key problems
    \item Knowledge of problem encodings
    \item Being able to formalize problems
}{2009-04-08}
\end{tikzpicture}
```

In the same fashion Johannes can now add the other lecture annotations. Obviously, Johannes will have some trouble fitting everything on a single A4-sized page, but by adjusting the spacing and some experimentation he can quickly arrange all the annotations as needed.

6.5 Adding the Background

Johannes has already used colors to organize his lecture map into four regions, each having a different color. In order to emphasize these regions even more strongly, he wishes to add a background coloring to each of these regions.

Adding these background colors turns out to be more tricky than Johannes would have thought. At first sight, what he needs is some sort of "color wheel" that is blue in the lower right direction and then changes smoothly to orange in the upper right direction and then to green in the upper left direction and so on. Unfortunately, there is no easy way of creating such a color wheel shading (although it can be done, in principle, but only at a very high cost, see page 454 for an example).

Johannes decides to do something a bit more basic: He creates four large rectangles, one for each of the four quadrants around the central concept, each colored with a light version of the quadrant. Then, in order to "smooth" the change between adjacent rectangles, he puts four shadings on top of them.

Since these background rectangles should go "behind" everything else, Johannes puts all his background stuff on the background layer.

In the following code, only the central concept is shown to save some space:



```
\begin{tikzpicture}[
 mindmap,
 concept color=black,
 root concept/.append style={
   concept,
   circular drop shadow,
   fill=white, line width=1ex,
   text=black, font=\large\scshape}
 \clip (-1.5,-5) rectangle ++(4,10);
 \node [root concept] (Computational Complexity) {Computational Complexity};
  \begin{pgfonlayer}{background}
   \clip (-1.5, -5) rectangle ++(4, 10);
   \colorlet{upperleft}{green!50!black!25}
   \colorlet{upperright}{orange!25}
    \colorlet{lowerleft}{red!25}
   \colorlet{lowerright}{blue!25}
    % The large rectangles:
   \fill [upperleft] (Computational Complexity) rectangle ++(-20,20);
   \fill [upperright] (Computational Complexity) rectangle ++(20,20);
                      (Computational Complexity) rectangle ++(-20,-20);
   \fill [lowerleft]
   \fill [lowerright] (Computational Complexity) rectangle ++(20,-20);
   % The shadings:
   \shade [left color=upperleft,right color=upperright]
      ([xshift=-1cm]Computational Complexity) rectangle ++(2,20);
   \shade [left color=lowerleft,right color=lowerright]
     ([xshift=-1cm]Computational Complexity) rectangle ++(2,-20);
   \shade [top color=upperleft,bottom color=lowerleft]
      ([yshift=-1cm]Computational Complexity) rectangle ++(-20,2);
    \shade [top color=upperright,bottom color=lowerright]
      ([yshift=-1cm]Computational Complexity) rectangle ++(20,2);
  \end{pgfonlayer}
\end{tikzpicture}
```

6.6 Adding the Calendar

Johannes intends to plan his lecture rather carefully. In particular, he already knows when each of his lectures will be held during the course. Naturally, this does not mean that Johannes will slavishly follow the plan and he might need longer for some subjects than he anticipated, but nevertheless he has a detailed plan of when which subject will be addressed.

Johannes intends to share this plan with his students by adding a calendar to the lecture map. In addition to serving as a reference on which particular day a certain topic will be addressed, the calendar is also useful to show the overall chronological order of the course.

In order to add a calendar to a TikZ graphic, the calendar library is most useful. The library provides the \calendar command, which takes a large number of options and which can be configured in many ways to produce just about any kind of calendar imaginable. For Johannes' purposes, a simple day list downward will be a nice option since it produces a list of days that go "downward".

Using the name option, we gave a name to the calendar, which will allow us to reference the nodes that make up the individual days of the calendar later on. For instance, the rectangular node containing the 1 that represents April 1st, 2009, can be referenced as (cal-2009-04-01). The dates option is used to specify

an interval for which the calendar should be drawn. Johannes will need several months in his calendar, but the above example only shows two weeks to save some space.

Note the if (weekend) construct. The \calendar command is followed by options and then by if-statements. These if-statements are checked for each day of the calendar and when a date passes this test, the options or the code following the if-statement is executed. In the above example, we make weekend days (Saturdays and Sundays, to be precise) lighter than normal days. (Use your favorite calendar to check that, indeed, April 5th, 2009, is a Sunday.)

As mentioned above, Johannes can reference the nodes that are used to typeset days. Recall that his \lecture macro already got passed a date, which we did not use, yet. We can now use it to place the lecture's title next to the date when the lecture will be held:

Johannes can now use this new **\lecture** command as follows (in the example, only the new part of the definition is used):

```
\tinv
                          \begin{tikzpicture}
                             \calendar [day list downward,
                                       name=cal.
                                        dates=2009-04-01 to 2009-04-14]
  Computational Problems
                              if (weekend)
                                 [black!25];
10
                            % As before:
                             \lecture{1}{Computational Problems}{above,xshift=-3mm}
                            {Computational Problems.north}{
                              \item Knowledge of several key problems
                              \item Knowledge of problem encodings
                               \item Being able to formalize problems
                            1{2009-04-08}
                          \end{tikzpicture}
```

As a final step, Johannes needs to add a few more options to the calendar command: He uses the month text option to configure how the text of a month is rendered (see Section 28 for details) and then typesets the month text at a special position at the beginning of each month.

```
\tiny
           April 2009
                                                                                                 \begin{tikzpicture}
                                                                                                       \calendar [day list downward,
   3
                                                                                                                                                month text=\%mt\ \%y0,
                                                                                                                                                month yshift=3.5em,
                                                                                                                                                name=cal,
                                                                                                                                                dates=2009-04-01 to 2009-05-01]
          Computational Problems
                                                                                                              if (weekend)
10
                                                                                                                      [black!25]
                                                                                                               if (day of month=1) {
13
                                                                                                                      \node at (0pt,1.5em) [anchor=base west] {\small\tikzmonthtext};
           Computational Models
 16
17
                                                                                                        \lecture{1}{Computational Problems}{above,xshift=-3mm}
                                                                                                        {Computational Problems.north}{
20
21
22
23
                                                                                                               \item Knowledge of several key problems
                                                                                                                \item Knowledge of problem encodings
                                                                                                               \item Being able to formalize problems
24
                                                                                                       }{2009-04-08}
27
                                                                                                       \label{lem:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma:lemma
                                                                                                       {Computational Models.north}{
30
                                                                                                                \item Knowledge of Turing machines
                                                                                                               \item Being able to compare the computational power of different
          May 2009
                                                                                                       }{2009-04-15}
                                                                                                 \end{tikzpicture}
```

6.7 The Complete Code

Putting it all together, Johannes gets the following code: First comes the definition of the \lecture command:

This is followed by the main mindmap setup...

```
\noindent
\begin{tikzpicture}
 \begin{scope}[
   mindmap,
   every node/.style={concept, circular drop shadow,execute at begin node=\hskipOpt},
   root concept/.append style={
     concept color=black,
     fill=white, line width=1ex,
     text=black, font=\large\scshape},
   text=white.
   computational problems/.style={concept color=red,faded/.style={concept color=red!50}},
   computational models/.style={concept color=blue,faded/.style={concept color=blue!50}},
   {\tt measuring\ complexity/.style=\{concept\ color=orange,faded/.style=\{concept\ color=orange!50\}\},}
   solving problems/.style={concept color=green!50!black,faded/.style={concept color=green!50!black!50}},
   level 1/.append style={level distance=4.5cm, sibling angle=90, font=\scshape},
   level 2/.append style={level distance=3cm,sibling angle=45,font=\scriptsize}]
```

... and contents:

```
\node [root concept] (Computational Complexity) {Computational Complexity} % root
   child [computational problems] { node [yshift=-1cm] (Computational Problems) {Computational Problems}
     child
                   { node (Problem Measures) {Problem Measures} }
      child
                    { node (Problem Aspects) {Problem Aspects} }
     child [faded] { node (problem Domains) {Problem Domains} }
                   { node (Key Problems) {Key Problems} }
     child
   child [computational models] { node [yshift=-1cm] (Computational Models) {Computational Models}
                   { node (Turing Machines) {Turing Machines} }
      child [faded] { node (Random-Access Machines) {Random-Access Machines} }
     child
                   { node (Circuits) {Circuits} }
     child [faded] { node (Binary Decision Diagrams) {Binary Decision Diagrams} }
              { node (Oracle Machines) {Oracle Machines} }
     child
     child
                   { node (Programming in Logic) {Programming in Logic} }
   child [measuring complexity] { node [yshift=1cm] (Measuring Complexity) {Measuring Complexity}
                    { node (Complexity Measures) {Complexity Measures} }
                    { node (Classifying Complexity) {Classifying Complexity} }
     child
                    { node (Comparing Complexity) {Comparing Complexity} }
     child
     child [faded] { node (Describing Complexity) {Describing Complexity} }
   child [solving problems] { node [yshift=1cm] (Solving Problems) {Solving Problems}
                  { node (Exact Algorithms) {Exact Algorithms} }
     child
                   { node (Randomization) {Randomization} }
     child
     child
                   { node (Fixed-Parameter Algorithms) {Fixed-Parameter Algorithms} }
     child
                   { node (Parallel Computation) {Parallel Computation} }
                   { node (Partial Solutions) {Partial Solutions} }
     child
                   { node (Approximation) {Approximation} }
     child
   }:
\end{scope}
```

Now comes the calendar code:

The lecture annotations:

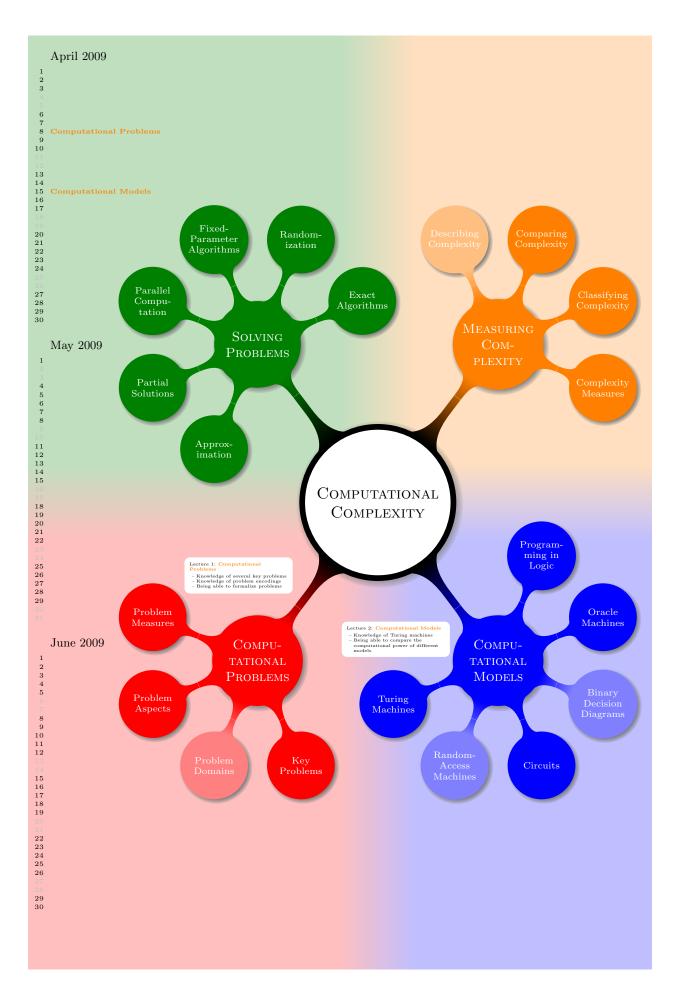
```
\lecture{1}{Computational Problems}{above,xshift=-5mm,yshift=5mm}{Computational Problems.north}{
   \item Knowledge of several key problems
   \item Knowledge of problem encodings
   \item Being able to formalize problems
}{2009-04-08}

\lecture{2}{Computational Models}{above left}
{Computational Models.west}{
   \item Knowledge of Turing machines
   \item Being able to compare the computational power of different
   models
}{2009-04-15}
```

Finally, the background:

```
\begin{pgfonlayer}{background}
    \clip[xshift=-1cm] (-.5\textwidth,-.5\textheight) rectangle ++(\textwidth,\textheight);
    \colorlet{upperleft}{green!50!black!25}
    \colorlet{upperright}{orange!25}
    \colorlet{lowerleft}{red!25}
    \colorlet{lowerright}{blue!25}
    % The large rectangles:
    \fill [upperleft] (Computational Complexity) rectangle ++(-20,20); \fill [upperright] (Computational Complexity) rectangle ++(20,20);
    \fill [lowerleft] (Computational Complexity) rectangle ++(-20,-20);
    \fill [lowerright] (Computational Complexity) rectangle ++(20,-20);
    % The shadings:
    \shade [left color=upperleft,right color=upperright]
      ([xshift=-1cm]Computational Complexity) rectangle ++(2,20);
    \shade [left color=lowerleft,right color=lowerright]
      ([xshift=-1cm]Computational Complexity) rectangle ++(2,-20);
    \shade [top color=upperleft,bottom color=lowerleft]
      ([yshift=-1cm]Computational Complexity) rectangle ++(-20,2);
    \shade [top color=upperright,bottom color=lowerright]
      ([yshift=-1cm]Computational Complexity) rectangle ++(20,2);
  \end{pgfonlayer}
\end{tikzpicture}
```

The next page shows the resulting lecture map in all its glory (it would be somewhat more glorious, if there were more lecture annotations, but you should get the idea).



7 Guidelines on Graphics

The present section is not about PGF or TikZ, but about general guidelines and principles concerning the creation of graphics for scientific presentations, papers, and books.

The guidelines in this section come from different sources. Many of them are just what I would like to claim is "common sense," some reflect my personal experience (though, hopefully, not my personal preferences), some come from books (the bibliography is still missing, sorry) on graphic design and typography. The most influential source are the brilliant books by Edward Tufte. While I do not agree with everything written in these books, many of Tufte's arguments are so convincing that I decided to repeat them in the following guidelines.

The first thing you should ask yourself when someone presents a bunch of guidelines is: Should I really follow these guidelines? This is an important question, because there are good reasons not to follow general guidelines. The person who set up the guidelines may have had other objectives than you do. For example, a guideline might say "use the color red for emphasis." While this guideline makes perfect sense for, say, a presentation using a projector, red "color" has the *opposite* effect of "emphasis" when printed using a black-and-white printer. Guidelines were almost always set up to address a specific situation. If you are not in this situation, following a guideline can do more harm than good.

The second thing you should be aware of is the basic rule of typography is: "Every rule can be broken, as long as you are *aware* that you are breaking a rule." This rule also applies to graphics. Phrased differently, the basic rule states: "The only mistakes in typography are things done in ignorance." When you are aware of a rule and when you decide that breaking the rule has a desirable effect, break the rule.

7.1 Planning the Time Needed for the Creation of Graphics

When you create a paper with numerous graphics, the time needed to create these graphics becomes an important factor. How much time should you calculate for the creation of graphics?

As a general rule, assume that a graphic will need as much time to create as would a text of the same length. For example, when I write a paper, I need about one hour per page for the first draft. Later, I need between two and four hours per page for revisions. Thus, I expect to need about half an hour for the creation of a first draft of a half page graphic. Later on, I expect another one to two hours before the final graphic is finished.

In many publications, even in good journals, the authors and editors have obviously invested a lot of time on the text, but seem to have spend about five minutes to create all of the graphics. Graphics often seem to have been added as an "afterthought" or look like a screen shot of whatever the authors's statistical software shows them. As will be argued later on, the graphics that programs like GNUPLOT produce by default are of poor quality.

Creating informative graphics that help the reader and that fit together with the main text is a difficult, lengthy process.

- Treat graphics as first-class citizens of your papers. They deserve as much time and energy as the text does. Indeed, the creation of graphics might deserve *even more* time than the writing of the main text since more attention will be paid to the graphics and they will be looked at first.
- Plan as much time for the creation and revision of a graphic as you would plan for text of the same size.
- Difficult graphics with a high information density may require even more time.
- Very simple graphics will require less time, but most likely you do not want to have "very simple graphics" in your paper, anyway; just as you would not like to have a "very simple text" of the same size.

7.2 Workflow for Creating a Graphic

When you write a (scientific) paper, you will most likely follow the following pattern: You have some results/ideas that you would like to report about. The creation of the paper will typically start with compiling a rough outline. Then, the different sections are filled with text to create a first draft. This draft is then revised repeatedly until, often after substantial revision, a final paper results. In a good journal paper there is typically not be a single sentence that has survived unmodified from the first draft.

Creating a graphics follows the same pattern:

- Decide on what the graphic should communicate. Make this a conscious decision, that is, determine "What is the graphic supposed to tell the reader?"
- Create an "outline," that is, the rough overall "shape" of the graphic, containing the most crucial elements. Often, it is useful to do this using pencil and paper.
- Fill out the finer details of the graphic to create a first draft.
- Revise the graphic repeatedly along with the rest of the paper.

7.3 Linking Graphics With the Main Text

Graphics can be placed at different places in a text. Either, they can be inlined, meaning they are somewhere "in the middle of the text" or they can be placed in stand-alone "figures." Since printers (the people) like to have their pages "filled," (both for aesthetic and economic reasons) stand-alone figures may traditionally be placed on pages in the document far away from the main text that refers to them. LATEX and TEX tend to encourage this "drifting away" of graphics for technical reasons.

When a graphic is inlined, it will more or less automatically be linked with the main text in the sense that the labels of the graphic will be implicitly explained by the surrounding text. Also, the main text will typically make it clear what the graphic is about and what is shown.

Quite differently, a stand-alone figure will often be viewed at a time when the main text that this graphic belongs to either has not yet been read or has been read some time ago. For this reason, you should follow the following guidelines when creating stand-alone figures:

- Stand-alone figures should have a caption than should make them "understandable by themselves." For example, suppose a graphic shows an example of the different stages of a quicksort algorithm. Then the figure's caption should, at the very least, inform the reader that "The figure shows the different stages of the quicksort algorithm introduced on page xyz." and not just "Quicksort algorithm."
- A good caption adds as much context information as possible. For example, you could say: "The figure shows the different stages of the quicksort algorithm introduced on page xyz. In the first line, the pivot element 5 is chosen. This causes..." While this information can also be given in the main text, putting it in the caption will ensure that the context is kept. Do not feel afraid of a 5-line caption. (Your editor may hate you for this. Consider hating them back.)
- Reference the graphic in your main text as in "For an example of quicksort 'in action,' see Figure 2.1 on page xyz."
- Most books on style and typography recommend that you do not use abbreviations as in "Fig. 2.1" but write "Figure 2.1."

The main argument against abbreviations is that "a period is too valuable to waste it on an abbreviation." The idea is that a period will make the reader assume that the sentence ends after "Fig" and it takes a "conscious backtracking" to realize that the sentence did not end after all.

The argument in favor of abbreviations is that they save space.

Personally, I am not really convinced by either argument. On the one hand, I have not yet seen any hard evidence that abbreviations slow readers down. On the other hand, abbreviating all "Figure" by "Fig." is most unlikely to save even a single line in most documents. I avoid abbreviations.

7.4 Consistency Between Graphics and Text

Perhaps the most common "mistake" people do when creating graphics (remember that a "mistake" in design is always just "ignorance") is to have a mismatch between the way their graphics look and the way their text looks.

It is quite common that authors use several different programs for creating the graphics of a paper. An author might produce some plots using GNUPLOT, a diagram using XFIG, and include an .eps graphic a coauthor contributed using some unknown program. All these graphics will, most likely, use different line widths, different fonts, and have different sizes. In addition, authors often use options like [height=5cm] when including graphics to scale them to some "nice size."

If the same approach were taken to writing the main text, every section would be written in a different font at a different size. In some sections all theorems would be underlined, in another they would be printed all in uppercase letters, and in another in red. In addition, the margins would be different on each page. Readers and editors would not tolerate a text if it were written in this fashion, but with graphics they often have to.

To create consistency between graphics and text, stick to the following guidelines:

- Do not scale graphics.
 - This means that when generating graphics using an external program, create them "at the right size."
- Use the same font(s) both in graphics and the body text.
- Use the same line width in text and graphics.
 - The "line width" for normal text is the width of the stem of letters like T. For T_EX, this is usually 0.4 pt. However, some journals will not accept graphics with a normal line width below 0.5 pt.
- When using colors, use a consistent color coding in the text and in graphics. For example, if red is supposed to alert the reader to something in the main text, use red also in graphics for important parts of the graphic. If blue is used for structural elements like headlines and section titles, use blue also for structural elements of your graphic.
 - However, graphics may also use a logical intrinsic color coding. For example, no matter what colors you normally use, readers will generally assume, say, that the color green as "positive, go, ok" and red as "alert, warning, action."

Creating consistency when using different graphic programs is almost impossible. For this reason, you should consider sticking to a single graphics program.

7.5 Labels in Graphics

Almost all graphics will contain labels, that is, pieces of text that explain parts of the graphics. When placing labels, stick to the following guidelines:

- Follow the rule of consistency when placing labels. You should do so in two ways: First, be consistent with the main text, that is, use the same font as the main text also for labels. Second, be consistent between labels, that is, if you format some labels in some particular way, format all labels in this way.
- In addition to using the same fonts in text and graphics, you should also use the same notation. For example, if you write 1/2 in your main text, also use "1/2" as labels in graphics, not "0.5". A π is a " π " and not "3.141". Finally, $e^{-i\pi}$ is " $e^{-i\pi}$ ", not "-1", let alone "-1".
- Labels should be legible. They should not only have a reasonably large size, they also should not be obscured by lines or other text. This also applies to labels of lines and text *behind* the labels.
- Labels should be "in place." Whenever there is enough space, labels should be placed next to the thing they label. Only if necessary, add a (subdued) line from the label to the labeled object. Try to avoid labels that only reference explanations in external legends. Reader have to jump back and forth between the explanation and the object that is described.
- Consider subduing "unimportant" labels using, for example, a gray color. This will keep the focus on the actual graphic.

7.6 Plots and Charts

One of the most frequent kind of graphics, especially in scientific papers, are *plots*. They come in a large variety, including simple line plots, parametric plots, three dimensional plots, pie charts, and many more.

Unfortunately, plots are notoriously hard to get right. Partly, the default settings of programs like GNUPLOT or Excel are to blame for this since these programs make it very convenient to create bad plots.

The first question you should ask yourself when creating a plot is: Are there enough data points to merit a plot? If the answer is "not really," use a table.

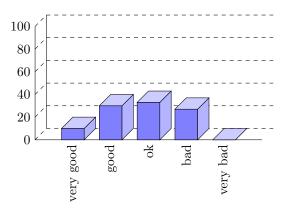
A typical situation where a plot is unnecessary is when people present a few numbers in a bar diagram. Here is a real-life example: At the end of a seminar a lecturer asked the participants for feedback. Of the 50 participants, 30 returned the feedback form. According to the feedback, three participants considered the

seminar "very good," nine considered it "good," ten "ok," eight "bad," and no one thought that the seminar was "very bad."

A simple way of summing up this information is the following table:

Rating given	Participants (out of 50) who gave this rating	Percentage
"very good"	3	6%
"good"	9	18%
"ok"	10	20%
"bad"	8	16%
"very bad"	0	0%
none	20	40%

What the lecturer did was to visualize the data using a 3D bar diagram. It looked like this (except that in reality the numbers where typeset using some extremely low-resolution bitmap font and were near-unreadable):



Both the table and the "plot" have about the same size. If your first thought is "the graphic looks nicer than the table," try to answer the following questions based on the information in the table or in the graphic:

- 1. How many participants where there?
- 2. How many participants returned the feedback form?
- 3. What percentage of the participants returned the feedback form?
- 4. How many participants checked "very good"?
- 5. What percentage out of all participants checked "very good"?
- 6. Did more than a quarter of the participants check "bad" or "very bad"?
- 7. What percentage of the participants that returned the form checked "very good"?

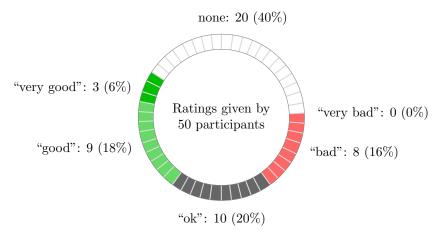
Sadly, the graphic does not allow us to answer a single one of these questions. The table answers all of them directly, except for the last one. In essence, the information density of the graphic is very close to zero. The table has a much higher information density; despite the fact that it uses quite a lot of white space to present a few numbers. Here is the list of things that went wrong with the 3D-bar diagram:

- The whole graphic is dominated by irritating background lines.
- It is not clear what the numbers at the left mean; presumably percentages, but it might also be the absolute number of participants.
- The labels at the bottom are rotated, making them hard to read.

 (In the real presentation that I saw, the text was rendered at a very low resolution with about 10 by 6 pixels per letter with wrong kerning, making the rotated text almost impossible to read.)
- The third dimension adds complexity to the graphic without adding information.

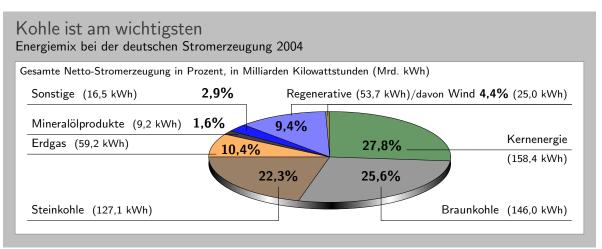
- The three dimensional setup makes it much harder to gauge the height of the bars correctly. Consider the "bad" bar. It the number this bar stands for more than 20 or less? While the front of the bar is below the 20 line, the back of the bar (which counts) is above.
- It is impossible to tell which numbers are represented by the bars. Thus, the bars needlessly hide the information these bars are all about.
- What do the bar heights add up to? Is it 100% or 60%?
- Does the bar for "very bad" represent 0 or 1?
- Why are the bars blue?

You might argue that in the example the exact numbers are not important for the graphic. The important things is the "message," which is that there are more "very good" and "good" ratings than "bad" and "very bad." However, to convey this message either use a sentence that says so or use a graphic that conveys this message more clearly:



The above graphic has about the same information density as the table (about the same size and the same numbers are shown). In addition, one can directly "see" that there are more good or very good ratings than bad ones. One can also "see" that the number of people who gave no rating at all is not negligible, which is quite common for feedback forms.

Charts are not always a good idea. Let us look at an example that I redrew from a pie chart in *Die Zeit*, June 4th, 2005:



This graphic has been redrawn in TikZ, but the original looks almost exactly the same. At first sight, the graphic looks "nice and informative," but there are a lot of things that went wrong:

• The chart is three dimensional. However, the shadings add nothing "information-wise," at best, they distract.

- In a 3D-pie-chart the relative sizes are very strongly distorted. For example, the area taken up by the gray color of "Braunkohle" is larger than the area taken up by the green color of "Kernenergie" despite the fact that the percentage of Braunkohle is less than the percentage of Kernenergie.
- The 3D-distortion gets worse for small areas. The area of "Regenerative" somewhat larger than the area of "Erdgas." The area of "Wind" is slightly smaller than the area of "Mineralölprodukte" although the percentage of Wind is nearly three times larger than the percentage of Mineralölprodukte.
 - In the last case, the different sizes are only partly due to distortion. The designer(s) of the original graphic have also made the "Wind" slice too small, even taking distortion into account. (Just compare the size of "Wind" to "Regenerative" in general.)
- According to its caption, this chart is supposed to inform us that coal was the most important energy source in Germany in 2004. Ignoring the strong distortions caused by the superfluous and misleading 3D-setup, it takes quite a while for this message to get across.
 - Coal as an energy source is split up into two slices: one for "Steinkohle" and one for "Braunkohle" (two different kinds of coal). When you add them up, you see that the whole lower half of the pie chart is taken up by coal.
 - The two areas for the different kinds of coal are not visually linked at all. Rather, two different colors are used, the labels are on different sides of the graphic. By comparison, "Regenerative" and "Wind" are very closely linked.
- The color coding of the graphic follows no logical pattern at all. Why is nuclear energy green? Regenerative energy is light blue, "other sources" are blue. It seems more like a joke that the area for "Braunkohle" (which literally translates to "brown coal") is stone gray, while the area for "Steinkohle" (which literally translates to "stone coal") is brown.
- The area with the lightest color is used for "Erdgas." This area stands out most because of the brighter color. However, for this chart "Erdgas" is not really important at all.

Edward Tufte calls graphics like the above "chart junk." (I am happy to announce, however, that *Die Zeit* has stopped using 3D pie charts and their information graphics have got somewhat better.)

Here are a few recommendations that may help you avoid producing chart junk:

- Do not use 3D pie charts. They are evil.
- Consider using a table instead of a pie chart.
- Do not apply colors randomly; use them to direct the readers's focus and to group things.
- Do not use background patterns, like a crosshatch or diagonal lines, instead of colors. They distract. Background patterns in information graphics are *evil*.

7.7 Attention and Distraction

Pick up your favorite fiction novel and have a look at a typical page. You will notice that the page is very uniform. Nothing is there to distract the reader while reading; no large headlines, no bold text, no large white areas. Indeed, even when the author does wish to emphasize something, this is done using italic letters. Such letters blend nicely with the main text—at a distance you will not be able to tell whether a page contains italic letters, but you would notice a single bold word immediately. The reason novels are typeset this way is the following paradigm: Avoid distractions.

Good typography (like good organization) is something you do *not* notice. The job of typography is to make reading the text, that is, "absorbing" its information content, as effortless as possible. For a novel, readers absorb the content by reading the text line-by-line, as if they were listening to someone telling the story. In this situation anything on the page that distracts the eye from going quickly and evenly from line to line will make the text harder to read.

Now, pick up your favorite weekly magazine or newspaper and have a look at a typical page. You will notice that there is quite a lot "going on" on the page. Fonts are used at different sizes and in different arrangements, the text is organized in narrow columns, typically interleaved with pictures. The reason magazines are typeset in this way is another paradigm: Steer attention.

Readers will not read a magazine like a novel. Instead of reading a magazine line-by-line, we use headlines and short abstracts to check whether we want to read a certain article or not. The job of typography is to

steer our attention to these abstracts and headlines, first. Once we have decided that we want to read an article, however, we no longer tolerate distractions, which is why the main text of articles is typeset exactly the same way as a novel.

The two principles "avoid distractions" and "steer attention" also apply to graphics. When you design a graphic, you should eliminate everything that will "distract the eye." At the same time, you should try to actively help the reader "through the graphic" by using fonts/colors/line widths to highlight different parts.

Here is a non-exhaustive list of things that can distract readers:

• Strong contrasts will always be registered first by the eye. For example, consider the following two grids:



Even though the left grid comes first in English reading order, the right one is much more likely to be seen first: The white-to-black contrast is higher than the gray-to-white contrast. In addition, there are more "places" adding to the overall contrast in the right grid.

Things like grids and, more generally, help lines usually should not grab the attention of the readers and, hence, should be typeset with a low contrast to the background. Also, a loosely-spaced grid is less distracting than a very closely-spaced grid.

• Dashed lines create many points at which there is black-to-white contrast. Dashed or dotted lines can be very distracting and, hence, should be avoided in general.

Do not use different dashing patterns to differentiate curves in plots. You lose data points this way and the eye is not particularly good at "grouping things according to a dashing pattern." The eye is much better at grouping things according to colors.

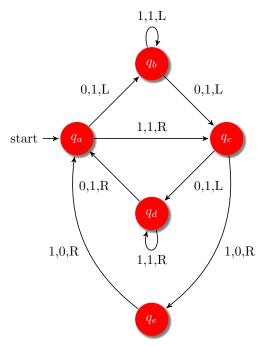
- Background patterns filling an area using diagonal lines or horizontal and vertical lines or just dots are almost always distracting and, usually, serve no real purpose.
- Background images and shadings distract and only seldom add anything of importance to a graphic.
- Cute little clip arts can easily draw attention away from the data.

Part II

Installation and Configuration

by Till Tantau

This part explains how the system is installed. Typically, someone has already done so for your system, so this part can be skipped; but if this is not the case and you are the poor fellow who has to do the installation, read the present part.



The current candidate for the busy beaver for five states. It is presumed that this Turing machine writes a maximum number of 1's before halting among all Turing machines with five states and the tape alphabet $\{0,1\}$. Proving this conjecture is an open research problem.

```
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto,node distance=2.8cm,on grid,semithick,
                    every state/.style={fill=red,draw=none,circular drop shadow,text=white}]
  \node[initial,state] (A)
                                               {$q_a$};
  \node[state]
                       (B)
                           [above right=of A]
                                               {$q_b$};
  \node[state]
                       (D)
                           [below right=of A] {$q_d$};
  \node[state]
                       (C) [below right=of B] {$q_c$};
  \node[state]
                       (E) [below=of D]
                                               {$q_e$};
  \path (A) edge
                              node {0,1,L} (B)
            edge
                              node {1,1,R} (C)
        (B) edge [loop above] node {1,1,L} (B)
            edge
                              node {0,1,L} (C)
        (C) edge
                              node {0,1,L} (D)
                 [bend left]
                              node {1,0,R} (E)
            edge
        (D) edge [loop below] node {1,1,R} (D)
            edge
                              node \{0,1,R\} (A)
        (E) edge [bend left]
                              node {1,0,R} (A);
   \node [right=1cm,text width=8cm] at (C)
     The current candidate for the busy beaver for five states. It is
     presumed that this Turing machine writes a maximum number of
     $1$'s before halting among all Turing machines with five states
     and the tape alphabet \{0, 1\}. Proving this conjecture is an
     open research problem.
\end{tikzpicture}
```

8 Installation

There are different ways of installing PGF, depending on your system and needs, and you may need to install other packages as well as, see below. Before installing, you may wish to review the licenses under which the package is distributed, see Section 9.

Typically, the package will already be installed on your system. Naturally, in this case you do not need to worry about the installation process at all and you can skip the rest of this section.

8.1 Package and Driver Versions

This documentation is part of version 2.10-cvs of the PGF package. In order to run PGF, you need a reasonably recent TeX installation. When using IATeX, you need the following packages installed (newer versions should also work):

• xcolor version 2.00.

With plain T_EX, xcolor is not needed, but you obviously do not get its (full) functionality. Currently, PGF supports the following backend drivers:

- pdftex version 0.14 or higher. Earlier versions do not work.
- dvips version 5.94a or higher. Earlier versions may also work.
 For inter-picture connections, you need process pictures using pdftex version 1.40 or higher running in DVI mode.
- dvipdfm version 0.13.2c or higher. Earlier versions may also work.
 For inter-picture connections, you need process pictures using pdftex version 1.40 or higher running in DVI mode.
- tex4ht version 2003-05-05 or higher. Earlier versions may also work.
- vtex version 8.46a or higher. Earlier versions may also work.
- textures version 2.1 or higher. Earlier versions may also work.
- xetex version 0.996 or higher. Earlier versions may also work.

Currently, PGF supports the following formats:

- latex with complete functionality.
- plain with complete functionality, except for graphics inclusion, which works only for pdfT_FX.
- context with complete functionality⁶, except for graphics inclusion, which works only for pdfT_FX.

For more details, see Section 10.

8.2 Installing Prebundled Packages

I do not create or manage prebundled packages of PGF, but, fortunately, nice other people do. I cannot give detailed instructions on how to install these packages, since I do not manage them, but I can tell you were to find them. If you have a problem with installing, you might wish to have a look at the Debian page or the MiKTEX page first.

8.2.1 Debian

The command "aptitude install pgf" should do the trick. Sit back and relax. In detail, the following packages are installed:

http://packages.debian.org/pgf http://packages.debian.org/latex-xcolor

⁶Note that PGF/TikZ is not supported by recent ConTEXt versions (like mark IV, the LuaTEX-aware part of ConTEXt).

8.2.2 MiKTeX

For MiKTEX, use the update wizard to install the (latest versions of the) packages called pgf and xcolor.

8.3 Installation in a texmf Tree

For a permanent installation, you place the files of the PGF package in an appropriate texmf tree.

When you ask TEX to use a certain class or package, it usually looks for the necessary files in so-called texmf trees. These trees are simply huge directories that contain these files. By default, TEX looks for files in three different texmf trees:

- The root texmf tree, which is usually located at /usr/share/texmf/ or c:\texmf\ or somewhere similar.
- The local texmf tree, which is usually located at /usr/local/share/texmf/ or c:\localtexmf\ or somewhere similar.
- Your personal texmf tree, which is usually located in your home directory at ~/texmf/ or ~/Library/texmf/.

You should install the packages either in the local tree or in your personal tree, depending on whether you have write access to the local tree. Installation in the root tree can cause problems, since an update of the whole T_EX installation will replace this whole tree.

8.3.1 Installation that Keeps Everything Together

Once you have located the right texmf tree, you must decide whether you want to install PGF in such a way that "all its files are kept in one place" or whether you want to be "TDS-compliant," where TDS means "TEX directory structure."

If you want to keep "everything in one place," inside the texmf tree that you have chosen create a subsub-directory called texmf/tex/generic/pgf or texmf/tex/generic/pgf-2.10-cvs, if you prefer. Then place all files of the pgf package in this directory. Finally, rebuild TEX's filename database. This is done by running the command texhash or mktexlsr (they are the same). In MiKTEX, there is a menu option to do this.

8.3.2 Installation that is TDS-Compliant

While the above installation process is the most "natural" one and although I would like to recommend it since it makes updating and managing the PGF package easy, it is not TDS-compliant. If you want to be TDS-compliant, proceed as follows: (If you do not know what TDS-compliant means, you probably do not want to be TDS-compliant.)

The .tar file of the pgf package contains the following files and directories at its root: README, doc, generic, plain, and latex. You should "merge" each of the four directories with the following directories texmf/doc, texmf/tex/generic, texmf/tex/plain, and texmf/tex/latex. For example, in the .tar file the doc directory contains just the directory pgf, and this directory has to be moved to texmf/doc/pgf. The root README file can be ignored since it is reproduced in doc/pgf/README.

You may also consider keeping everything in one place and using symbolic links to point from the TDS-compliant directories to the central installation.

For a more detailed explanation of the standard installation process of packages, you might wish to consult http://www.ctan.org/installationadvice/. However, note that the PGF package does not come with a .ins file (simply skip that part).

8.4 Updating the Installation

To update your installation from a previous version, all you need to do is to replace everything in the directory texmf/tex/generic/pgf with the files of the new version (or in all the directories where pgf was installed, if you chose a TDS-compliant installation). The easiest way to do this is to first delete the old version and then proceed as described above. Sometimes, there are changes in the syntax of certain command from version to version. If things no longer work that used to work, you may wish to have a look at the release notes and at the change log.

9 Licenses and Copyright

9.1 Which License Applies?

Different parts of the PGF package are distributed under different licenses:

- 1. The *code* of the package is dual-license. This means that you can decide which license you wish to use when using the PGF package. The two options are:
 - (a) You can use the GNU Public License, version 2.
 - (b) You can use the LATEX Project Public License, version 1.3c.
- 2. The documentation of the package is also dual-license. Again, you can choose between two options:
 - (a) You can use the GNU Free Documentation License, version 1.2.
 - (b) You can use the LATEX Project Public License, version 1.3c.

The "documentation of the package" refers to all files in the subdirectory doc of the pgf package. A detailed listing can be found in the file doc/generic/pgf/licenses/manifest-documentation.txt. All files in other directories are part of the "code of the package." A detailed listing can be found in the file doc/generic/pgf/licenses/manifest-code.txt.

In the rest of this section, the licenses are presented. The following text is copyrighted, see the plain text versions of these licenses in the directory doc/generic/pgf/licenses for details.

The example picture used in this manual, the Brave GNU World logo, is taken from the Brave GNU World homepage, where it is copyrighted as follows: "Copyright (C) 1999, 2000, 2001, 2002, 2003, 2004 Georg C. F. Greve. Permission is granted to make and distribute verbatim copies of this transcript as long as the copyright and this permission notice appear."

9.2 The GNU Public License, Version 2

9.2.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

9.2.2 Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

- 2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

- 3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsubsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

- 4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
- 5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
- 6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
- 7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

- 8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
- 9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.
 - Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
- 10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

9.2.3 No Warranty

- 10. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.
- 11. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

9.3 The LATEX Project Public License, Version 1.3c 2006-05-20

9.3.1 Preamble

The LaTeX Project Public License (LPPL) is the primary license under which the LaTeX kernel and the base LaTeX packages are distributed.

You may use this license for any work of which you hold the copyright and which you wish to distribute. This license may be particularly suitable if your work is TEX-related (such as a LATEX package), but it is written in such a way that you can use it even if your work is unrelated to TEX.

The section 'WHETHER AND HOW TO DISTRIBUTE WORKS UNDER THIS LICENSE', below, gives instructions, examples, and recommendations for authors who are considering distributing their works under this license.

This license gives conditions under which a work may be distributed and modified, as well as conditions under which modified versions of that work may be distributed.

We, the LATEX3 Project, believe that the conditions below give you the freedom to make and distribute modified versions of your work that conform with whatever technical specifications you wish while maintaining the availability, integrity, and reliability of that work. If you do not see how to achieve your goal while meeting these conditions, then read the document 'cfgguide.tex' and 'modguide.tex' in the base LATEX distribution for suggestions.

9.3.2 Definitions

In this license document the following terms are used:

Work Any work being distributed under this License.

Derived Work Any work that under any applicable law is derived from the Work.

Modification Any procedure that produces a Derived Work under any applicable law – for example, the production of a file containing an original file associated with the Work or a significant portion of such a file, either verbatim or with modifications and/or translated into another language.

Modify To apply any procedure that produces a Derived Work under any applicable law.

Distribution Making copies of the Work available from one person to another, in whole or in part. Distribution includes (but is not limited to) making any electronic components of the Work accessible by file transfer protocols such as FTP or HTTP or by shared file systems such as Sun's Network File System (NFS).

Compiled Work A version of the Work that has been processed into a form where it is directly usable on a computer system. This processing may include using installation facilities provided by the Work, transformations of the Work, copying of components of the Work, or other activities. Note that modification of any installation facilities provided by the Work constitutes modification of the Work.

Current Maintainer A person or persons nominated as such within the Work. If there is no such explicit nomination then it is the 'Copyright Holder' under any applicable law.

Base Interpreter A program or process that is normally needed for running or interpreting a part or the whole of the Work.

A Base Interpreter may depend on external components but these are not considered part of the Base Interpreter provided that each external component clearly identifies itself whenever it is used interactively. Unless explicitly specified when applying the license to the Work, the only applicable Base Interpreter is a 'LATEX-Format' or in the case of files belonging to the 'LATEX-format' a program implementing the 'TEX language'.

9.3.3 Conditions on Distribution and Modification

- 1. Activities other than distribution and/or modification of the Work are not covered by this license; they are outside its scope. In particular, the act of running the Work is not restricted and no requirements are made concerning any offers of support for the Work.
- 2. You may distribute a complete, unmodified copy of the Work as you received it. Distribution of only part of the Work is considered modification of the Work, and no right to distribute such a Derived Work may be assumed under the terms of this clause.
- 3. You may distribute a Compiled Work that has been generated from a complete, unmodified copy of the Work as distributed under Clause 2 above, as long as that Compiled Work is distributed in such a way that the recipients may install the Compiled Work on their system exactly as it would have been installed if they generated a Compiled Work directly from the Work.
- 4. If you are the Current Maintainer of the Work, you may, without restriction, modify the Work, thus creating a Derived Work. You may also distribute the Derived Work without restriction, including Compiled Works generated from the Derived Work. Derived Works distributed in this manner by the Current Maintainer are considered to be updated versions of the Work.
- 5. If you are not the Current Maintainer of the Work, you may modify your copy of the Work, thus creating a Derived Work based on the Work, and compile this Derived Work, thus creating a Compiled Work based on the Derived Work.
- 6. If you are not the Current Maintainer of the Work, you may distribute a Derived Work provided the following conditions are met for every component of the Work unless that component clearly states in the copyright notice that it is exempt from that condition. Only the Current Maintainer is allowed to add such statements of exemption to a component of the Work.

- (a) If a component of this Derived Work can be a direct replacement for a component of the Work when that component is used with the Base Interpreter, then, wherever this component of the Work identifies itself to the user when used interactively with that Base Interpreter, the replacement component of this Derived Work clearly and unambiguously identifies itself as a modified version of this component to the user when used interactively with that Base Interpreter.
- (b) Every component of the Derived Work contains prominent notices detailing the nature of the changes to that component, or a prominent reference to another file that is distributed as part of the Derived Work and that contains a complete and accurate log of the changes.
- (c) No information in the Derived Work implies that any persons, including (but not limited to) the authors of the original version of the Work, provide any support, including (but not limited to) the reporting and handling of errors, to recipients of the Derived Work unless those persons have stated explicitly that they do provide such support for the Derived Work.
- (d) You distribute at least one of the following with the Derived Work:
 - i. A complete, unmodified copy of the Work; if your distribution of a modified component is made by offering access to copy the modified component from a designated place, then offering equivalent access to copy the Work from the same or some similar place meets this condition, even though third parties are not compelled to copy the Work along with the modified component;
 - ii. Information that is sufficient to obtain a complete, unmodified copy of the Work.
- 7. If you are not the Current Maintainer of the Work, you may distribute a Compiled Work generated from a Derived Work, as long as the Derived Work is distributed to all recipients of the Compiled Work, and as long as the conditions of Clause 6, above, are met with regard to the Derived Work.
- 8. The conditions above are not intended to prohibit, and hence do not apply to, the modification, by any method, of any component so that it becomes identical to an updated version of that component of the Work as it is distributed by the Current Maintainer under Clause 4, above.
- 9. Distribution of the Work or any Derived Work in an alternative format, where the Work or that Derived Work (in whole or in part) is then produced by applying some process to that format, does not relax or nullify any sections of this license as they pertain to the results of applying that process.
- 10. (a) A Derived Work may be distributed under a different license provided that license itself honors the conditions listed in Clause 6 above, in regard to the Work, though it does not have to honor the rest of the conditions in this license.
 - (b) If a Derived Work is distributed under a different license, that Derived Work must provide sufficient documentation as part of itself to allow each recipient of that Derived Work to honor the restrictions in Clause 6 above, concerning changes from the Work.
- 11. This license places no restrictions on works that are unrelated to the Work, nor does this license place any restrictions on aggregating such works with the Work by any means.
- 12. Nothing in this license is intended to, or may be used to, prevent complete compliance by all parties with all applicable laws.

9.3.4 No Warranty

There is no warranty for the Work. Except when otherwise stated in writing, the Copyright Holder provides the Work 'as is', without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Work is with you. Should the Work prove defective, you assume the cost of all necessary servicing, repair, or correction.

In no event unless required by applicable law or agreed to in writing will The Copyright Holder, or any author named in the components of the Work, or any other party who may distribute and/or modify the Work as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of any use of the Work or out of inability to use the Work (including, but not limited to, loss of data, data being rendered inaccurate, or losses sustained by anyone as a result of any failure of the Work to operate with any other programs), even if the Copyright Holder or said author or said other party has been advised of the possibility of such damages.

9.3.5 Maintenance of The Work

The Work has the status 'author-maintained' if the Copyright Holder explicitly and prominently states near the primary copyright notice in the Work that the Work can only be maintained by the Copyright Holder or simply that it is 'author-maintained'.

The Work has the status 'maintained' if there is a Current Maintainer who has indicated in the Work that they are willing to receive error reports for the Work (for example, by supplying a valid e-mail address). It is not required for the Current Maintainer to acknowledge or act upon these error reports.

The Work changes from status 'maintained' to 'unmaintained' if there is no Current Maintainer, or the person stated to be Current Maintainer of the work cannot be reached through the indicated means of communication for a period of six months, and there are no other significant signs of active maintenance.

You can become the Current Maintainer of the Work by agreement with any existing Current Maintainer to take over this role.

If the Work is unmaintained, you can become the Current Maintainer of the Work through the following steps:

- 1. Make a reasonable attempt to trace the Current Maintainer (and the Copyright Holder, if the two differ) through the means of an Internet or similar search.
- 2. If this search is successful, then enquire whether the Work is still maintained.
 - (a) If it is being maintained, then ask the Current Maintainer to update their communication data within one month.
 - (b) If the search is unsuccessful or no action to resume active maintenance is taken by the Current Maintainer, then announce within the pertinent community your intention to take over maintenance. (If the Work is a LATEX work, this could be done, for example, by posting to comp.text.tex.)
- 3. (a) If the Current Maintainer is reachable and agrees to pass maintenance of the Work to you, then this takes effect immediately upon announcement.
 - (b) If the Current Maintainer is not reachable and the Copyright Holder agrees that maintenance of the Work be passed to you, then this takes effect immediately upon announcement.
- 4. If you make an 'intention announcement' as described in 2b above and after three months your intention is challenged neither by the Current Maintainer nor by the Copyright Holder nor by other people, then you may arrange for the Work to be changed so as to name you as the (new) Current Maintainer.
- 5. If the previously unreachable Current Maintainer becomes reachable once more within three months of a change completed under the terms of 3b or 4, then that Current Maintainer must become or remain the Current Maintainer upon request provided they then update their communication data within one month.

A change in the Current Maintainer does not, of itself, alter the fact that the Work is distributed under the LPPL license.

If you become the Current Maintainer of the Work, you should immediately provide, within the Work, a prominent and unambiguous statement of your status as Current Maintainer. You should also announce your new status to the same pertinent community as in 2b above.

9.3.6 Whether and How to Distribute Works under This License

This section contains important instructions, examples, and recommendations for authors who are considering distributing their works under this license. These authors are addressed as 'you' in this section.

9.3.7 Choosing This License or Another License

If for any part of your work you want or need to use *distribution* conditions that differ significantly from those in this license, then do not refer to this license anywhere in your work but, instead, distribute your work under a different license. You may use the text of this license as a model for your own license, but your license should not refer to the LPPL or otherwise give the impression that your work is distributed under the LPPL.

The document 'modguide.tex' in the base IATEX distribution explains the motivation behind the conditions of this license. It explains, for example, why distributing IATEX under the GNU General Public

License (GPL) was considered inappropriate. Even if your work is unrelated to LATEX, the discussion in 'modguide.tex' may still be relevant, and authors intending to distribute their works under any license are encouraged to read it.

9.3.8 A Recommendation on Modification Without Distribution

It is wise never to modify a component of the Work, even for your own personal use, without also meeting the above conditions for distributing the modified component. While you might intend that such modifications will never be distributed, often this will happen by accident – you may forget that you have modified that component; or it may not occur to you when allowing others to access the modified version that you are thus distributing it and violating the conditions of this license in ways that could have legal implications and, worse, cause problems for the community. It is therefore usually in your best interest to keep your copy of the Work identical with the public one. Many works provide ways to control the behavior of that work without altering any of its licensed components.

9.3.9 How to Use This License

To use this license, place in each of the components of your work both an explicit copyright notice including your name and the year the work was authored and/or last substantially modified. Include also a statement that the distribution and/or modification of that component is constrained by the conditions in this license.

Here is an example of such a notice and statement:

```
%% pig.dtx
%% Copyright 2005 M. Y. Name
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3
% of this license or (at your option) any later version.
% The latest version of this license is in
% http://www.latex-project.org/lppl.txt
% and version 1.3 or later is part of all distributions of LaTeX
% version 2005/12/01 or later.
%
% This work has the LPPL maintenance status 'maintained'.
%
% The Current Maintainer of this work is M. Y. Name.
%
% This work consists of the files pig.dtx and pig.ins
% and the derived file pig.sty.
```

Given such a notice and statement in a file, the conditions given in this license document would apply, with the 'Work' referring to the three files 'pig.dtx', 'pig.ins', and 'pig.sty' (the last being generated from 'pig.dtx' using 'pig.ins'), the 'Base Interpreter' referring to any 'IATEX-Format', and both 'Copyright Holder' and 'Current Maintainer' referring to the person 'M. Y. Name'.

If you do not want the Maintenance section of LPPL to apply to your Work, change 'maintained' above into 'author-maintained'. However, we recommend that you use 'maintained' as the Maintenance section was added in order to ensure that your Work remains useful to the community even when you can no longer maintain and support it yourself.

9.3.10 Derived Works That Are Not Replacements

Several clauses of the LPPL specify means to provide reliability and stability for the user community. They therefore concern themselves with the case that a Derived Work is intended to be used as a (compatible or incompatible) replacement of the original Work. If this is not the case (e.g., if a few lines of code are reused for a completely different task), then clauses 6b and 6d shall not apply.

9.3.11 Important Recommendations

Defining What Constitutes the Work The LPPL requires that distributions of the Work contain all the files of the Work. It is therefore important that you provide a way for the licensee to determine which

files constitute the Work. This could, for example, be achieved by explicitly listing all the files of the Work near the copyright notice of each file or by using a line such as:

% This work consists of all files listed in manifest.txt.

in that place. In the absence of an unequivocal list it might be impossible for the licensee to determine what is considered by you to comprise the Work and, in such a case, the licensee would be entitled to make reasonable conjectures as to which files comprise the Work.

9.4 GNU Free Documentation License, Version 1.2, November 2002

9.4.1 Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

9.4.2 Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

9.4.3 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

9.4.4 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

9.4.5 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified

Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any

one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

9.4.6 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

9.4.7 Collection of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

9.4.8 Aggregating with independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9.4.9 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9.4.10 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9.4.11 Future Revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

9.4.12 Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

10 Input and Output Formats

TEX was designed to be a flexible system. This is true both for the *input* for TEX as well as for the *output*. The present section explains which input formats there are and how they are supported by PGF. It also explains which different output formats can be produced.

10.1 Supported Input Formats

TEX does not prescribe exactly how your input should be formatted. While it is *customary* that, say, an opening brace starts a scope in TEX, this is by no means necessary. Likewise, it is *customary* that environments start with \begin, but TEX could not really care less about the exact command name.

Even though TEX can be reconfigured, users can not. For this reason, certain *input formats* specify a set of commands and conventions how input for TEX should be formatted. There are currently three "major" formats: Donald Knuth's original plain TEX format, Leslie Lamport's popular LATEX format, and Hans Hangen's ConTEXt format.

10.1.1 Using the LATEX Format

Using PGF and TikZ with the LaTeX format is easy: You say \usepackage{pgf} or \usepackage{tikz}. Usually, that is all you need to do, all configuration will be done automatically and (hopefully) correctly.

The style files used for the LATEX format reside in the subdirectory latex/pgf/ of the PGF-system. Mainly, what these files do is to include files in the directory generic/pgf. For example, here is the content of the file latex/pgf/frontends/tikz.sty:

```
% Copyright 2006 by Till Tantau
%
% This file may be distributed and/or modified
%
% 1. under the LaTeX Project Public License and/or
% 2. under the GNU Public License.
%
% See the file doc/generic/pgf/licenses/LICENSE for more details.

\RequirePackage{pgf,pgffor}
\input{tikz.code.tex}
\endinput
```

The files in the generic/pgf directory do the actual work.

10.1.2 Using the Plain TeX Format

When using the plain T_EX format, you say \input{pgf.tex} or \input{tikz.tex}. Then, instead of \begin{pgfpicture} and \end{pgfpicture} you use \pgfpicture and \endpgfpicture.

Unlike for the LATEX format, PGF is not as good at discerning the appropriate configuration for the plain TEX format. In particular, it can only automatically determine the correct output format if you use pdftex or tex plus dvips. For all other output formats you need to set the macro \pgfsysdriver to the correct value. See the description of using output formats later on.

PGF was originally written for use with IATEX and this shows in a number of places. Nevertheless, the plain TEX support is reasonably good.

Like the LATEX style files, the plain TEX files like tikz.tex also just include the correct tikz.code.tex file.

10.1.3 Using the ConTeXt Format

When using the ConTEXt format⁷, you say \usemodule[pgf] or \usemodule[tikz]. As for the plain TEX format you also have to replace the start- and end-of-environment tags as follows: Instead of \begin{pgfpicture} and \end{pgfpicture} you use \starttpgfpicture and \stoppgfpicture; similarly, instead of \begin{tikzpicture} and \end{tikzpicture} you use must now use \starttikzpicture and \stoptikzpicture; and so on for other environments.

⁷Note that PGF/TikZ is not supported by recent ConTEXt versions (like mark IV, the LuaTEX-aware part of ConTEXt).

The ConT_EXt support is very similar to the plain T_EX support, so the same restrictions apply: You may have to set the output format directly and graphics inclusion may be a problem.

In addition to pgf and tikz there also exist modules like pgfcore or pgfmodulematrix. To use them, you may need to include the module pgfmod first (the modules pgf and tikz both include pgfmod for you, so typically you can skip this). This special module is necessary since ConTEXt satanically restricts the length of module names to 6 characters and PGF's long names are mapped to cryptic 6-letter-names for you by the module pgfmod.

10.2 Supported Output Formats

An output format is a format in which TeX outputs the text it has typeset. Producing the output is (conceptually) a two-stage process:

- 1. TEX typesets your text and graphics. The result of this typesetting is mainly a long list of letter—coordinate pairs, plus (possibly) some "special" commands. This long list of pairs is written to something called a .dvi-file.
- 2. Some other program reads this .dvi-file and translates the letter-coordinate pairs into, say, PostScript commands for placing the given letter at the given coordinate.

The classical example of this process is the combination of latex and dvips. The latex program (which is just the tex program called with the LATEX-macros preinstalled) produces a .dvi-file as its output. The dvips program takes this output and produces a .ps-file (a PostScript) file. Possibly, this file is further converted using, say, ps2pdf, whose name is supposed to mean "PostScript to PDF." Another example of programs using this process is the combination of tex and dvipdfm. The dvipdfm program takes a .dvi-file as input and translates the letter-coordinate pairs therein into PDF-commands, resulting in a .pdf file directly. Finally, the tex4ht is also a program that takes a .dvi-file and produces an output, this time it is a .html file. The programs pdftex and pdflatex are special: They directly produce a .pdf-file without the intermediate .dvi-stage. However, from the programmer's point of view they behave exactly as if there where an intermediate stage.

Normally, TeX only produces letter—coordinate pairs as its "output." This obviously makes it difficult to draw, say, a curve. For this, "special" commands can be used. Unfortunately, these special commands are not the same for the different programs that process the .dvi-file. Indeed, every program that takes a .dvi-file as input has a totally different syntax for the special commands.

One of the main jobs of PGF is to "abstract away" the difference in the syntax of the different programs. However, this means that support for each program has to be "programmed," which is a time-consuming and complicated process.

10.2.1 Selecting the Backend Driver

When TEX typesets your document, it does not know which program you are going to use to transform the .dvi-file. If your .dvi-file does not contain any special commands, this would be fine; but these days almost all .dvi-files contain lots of special commands. It is thus necessary to tell TEX which program you are going to use later on.

Unfortunately, there is no "standard" way of telling this to TeX. For the LATeX format a sophisticated mechanism exists inside the graphics package and PGF plugs into this mechanism. For other formats and when this plugging does not work as expected, it is necessary to tell PGF directly which program you are going to use. This is done by redefining the macro \pgfsysdriver to an appropriate value before you load pgf. If you are going to use the dvips program, you set this macro to the value pgfsys-dvips.def; if you use pdftex or pdflatex, you set it to pgfsys-pdftex.def; and so on. In the following, details of the support of the different programs are discussed.

10.2.2 Producing PDF Output

PGF supports three programs that produce PDF output (PDF means "portable document format" and was invented by the Adobe company): dvipdfm, pdftex, and vtex. The pdflatex program is the same as the pdftex program: it uses a different input format, but the output is exactly the same.

File pgfsys-pdftex.def

This is the driver file for use with pdfTEX, that is, with the pdftex or pdflatex command. It includes pgfsys-common-pdf.def.

This driver has the "complete" functionality. This means, everything PGF "can do at all" is implemented in this driver.

File pgfsys-dvipdfm.def

This is a driver file for use with (la)tex followed by dvipdfm. It includes pgfsys-common-pdf.def.

This driver supports most of PGF's features, but there are some restrictions:

- 1. In IATEX mode it uses graphicx for the graphics inclusion and does not support masking.
- 2. In plain T_FX mode it does not support image inclusion.
- 3. For remembering of pictures (inter-picture connections) you need to use a recent version of pdftex running in DVI-mode.
- 4. Patterns are not (cannot be) supported.
- 5. Functional shadings are not (cannot be) supported.

File pgfsys-xetex.def

This is a driver file for use with xe(la)tex followed by xdvipdfmx. This driver supports the same operations as the dvipdfm driver, except that remembering of pictures (inter-picture connections) always works.

File pgfsys-vtex.def

This is the driver file for use with the commercial VTEX program. Even though it produces PDF output, it includes pgfsys-common-postscript.def. Note that the VTEX program can produce both Postscript and PDF output, depending on the command line parameters. However, whether you produce Postscript or PDF output does not change anything with respect to the driver.

This driver supports most of PGF's features, except for the following restrictions:

- 1. In IATEX mode it uses graphicx for the graphics inclusion and does not support masking.
- 2. In plain TeX mode it does not support image inclusion.
- 3. Shading is fully implemented, but yields the same quality as the implementation for dvips.
- 4. Opacity is not supported.
- 5. Remembering of pictures (inter-picture connections) is not supported.

It is also possible to produce a .pdf-file by first producing a PostScript file (see below) and then using a PostScript-to-PDF conversion program like ps2pdf or the Acrobat Distiller.

10.2.3 Producing PostScript Output

File pgfsys-dvips.def

This is a driver file for use with (la)tex followed by dvips. It includes pgfsys-common-postscript.def. This driver also supports most of PGF's features, except for the following restrictions:

- 1. In IATEX mode it uses graphicx for the graphics inclusion and does not support masking.
- 2. In plain TEX mode it does not support image inclusion.
- 3. Shading is fully implemented, but the results will not be as good as with a driver producing .pdf as output.
- 4. Opacity works only in conjunction with newer versions of Ghostscript.
- 5. For remembering of pictures (inter-picture connections) you need to use a recent version of pdftex running in DVI-mode.

File pgfsys-textures.def

This is a driver file for use with the TEXTURES program. It includes pgfsys-common-postscript.def. This driver has exactly the same restrictions as the driver for dvips.

You can also use the vtex program together with pgfsys-vtex.def to produce Postscript output.

10.2.4 Producing HTML / SVG Output

The tex4ht program converts .dvi-files to .html-files. While the HTML-format cannot be used to draw graphics, the SVG-format can. Using the following driver, you can ask PGF to produce an SVG-picture for each PGF graphic in your text.

File pgfsys-tex4ht.def

This is a driver file for use with the tex4ht program. It includes pgfsys-common-svg.def.

When using this driver you should be aware of the following restrictions:

- 1. In LATEX mode it uses graphicx for the graphics inclusion.
- 2. In plain T_EX mode it does not support image inclusion.
- 3. Remembering of pictures (inter-picture connections) is not supported.
- 4. Text inside pgfpictures is not supported very well. The reason is that the SVG specification currently does not support text very well and, although it is possible to "escape back" to HTML, TikZ has then to guess what size the text rendered by the browser would have.
- 5. Unlike for other output formats, the bounding box of a picture "really crops" the picture.
- 6. Matrices do not work.
- 7. Functional shadings are not supported.

The driver basically works as follows: When a {pgfpicture} is started, appropriate \special commands are used to directed the output of tex4ht to a new file called \jobname-xxx.svg, where xxx is a number that is increased for each graphic. Then, till the end of the picture, each (system layer) graphic command creates a special that inserts appropriate SVG literal text into the output file. The exact details are a bit complicated since the imaging model and the processing model of PostScript/PDF and SVG are not quite the same; but they are "close enough" for PGF's purposes.

Because text is not supported very well in the SVG standard, you may wish to use the following options to modify the way text is handled:

/tikz/tex4ht node/escape= $\langle boolean \rangle$

(default false)

Selects the rendering method for a text node with the tex4ht driver.

When this key is set to false, text is translated into SVG text, which is somewhat limited: simple characters (letters, numerals, punctuation, \sum , \int , ...), subscripts and superscripts (but not subsubscripts) will display but everything else will be filtered out, ignored or will produce invalid HTML code (in the worst case). This means that two kind of texts render reasonably well:

- 1. First, plain text without math mode, special characters or anything else special.
- 2. Second, *very* simple mathematical text that contains subscripts or superscripts. Even then, variables are not correctly set in italics and, in general, text simple does not look very nice.

If you use text that contains anything special, even something as simple as \$\alpha\$, this may corrupt the graphic.

```
\tikz \node[draw,tex4ht node/escape=false] {Example : $(a+b)^2=a^2+2ab+b^2$};
```

When you write node[tex4ht node/escape=true] $\{\langle text \rangle\}$, TikZ escapes back to HTML to render the $\langle text \rangle$. This method produces valid HTML code in most cases and the support for complicated text nodes is much better since code that renders well outside a $\{tikzpicture\}$, should also render well inside a text node. Another advantage is that inside text nodes with fixed width, HTML will produce line breaks for long lines. On the other hand, you need a browser with good SVG support to display the picture. Also, the text will display differently, depending on your browsers, the fonts you have on your system and your settings. Finally, TikZ has to guess the size of the text rendered by the browser to scale it and prevent it from sticking from the node. When it fails, the text will be either cropped or too small.

```
\tikz \node[draw,tex4ht node/escape=true]
{Example : $\int_0^\infty\frac{1}{1+t^2}dt=\frac{\pi}{2}$};
```

/tikz/tex4ht node/css=\langle filename\rangle

(default \jobname)

This option allows you to tell the browser what CSS file it should use to style the display of the node (only with tex4ht node/escape=true).

/tikz/tex4ht node/class= $\langle class \ name \rangle$

(default foreignobject)

This option allows you to give a class name to the node, allowing it to be styled by a CSS file (only with tex4ht node/escape=true).

/tikz/tex4ht node/id= $\langle id \ name \rangle$

(default \jobname picture number-node number)

This option allows you to give a unique id to the node, allowing it to be styled by a CSS file (only with tex4ht node/escape=true).

10.2.5 Producing Perfectly Portable DVI Output

File pgfsys-dvi.def

This is a driver file that can be used with any output driver, except for tex4ht.

The driver will produce perfectly portable .dvi files by composing all pictures entirely of black rectangles, the basic and only graphic shape supported by the TEX core. Even straight, but slanted lines are tricky to get right in this model (they need to be composed of lots of little squares).

Naturally, very little is possible with this driver. In fact, so little is possible that it is easier to list what is possible:

- Text boxes can be placed in the normal way.
- Lines and curves can be drawn (stroked). If they are not horizontal or vertical, they are composed of hundred of small rectangles.
- Lines of different width are supported.
- Transformations are supported.

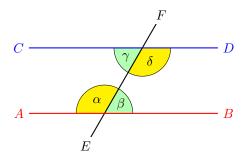
Note that, say, even filling is not supported! (Let alone color or anything fancy.)

This driver has only one real application: It might be useful when you only need horizontal or vertical lines in a picture. Then, the results are quite satisfactory.

Part III

TikZ ist kein Zeichenprogramm

by Till Tantau



When we assume that AB and CD are parallel, i. e., $AB \parallel CD$, then $\alpha = \delta$ and $\beta = \gamma$.

```
\begin{tikzpicture}
 \draw[fill=yellow] (0,0) -- (60:.75cm) arc (60:180:.75cm);
 \draw(120:0.4cm) node {$\alpha$};
 \draw[fill=green!30] (0,0) -- (right:.75cm) arc (0:60:.75cm);
 \draw(30:0.5cm) node {$\beta$};
 \begin{scope}[shift={(60:2cm)}]
   \draw[fill=green!30] (0,0) -- (180:.75cm) arc (180:240:.75cm);
   \draw (30:-0.5cm) node {$\gamma$};
   \draw[fill=yellow] (0,0) -- (240:.75cm) arc (240:360:.75cm);
   \draw (-60:0.4cm) node {$\delta$};
 \end{scope}
 \begin{scope}[thick]
   \draw (60:-1cm) node[fill=white] {$E$} -- (60:3cm) node[fill=white] {$F$};
                                 (-2,0) node[left] {$A$} -- (3,0) node[right]{$B$};
   \draw[blue,shift={(60:2cm)}] (-3,0) node[left] {$C$} -- (2,0) node[right]{$D$};
   \draw[shift={(60:1cm)},xshift=4cm]
   node [right,text width=6cm,rounded corners,fill=red!20,inner sep=1ex]
     When we assume that $\color{red}AB$ and $\color{blue}CD$ are
     parallel, i.\,e., ${\color{red}AB} \mathbin{\|} \color{blue}CD$,
     then $\alpha = \delta$ and $\beta = \gamma$.
 \end{scope}
\end{tikzpicture}
```

11 Design Principles

This section describes the design principles behind the TikZ frontend, where TikZ means "TikZ ist kein Zeichenprogramm." To use TikZ, as a LaTeX user say \usepackage{tikz} somewhere in the preamble, as a plain TeX user say \input tikz.tex. TikZ's job is to make your life easier by providing an easy-to-learn and easy-to-use syntax for describing graphics.

The commands and syntax of TikZ were influenced by several sources. The basic command names and the notion of path operations is taken from METAFONT, the option mechanism comes from PSTRICKS, the notion of styles is reminiscent of SVG. To make it all work together, some compromises were necessary. I also added some ideas of my own, like coordinate transformations.

The following basic design principles underlie TikZ:

- 1. Special syntax for specifying points.
- 2. Special syntax for path specifications.
- 3. Actions on paths.
- 4. Key-value syntax for graphic parameters.
- 5. Special syntax for nodes.
- 6. Special syntax for trees.
- 7. Grouping of graphic parameters.
- 8. Coordinate transformation system.

11.1 Special Syntax For Specifying Points

TikZ provides a special syntax for specifying points and coordinates. In the simplest case, you provide two TFX dimensions, separated by commas, in round brackets as in (1cm, 2pt).

You can also specify a point in polar coordinates by using a colon instead of a comma as in (30:1cm), which means "1cm in a 30 degrees direction."

If you do not provide a unit, as in (2,1), you specify a point in PGF's xy-coordinate system. By default, the unit x-vector goes 1cm to the right and the unit y-vector goes 1cm upward.

By specifying three numbers as in (1,1,1) you specify a point in PGF's xyz-coordinate system.

It is also possible to use an anchor of a previously defined shape as in (first node.south).

You can add two plus signs before a coordinate as in ++(1cm,0pt). This means "1cm to the right of the last point used." This allows you to easily specify relative movements. For example, (1,0) ++(1,0) ++(0,1) specifies the three coordinates (1,0), then (2,0), and (2,1).

Finally, instead of two plus signs, you can also add a single one. This also specifies a point in a relative manner, but it does not "change" the current point used in subsequent relative commands. For example, (1,0) + (0,1) specifies the three coordinates (1,0), then (2,0), and (1,1).

11.2 Special Syntax For Path Specifications

When creating a picture using TikZ, your main job is the specification of paths. A path is a series of straight or curved lines, which need not be connected. TikZ makes it easy to specify paths, partly using the syntax of METAPOST. For example, to specify a triangular path you use

```
(5pt,0pt) -- (0pt,0pt) -- (0pt,5pt) -- cycle
```

and you get \triangleright when you draw this path.

11.3 Actions on Paths

A path is just a series of straight and curved lines, but it is not yet specified what should happen with it. One can draw a path, fill a path, shade it, clip it, or do any combination of these. Drawing (also known as stroking) can be thought of as taking a pen of a certain thickness and moving it along the path, thereby drawing on the canvas. Filling means that the interior of the path is filled with a uniform color. Obviously, filling makes sense only for closed paths and a path is automatically closed prior to filling, if necessary.

Given a path as in \path (0,0) rectangle (2ex,1ex);, you can draw it by adding the draw option as in \path[draw] (0,0) rectangle (2ex,1ex);, which yields \subseteq. The \draw command is just an abbreviation for \path[draw]. To fill a path, use the fill option or the \fill command, which is an abbreviation for \path[fill]. The \filldraw command is an abbreviation for \path[fill], draw]. Shading is caused by the shade option (there are \shade and \shadedraw abbreviations) and clipping by the clip option. There is also a \clip command, which does the same as \path[clip], but not commands like \drawclip. Use, say, \draw[clip] or \path[draw,clip] instead.

All of these commands can only be used inside {tikzpicture} environments.

TikZ allows you to use different colors for filling and stroking.

11.4 Key-Value Syntax for Graphic Parameters

Whenever TikZ draws or fills a path, a large number of graphic parameters influences the rendering. Examples include the colors used, the dashing pattern, the clipping area, the line width, and many others. In TikZ, all these options are specified as lists of so called key-value pairs, as in color=red, that are passed as optional parameters to the path drawing and filling commands. This usage is similar to PSTRICKS. For example, the following will draw a thick, red triangle;



\tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (0,1) -- cycle;

11.5 Special Syntax for Specifying Nodes

TikZ introduces a special syntax for adding text or, more generally, nodes to a graphic. When you specify a path, add nodes as in the following example:

```
\tikz \draw (1,1) node {text} -- (2,2);

text
```

Nodes are inserted at the current position of the path, but only after the path has been rendered. When special options are given, as in \draw (1,1) node[circle,draw] {text};, the text is not just put at the current position. Rather, it is surrounded by a circle and this circle is "drawn."

You can add a name to a node for later reference either by using the option name=\(\name\) or by stating the node name in parentheses outside the text as in node[circle] (name) \(\text{text}\).

Predefined shapes include rectangle, circle, and ellipse, but it is possible (though a bit challenging) to define new shapes.

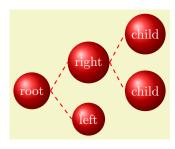
11.6 Special Syntax for Specifying Trees

In addition to the "node syntax," TikZ also introduces a special syntax for drawing trees. The syntax is intergraded with the special node syntax and only few new commands need to be remembered. In essence, a node can be followed by any number of children, each introduced by the keyword child. The children are nodes themselves, each of which may have children in turn.

Since trees are made up from nodes, it is possible to use options to modify the way trees are drawn. Here are two examples of the above tree, redrawn with different options:

```
left right child
```

```
\begin{tikzpicture}
  [edge from parent fork down, sibling distance=15mm, level distance=15mm,
  every node/.style={fill=red!30,rounded corners},
  edge from parent/.style={red,-o,thick,draw}]
  \node {root}
    child {node {left}}
    child {node {right}
        child {node {child}}
        child {node {child}}
    }
    ;
\end{tikzpicture}
```



11.7 Grouping of Graphic Parameters

Graphic parameters should often apply to several path drawing or filling commands. For example, we may wish to draw numerous lines all with the same line width of 1pt. For this, we put these commands in a {scope} environment that takes the desired graphic options as an optional parameter. Naturally, the specified graphic parameters apply only to the drawing and filling commands inside the environment. Furthermore, nested {scope} environments or individual drawing commands can override the graphic parameters of outer {scope} environments. In the following example, three red lines, two green lines, and one blue line are drawn:

```
\begin{tikzpicture}
\begin{scope}[color=red]
\draw (0mm,10mm) -- (10mm,10mm);
\draw (0mm, 8mm) -- (10mm, 8mm);
\draw (0mm, 6mm) -- (10mm, 6mm);
\end{scope}
\begin{scope}[color=green]
\draw (0mm, 4mm) -- (10mm, 4mm);
\draw (0mm, 2mm) -- (10mm, 2mm);
\draw[color=blue] (0mm, 0mm) -- (10mm, 0mm);
\end{scope}
\end{tikzpicture}
```

The {tikzpicture} environment itself also behaves like a {scope} environment, that is, you can specify graphic parameters using an optional argument. These optional apply to all commands in the picture.

11.8 Coordinate Transformation System

TikZ supports both PGF's coordinate transformation system to perform transformations as well as canvas transformations, a more low-level transformation system. (For details on the difference between coordinate transformations and canvas transformations see Section 76.4.)

The syntax is set up in such a way that it is harder to use canvas transformations than coordinate transformations. There are two reasons for this: First, the canvas transformation must be used with great care and often results in "bad" graphics with changing line width and text in wrong sizes. Second, PGF loses track of where nodes and shapes are positioned when canvas transformations are used. So, in almost all circumstances, you should use coordinate transformations rather than canvas transformations.

12 Hierarchical Structures: Package, Environments, Scopes, and Styles

The present section explains how your files should be structured when you use TikZ. On the top level, you need to include the tikz package. In the main text, each graphic needs to be put in a {tikzpicture} environment. Inside these environments, you can use {scope} environments to create internal groups. Inside the scopes you use \path commands to actually draw something. On all levels (except for the package level), graphic options can be given that apply to everything within the environment.

12.1 Loading the Package and the Libraries

```
\usepackage{tikz} % MEX
\input tikz.tex % plain TEX
\usemodule[tikz] % ConTEXt
```

This package does not have any options.

This will automatically load the PGF and the pgffor package.

PGF needs to know what TEX driver you are intending to use. In most cases PGF is clever enough to determine the correct driver for you; this is true in particular if you use LATEX. Currently, the only situation where PGF cannot know the driver "by itself" is when you use plain TEX or ConTEXt together with dvipdfm. In this case, you have to write \def\pgfsysdriver{pgfsys-dvipdfm.def} before you input tikz.tex.

```
\usetikzlibrary{\langle list\ of\ libraries\rangle}
```

Once TikZ has been loaded, you can use this command to load further libraries. The list of libraries should contain the names of libraries separated by commas. Instead of curly braces, you can also use square brackets, which is something $ConT_EXt$ users will like. If you try to load a library a second time, nothing will happen.

```
Example: \usetikzlibrary{arrows}
```

The above command will load a whole bunch of extra arrow tip definitions.

What this command does is to load the file tikzlibrary $\langle library \rangle$.code.tex for each $\langle library \rangle$ in the $\langle list\ of\ libraries \rangle$. If this file does not exist, the file pgflibrary $\langle library \rangle$.code.tex is loaded instead. If this file also does not exist, an error message is printed. Thus, to write your own library file, all you need to do is to place a file of the appropriate name somewhere where TeX can find it. LaTeX, plain TeX, and ConTeXt users can then use your library.

12.2 Creating a Picture

12.2.1 Creating a Picture Using an Environment

The "outermost" scope of TikZ is the {tikzpicture} environment. You may give drawing commands only inside this environment, giving them outside (as is possible in many other packages) will result in chaos.

In TikZ, the way graphics are rendered is strongly influenced by graphic options. For example, there is an option for setting the color used for drawing, another for setting the color used for filling, and also more obscure ones like the option for setting the prefix used in the filenames of temporary files written while plotting functions using an external program. The graphic options are specified in $key\ lists$, see Section 12.4 below for details. All graphic options are local to the $\{tikzpicture\}$ to which they apply.

```
\begin{tikzpicture} [⟨options⟩] 
⟨environment contents⟩ 
\end{tikzpicture}
```

All TikZ commands should be given inside this environment, except for the **\tikzset** command. Unlike other packages, it is not possible to use, say, **\pgfpathmoveto** outside this environment and doing so will result in chaos. For TikZ, commands like **\path** are only defined inside this environment, so there is little chance that you will do something wrong here.

When this environment is encountered, the $\langle options \rangle$ are parsed, see Section 12.4. All options given here will apply to the whole picture.

Next, the contents of the environment is processed and the graphic commands therein are put into a box. Non-graphic text is suppressed as well as possible, but non-PGF commands inside a {tikzpicture} environment should not produce any "output" since this may totally scramble the positioning system of the backend drivers. The suppressing of normal text, by the way, is done by temporarily switching the font to \nullfont. You can, however, "escape back" to normal TEX typesetting. This happens, for example, when you specify a node.

At the end of the environment, PGF tries to make a good guess at the size of a bounding box of the graphic and then resizes the picture box such that the box has this size. To "make its guess," everytime PGF encounters a coordinate, it updates the bounding box's size such that it encompasses all these coordinates. This will usually give a good approximation of the bounding box, but will not always be accurate. First, the line thickness of diagonal lines is not taken into account correctly. Second, control points of a curve often lie far "outside" the curve and make the bounding box too large. In this case, you should use the [use as bounding box] option.

The following key influences the baseline of the resulting picture:

/tikz/baseline=⟨dimension or coordinate or default⟩

Normally, the lower end of the picture is put on the baseline of the surrounding text. For example, when you give the code \tikz\draw(0,0)circle(.5ex);, PGF will find out that the lower end of the picture is at -.5ex and that the upper end is at .5ex. Then, the lower end will be put on the baseline, resulting in the following: o.

Using this option, you can specify that the picture should be raised or lowered such that the height $\langle dimension \rangle$ is on the baseline. For example, tikz[baseline=0pt] draw(0,0)circle(.5ex); yields $_{\odot}$ since, now, the baseline is on the height of the x-axis.

This options is often useful for "inlined" graphics as in

```
A \longrightarrow B $A \mathbin{\tikz[baseline] \draw[->>] (0pt,.5ex) -- (3ex,.5ex);} B$
```

(default Opt)

Instead of a $\langle dimension \rangle$ you can also provide a coordinate in parentheses. Then the effect is to put the baseline on the y-coordinate that the given $\langle coordinate \rangle$ has at the end of the picture. This means that, at the end of the picture, the $\langle coordinate \rangle$ is evaluated and then the baseline is set to the y-coordinate of the resulting point. This makes it easy to reference the y-coordinate of, say, the baseline of nodes.

```
Hello world.

Hello \tikz[baseline=(X.base)] \node [cross out,draw] (X) {world.};

Top align: \tikz[baseline=(current bounding box.north)] \draw (0,0) rectangle (1cm,1ex);
```

Use baseline=default to reset the baseline option to its initial configuration.

/tikz/execute at begin picture= $\langle code \rangle$ (no default)

This option causes $\langle code \rangle$ to be executed at the beginning of the picture. This option must be given in the argument of the $\{tikzpicture\}$ environment itself since this option will not have an effect otherwise. After all, the picture has already "started" later on. The effect of multiply setting this option accumulates.

This option is mainly used in styles like the every picture style to execute certain code at the start of a picture.

/tikz/execute at end picture= $\langle code \rangle$ (no default)

This option installs $\langle code \rangle$ that will be executed at the end of the picture. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the $\{tikzpicture\}$ environment.



All options "end" at the end of the picture. To set an option "globally" change the following style:

/tikz/every picture

(style, initially empty)

This style is installed at the beginning of each picture.

```
\tikzset{every picture/.style=semithick}
```

Note that you should not use \tikzset to set options directly. For instance, if you want to use a line width of 1pt by default, do not try to say \tikzset{line width=1pt} at the beginning of your document. This will not work since the line width is changed in many places. Instead, say

```
\tikzset{every picture/.style={line width=1pt}}
```

This will have the desired effect.

In other T_EX formats, you should use the following commands instead:

```
\tikzpicture[\langle options \rangle] \langle environment contents \rangle \text{endtikzpicture}
```

This is the plain T_EX version of the environment.

```
\starttikzpicture[\langle options \rangle]
\langle environment contents \rangle
\stoptikzpicture
```

This is the ConTEXt version of the environment.

12.2.2 Creating a Picture Using a Command

The following command is an alternative to {tikzpicture} that is particular useful for graphics consisting of a single or few commands.

```
\texttt{\tikz}[\langle options \rangle] \{\langle path\ commands \rangle\}
```

This command places the $\langle path\ commands \rangle$ inside a {tikzpicture} environment. The $\langle path\ commands \rangle$ may contain paragraphs and fragile material (like verbatim text).

If there is only one path command, it need not be surrounded by curly braces, if there are several, you need to add them (this is similar to the \foreach statement and also to the rules in programming languages like Java or C concerning the placement of curly braces).

```
Example: \text{tikz}(\text{draw (0,0) rectangle (2ex,1ex);}) yields \square Example: \text{tikz (0,0) rectangle (2ex,1ex);} yields \square
```

12.2.3 Adding a Background

By default, pictures do not have any background, that is, they are "transparent" on all parts on which you do not draw anything. You may instead wish to have a colored background behind your picture or a black frame around it or lines above and below it or some other kind of decoration.

Since backgrounds are often not needed at all, the definition of styles for adding backgrounds has been put in the library package backgrounds. This package is documented in Section 26.

12.3 Using Scopes to Structure a Picture

Inside a {tikzpicture} environment you can create scopes using the {scope} environment. This environment is available only inside the {tikzpicture} environment, so once more, there is little chance of doing anything wrong.

12.3.1 The Scope Environment

```
\begin{scope}[⟨options⟩]
  ⟨environment contents⟩
\end{scope}
```

All $\langle options \rangle$ are local to the $\langle environment\ contents \rangle$. Furthermore, the clipping path is also local to the environment, that is, any clipping done inside the environment "ends" at its end.

The following style influences scopes:

```
/tikz/every scope
```

(style, initially empty)

This style is installed at the beginning of every scope.

The following options are useful for scopes:

```
/tikz/execute at begin scope=\langle code \rangle
```

(no default)

This option install some code that will be executed at the beginning of the scope. This option must be given in the argument of the {scope} environment.

The effect applies only to the current scope, not to subscopes.

```
/tikz/execute at end scope=\langle code \rangle
```

(no default)

This option installs some code that will be executed at the end of the current scope. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the {scope} environment.

Again, the effect applies only to the current scope, not to subscopes.

```
\scope[\langle options \rangle]
\langle environment\ contents \rangle
\endscope
```

Plain T_EX version of the environment.

```
\startscope[\langle options \rangle] \ \langle environment\ contents \rangle
\startscope
```

ConTEXt version of the environment.

12.3.2 Shorthand for Scope Environments

There is a small library that makes using scopes a bit easier:

```
\usetikzlibrary{scopes} % MEX and plain TeX \usetikzlibrary[scopes] % ConTeXt
```

This library defines a shorthand for starting and ending {scope} environments.

When this library is loaded, the following happens: At certain places inside a TikZ picture, it is allowed to start a scope just using a single brace, provided the single brace is followed by options in square brackets:

In the above example, { [thick] actually causes a \begin{scope}[thick] to be inserted, and the corresponding closing } causes an \end{scope} to be inserted.

The "certain places" where an opening brace has this special meaning are the following: First, right after the semicolon that ends a path. Second, right after the end of a scope. Third, right at the beginning of a scope, which includes the beginning of a picture. Also note that some square bracket must follow, otherwise the brace is treated as a normal TeX scope.

12.3.3 Single Command Scopes

In some situations it is useful to create a scope for a single command. For instance, when you wish to use algorithm graph drawing in order to layout a tree, the path of the tree needs to be surrounded by a scope whose only purpose is to take a key that selects a layout for the scope. Similarly, in order to put something on a background layer, a scope needs to be created. In such cases, where it will cumbersome to create a \begin{scope} and \end{scope} pair just for a single command, the \scoped command may be useful:

```
\colone{cond} [\langle options \rangle] \langle path \ command \rangle
```

This command works like $\texttt{\tikz}$, only you can use it inside a $\texttt{\tikzpicture}$. It will take the following $\langle path\ command \rangle$ and put it inside a $\texttt{\tikzpicture}$ with the $\langle options \rangle$ set. The $\langle path\ command \rangle$ may either be a single command ended by a semicolon or it may contain multiple commands, but then they must be surrounded by curly braces.

```
Hello world

Hello world

Hello world

Hello world

Hello world

\[
\text{Node [fill=white] at (1,1) {Hello world};} \\
\text{Nscoped [on background layer]} \\
\text{draw (0,0) grid (3,2);} \\
\text{end{tikzpicture}}
\]
```

12.3.4 Using Scopes Inside Paths

The \path command, which is described in much more detail in later sections, also takes graphic options. These options are local to the path. Furthermore, it is possible to create local scopes within a path simply by using curly braces as in

```
\tikz \draw (0,0) -- (1,1)
{[rounded corners] -- (2,0) -- (3,1)}
-- (3,0) -- (2,1);
```

Note that many options apply only to the path as a whole and cannot be scoped in this way. For example, it is not possible to scope the color of the path. See the explanations in the section on paths for more details.

Finally, certain elements that you specify in the argument to the \path command also take local options. For example, a node specification takes options. In this case, the options apply only to the node, not to the surrounding path.

12.4 Using Graphic Options

12.4.1 How Graphic Options Are Processed

Many commands and environments of TikZ accept options. These options are so-called key lists. To process the options, the following command is used, which you can also call yourself. Note that it is usually better not to call this command directly, since this will ensure that the effect of options are local to a well-defined scope.

$\tikzset{\langle options \rangle}$

This command will process the $\langle options \rangle$ using the \pgfkeys command, documented in detail in Section 63, with the default path set to /tikz. Under normal circumstances, the $\langle options \rangle$ will be lists of comma-separated pairs of the form $\langle key \rangle = \langle value \rangle$, but more fancy things can happen when you use the power of the pgfkeys mechanism, see Section 63 once more.

When a pair $\langle key \rangle = \langle value \rangle$ is processed, the following happens:

- 1. If the $\langle key \rangle$ is a full key (starts with a slash) it is handled directly as described in Section 63.
- 2. Otherwise (which is usually the case), it is checked whether $/\text{tikz}/\langle key \rangle$ is a key and, if so, it is executed.
- 3. Otherwise, it is checked whether $/pgf/\langle key \rangle$ is a key and, if so, it is executed.
- 4. Otherwise, it is checked whether $\langle key \rangle$ is a color and, if so, color= $\langle key \rangle$ is executed.
- 5. Otherwise, it is checked whether $\langle key \rangle$ contains a dash and, if so, arrows= $\langle key \rangle$ is executed.
- 6. Otherwise, it is checked whether $\langle key \rangle$ is the name of a shape and, if so, shape= $\langle key \rangle$ is executed.
- 7. Otherwise, an error message is printed.

Note that by the above description, all keys starting with /tikz and also all keys starting with /pgf can be used as $\langle key \rangle$ s in an $\langle options \rangle$ list.

12.4.2 Using Styles to Manage How Pictures Look

There is a way of organizing sets of graphic options "orthogonally" to the normal scoping mechanism. For example, you might wish all your "help lines" to be drawn in a certain way like, say, gray and thin (do not dash them, that distracts). For this, you can use styles.

A style is a key that, when used, causes a set of graphic options to be processed. Once a style has been defined, it can be used like any other key. For example, the predefined help lines style, which you should use for lines in the background like grid lines or construction lines.

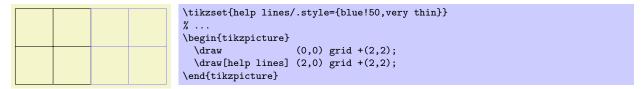
Defining styles is also done using options. Suppose we wish to define a style called my style and when this style is used, we want the draw color to be set to red and the fill color be set to red!20. To achieve this, we use the following option:

```
my style/.style={draw=red,fill=red!20}
```

The meaning of the curious /.style is the following: "The key my style should not be used here but, rather, be defined. So, set up things such that using the key my style will, in the following, have the same effect as if we had written draw=red,fill=red!20 instead."

Returning to the help lines example, suppose we prefer blue help lines. This could be achieved as follows:

Naturally, one of the main ideas behind styles is that they can be used in different pictures. In this case, we have to use the **\tikzset** command somewhere at the beginning.



Since styles are just special cases of pgfkeys's general style facility, you can actually do quite a bit more. Let us start with adding options to an already existing style. This is done using /.append style instead of /.style:

```
\begin{tikzpicture}[help lines/.append style=blue!50]
\draw (0,0) grid +(2,2);
\draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

In the above example, the option blue!50 is appended to the style help lines, which now has the same effect as black!50, very thin, blue!50. Note that two colors are set, so the last one will "win." There also exists a handler called /.prefix style that adds something at the beginning of the style.

Just as normal keys, styles can be parameterized. This means that you write $\langle style \rangle = \langle value \rangle$ when you use the style instead of just $\langle style \rangle$. In this case, all occurrences of #1 in $\langle style \rangle$ are replaced by $\langle value \rangle$. Here is an example that shows how this can be used.

For parameterized styles you can also set a default value using the /.default handler:

For more details on using and setting styles, see also Section 63.

13 Specifying Coordinates

13.1 Overview

A coordinate is a position on the canvas on which your picture is drawn. TikZ uses a special syntax for specifying coordinates. Coordinates are always put in round brackets. The general syntax is $([\langle options \rangle] \langle coordinates)$.

The $\langle coordinate\ specification \rangle$ specifies coordinates using one of many different possible $coordinate\ systems$. Examples are the Cartesian coordinate system or polar coordinates or spherical coordinates. No matter which coordinate system is used, in the end, a specific point on the canvas is represented by the coordinate.

There are two ways of specifying which coordinate system should be used:

Explicitly You can specify the coordinate system explicitly. To do so, you give the name of the coordinate system at the beginning, followed by cs:, which stands for "coordinate system," followed by a specification of the coordinate using the key-value syntax. Thus, the general syntax for \(\lambda coordinate specification \rangle\) in the explicit case is \(\lambda coordinate system \rangle\) cs:\(\lambda list of key-value pairs specific to the coordinate system \rangle\).

Implicitly The explicit specification is often too verbose when numerous coordinates should be given. Because of this, for the coordinate systems that you are likely to use often a special syntax is provided. TikZ will notice when you use a coordinate specified in a special syntax and will choose the correct coordinate system automatically.

Here is an example in which explicit the coordinate systems are specified explicitly:

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (canvas cs:x=0cm,y=2mm)
  -- (canvas polar cs:radius=2cm,angle=30);
  \end{tikzpicture}
```

In the next example, the coordinate systems are implicit:

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0cm,2mm) -- (30:2cm);
  \end{tikzpicture}
```

It is possible to give options that apply only to a single coordinate, although this makes sense for transformation options only. To give transformation options for a single coordinate, give these options at the beginning in brackets:

13.2 Coordinate Systems

13.2.1 Canvas, XYZ, and Polar Coordinate Systems

Let us start with the basic coordinate systems.

Coordinate system canvas

The simplest way of specifying a coordinate is to use the canvas coordinate system. You provide a dimension d_x using the x= option and another dimension d_y using the y= option. The position on the canvas is located at the position that is d_x to the right and d_y above the origin.

```
/tikz/cs/x=\langle dimension\rangle
```

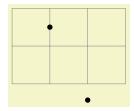
(no default, initially Opt)

Distance by which the coordinate is to the right of the origin. You can also write things like 1cm+2pt since the mathematical engine is used to evaluate the $\langle dimension \rangle$.

```
/tikz/cs/y=\langle dimension\rangle
```

(no default, initially Opt)

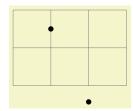
Distance by which the coordinate is above the origin.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\fill (canvas cs:x=1cm,y=1.5cm) circle (2pt);
  \fill (canvas cs:x=2cm,y=-5mm+2pt) circle (2pt);
\end{tikzpicture}
```

To specify a coordinate in the coordinate system implicitly, you use two dimensions that are separated by a comma as in (0cm,3pt) or (2cm,\textheight).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\fill (1cm,1.5cm) circle (2pt);
  \fill (2cm,-5mm+2pt) circle (2pt);
\end{tikzpicture}
```

Coordinate system xyz

The xyz coordinate system allows you to specify a point as a multiple of three vectors called the x-, y-, and z-vectors. By default, the x-vector points 1cm to the right, the y-vector points 1cm upwards, but this can be changed arbitrarily as explained in Section 23.2. The default z-vector points to (-3.85 mm, -3.85 mm).

To specify the factors by which the vectors should be multiplied before being added, you use the following three options:

```
/tikz/cs/x=\langle factor\rangle
```

(no default, initially 0)

Factor by which the x-vector is multiplied.

```
/tikz/cs/y=\langle factor\rangle
```

(no default, initially 0)

Works like x.

```
/tikz/cs/z=\factor\
```

(no default, initially 0)

Works like x.

```
\text{begin{tikzpicture}[->]}
\draw (0,0) -- (xyz cs:x=1);
\draw (0,0) -- (xyz cs:y=1);
\draw (0,0) -- (xyz cs:z=1);
\end{tikzpicture}
```

This coordinate system can also be selected implicitly. To do so, you just provide two or three commaseparated factors (not dimensions).

```
\text{\login{tikzpicture}[->] \\draw (0,0) -- (1,0); \\draw (0,0) -- (0,1,0); \\draw (0,0) -- (0,0,1); \\end{tikzpicture}
```

Note: It is possible to use coordinates like (1,2cm), which are neither canvas coordinates nor xyz coordinates. The rule is the following: If a coordinate is of the implicit form $(\langle x \rangle, \langle y \rangle)$, then $\langle x \rangle$ and $\langle y \rangle$

are checked, independently, whether they have a dimension or whether they are dimensionless. If both have a dimension, the canvas coordinate system is used. If both lack a dimension, the xyz coordinate system is used. If $\langle x \rangle$ has a dimension and $\langle y \rangle$ has not, then the sum of two coordinate $(\langle x \rangle, 0pt)$ and $(0, \langle y \rangle)$ is used. If $\langle y \rangle$ has a dimension and $\langle x \rangle$ has not, then the sum of two coordinate $(\langle x \rangle, 0)$ and $(0pt, \langle y \rangle)$ is used.

Note furthermore: An expression like (2+3cm,0) does not mean the same as (2cm+3cm,0). Instead, if $\langle x \rangle$ or $\langle y \rangle$ internally uses a mixture of dimensions and dimensionless values, then all dimensionless values are "upgraded" to dimensions by interpreting them as pt. So, 2+3cm is the same dimension as 2pt+3cm.

Coordinate system canvas polar

The canvas polar coordinate system allows you to specify polar coordinates. You provide an angle using the angle= option and a radius using the radius= option. This yields the point on the canvas that is at the given radius distance from the origin at the given degree. An angle of zero degrees to the right, a degree of 90 upward.

$/tikz/cs/angle=\langle degrees \rangle$

(no default)

The angle of the coordinate. The angle must always be given in degrees and should be between -360 and 720.

$\texttt{/tikz/cs/radius=} \langle dimension \rangle$

(no default)

The distance from the origin.

/tikz/cs/x radius=⟨dimension⟩

(no default)

A polar coordinate is, after all, just a point on a circle of the given $\langle radius \rangle$. When you provide an x-radius and also a y-radius, you specify an ellipse instead of a circle. The radius option has the same effect as specifying identical x radius and y radius options.

/tikz/cs/y radius= $\langle dimension \rangle$

(no default)

Works like x radius.

```
/
```

```
\tikz \draw (0,0) -- (canvas polar cs:angle=30,radius=1cm);
```

The implicit form for canvas polar coordinates is the following: you specify the angle and the distance, separated by a colon as in (30:1cm).



```
\tikz \draw (0cm,0cm) -- (30:1cm) -- (60:1cm) -- (120:1cm) -- (150:1cm) -- (180:1cm);
```

Two different radii are specified by writing (30:1cm and 2cm).

For the implicit form, instead of an angle given as a number you can also use certain words. For example, up is the same as 90, so that you can write \tikz \draw (0,0) -- (2ex,0pt) -- +(up:1ex); and get \(\to \). Apart from up you can use down, left, right, north, south, west, east, north east, north west, south east, south west, all of which have their natural meaning.

Coordinate system xyz polar

This coordinate system work similarly to the canvas polar system. However, the radius and the angle are interpreted in the xy-coordinate system, not in the canvas system. More detailed, consider the circle or ellipse whose half axes are given by the current x-vector and the current y-vector. Then, consider the point that lies at a given angle on this ellipse, where an angle of zero is the same as the x-vector and an angle of 90 is the y-vector. Finally, multiply the resulting vector by the given radius factor. Voilà.

```
/tikz/cs/angle=\langle degrees\rangle
```

(no default)

The angle of the coordinate interpreted in the ellipse whose axes are the x-vector and the y-vector.

$/tikz/cs/radius=\langle factor \rangle$

(no default)

A factor by which the x-vector and y-vector are multiplied prior to forming the ellipse.

```
/tikz/cs/x radius=\langle dimension \rangle
```

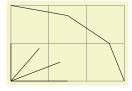
(no default)

A specific factor by which only the x-vector is multiplied.

```
/tikz/cs/y radius=\langle dimension\rangle
```

(no default)

Works like x radius.



```
\begin{tikzpicture} [x=1.5cm,y=1cm]
  \draw[help lines] (0cm,0cm) grid (3cm,2cm);

\draw (0,0) -- (xyz polar cs:angle=0,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=30,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=60,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=90,radius=1);

\draw (xyz polar cs:angle=0,radius=2)
  -- (xyz polar cs:angle=30,radius=2)
  -- (xyz polar cs:angle=60,radius=2)
  -- (xyz polar cs:angle=90,radius=2);
  \end{tikzpicture}
```

The implicit version of this option is the same as the implicit version of canvas polar, only you do not provide a unit.



```
\tikz[x={(0cm,1cm)},y={(-1cm,0cm)}]
\draw (0,0) -- (30:1) -- (60:1) -- (90:1)
-- (120:1) -- (150:1);
```

Coordinate system xy polar

This is just an alias for xyz polar, which some people might prefer as there is no z-coordinate involved in the xyz polar coordinates.

13.2.2 Barycentric Systems

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors v_1, v_2, \ldots, v_n and numbers $\alpha_1, \alpha_2, \ldots, \alpha_n$. Then the barycentric coordinate specified by these vectors and numbers is

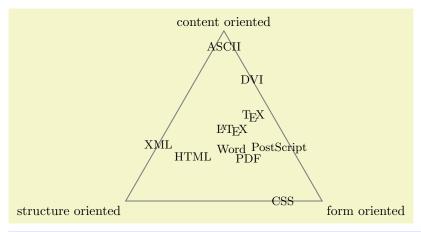
$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \dots + \alpha_n}$$

The barycentric cs allows you to specify such coordinates easily.

Coordinate system barycentric

For this coordinate system, the $\langle coordinate\ specification \rangle$ should be a comma-separated list of expressions of the form $\langle node\ name \rangle = \langle number \rangle$. Note that (currently) the list should not contain any spaces before or after the $\langle node\ name \rangle$ (unlike normal key-value pairs).

The specified coordinate is now computed as follows: Each pair provides one vector and a number. The vector is the center anchor of the $\langle node\ name \rangle$. The number is the $\langle number \rangle$. Note that (currently) you cannot specify a different anchor, so that in order to use, say, the north anchor of a node you first have to create a new coordinate at this north anchor. (Using for instance \coordinate(mynorth) at (mynode.north);.)



```
\begin{tikzpicture}
                           at (90:3cm);
  \coordinate (content)
  \coordinate (structure) at (210:3cm);
                           at (-30:3cm);
  \coordinate (form)
                       at (content)
                                       {content oriented}:
  \node [above]
  \node [below left] at (structure) {structure oriented};
  \node [below right] at (form)
                                       {form oriented};
  \draw [thick,gray] (content.south) -- (structure.north east) -- (form.north west) -- cycle;
  \small
  \node at (barycentric cs:content=0.5,structure=0.1 ,form=1)
                                                                     {PostScript};
  \node at (barycentric cs:content=1 ,structure=0 ,form=0.4)
                                                                     {DVI}:
  \node at (barycentric cs:content=0.5,structure=0.5 ,form=1)
                                                                     {PDF};
  \node at (barycentric cs:content=0 ,structure=0.25,form=1)
                                                                     {CSS};
  \node at (barycentric cs:content=0.5,structure=1 ,form=0)
                                                                     {XML}:
  \node at (barycentric cs:content=0.5,structure=1
                                                                     {HTML};
 \node at (barycentric cs:content=1 ,structure=0.2 ,form=0.8)
\node at (barycentric cs:content=1 ,structure=0.6 ,form=0.8)
                                                                     {\TeX}:
                                                                     {\LaTeX};
  \node at (barycentric cs:content=0.8,structure=0.8 ,form=1)
                                                                     {Word}:
  \node at (barycentric cs:content=1 ,structure=0.05,form=0.05) {ASCII};
\end{tikzpicture}
```

13.2.3 Node Coordinate System

In PGF and in TikZ it is quite easy to define a node that you wish to reference at a later point. Once you have defined a node, there are different ways of referencing points of the node. To do so, you use the following coordinate system:

Coordinate system node

This coordinate system is used to reference a specific point inside or on the border of a previously defined node. It can be used in different ways, so let us go over them one by one.

You can use three options to specify which coordinate you mean:

/tikz/cs/name= $\langle node \ name \rangle$

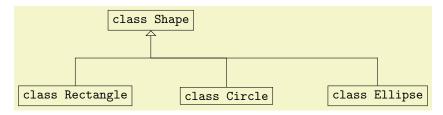
(no default)

Specifies the node in which you which to specify a coordinate. The $\langle node\ name \rangle$ is the name that was previously used to name the node using the name= $\langle node\ name \rangle$ option or the special node name syntax.

/tikz/anchor=\lanchor\range

(no default)

Specifies an anchor of the node. Here is an example:



/tikz/cs/angle=\langle degrees\rangle

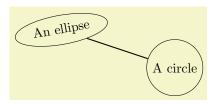
(no default)

It is also possible to provide an angle *instead* of an anchor. This coordinate refers to a point of the node's border where a ray shot from the center in the given angle hits the border. Here is an example:



```
\begin{tikzpicture}
  \node (start) [draw,shape=ellipse] {start};
  \foreach \angle in {-90, -80, ..., 90}
   \draw (node cs:name=start,angle=\angle)
    .. controls +(\angle:1cm) and +(-1,0) .. (2.5,0);
  \end{tikzpicture}
```

It is possible to provide *neither* the anchor= option nor the angle= option. In this case, TikZ will calculate an appropriate border position for you. Here is an example:



TikZ will be reasonably clever at determining the border points that you "mean," but, naturally, this may fail in some situations. If TikZ fails to determine an appropriate border point, the center will be used instead.

Automatic computation of anchors works only with the line-to operations --, the vertical/horizontal versions |- and -|, and with the curve-to operation ... For other path commands, such as parabola or plot, the center will be used. If this is not desired, you should give a named anchor or an angle anchor.

Note that if you use an automatic coordinate for both the start and the end of a line-to, as in --(node cs:name=b)--, then two border coordinates are computed with a move-to between them. This is usually exactly what you want.

If you use relative coordinates together with automatic anchor coordinates, the relative coordinates are computed relative to the node's center, not relative to the border point. Here is an example:

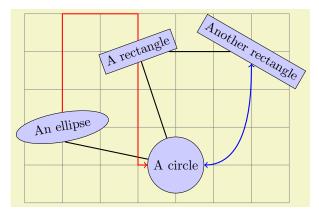
```
\tikz \draw (0,0) node(x) [draw] {Text}
rectangle (1,1)
(node cs:name=x) -- +(1,1);
```

Similarly, in the following examples both control points are (1,1):



The implicit way of specifying the node coordinate system is to simply use the name of the node in parentheses as in (a) or to specify a name together with an anchor or an angle separated by a dot as in (a.north) or (a.10).

Here is a more complete example:



13.2.4 Tangent Coordinate Systems

Coordinate system tangent

This coordinate system, which is available only when the TikZ library calc is loaded, allows you to compute the point that lies tangent to a shape. In detail, consider a $\langle node \rangle$ and a $\langle point \rangle$. Now, draw a straight line from the $\langle point \rangle$ so that it "touches" the $\langle node \rangle$ (more formally, so that it is tangent to this $\langle node \rangle$). The point where the line touches the shape is the point referred to by the tangent coordinate system.

The following options may be given:

```
/\text{tikz/cs/node} = \langle node \rangle (no default)
```

This key specifies the node on whose border the tangent should lie.

```
/\text{tikz/cs/point} = \langle point \rangle (no default)
```

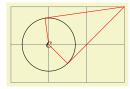
This key specifies the point through which the tangent should go.

```
/\text{tikz/cs/solution} = \langle number \rangle (no default)
```

Specifies which solution should be used if there are more than one.

A special algorithm is needed in order to compute the tangent for a given shape. Currently, tangents can be computed for nodes whose shape is one of the following:

- coordinate
- circle



There is no implicit syntax for this coordinate system.

13.2.5 Defining New Coordinate Systems

While the set of coordinate systems that TikZ can parse via their special syntax is fixed, it is possible and quite easy to define new explicitly named coordinate systems. For this, the following commands are used:

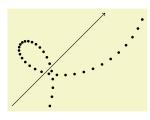
```
\tikzdeclarecoordinatesystem{\langle name \rangle} {\langle code \rangle}
```

This command declares a new coordinate system named $\langle name \rangle$ that can later on be used by writing $(\langle name \rangle \ cs: \langle arguments \rangle)$. When TikZ encounters a coordinate specified in this way, the $\langle arguments \rangle$ are passed to $\langle code \rangle$ as argument #1.

It is now the job of $\langle code \rangle$ to make sense of the $\langle arguments \rangle$. At the end of $\langle code \rangle$, the two TeX dimensions \pgf@x and \pgf@y should be have the x- and y-canvas coordinate of the coordinate.

It is not necessary, but customary, to parse $\langle arguments \rangle$ using the key-value syntax. However, you can also parse it in any way you like.

In the following example, a coordinate system cylindrical is defined.



```
\makeatletter
\define@key{cylindricalkeys}{angle}{\def\myangle{#1}}
\define@key{cylindricalkeys}{radius}{\def\myradius{#1}}
\define@key{cylindricalkeys}{z}{\def\myz{#1}}
\tikzdeclarecoordinatesystem{cylindrical}%
{%
  \setkeys{cylindricalkeys}{#1}%
  \pgfpointadd{\pgfpointxyz{0}{0}{\myz}}{\pgfpointpolarxy{\myangle}{\myradius}}
}
\begin{tikzpicture}[z=0.2pt]
  \draw [->] (0,0,0) -- (0,0,350);
  \foreach \num in {0,10,...,350}
  \fill (cylindrical cs:angle=\num,radius=1,z=\num) circle (1pt);
\end{tikzpicture}
```

 $\tikzaliascoordinatesystem{\langle new \ name \rangle}{\langle old \ name \rangle}$

Creates an alias of $\langle old \ name \rangle$.

13.3 Coordinates at Intersections

You will wish to compute the intersection of two paths. For the special and frequent case of two perpendicular lines, a special coordinate system called **perpendicular** is available. For more general cases, the intersection library can be used.

13.3.1 Intersections of Perpendicular Lines

A frequent special case of path intersections is the intersection of a vertical line going through a point p and a horizontal line going through some other point q. For this situation there is a useful coordinate system.

Coordinate system perpendicular

You can specify the two lines using the following keys:

```
/tikz/cs/horizontal line through={(\langle coordinate \rangle)} (no default)
```

Specifies that one line is a horizontal line that goes through the given coordinate.

```
tikz/cs/vertical line through={((coordinate))} (no default)
```

Specifies that the other line is vertical and goes through the given coordinate.

However, in almost all cases you should, instead, use the implicit syntax. Here, you write $(\langle p \rangle \mid - \langle q \rangle)$ or $(\langle q \rangle \mid - | \langle p \rangle)$.

For example, $(2,1 \mid -3,4)$ and $(3,4 \mid 2,1)$ both yield the same as (2,4) (provided the *xy*-coordinate system has not been modified).

The most useful application of the syntax is to draw a line up to some point on a vertical or horizontal line. Here is an example:

```
\begin{array}{c} q_2 \\ p_1 \\ \hline \end{array} \begin{array}{c} p_1 \\ \hline \end{array} \end{array} \begin{array}{c} p_1 \\ \hline \end{array} \begin{array}{c} p_1 \\
```

13.3.2 Intersections of Arbitrary Paths

```
\usetikzlibrary{intersections} % MEX and plain TEX \usetikzlibrary[intersections] % ConTEXt
```

This library enables the calculation of intersections of two arbitrary paths. However, due to the low accuracy of TEX, the paths should not be "too complicated". In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths.

To find the intersections of two paths in TikZ, they must be "named". A "named path" is, quite simply, a path that has been named using the following key:

```
\label{local_continuity} $$ \text{tikz/name path} = \langle name \rangle $$ (no default) $$
```

The effect of this key is that, after the path has been constructed, just before it is used, it is associated with $\langle name \rangle$. For name path, this association survives beyond the final semi-colon of the path but not the end of the surrounding scope. For name path global, the association will survive beyond any scope as well. Handle with care.

Any paths created by nodes on the (main) path are ignored, unless this key is explicitly used. If the same $\langle name \rangle$ is used for the main path and the node path(s), then the paths will be added together and then associated with $\langle name \rangle$.

To find the intersection of named paths, the following key is used:

```
\text{/tikz/name intersections=}\{\langle options \rangle\}\ (no default)
```

This key changes the key path to /tikz/intersection and processes (options). These options determine, among other things, which paths to use for the intersection. Having processed the options, any intersections are then found. A coordinate is created at each intersection, which by default, will be named intersection-1, intersection-2, and so on. Optionally, the prefix intersection can be changed, and the total number of intersections stored in a TEX-macro.

The following keys can be used in $\langle options \rangle$:

```
/tikz/intersection/of=\langle name\ path\ 1 \rangle and \langle name\ path\ 2 \rangle (no default)
```

This key is used to specify the names of the paths to use for the intersection.

/tikz/intersection/name= $\langle prefix \rangle$

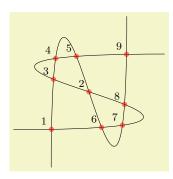
(no default, initially intersection)

This key specifies the prefix name for the coordinate nodes placed at each intersection.

/tikz/intersection/total= $\langle macro \rangle$

(no default)

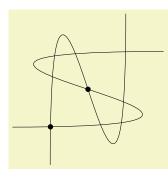
This key means that the total number of intersections found will be stored in $\langle macro \rangle$.



/tikz/intersection/by=\langle comma-separated list\rangle

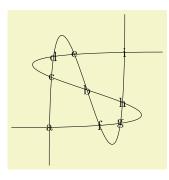
(no default)

This key allows you to specify a list of names for the intersection coordinates. The intersection coordinates will still be named $\langle prefix \rangle - \langle number \rangle$, but additionally the first coordinate will also be named by the first element of the $\langle comma-separated\ list \rangle$. What happens is that the $\langle comma-separated\ list \rangle$ is passed to the $\langle comma-separated\ list \rangle$ a coordinate is created at the already-named intersection.



You can also use the ... notation of the \foreach statement inside the $\langle comma\text{-}separated\ list \rangle$. In case an element of the $\langle comma\text{-}separated\ list \rangle$ starts with options in square brackets, these

options are used when the coordinate is created. A coordinate name can still, but need not, follow the options. This makes it easy to add labels to intersections:



```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) ... controls (8,-1) and (-8,1) ... (2,1);
  \draw [name path=curve 2] (-1,-2) ... controls (-1,8) and (1,-8) ... (1,2);

\fill [name intersections={
      of=curve 1 and curve 2,
      by={[label=center:a],[label=center:...],[label=center:i]}};

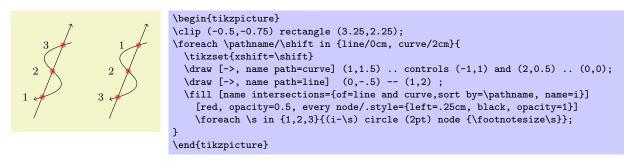
\end{tikzpicture}
```

/tikz/intersection/sort by=⟨path name⟩

(no default)

By default, the intersections are simply returned in the order that the intersection algorithm finds them. Unfortunately, this is not necessarily a "helpful" ordering. This key can be used to sort the

intersections along the path specified by $\langle path \ name \rangle$, which should be one of the paths mentioned in the /tikz/intersection/of key.



13.4 Relative and Incremental Coordinates

13.4.1 Specifying Relative Coordinates

You can prefix coordinates by ++ to make them "relative." A coordinate such as ++(1cm,0pt) means "1cm to the right of the previous position." Relative coordinates are often useful in "local" contexts:

```
\begin{tikzpicture}
\draw (0,0) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
\draw (2,0) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
\draw (1.5,1.5) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
\end{tikzpicture}
```

Instead of ++ you can also use a single +. This also specifies a relative coordinate, but it does not "update" the current point for subsequent usages of relative coordinates. Thus, you can use this notation to specify numerous points, all relative to the same "initial" point:

```
\begin{tikzpicture}
\draw (0,0) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\draw (2,0) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\draw (1.5,1.5) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\end{tikzpicture}
```

There is a special situation, where relative coordinates are interpreted differently. If you use a relative coordinate as a control point of a Bézier curve, the following rule applies: First, a relative first control point is taken relative to the beginning of the curve. Second, a relative second control point is taken relative to the end of the curve. Third, a relative end point of a curve is taken relative to the start of the curve.

This special behavior makes it easy to specify that a curve should "leave or arrive from a certain direction" at the start or end. In the following example, the curve "leaves" at 30° and "arrives" at 60°:

```
\begin{tikzpicture}
\draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
\draw[gray,->] (1,0) -- +(30:1cm);
\draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

13.4.2 Relative Coordinates and Scopes

An interesting question is, how do relative coordinates behave in the presence of scopes? That is, suppose we use curly braces in a path to make part of it "local," how does that affect the current position? On the one hand, the current position certainly changes since the scope only affects options, not the path itself. On the other hand, it may be useful to "temporarily escape" from the updating of the current point.

Since both interpretations of how the current point and scopes should "interact" are useful, there is a (local!) option that allows you to decide which you need.

```
/tikz/current point is local=\langle boolean \rangle
```

(no default, initially false)

Normally, the scope path operation has no effect on the current point. That is, curly braces on a path have no effect on the current position:

```
\begin{tikzpicture}
\draw (0,0) -- ++(1,0) -- ++(0,1) -- ++(-1,0);
\draw[red] (2,0) -- ++(1,0) { -- ++(0,1) } -- ++(-1,0);
\end{tikzpicture}
```

If you set this key to true, this behaviour changes. In this case, at the end of a group created on a path, the last current position reverts to whatever value it had at the beginning of the scope. More precisely, when TikZ encounters $\}$ on a path, it checks whether at this particular moment the key is set to true. If so, the current position reverts to the value is had when the matching $\{$ was read.

```
\begin\{tikzpicture\}
\draw (0,0) -- ++(1,0) -- ++(0,1) -- ++(-1,0);
\draw[red] (2,0) -- ++(1,0)
{ [current point is local] -- ++(0,1) } -- ++(-1,0);
\end\{tikzpicture\}
```

In the above example, we could also have given the option outside the scope, for instance as a parameter to the whole scope.

13.5 Coordinate Calculations

```
\usetikzlibrary{calc} % MEX and plain TeX \usetikzlibrary[calc] % ConTeXt
```

You need to load this library in order to use the coordinate calculation functions described in the present section.

It is possible to do some basic calculations that involve coordinates. In essence, you can add and subtract coordinates, scale them, compute midpoints, and do projections. For instance, ((a) + 1/3*(1cm,0)) is the coordinate that is 1/3cm to the right of the point a:

```
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\node (a) at (1,1) {A};
\fill [red] ($(a) + 1/3*(1cm,0)$) circle (2pt);
\end{tikzpicture}
```

13.5.1 The General Syntax

The general syntax is the following:

```
([\langle options \rangle] \ (coordinate\ computation) \ ).
```

As you can see, the syntax uses the TEX math symbol \$ to indicate that a "mathematical computation" is involved. However, the \$ has no other effect, in particular, no mathematical text is typeset.

The $\langle coordinate\ computation \rangle$ has the following structure:

1. It starts with

```
\langle factor \rangle * \langle coordinate \rangle \langle modifiers \rangle
```

2. This is optionally followed by + or - and then another

```
\langle factor \rangle * \langle coordinate \rangle \langle modifiers \rangle
```

3. This is once more followed by + or - and another of the above modified coordinate; and so on.

In the following, the syntax of factors and of the different modifiers is explained in detail.

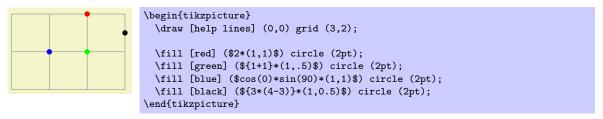
13.5.2 The Syntax of Factors

The $\langle factor \rangle$ s are optional and detected by checking whether the $\langle coordinate\ computation \rangle$ starts with a (. Also, after each \pm a $\langle factor \rangle$ is present if, and only if, the + or - sign is not directly followed by (.

If a $\langle factor \rangle$ is present, it is evaluated using the \pgfmathparse macro. This means that you can use pretty complicated computations inside a factor. A $\langle factor \rangle$ may even contain opening parentheses, which creates a complication: How does TikZ know where a $\langle factor \rangle$ ends and where a coordinate starts? For instance, if the beginning of a $\langle coordinate \ computation \rangle$ is 2*(3+4...), it is not clear whether 3+4 is part of a $\langle coordinate \rangle$ or part of a $\langle factor \rangle$. Because of this, the following rule is used: Once it has been determined, that a $\langle factor \rangle$ is present, in principle, the $\langle factor \rangle$ contains everything up to the next occurrence of *(. Note that there is no space between the asterisk and the parenthesis.

It is permissible to put the $\langle factor \rangle$ in curly braces. This can be used whenever it is unclear where the $\langle factor \rangle$ would end.

Here are some examples of coordinate specifications that consist of exactly one $\langle factor \rangle$ and one $\langle coordinate \rangle$:



13.5.3 The Syntax of Partway Modifiers

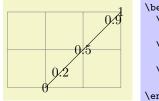
A $\langle coordinate \rangle$ can be followed by different $\langle modifiers \rangle$. The first kind of modifier is the partway modifier. The syntax (which is loosely inspired by Uwe Kern's xcolor package) is the following:

```
\langle coordinate \rangle ! \langle number \rangle ! \langle angle \rangle : \langle second\ coordinate \rangle
```

One could write for instance

(1,2)!.75!(3,4)

The meaning of this is: "Use the coordinate that is three quarters on the way from (1,2) to (3,4)." In general, $\langle coordinate \ x \rangle! \langle number \rangle! \langle coordinate \ y \rangle$ yields the coordinate $(1 - \langle number \rangle) \langle coordinate \ x \rangle + \langle number \rangle \langle coordinate \ y \rangle$. Note that this is a bit different from the way the $\langle number \rangle$ is interpreted in the xcolor package: First, you use a factor between 0 and 1, not a percentage, and, second, as the $\langle number \rangle$ approaches 1, we approach the second coordinate, not the first. It is permissible to use a $\langle number \rangle$ that is smaller than 0 or larger than 1. The $\langle number \rangle$ is evaluated using the \pgfmathparse command and, thus, it can involve complicated computations.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

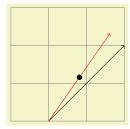
\draw (1,0) -- (3,2);

\foreach \i in {0,0.2,0.5,0.9,1}
  \node at ($(1,0)!\i!(3,2)$) {\i};

\end{tikzpicture}
```

The $\langle second\ coordinate \rangle$ may be prefixed by an $\langle angle \rangle$, separated with a colon, as in (1,1)!.5!60:(2,2). The general meaning of $\langle a \rangle$! $\langle factor \rangle$! $\langle angle \rangle$: $\langle b \rangle$ is "First, consider the line from $\langle a \rangle$ to $\langle b \rangle$. Then rotate this line by $\langle angle \rangle$ around the point $\langle a \rangle$. Then the two endpoints of this line will be $\langle a \rangle$ and some point $\langle c \rangle$. Use this point $\langle c \rangle$ for the subsequent computation, namely the partway computation."

Here are two examples:



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);

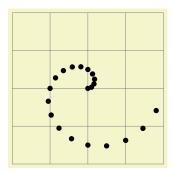
\coordinate (a) at (1,0);
  \coordinate (b) at (3,2);

\draw[->] (a) -- (b);

\coordinate (c) at ($ (a)!1! 10:(b) $);

\draw[->,red] (a) -- (c);

\fill ($ (a)!.5! 10:(b) $) circle (2pt);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (4,4);

\foreach \i in {0,0.1,...,2}
  \fill ($(2,2) !\i! \i*180:(3,2)$) circle (2pt);
\end{tikzpicture}
```

You can repeatedly apply modifiers. That is, after any modifier you can add another (possibly different) modifier.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

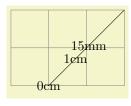
  \draw (0,0) -- (3,2);
  \draw[red] ($(0,0)!.3!(3,2)$) -- (3,0);
  \fill[red] ($(0,0)!.3!(3,2)!.7!(3,0)$) circle (2pt);
  \end{tikzpicture}
```

13.5.4 The Syntax of Distance Modifiers

A distance modifier has nearly the same syntax as a partway modifier, only you use a $\langle dimension \rangle$ (something like 1cm) instead of a $\langle factor \rangle$ (something like 0.5):

```
\langle coordinate \rangle ! \langle dimension \rangle ! \langle angle \rangle : \langle second\ coordinate \rangle
```

When you write $\langle a \rangle ! \langle dimension \rangle ! \langle b \rangle$, this means the following: Use the point that is distanced $\langle dimension \rangle$ from $\langle a \rangle$ on the straight line from $\langle a \rangle$ to $\langle b \rangle$. Here is an example:



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

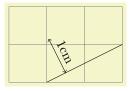
\draw (1,0) -- (3,2);

\foreach \i in {0cm,1cm,15mm}
  \node at ($(1,0)!\i!(3,2)$) {\i};

\end{tikzpicture}
```

As before, if you use a $\langle angle \rangle$, the $\langle second\ coordinate \rangle$ is rotated by this much around the $\langle coordinate \rangle$ before it is used.

The combination of an $\langle angle \rangle$ of 90 degrees with a distance can be used to "offset" a point relative to a line. Suppose, for instance, that you have computed a point (c) that lies somewhere on a line from (a) to (b) and you now wish to offset this point by 1cm so that the distance from this offset point to the line is 1cm. This can be achieved as follows:



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \coordinate (a) at (1,0);
  \coordinate (b) at (3,1);

  \draw (a) -- (b);

  \coordinate (c) at ($ (a)!.25!(b) $);
  \coordinate (d) at ($ (c)!1cm!90:(b) $);

  \draw [<->] (c) -- (d) node [sloped,midway,above] {1cm};
  \end{tikzpicture}
```

13.5.5 The Syntax of Projection Modifiers

The projection modifier is also similar to the above modifiers: It also gives a point on a line from the $\langle coordinate \rangle$ to the $\langle second\ coordinate \rangle$. However, the $\langle number \rangle$ or $\langle dimension \rangle$ is replaced by a $\langle projection\ coordinate \rangle$:

```
\langle coordinate \rangle ! \langle projection \ coordinate \rangle ! \langle angle \rangle : \langle second \ coordinate \rangle
```

Here is an example:

(1,2)!(0,5)!(3,4)

The effect is the following: We project the $\langle projection \ coordinate \rangle$ orthogonally onto the line from $\langle coordinate \rangle$ to $\langle second \ coordinate \rangle$. This makes it easy to compute projected points:



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

\coordinate (a) at (0,1);
  \coordinate (b) at (3,2);
  \coordinate (c) at (2.5,0);

\draw (a) -- (b) -- (c) -- cycle;

\draw[red] (a) -- ($(b)!(a)!(c)$);
  \draw[orange] (b) -- ($(a)!(b)!(c)$);
  \draw[blue] (c) -- ($(a)!(c)!(b)$);

\end{tikzpicture}
```

14 Syntax for Path Specifications

A path is a series of straight and curved line segments. It is specified following a \path command and the specification must follow a special syntax, which is described in the subsections of the present section.

```
\operatorname{\mathtt{ar{path}}} \langle specification \rangle;
```

This command is available only inside a {tikzpicture} environment.

The $\langle specification \rangle$ is a long stream of path operations. Most of these path operations tell TikZ how the path is built. For example, when you write --(0,0), you use a line-to operation and it means "continue the path from wherever you are to the origin."

At any point where TikZ expects a path operation, you can also give some graphic options, which is a list of options in brackets, such as [rounded corners]. These options can have different effects:

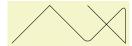
 Some options take "immediate" effect and apply to all subsequent path operations on the path. For example, the rounded corners option will round all following corners, but not the corners "before" and if the sharp corners is given later on the path (in a new set of brackets), the rounding effect will end.

```
\t
```

```
\tikz \draw (0,0) -- (1,1)
[rounded corners] -- (2,0) -- (3,1)
[sharp corners] -- (3,0) -- (2,1);
```

Another example are the transformation options, which also apply only to subsequent coordinates.

2. The options that have immediate effect can be "scoped" by putting part of a path in curly braces. For example, the above example could also be written as follows:



```
\tikz \draw (0,0) -- (1,1)
{[rounded corners] -- (2,0) -- (3,1)}
-- (3,0) -- (2,1);
```

3. Some options only apply to the path as a whole. For example, the color= option for determining the color used for, say, drawing the path always applies to all parts of the path. If several different colors are given for different parts of the path, only the last one (on the outermost scope) "wins":

```
\tikz \draw (0,0) -- (1,1)
[color=red] -- (2,0) -- (3,1)
[color=blue] -- (3,0) -- (2,1);
```

Most options are of this type. In the above example, we would have had to "split up" the path into several **\path** commands:



```
\tikz{\draw (0,0) -- (1,1);
\draw [color=red] (2,0) -- (3,1);
\draw [color=blue] (3,0) -- (2,1);}
```

By default, the \path command does "nothing" with the path, it just "throws it away." Thus, if you write \path(0,0)--(1,1);, nothing is drawn in your picture. The only effect is that the area occupied by the picture is (possibly) enlarged so that the path fits inside the area. To actually "do" something with the path, an option like draw or fill must be given somewhere on the path. Commands like \draw do this implicitly.

Finally, it is also possible to give *node specifications* on a path. Such specifications can come at different locations, but they are always allowed when a normal path operation could follow. A node specification starts with node. Basically, the effect is to typeset the node's text as normal TEX text and to place it at the "current location" on the path. The details are explained in Section 16.

Note, however, that the nodes are *not* part of the path in any way. Rather, after everything has been done with the path what is specified by the path options (like filling and drawing the path due to a fill and a draw option somewhere in the *(specification)*), the nodes are added in a post-processing step.

The following style influences scopes:

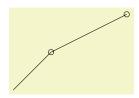
/tikz/every path (style, initially empty)

This style is installed at the beginning of every path. This can be useful for (temporarily) adding, say, the draw option to everything in a scope.

/tikz/insert path= $\langle path \rangle$

(no default)

This key can be used inside an option to add something to the current path. This is mostly useful for defining styles that create graphic contents. This option should be used with care, for instance it should not be used as an argument of, say, a node. In the following example, we use a style to add little circles to a path.



```
\tikz [c/.style={insert path={circle[radius=2pt]}}]
\draw (0,0) -- (1,1) [c] -- (3,2) [c];
```

The effect is the same as of (0,0) -- (1,1) circle[radius=2pt] -- (3,2) circle[radius=2pt].

The following options are for experts only:

/tikz/append after command= $\langle path \rangle$

(no default)

Some of the path commands described in the following sections take optional arguments. For these commands, when you use this key inside these options, the $\langle path \rangle$ will be inserted *after* the path command is done. For instance, when you give this command in the option list of a node, the $\langle path \rangle$ will be added after the node. This is used by, for instance, the label option to allow you to specify a label in the option list of a node, but have this label cause a node to be added after another node.



```
\tikz \draw node [append after command={(foo)--(1,1)},draw] (foo){foo};
```

If this key is called multiple times, the effects accumulate, that is, all of the paths are added in the order to keys were found.

/tikz/prefix after command= $\langle path \rangle$

(no default)

Works like append after command, only the accumulation order is inverse: The $\langle path \rangle$ is added before any earlier paths added using either append after command or prefix after command.

14.1 The Move-To Operation

The perhaps simplest operation is the move-to operation, which is specified by just giving a coordinate where a path operation is expected.

```
\path \dots \langle coordinate \rangle \dots;
```

The move-to operation normally starts a path at a certain point. This does not cause a line segment to be created, but it specifies the starting point of the next segment. If a path is already under construction, that is, if several segments have already been created, a move-to operation will start a new part of the path that is not connected to any of the previous segments.

```
\begin{tikzpicture}
\draw (0,0) --(2,0) (0,1) --(2,1);
\end{tikzpicture}
```

In the specification (0,0) --(2,0) (0,1) --(2,1) two move-to operations are specified: (0,0) and (0,1). The other two operations, namely --(2,0) and --(2,1) are line-to operations, described next.

14.2 The Line-To Operation

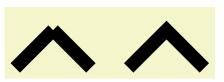
14.2.1 Straight Lines

```
\path ... --\langle coordinate \rangle ...;
```

The line-to operation extends the current path from the current point in a straight line to the given coordinate. The "current point" is the endpoint of the previous drawing operation or the point specified by a prior move-to operation.

You use two minus signs followed by a coordinate in round brackets. You can add spaces before and after the --.

When a line-to operation is used and some path segment has just been constructed, for example by another line-to operation, the two line segments become joined. This means that if they are drawn, the point where they meet is "joined" smoothly. To appreciate the difference, consider the following two examples: In the left example, the path consists of two path segments that are not joined, but they happen to share a point, while in the right example a smooth join is shown.



```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) --(1,1) (1,1) --(2,0);
  \draw (3,0) -- (4,1) -- (5,0);
  \useasboundingbox (0,1.5); % make bounding box higher
  \end{tikzpicture}
```

14.2.2 Horizontal and Vertical Lines

Sometimes you want to connect two points via straight lines that are only horizontal and vertical. For this, you can use two path construction operations.

```
\path \ldots - \ldot\coordinate\rangle \ldots;
\ldot\coordinate\rangle \ldots;
\ldot\coordinate\rangle \ldots;
```

```
\lambda begin{tikzpicture}
\draw (0,0) node(a) [draw] {A} (1,1) node(b) [draw] {B};
\draw (a.north) |- (b.west);
\draw[color=red] (a.east) -| (2,1.5) -| (b.north);
\end{tikzpicture}
```

```
\path ... |-\langle coordinate \rangle ...;
```

—This operations means "first vertical, then horizontal."

14.3 The Curve-To Operation

The curve-to operation allows you to extend a path using a Bézier curve.

```
\path ... ..controls\langle c \rangleand\langle d \rangle ... \langle y \rangle ...;
```

This operation extends the current path from the current point, let us call it x, via a curve to a the current point y. The curve is a cubic Bézier curve. For such a curve, apart from y, you also specify two control points c and d. The idea is that the curve starts at x, "heading" in the direction of c. Mathematically spoken, the tangent of the curve at x goes through c. Similarly, the curve ends at y, "coming from" the other control point, d. The larger the distance between x and c and between d and d, the larger the curve will be.

If the "and $\langle d \rangle$ " part is not given, d is assumed to be equal to c.



As with the line-to operation, it makes a difference whether two curves are joined because they resulted from consecutive curve-to or line-to operations, or whether they just happen to have a common (end) point:



```
\begin{tikzpicture}[line width=10pt]
\draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
\draw (3,0) -- (4,1) .. controls (4,0) and (5,0) .. (5,0);
\useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

14.4 The Cycle Operation

```
\path ... --cycle ...;
```

This operation adds a straight line from the current point to the last point specified by a move-to operation. Note that this need not be the beginning of the path. Furthermore, a smooth join is created between the first segment created after the last move-to operation and the straight line appended by the cycle operation.

Consider the following example. In the left example, two triangles are created using three straight lines, but they are not joined at the ends. In the second example cycle operations are used.



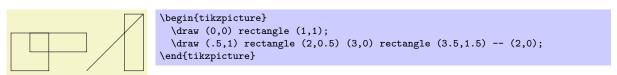
```
\begin{tikzpicture}[line width=10pt]
\draw (0,0) -- (1,1) -- (1,0) -- (0,0) (2,0) -- (3,1) -- (3,0) -- (2,0);
\draw (5,0) -- (6,1) -- (6,0) -- cycle (7,0) -- (8,1) -- (8,0) -- cycle;
\useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

14.5 The Rectangle Operation

A rectangle can obviously be created using four straight lines and a cycle operation. However, since rectangles are needed so often, a special syntax is available for them.

```
\path ... rectangle \langle corner \rangle ...;
```

When this operation is used, one corner will be the current point, another corner is given by $\langle corner \rangle$, which becomes the new current point.



14.6 Rounding Corners

All of the path construction operations mentioned up to now are influenced by the following option:

```
/tikz/rounded corners=\langle inset \rangle
```

(default 4pt)

When this option is in force, all corners (places where a line is continued either via line-to or a curve-to operation) are replaced by little arcs so that the corner becomes smooth.



```
\tikz \draw [rounded corners] (0,0) -- (1,1)
-- (2,0) .. controls (3,1) .. (4,0);
```

The $\langle inset \rangle$ describes how big the corner is. Note that the $\langle inset \rangle$ is not scaled along if you use a scaling option like scale=2.



```
\begin{tikzpicture}
  \draw[color=gray,very thin] (10pt,15pt) circle[radius=10pt];
  \draw[rounded corners=10pt] (0,0) -- (0pt,25pt) -- (40pt,25pt);
  \end{tikzpicture}
```

You can switch the rounded corners on and off "in the middle of path" and different corners in the same path can have different corner radii:



Here is a rectangle with rounded corners:



```
\tikz \draw[rounded corners=1ex] (0,0) rectangle (20pt,2ex);
```

You should be aware, that there are several pitfalls when using this option. First, the rounded corner will only be an arc (part of a circle) if the angle is 90°. In other cases, the rounded corner will still be round, but "not as nice."

Second, if there are very short line segments in a path, the "rounding" may cause inadvertent effects. In such case it may be necessary to temporarily switch off the rounding using sharp corners.

/tikz/sharp corners

(no value)

This options switches off any rounding on subsequent corners of the path.

14.7 The Circle and Ellipse Operations

Circles and ellipses are common path elements for which there is a special path operation.

```
\path ... circle[\langle options \rangle] ...;
```

This command adds a circle to the current path where the center of the circle is the current point by default, but you can use the at option to change this. The new current point of the path will be (typically just remain) the center of the circle.

The radius of the circle is specified using the following options:

/tikz/x radius= $\langle value \rangle$

(no default)

Sets the horizontal radius of the circle (which, when this value is different form the vertical radius, is actually an ellipse). The $\langle value \rangle$ may either be a dimension or a dimensionless number. In the latter case, the number is interpreted in the xy-coordinate system (if the x-unit is set to, say, 2cm, then x radius=3 will have the same effect as x radius=6cm).

```
/tikz/y radius=\langle value \rangle
```

(no default)

Works like the x radius.

 $/tikz/radius=\langle value \rangle$

(no default)

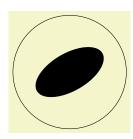
Sets the x radius and y radius simultaneously.

```
/tikz/at=\langle coordinate \rangle
```

(no default)

If this option is explicitly set inside the $\langle options \rangle$ (or indirectly via the every circle style), the $\langle coordinate \rangle$ is used as the center of the circle instead of the current point. Setting at to some value in an enclosing scope has no effect.

The $\langle options \rangle$ may also contain additional options like, say, a rotate or scale, that will only have an effect on the circle.



```
\begin{tikzpicture}
  \draw (1,0) circle [radius=1.5];
  \fill (1,0) circle [x radius=1cm, y radius=5mm, rotate=30];
  \end{tikzpicture}
```

It is possible to set the radius also in some enclosing scope, in this case the options can be left out (but see the note below on what may follow:



```
\begin{tikzpicture}[radius=2pt]
  \draw (0,0) circle -- (1,1) circle -- ++(0,1) circle;
\end{tikzpicture}
```

The following style is used with every circle:

/tikz/every circle

(style, no value)

You can use this key to set up, say, a default radius for every circle. The key will also be used with the ellipse operation.

In case you feel that the names radius and x radius are too long for your taste, you can easily created shorter aliases:

```
\tikzset{r/.style={radius=#1},rx/.style={x radius=#1},ry/.style={y radius=#1}}
```

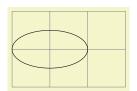
You can then say circle [r=1cm] or circle [rx=1,ry=1.5]. The reason TikZ uses the longer names by default is that it encourages people to write more readable code.

Note: There also exists an older syntax for circles, where the radius of the circle is given in parentheses right after the circle command as in circle (1pt). Although this syntax is a bit more succinct, it is harder to understand for readers of the code and the use of parentheses for something other than a coordinate is ill-chosen.

TikZ will use the following rule to determine whether the old or the normal syntax is used: If circle is directly followed by something that (expands to) an opening parenthesis, then the old syntax is used and inside these following parentheses there must be a single number or dimension representing a radius. In all other cases the new syntax is used.

```
\path ... ellipse[\langle options \rangle] ...;
```

This command has exactly the same effect as circle. The older syntax for this command is ellipse ($\langle x \, radius \rangle$) and $\langle y \, radius \rangle$). As for the circle command, this syntax is not as good as the standard syntax.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (1,1) ellipse [x radius=1cm,y radius=.5cm];
  \end{tikzpicture}
```

14.8 The Arc Operation

The arc operation allows you to add an arc to the current path.

```
\path ... arc[\langle options \rangle] ...;
```

The arc operation adds a part of an ellipse to the current path. The radii of the ellipse are given by the values of x radius and y radius, which should be set in the $\langle options \rangle$. The arc will start at the current point and will end at the end of the arc. The arc will start and end at angles computed from the three keys start angle, end angle, and delta angle. Normally, the first two keys specify the start and end angle. However, in case one of them is empty, it is computed from the other key plus or minus the delta angle. In detail, if end angle is empty, it is set to the start angle plus the delta angle. If the start angle is missing, it is set to the end angle minus the delta angle. If all three keys are set, the delta angle is ignored.

```
\texttt{/tikz/start angle=}\langle degrees \rangle (no default)
```

Sets the start angle.

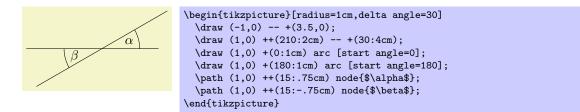
```
/tikz/end angle=\langle degrees \rangle (no default)
```

Sets the end angle.

```
/\text{tikz/delta angle=}\langle degrees \rangle (no default)
```

Sets the delta angle.





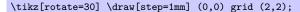
There also exists a shorter syntax for the arc operation, namely **arc** begin directly followed by $(\langle start\ angle \rangle : \langle end\ angle \rangle : \langle radius \rangle)$. However, this syntax is harder to read, so the normal syntax should be preferred in general.

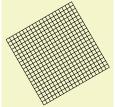
14.9 The Grid Operation

You can add a grid to the current path using the grid path operation.

```
\path ... grid[\langle options \rangle] \langle corner \rangle ...;
```

This operations adds a grid filling a rectangle whose two corners are given by $\langle corner \rangle$ and by the previous coordinate. Thus, the typical way in which a grid is drawn is \draw (1,1) grid (3,3);, which yields a grid filling the rectangle whose corners are at (1,1) and (3,3). All coordinate transformations apply to the grid.



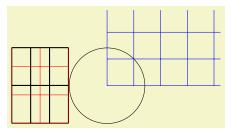


The $\langle options \rangle$, which are local to the grid operation, can be used to influence the appearance of the grid. The stepping of the grid is governed by the following options:

/tikz/step=\(number or dimension or coordinate\)

(no default, initially 1cm)

Sets the stepping in both the x and y-direction. If a dimension is provided, this is used directly. If a number is provided, this number is interpreted in the xy-coordinate system. For example, if you provide the number 2, then the x-step is twice the x-vector and the y-step is twice the y-vector set by the x= and y= options. Finally, if you provide a coordinate, then the x-part of this coordinate will be used as the x-step and the y-part will be used as the y-coordinate.



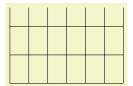
```
\begin{tikzpicture}[x=.5cm]
  \draw[thick] (0,0) grid [step=1] (3,2);
  \draw[red] (0,0) grid [step=.75cm] (3,2);
  \end{tikzpicture}
  \begin{tikzpicture}
  \draw (0,0) circle [radius=1];
  \draw[blue] (0,0) grid [step=(45:1)] (3,2);
  \end{tikzpicture}
```

A complication arises when the x- and/or y-vector do not point along the axes. Because of this, the actual rule for computing the x-step and the y-step is the following: As the x- and y-steps we use the x- and y-components or the following two vectors: The first vector is either ($\langle x$ -grid-step- $number \rangle$, 0) or ($\langle x$ -grid-step- $dimension \rangle$, 0pt), the second vector is $\langle 0, \langle y$ -grid-step- $number \rangle$) or $\langle 0$ -grid-step- $dimension \rangle$).

/tikz/xstep=\langle dimension or number\rangle

(no default, initially 1cm)

Sets the stepping in the x-direction.



```
\tikz \draw (0,0) grid [xstep=.5,ystep=.75] (3,2);
```

 $\verb|/tikz/ystep=| \langle \mathit{dimension} \ \mathit{or} \ \mathit{number} \rangle|$

(no default, initially 1cm)

Sets the stepping in the y-direction.

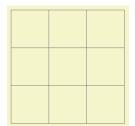
It is important to note that the grid is always "phased" such that it contains the point (0,0) if that point happens to be inside the rectangle. Thus, the grid does *not* always have an intersection at the corner points; this occurs only if the corner points are multiples of the stepping. Note that due to rounding errors, the "last" lines of a grid may be omitted. In this case, you have to add an epsilon to the corner points.

The following style is useful for drawing grids:

/tikz/help lines

(style, initially line width=0.2pt,gray)

This style makes lines "subdued" by using thin gray lines for them. However, this style is not installed automatically and you have to say for example:



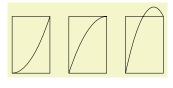
```
\tikz \draw[help lines] (0,0) grid (3,3);
```

14.10 The Parabola Operation

The parabola path operation continues the current path with a parabola. A parabola is a (shifted and scaled) curve defined by the equation $f(x) = x^2$ and looks like this: \vee .

```
\path ... parabola[\langle options \rangle] bend \langle bend \ coordinate \rangle \langle coordinate \rangle ...;
```

This operation adds a parabola through the current point and the given $\langle coordinate \rangle$. If the bend is given, it specifies where the bend should go; the $\langle options \rangle$ can also be used to specify where the bend is. By default, the bend is at the old current point.



```
\begin{tikzpicture}
  \draw
                       (0,0) rectangle
                                                        (1,1.5)
                                                        (1,1.5);
                       (0,0) parabola
  \draw[xshift=1.5cm]
                      (0,0) rectangle
                                                        (1,1.5)
                       (0,0) parabola[bend at end]
                                                       (1,1.5);
  \draw[xshift=3cm]
                       (0,0) rectangle
                                                        (1.1.5)
                       (0,0) parabola bend (.75,1.75) (1,1.5);
\end{tikzpicture}
```

The following options influence parabolas:

```
/tikz/bend=\langle coordinate\rangle
```

(no default)

Has the same effect as saying bend $\langle coordinate \rangle$ outside the $\langle options \rangle$. The option specifies that the bend of the parabola should be at the given $\langle coordinate \rangle$. You have to take care yourself that the bend position is a "valid" position; which means that if there is no parabola of the form $f(x) = ax^2 + bx + c$ that goes through the old current point, the given bend, and the new current point, the result will not be a parabola.

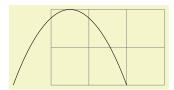
There is one special property of the $\langle coordinate \rangle$: When a relative coordinate is given like +(0,0), the position relative to this coordinate is "flexible." More precisely, this position lies somewhere on a line from the old current point to the new current point. The exact position depends on the next option.

```
/tikz/bend pos=\langle fraction \rangle
```

(no default)

Specifies where the "previous" point is relative to which the bend is calculated. The previous point will be at the $\langle fraction \rangle$ th part of the line from the old current point to the new current point.

The idea is the following: If you say bend pos=0 and bend +(0,0), the bend will be at the old current point. If you say bend pos=1 and bend +(0,0), the bend will be at the new current point. If you say bend pos=0.5 and bend +(0,2cm) the bend will be 2cm above the middle of the line between the start and end point. This is most useful in situations such as the following:



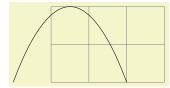
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[bend pos=0.5] bend +(0,2) +(3,0);
  \end{tikzpicture}
```

In the above example, the bend +(0,2) essentially means "a parabola that is 2cm high" and +(3,0) means "and 3cm wide." Since this situation arises often, there is a special shortcut option:

```
/tikz/parabola height=⟨dimension⟩
```

(no default)

This option has the same effect as [bend pos=0.5,bend= $\{+(Opt, \langle dimension \rangle)\}$].



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[parabola height=2cm] +(3,0);
  \end{tikzpicture}
```

The following styles are useful shortcuts:

```
/tikz/bend at start
```

(style, no value)

This places the bend at the start of a parabola. It is a shortcut for the following options: bend pos=0, bend= $\{+(0,0)\}$.

```
/tikz/bend at end
```

(style, no value)

This places the bend at the end of a parabola.

14.11 The Sine and Cosine Operation

The sin and cos operations are similar to the parabola operation. They, too, can be used to draw (parts of) a sine or cosine curve.

```
\path ... sin\langle coordinate \rangle ...;
```

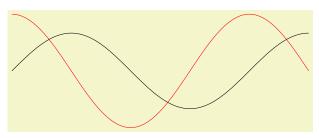
The effect of sin is to draw a scaled and shifted version of a sine curve in the interval $[0, \pi/2]$. The scaling and shifting is done in such a way that the start of the sine curve in the interval is at the old current point and that the end of the curve in the interval is at $\langle coordinate \rangle$. Here is an example that should clarify this:



```
\tikz \draw (0,0) rectangle (1,1) (0,0) sin (1,1) (2,0) rectangle +(1.57,1) (2,0) sin +(1.57,1);
```

```
\path ... \cos(coordinate) ...;
```

This operation works similarly, only a cosine in the interval $[0, \pi/2]$ is drawn. By correctly alternating \sin and \cos operations, you can create a complete sine or cosine curve:



```
\begin{tikzpicture}[xscale=1.57]
  \draw (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0) sin (5,1);
  \draw[color=red] (0,1.5) cos (1,0) sin (2,-1.5) cos (3,0) sin (4,1.5) cos (5,0);
  \end{tikzpicture}
```

Note that there is no way to (conveniently) draw an interval on a sine or cosine curve whose end points are not multiples of $\pi/2$.

14.12 The SVG Operation

The svg operation can be used to extend the current path by a path given in the SVG path data syntax. This syntax is described in detail in Section 8.3 of the SVG 1.1 specification, please consult this specification for details.

```
\path ... svg[\langle options \rangle] "\langle path \ data \rangle" ...;
```

This operation adds the path specified in the $\langle path \; data \rangle$ in SVG 1.1 PATH DATA syntax to the current path. Unlike the SVG-specification, it is permissible that the path data does not start with a moveto command (m or M), in which case the last point of the current path is used as start point. The optional $\langle options \rangle$ apply locally to this path operation, typically you will use them to set up, say, some transformations.

An svG coordinate like 10 20 is always interpreted as (10pt,20pt), so the basic unit is always points (pt). The xy-coordinate system is not used. However, you can use scaling to (locally) change the basic unit. For instance, svg[scale=1cm] (yes, this works, although some rather evil magic is involved) will cause 1cm to be the basic unit.

Warning: The arc operations (a and A) are not numerically stable. This means that they will be quite imprecise, except when the angle is a multiple of 90° (as is, fortunately, most often the case).

14.13 The Plot Operation

The plot operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

Since the syntax and the behaviour of this command are a bit complex, they are described in the separated Section 20.

14.14 The To Path Operation

The to operation is used to add a user-defined path from the previous coordinate to the following coordinate. When you write (a) to (b), a straight line is added from a to b, exactly as if you had written (a) -- (b). However, if you write (a) to [out=135,in=45] (b) a curve is added to the path, which leaves at an angle of 135° at a and arrives at an angle of 45° at b. This is because the options in and out trigger a special path to be used instead of the straight line.

```
\path ... to[\langle options \rangle] \langle nodes \rangle (\langle coordinate \rangle) ...;
```

This path operation inserts the path currently set via the to path option at the current position. The $\langle options \rangle$ can be used to modify (perhaps implicitly) the to path and to set up how the path will be rendered.

Before the to path is inserted, a number of macros are set up that can "help" the to path. These are \tikztostart, \tikztotarget, and \tikztonodes; they are explained in the following.

Start and Target Coordinates. The to operation is always followed by a \(\chico\cordinate\), called the target coordinate. The macro \tikztotarget is set to this coordinate (without the parentheses). There is also a start coordinate, which is the coordinate preceding the to operation. This coordinate can be accessed via the macro \tikztostart. In the following example, for the first to, the macro \tikztostart is Opt,Opt and the \tikztotarget is 0,2. For the second to, the macro \tikztostart is 10pt,10pt and \tikztotarget is a.

```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) to (0,2);
\node (a) at (2,2) {a};
\draw[red] (10pt,10pt) to (a);
\end{tikzpicture}
```

Nodes on to-paths. It is possible to add nodes to the paths constructed by a to operation. To do so, you specify the nodes between the to keyword and the coordinate (if there are options to the to

operation, these come first). The effect of (a) to node {x} (b) (typically) is the same as if you had written (a) -- node {x} (b), namely that the node is placed on the to. This can be used to add labels to tos:

```
\begin{tikzpicture}
  \draw (0,0) to node [sloped,above] {x} (3,2);

\draw (0,0) to[out=90,in=180] node [sloped,above] {x} (3,2);

\end{tikzpicture}
```

Instead of writing the node between the to keyword and the target coordinate, you may also use the following keys to create such nodes:

```
/tikz/edge node=\langle node \ specification \rangle
```

(no default)

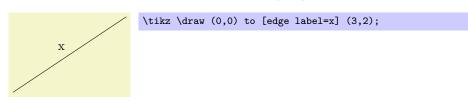
This key can be used inside the $\langle options \rangle$ of a to path command. It will add the $\langle node\ specification \rangle$ to the list of nodes to be placed on the connecting line, just as if you had written the $\langle node\ specification \rangle$ directly after the to keyword:

This key is mostly useful to create labels automatically using other keys.

```
/tikz/edge label=\langle text \rangle
```

(no default)

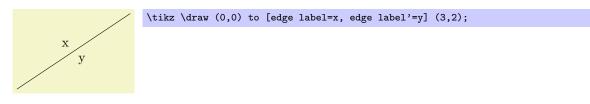
A shorthand for edge node={node[auto]{ $\langle text \rangle$ }}.



```
/tikz/edge label'=\langle text \rangle
```

(no default)

A shorthand for edge node={node[auto,swap]{ $\langle text \rangle$ }}.

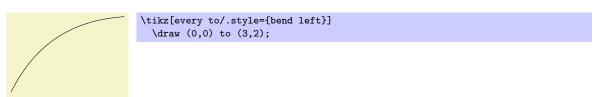


When the quotes library is loaded, additional ways of specifying nodes on to-paths become available, see Section 16.12.2.

Styles for to-paths. In addition to the $\langle options \rangle$ given after the to operation, the following style is also set at the beginning of the to path:

```
/tikz/every to (style, initially empty)
```

This style is installed at the beginning of every to.



Options. The $\langle options \rangle$ given with the to allow you to influence the appearance of the to path. Mostly, these options are used to change the to path. This can be used to change the path from a straight line to, say, a curve.

The path used is set using the following option:

```
/tikz/to path=\langle path \rangle
```

(no default)

Whenever an to operation is used, the $\langle path \rangle$ is inserted. More precisely, the following path is added:

```
{[every to,\langle options \rangle] \langle path \rangle }
```

The $\langle options \rangle$ are the options given to the to operation, the $\langle path \rangle$ is the path set by this option to path.

Inside the $\langle path \rangle$, different macros are used to reference the from- and to-coordinates. In detail, these are:

- \tikztostart will expand to the from-coordinate (without the parentheses).
- \tikztotarget will expand to the to-coordinate.
- \tikztonodes will expand to the nodes between the to operation and the coordinate. Furthermore, these nodes will have the pos option set implicitly.

Let us have a look at a simple example. The standard straight line for a to is achieved by the following $\langle path \rangle$:

```
-- (\tikztotarget) \tikztonodes
```

Indeed, this is the default setting for the path. When we write (a) to (b), the $\langle path \rangle$ will expand to (a) -- (b), when we write

```
(a) to[red] node {x} (b)
```

the $\langle path \rangle$ will expand to

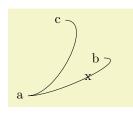
```
(a) -- (b) node[red] {x}
```

It is not possible to specify the path

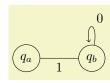
-- \tikztonodes (\tikztotarget)

since TikZ does not allow one to have a macro after -- that expands to a node.

Now let us have a look at how we can modify the $\langle path \rangle$ sensibly. The simplest way is to use a curve.



Here is another example:



```
/tikz/execute at begin to=\langle code \rangle
```

(no default)

The $\langle code \rangle$ is executed prior to the to. This can be used to draw one or more additional paths or to do additional computations.

```
/tikz/execute at end to=\langle code \rangle
```

(no default)

Works like the previous option, only this code is executed after the to path has been added.

```
/tikz/every to
```

(style, initially empty)

This style is installed at the beginning of every to.

There are a number of predefined to paths, see Section 52 for a reference.

14.15 The Let Operation

The *let operation* is the first of a number of path operations that do not actually extend that path, but have different, mostly local, effects.

```
\path ... let(assignment), (assignment), (assignment)... in ...;
```

When this path operation is encountered, the $\langle assignment \rangle$ s are evaluated, one by one. This will store coordinate and number in special registers (which are local to TikZ, they have nothing to do with TEX registers). Subsequently, one can access the contents of these registers using the macros p, x, y, and n.

The first kind of permissible $\langle assignment \rangle$ s have the following form:

```
\n\langle number\ register \rangle = \{\langle formula \rangle\}
```

When an assignment has this form, the $\langle formula \rangle$ is evaluated using the \pgfmathparse operation. The result is stored in the $\langle number\ register \rangle$. If the $\langle formula \rangle$ involves a dimension anywhere (as in 2*3cm/2), then the $\langle number\ register \rangle$ stores the resulting dimension with a trailing pt. A $\langle number\ register \rangle$ can be named arbitrarily and is a normal T_EX parameter to the \n macro. Possible names are {left corner}, but also just a single digit like 5.

Let us call the path that follows a let operation its body. Inside the body, the \n macro can be used to access the register.

```
\n{\langle number\ register \rangle}
```

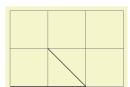
When this macro is used on the left-hand side of an =-sign in a let operation, it has no effect and is just there for readability. When the macro is used on the right-hand side of an =-sign or in the body of the let operation, then it expands to the value stored in the $\langle number\ register \rangle$. This will either be a dimensionless number like 2.0 or a dimension like 5.6pt.

For instance, if we say let $n1=\{1pt+2pt\}$, $n2=\{1+2\}$ in ..., then inside the ... part the macro n1 will expand to 3pt and n2 expands to 3.

The second kind of $\langle assignments \rangle$ have the following form:

```
\p\langle point\ register \rangle = \{\langle formula \rangle\}
```

Point position registers store a single point, consisting of an x-part and a y-part measured in TEX points (pt). In particular, point registers do not store nodes or node names. Here is an example:



```
\p{\langle point \ register \rangle}
```

When this macro is used on the left-hand side of an =-sign in a let operation, it has no effect and is just there for readability. When the macro is used on the right-hand side of an =-sign or in the body of the let operation, then it expands to the x-part (measured in TeX points) of the coordinate stored in the $\langle register \rangle$, followed, by a comma, followed by the y-part.

For instance, if we say let \p1=(1pt,1pt+2pt) in ..., then inside the ... part the macro \p1 will expand to exactly the seven characters "1pt,3pt". This means that you when you write (\p1), this expands to (1pt,3pt), which is presumably exactly what you intended.

$\mathbf{x}\{\langle point\ register \rangle\}$

This macro expands just to the x-part of the point register. If we say as above, as we did above, let p1=(1pt,1pt+2pt) in ..., then inside the ... part the macro x1 expands to 1pt.

```
\y{\langle point \ register \rangle}
```

Works like \x , only for the y-part.

Note that the above macros are available only inside a let operation.

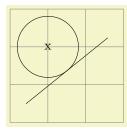
Here is an example where let clauses are used to assemble a coordinate from the x-coordinate of a first point and the y-coordinate of a second point. Naturally, using the |-| notation, this could be written much more compactly.



Note that the effect of a let operation is local to the body of the let operation. If you wish to access a computed coordinate outside the body, you must use a **coordinate** path operation:



For a more useful application of the let operation, let us draw a circle that touches a given line:



14.16 The Scoping Operation

When TikZ encounters and opening or a closing brace ($\{$ or $\}$) at some point where a path operation should come, it will open or close a scope. All options that can be applied "locally" will be scoped inside the

scope. For example, if you apply a transformation like [xshift=1cm] inside the scoped area, the shifting only applies to the scope. On the other hand, an option like color=red does not have any effect inside a scope since it can only be applied to the path as a whole.

Concerning the effect of scopes on relative coordinates, please see Section 13.4.2.

14.17 The Node and Edge Operations

There are two more operations that can be found in paths: **node** and **edge**. The first is used to add a so-called node to a path. This operation is special in the following sense: It does not change the current path in any way. In other words, this operation is not really a path operation, but has an effect that is "external" to the path. The **edge** operation has similar effect in that it adds something *after* the main path has been drawn. However, it works like the **to** operation, that is, it adds a **to** path to the picture after the main path has been drawn.

Since these operations are quite complex, they are described in the separate Section 16.

14.18 The Graph Operation

The graph operation can be used to specify easily how a large number of nodes are connected. This operation is documented in a separate section, see Section 17.

14.19 The PGF-Extra Operation

In some cases you may need to "do some calculations or some other stuff" while a path is constructed. For this, you would like to suspend the construction of the path and suspend TikZ's parsing of the path, you would then like to have some TEX code executed, and would then like to resume the parsing of the path. This effect can be achieved using the following path operation pgfextra. Note that this operation should only be used by real experts and should only be used deep inside clever macros, not on normal paths.

```
\protect{pgfextra}{\langle code \rangle}
```

This command may only be used inside a TikZ path. There it is used like a normal path operation. The construction of the path is temporarily suspended and the $\langle code \rangle$ is executed. Then, the path construction is resumed.

```
\newdimen\mydim
\begin{tikzpicture}
  \mydim=1cm
  \draw (0pt,\mydim) \pgfextra{\mydim=2cm} -- (0pt,\mydim);
\end{tikzpicture}
```

$\protect{\mathsf{pgfextra}}\langle code \rangle \protect{\mathsf{endpgfextra}}$

This is an alternative syntax for the \protect command. If the code following \protect does not start with a brace, the $\langle code \rangle$ is executed until \protect is encountered. What actually happens is that \protect that is not followed by a brace completely shuts down the \protect parse and \protect is a normal macro that restarts the parser.

```
\newdimen\mydim
\begin{tikzpicture}
 \mydim=1cm
 \draw (Opt,\mydim)
  \pgfextra \mydim=2cm \endpgfextra -- (Opt,\mydim);
\end{tikzpicture}
```

15 Actions on Paths

15.1 Overview

Once a path has been constructed, different things can be done with it. It can be drawn (or stroked) with a "pen," it can be filled with a color or shading, it can be used for clipping subsequent drawing, it can be used to specify the extend of the picture—or any combination of these actions at the same time.

To decide what is to be done with a path, two methods can be used. First, you can use a special-purpose command like \draw to indicate that the path should be drawn. However, commands like \draw and \fill are just abbreviations for special cases of the more general method: Here, the \path command is used to specify the path. Then, options encountered on the path indicate what should be done with the path.

For example, \path (0,0) circle (1cm); means "This is a path consisting of a circle around the origin. Do not do anything with it (throw it away)." However, if the option draw is encountered anywhere on the path, the circle will be drawn. "Anywhere" is any point on the path where an option can be given, which is everywhere where a path command like circle (1cm) or rectangle (1,1) or even just (0,0) would also be allowed. Thus, the following commands all draw the same circle:

```
\path [draw] (0,0) circle (1cm);
\path (0,0) [draw] circle (1cm);
\path (0,0) circle (1cm) [draw];
```

Finally, \draw (0,0) circle (1cm); also draws a path, because \draw is an abbreviation for \path [draw] and thus the command expands to the first line of the above example.

Similarly, \fill is an abbreviation for \path[fill] and \filldraw is an abbreviation for the command \path[fill,draw]. Since options accumulate, the following commands all have the same effect:

```
\path [draw,fill] (0,0) circle (1cm);
\path [draw] [fill] (0,0) circle (1cm);
\path [fill] (0,0) circle (1cm) [draw];
\draw [fill] (0,0) circle (1cm);
\fill (0,0) [draw] circle (1cm);
\filldraw (0,0) circle (1cm);
```

In the following subsection the different actions that can be performed on a path are explained. The following commands are abbreviations for certain sets of actions, but for many useful combinations there are no abbreviations:

\draw

Inside {tikzpicture} this is an abbreviation for \path[draw].

\fill

Inside {tikzpicture} this is an abbreviation for \path[fill].

\filldraw

Inside {tikzpicture} this is an abbreviation for \path[fill,draw].

\pattern

Inside {tikzpicture} this is an abbreviation for \path[pattern].

\shade

Inside {tikzpicture} this is an abbreviation for \path[shade].

\shadedraw

Inside {tikzpicture} this is an abbreviation for \path[shade,draw].

\clip

Inside {tikzpicture} this is an abbreviation for \path[clip].

\useasboundingbox

Inside {tikzpicture} this is an abbreviation for \path[use as bounding box].

15.2 Specifying a Color

The most unspecific option for setting colors is the following:

/tikz/color=⟨color name⟩

(no default)

This option sets the color that is used for fill, drawing, and text inside the current scope. Any special settings for filling colors or drawing colors are immediately "overruled" by this option.

The $\langle color \ name \rangle$ is the name of a previously defined color. For LATEX users, this is just a normal "LATEX-color" and the xcolor extensions are allowed. Here is an example:



\tikz \fill[color=red!20] (0,0) circle (1ex);

It is possible to "leave out" the color= part and you can also write:



\tikz \fill[red!20] (0,0) circle (1ex);

What happens is that every option that TikZ does not know, like red!20, gets a "second chance" as a color name.

For plain TEX users, it is not so easy to specify colors since plain TEX has no "standardized" color naming mechanism. Because of this, PGF emulates the xcolor package, though the emulation is *extremely basic* (more precisely, what I could hack together in two hours or so). The emulation allows you to do the following:

- Specify a new color using \definecolor. Only the two color models gray and rgb are supported⁸. Example: \definecolor{orange}{rgb}{1,0.5,0}
- Use \colorlet to define a new color based on an old one. Here, the ! mechanism is supported, though only "once" (use multiple \colorlet for more fancy colors).

Example: \colorlet{lightgray}{black!25}

• Use \color{\langle color name \rangle} to set the color in the current TeX group. \aftergroup-hackery is used to restore the color after the group.

As pointed out above, the color= option applies to "everything" (except to shadings), which is not always what you want. Because of this, there are several more specialized color options. For example, the draw= option sets the color used for drawing, but does not modify the color used for filling. These color options are documented where the path action they influence is described.

15.3 Drawing a Path

You can draw a path using the following option:

$/\text{tikz/draw=}\langle color \rangle$

(default is scope's color setting)

Causes the path to be drawn. "Drawing" (also known as "stroking") can be thought of as picking up a pen and moving it along the path, thereby leaving "ink" on the canvas.

There are numerous parameters that influence how a line is drawn, like the thickness or the dash pattern. These options are explained below.

If the optional $\langle color \rangle$ argument is given, drawing is done using the given $\langle color \rangle$. This color can be different from the current filling color, which allows you to draw and fill a path with different colors. If no $\langle color \rangle$ argument is given, the last usage of the color= option is used.

If the special color name **none** is given, this option causes drawing to be "switched off." This is useful if a style has previously switched on drawing and you locally wish to undo this effect.

Although this option is normally used on paths to indicate that the path should be drawn, it also makes sense to use the option with a {scope} or {tikzpicture} environment. However, this will *not* cause all path to drawn. Instead, this just sets the $\langle color \rangle$ to be used for drawing paths inside the environment.

⁸ConTEXt users should be aware that \definecolor has a different meaning in ConTEXt. There is a low-level equivalent named \pgfutil@definecolor which can be used instead.

```
\begin{tikzpicture}
  \path[draw=red] (0,0) -- (1,1) -- (2,1) circle (10pt);
\end{tikzpicture}
```

The following subsections list the different options that influence how a path is drawn. All of these options only have an effect if the draw option is given (directly or indirectly).

15.3.1 Graphic Parameters: Line Width, Line Cap, and Line Join

/tikz/line width=⟨dimension⟩

(no default, initially 0.4pt)

(style, no value)

Specifies the line width. Note the space.

\tikz \draw[line width=5pt] (0,0) -- (1cm,1.5ex);

There are a number of predefined styles that provide more "natural" ways of setting the line width. You can also redefine these styles.

/tikz/ultra thin (style, no value)

Sets the line width to 0.1pt.

\tikz \draw[ultra thin] (0,0) -- (1cm,1.5ex);

/tikz/very thin (style, no value)

Sets the line width to 0.2pt.

\tikz \draw[very thin] (0,0) -- (1cm,1.5ex);

/tikz/thin (style, no value)

Sets the line width to 0.4pt.

\tikz \draw[thin] (0,0) -- (1cm,1.5ex);

/tikz/semithick (style, no value)

\tikz \draw[semithick] (0,0) -- (1cm,1.5ex);

Sets the line width to $0.6 \mathrm{pt}$.

Sets the line width to 0.8pt.

/tikz/thick

\tikz \draw[thick] (0,0) -- (1cm,1.5ex);

/tikz/very thick (style, no value)

Sets the line width to 1.2pt.

\tikz \draw[very thick] (0,0) -- (1cm,1.5ex);

/tikz/ultra thick (style, no value)

Sets the line width to 1.6pt.

\tikz \draw[ultra thick] (0,0) -- (1cm,1.5ex);

/tikz/line cap= $\langle type \rangle$

(no default, initially butt)

Specifies how lines "end." Permissible $\langle type \rangle$ are round, rect, and butt. They have the following effects:

```
\begin{tikzpicture}
\begin{scope}[line width=10pt]
    \draw[line cap=rect] (0,0) -- (1,0);
    \draw[line cap=butt] (0,.5) -- (1,.5);
    \draw[line cap=round] (0,1) -- (1,1);
    \end{scope}
    \draw[white,line width=1pt]
    (0,0) -- (1,0) (0,.5) -- (1,.5) (0,1) -- (1,1);
    \end{tikzpicture}
```

/tikz/line join= $\langle type \rangle$

(no default, initially miter)

Specifies how lines "join." Permissible $\langle type \rangle$ are round, bevel, and miter. They have the following effects:

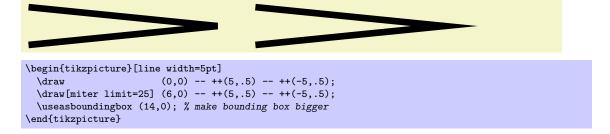


```
\begin{tikzpicture}[line width=10pt]
\draw[line join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
\draw[line join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
\draw[line join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
\useasboundingbox (0,1.5); % enlarge bounding box
\end{tikzpicture}
```

/tikz/miter limit=\langle factor\rangle

(no default, initially 10)

When you use the miter join and there is a very sharp corner (a small angle), the miter join may protrude very far over the actual joining point. In this case, if it were to protrude by more than $\langle factor \rangle$ times the line width, the miter join is replaced by a bevel join.



15.3.2 Graphic Parameters: Dash Pattern

/tikz/dash pattern= $\langle dash \ pattern \rangle$

(no default)

Sets the dashing pattern. The syntax is the same as in METAFONT. For example following pattern on 2pt off 3pt on 4pt off 4pt means "draw 2pt, then leave out 3pt, then draw 4pt once more, then leave out 4pt again, repeat".

```
\begin{tikzpicture}[dash pattern=on 2pt off 3pt on 4pt off 4pt]
\draw (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

/tikz/dash phase= $\langle dash \ phase \rangle$

(no default, initially Opt)

Shifts the start of the dash pattern by $\langle phase \rangle$.

As for the line thickness, some predefined styles allow you to set the dashing conveniently.

/tikz/solid (style, no value) Shorthand for setting a solid line as "dash pattern." This is the default. \tikz \draw[solid] (0pt,0pt) -- (50pt,0pt); /tikz/dotted (style, no value) Shorthand for setting a dotted dash pattern. \tikz \draw[dotted] (0pt,0pt) -- (50pt,0pt); /tikz/densely dotted (style, no value) Shorthand for setting a densely dotted dash pattern. \tikz \draw[densely dotted] (0pt,0pt) -- (50pt,0pt); /tikz/loosely dotted (style, no value) Shorthand for setting a loosely dotted dash pattern. \tikz \draw[loosely dotted] (0pt,0pt) -- (50pt,0pt); /tikz/dashed (style, no value) Shorthand for setting a dashed dash pattern. \tikz \draw[dashed] (0pt,0pt) -- (50pt,0pt); (style, no value) /tikz/densely dashed Shorthand for setting a densely dashed dash pattern. \tikz \draw[densely dashed] (Opt,Opt) -- (50pt,Opt); /tikz/loosely dashed (style, no value) Shorthand for setting a loosely dashed dash pattern. ----- \tikz \draw[loosely dashed] (0pt,0pt) -- (50pt,0pt); /tikz/dashdotted (style, no value) Shorthand for setting a dashed and dotted dash pattern. \tikz \draw[dashdotted] (0pt,0pt) -- (50pt,0pt); /tikz/densely dashdotted (style, no value) Shorthand for setting a densely dashed and dotted dash pattern. \tikz \draw[densely dashdotted] (0pt,0pt) -- (50pt,0pt); /tikz/loosely dashdotted (style, no value) Shorthand for setting a loosely dashed and dotted dash pattern. \tikz \draw[loosely dashdotted] (0pt,0pt) -- (50pt,0pt); /tikz/dashdotdotted (style, no value) Shorthand for setting a dashed and dotted dash pattern with more dots. \tikz \draw[dashdotdotted] (0pt,0pt) -- (50pt,0pt);

```
/tikz/densely dashdotdotted
```

(style, no value)

Shorthand for setting a densely dashed and dotted dash pattern with more dots.

```
\tikz \draw[densely dashdotdotted] (0pt,0pt) -- (50pt,0pt);
```

/tikz/loosely dashdotdotted

(style, no value)

Shorthand for setting a loosely dashed and dotted dash pattern with more dots.

```
\tikz \draw[loosely dashdotdotted] (0pt,0pt) -- (50pt,0pt);
```

15.3.3 Graphic Parameters: Draw Opacity

When a line is drawn, it will normally "obscure" everything behind it as if you has used perfectly opaque ink. It is also possible to ask TikZ to use an ink that is a little bit (or a big bit) transparent using the draw opacity option. This is explained in Section 21 on transparency in more detail.

15.3.4 Graphic Parameters: Arrow Tips

When you draw a line, you can add arrow tips at the ends. It is only possible to add one arrow tip at the start and one at the end. If the path consists of several segments, only the last segment gets arrow tips. The behavior for paths that are closed is not specified and may change in the future.

```
\ttikz/arrows=\tstart arrow kind\t-\tend arrow kind\t (no default)
```

This option sets the start and end arrow tips (an empty value as in -> indicates that no arrow tip should be drawn at the start).

Note: Since the arrow option is so often used, you can leave out the text arrows=. What happens is that every option that contains a - is interpreted as an arrow specification.

```
\text{\login{tikzpicture}} \draw[->] (0,0) -- (1,0); \draw[0-stealth] (0,0.3) -- (1,0.3); \end{tikzpicture}
```

The permissible values are all predefined arrow tips, though you can also define new arrow tip kinds as explained in Section 82. This is often necessary to obtain "double" arrow tips and arrow tips that have a fixed size. You need to load the **arrows** library if you need arrow tips other than the default ones, see Section 24.

One arrow tip kind is special: > (and all arrow tip kinds containing the arrow tip kind such as << or >|). This arrow tip type is not fixed. Rather, you can redefine it using the >= option, see below.

Example: You can also combine arrow tip types as in

```
\begin{tikzpicture}[thick]
\draw[to reversed-to] (0,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
\draw[[-latex reversed] (1,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
\draw[latex-)] (2,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
\useasboundingbox (-.1,-.1) rectangle (3.1,1.1); % make bounding box bigger
\end{tikzpicture}
```

```
/tikz/>=\langle end arrow kind\rangle
```

(no default)

This option can be used to redefine the "standard" arrow tip >. The idea is that different people have different ideas what arrow tip kind should normally be used. I prefer the arrow tip of T_EX 's \to command (which is used in things like $f: A \to B$). Other people will prefer LATEX's standard arrow tip, which looks like this: \nearrow . Since the arrow tip kind > is certainly the most "natural" one to use, it is kept free of any predefined meaning. Instead, you can change it by saying >=to to set the "standard" arrow tip kind to T_EX 's arrow tip, whereas >=latex will set it to LATEX's arrow tip and >=stealth will use a PSTRICKS-like arrow tip.

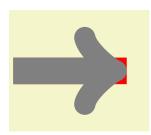
Apart from redefining the arrow tip kind > (and < for the start), this option also redefines the following arrow tip kinds: > and < as the swapped version of $\langle end \ arrow \ kind \rangle$, << and >> as doubled versions, >> and << as swapped doubled versions, and |< and >| as arrow tips ending with a vertical bar.

```
\begin{tikzpicture}[scale=2]
  \begin{scope}[>=latex]
    \draw[->]
                  (0pt,6ex) -- (1cm,6ex);
                 (Opt,5ex) -- (1cm,5ex);
    \draw[>->>]
    \draw[|<->|] (Opt,4ex) -- (1cm,4ex);
  \end{scope}
  \begin{scope}[>=diamond]
    \draw[->]
                  (0pt, 2ex) -- (1cm, 2ex);
    \draw[>->>]
                 (Opt,1ex) -- (1cm,1ex);
    \draw[|<->|] (Opt, Oex) -- (1cm, Oex);
  \end{scope}
\end{tikzpicture}
```

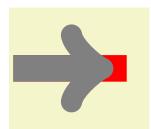
/tikz/shorten $>=\langle dimension \rangle$

(no default, initially Opt)

This option will shorten the end of lines by the given $\langle dimension \rangle$. If you specify an arrow tip, lines are already shortened a bit such that the arrow tip touches the specified endpoint and does not "protrude over" this point. Here is an example:



The shorten > option allows you to shorten the end on the line *additionally* by the given distance. This option can also be useful if you have not specified an arrow tip at all.



```
/tikz/shorten <=⟨dimension⟩
```

(no default)

Works like shorten >, but for the start.

15.3.5 Graphic Parameters: Double Lines and Bordered Lines

```
/tikz/double=\langle core \ color \rangle
```

(default white)

This option causes "two" lines to be drawn instead of a single one. However, this is not what really happens. In reality, the path is drawn twice. First, with the normal drawing color, secondly with the $\langle core\ color \rangle$, which is normally white. Upon the second drawing, the line width is reduced. The net effect is that it appears as if two lines had been drawn and this works well even with complicated, curved paths:



```
\tikz \draw[double]
plot[smooth cycle] coordinates{(0,0) (1,1) (1,0) (0,1)};
```

You can also use the doubling option to create an effect in which a line seems to have a certain "border":



```
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
  \draw[draw=white,double=red,very thick] (0,1) -- (1,0);
\end{tikzpicture}
```

/tikz/double distance= $\langle dimension \rangle$

(no default, initially 0.6pt)

Sets the distance the "two" lines are spaced apart. In reality, this is the thickness of the line that is used to draw the path for the second time. The thickness of the *first* time the path is drawn is twice the normal line width plus the given $\langle dimension \rangle$. As a side-effect, this option "selects" the double option.

/tikz/double distance between line centers= $\langle dimension \rangle$

(no default)

This option works like double distance, only the distance is not the distance between (inner) borders of the two main lines, but between their centers. Thus, the thickness the *first* time the path is drawn is the normal line width plus the given $\langle dimension \rangle$, while the line width of the *second* line that is drawn is $\langle dimension \rangle$ minus the normal line width. As a side-effect, this option "selects" the double option.

/tikz/double equal sign distance

(style, no value)

This style selects a double line distance such that it corresponds to the distance of the two lines in an equal sign.

```
Huge $=\implies$\tikz[baseline,double equal sign distance]
\draw[double,thick,-implies](0,0.55ex) --++(3ex,0);

= ⇒ ⇒ \normalsize $=\implies$\tikz[baseline,double equal sign distance]
\draw[double,-implies](0,0.6ex) --++(3ex,0);

| = ⇒ ⇒ \tiny $=\implies$\tikz[baseline,double equal sign distance]
\draw[double,very thin,-implies](0,0.5ex) -- ++(3ex,0);
```

15.4 Filling a Path

To fill a path, use the following option:

```
/tikz/fill = \langle color \rangle (default is scope's color setting)
```

This option causes the path to be filled. All unclosed parts of the path are first closed, if necessary. Then, the area enclosed by the path is filled with the current filling color, which is either the last color set using the general color= option or the optional color $\langle color \rangle$. For self-intersection paths and for paths consisting of several closed areas, the "enclosed area" is somewhat complicated to define and two different definitions exist, namely the nonzero winding number rule and the even odd rule, see the explanation of these options, below.

Just as for the draw option, setting $\langle color \rangle$ to none disables filling locally.



```
\begin{tikzpicture}
\fill (0,0) -- (1,1) -- (2,1);
\fill (4,0) circle (.5cm) (4.5,0) circle (.5cm);
\fill[even odd rule] (6,0) circle (.5cm) (6.5,0) circle (.5cm);
\fill (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

If the fill option is used together with the draw option (either because both are given as options or because a \filldraw command is used), the path is filled *first*, then the path is drawn *second*. This is especially useful if different colors are selected for drawing and for filling. Even if the same color is used, there is a difference between this command and a plain fill: A "filldrawn" area will be slightly larger than a filled area because of the thickness of the "pen."



```
\begin{tikzpicture}[fill=examplefill,line width=5pt]
  \filldraw (0,0) -- (1,1) -- (2,1);
  \filldraw (4,0) circle (.5cm) (4.5,0) circle (.5cm);
  \filldraw[even odd rule] (6,0) circle (.5cm) (6.5,0) circle (.5cm);
  \filldraw (8,0) -- (9,1) -- (10,0) circle (.5cm);
  \end{tikzpicture}
```

15.4.1 Graphic Parameters: Fill Pattern

Instead of filling a path with a single solid color, it is also possible to fill it with a *tiling pattern*. Imagine a small tile that contains a simple picture like a star. Then these tiles are (conceptually) repeated infinitely in all directions, but clipped against the path.

Tiling patterns come in two variants: *inherently colored patterns* and *form-only patterns*. An inherently colored pattern is, say, a red star with a black border and will always look like this. A form-only pattern may have a different color each time it is used, only the form of the pattern will stay the same. As such, form-only patterns do not have any colors of their own, but when it is used the current *pattern color* is used as its color.

Patterns are not overly flexible. In particular, it is not possible to change the size or orientation of a pattern without declaring a new pattern. For complicated cases, it may be easier to use two nested \foreach statements to simulate a pattern, but patterns are rendered *much* more quickly than simulated ones.

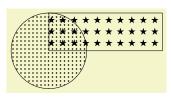
/tikz/pattern= $\langle name \rangle$

(default is scope's pattern)

This option causes the path to be filled with a pattern. If the $\langle name \rangle$ is given, this pattern is used, otherwise the pattern set in the enclosing scope is used. As for the draw and fill options, setting $\langle name \rangle$ to none disables filling locally.

The pattern works like a fill color. In particular, setting a new fill color will fill the path with a solid color once more.

Strangely, no $\langle name \rangle$ s are permissible by default. You need to load for instance the patterns library, see Section 42, to install predefined patterns.

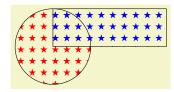


```
\begin{tikzpicture}
  \draw[pattern=dots] (0,0) circle (1cm);
  \draw[pattern=fivepointed stars] (0,0) rectangle (3,1);
  \end{tikzpicture}
```

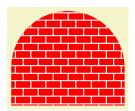
/tikz/pattern color= $\langle color \rangle$

(no default)

This option is used to set the color to be used for form-only patterns. This option has no effect on inherently colored patterns.



```
\begin{tikzpicture}
  \draw[pattern color=red,pattern=fivepointed stars] (0,0) circle (1cm);
  \draw[pattern color=blue,pattern=fivepointed stars] (0,0) rectangle (3,1);
  \end{tikzpicture}
```



```
\begin{tikzpicture}
\def\mypath{(0,0) -- +(0,1) arc (180:0:1.5cm) -- +(0,-1)}
\fill [red] \mypath;
\pattern[pattern color=white,pattern=bricks] \mypath;
\end{tikzpicture}
```

15.4.2 Graphic Parameters: Interior Rules

The following two options can be used to decide how interior points should be determined:

/tikz/nonzero rule (no value)

If this rule is used (which is the default), the following method is used to determine whether a given point is "inside" the path: From the point, shoot a ray in some direction towards infinity (the direction is chosen such that no strange borderline cases occur). Then the ray may hit the path. Whenever it hits the path, we increase or decrease a counter, which is initially zero. If the ray hits the path as the path goes "from left to right" (relative to the ray), the counter is increased, otherwise it is decreased. Then, at the end, we check whether the counter is nonzero (hence the name). If so, the point is deemed to lie "inside," otherwise it is "outside." Sounds complicated? It is.

```
crossings: -1 + 1 = 0

crossings: 1 + 1 = 2
```

```
\begin{tikzpicture}
  \filldraw[fill=examplefill]
  % Clockwise rectangle
  (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
  % Counter-clockwise rectangle
  (0.25,0.25) -- (0.75,0.25) -- (0.75,0.75) -- (0.25,0.75) -- cycle;
  draw[->] (0,1) -- (.4,1);
  \draw[->] (0.75,0.75) -- (0.3,.75);
  \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $-1+1 = 0$};
  \begin{scope}[yshift=-3cm]
    \filldraw[fill=examplefill]
    % Clockwise rectangle
    (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
    % Clockwise rectangle
    (0.25,0.25) -- (0.25,0.75) -- (0.75,0.75) -- (0.75,0.25) -- cycle;
    \draw[->] (0,1) -- (.4,1);
    \draw[->] (0.25,0.75) -- (0.4,.75);
    \frac{-}{0.5,0.5} -- +(0,1) \text{ node[above] {crossings: $1+1 = 2$};}
  \end{scope}
\end{tikzpicture}
```

/tikz/even odd rule (no value)

This option causes a different method to be used for determining the inside and outside of paths. While it is less flexible, it turns out to be more intuitive.

With this method, we also shoot rays from the point for which we wish to determine whether it is inside or outside the filling area. However, this time we only count how often we "hit" the path and declare the point to be "inside" if the number of hits is odd.

Using the even-odd rule, it is easy to "drill holes" into a path.

```
crossings: 1+1=2
```

```
\begin{tikzpicture}
  \filldraw[fill=examplefill,even odd rule]
  (0,0) rectangle (1,1) (0.5,0.5) circle (0.4cm);
  \draw[->] (0.5,0.5) -- +(0,1) [above] node{crossings: $1+1 = 2$};
  \end{tikzpicture}
```

15.4.3 Graphic Parameters: Fill Opacity

Analogously to the draw opacity, you can also set the fill opacity. Please see Section 21 for more details.

15.5 Generalized Filling: Using Arbitrary Pictures to Fill a Path

Sometimes you wish to "fill" a path with something even more complicated than a pattern, let alone a single color. For instance, you might wish to use an image to fill the path or some other, complicated drawing. In principle, this effect can be achieved by first using the path for clipping and then, subsequently, drawing the desired image or picture. However, there is an option that makes this process much easier:

```
/tikz/path picture=\langle code \rangle
```

(no default)

When this option is given on a path and when the $\langle code \rangle$ is not empty, the following happens: After all other "filling" operations are done with the path, which are caused by the options fill, pattern and shade, a local scope is opened and the path is temporarily installed as a clipping path. Then, the $\langle code \rangle$ is executed, which can now draw something. Then, the local scope ends and, possibly, the path is stroked, provided the draw option has been given.

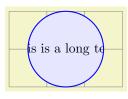
As with other keys like fill or draw this option needs to be given on a path, setting the path picture outside a path has no effect (the path picture is cleared at the beginning of each path).

The $\langle code \rangle$ can be any normal TikZ code like \draw ... or \node As always, when you include an external graphic, you need to put it inside a \node.

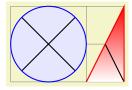
Note that no special actions are taken to transform the origin in any way. This means that the coordinate (0,0) is still where is was when the path was being constructed and not – as one might expect – at the lower left corner of the path. However, you can use the following special node to access the size of the path:

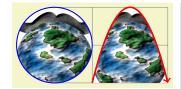
Predefined node path picture bounding box

This node is of shape rectangle. Its size and position are those of current path bounding box just before the $\langle code \rangle$ of the path picture started to be executed. The $\langle code \rangle$ can construct its own paths, so accessing the current path bounding box inside the $\langle code \rangle$ yields the bounding box of any path that is currently being constructed inside the $\langle code \rangle$.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \filldraw [fill=blue!10,draw=blue,thick] (1.5,1) circle (1)
  [path picture={
      \node at (path picture bounding box.center) {
      This is a long text.
      };}
  ];
  \end{tikzpicture}
```





```
\begin{tikzpicture}[path image/.style={
    path picture={
        \node at (path picture bounding box.center) {
            \includegraphics[height=3cm] {#1}
        };}}]
\draw        [help lines] (0,0) grid (3,2);

\draw [path image=brave-gnu-world-logo,draw=blue,thick]
            (0,1) circle (1);
\draw [path image=brave-gnu-world-logo,draw=red,very thick,->]
            (1,0) parabola[parabola height=2cm] (3,0);

\end{tikzpicture}
```

15.6 Shading a Path

You can shade a path using the **shade** option. A shading is like a filling, only the shading changes its color smoothly from one color to another.

/tikz/shade (no value)

Causes the path to be shaded using the currently selected shading (more on this later). If this option is used together with the draw option, then the path is first shaded, then drawn.

It is not an error to use this option together with the fill option, but it makes no sense.

```
\tikz \shade (0,0) circle (1ex);

\tikz \shadedraw (0,0) circle (1ex);
```

For some shadings it is not really clear how they can "fill" the path. For example, the ball shading normally looks like this: •. How is this supposed to shade a rectangle? Or a triangle?

To solve this problem, the predefined shadings like ball or axis fill a large rectangle completely in a sensible way. Then, when the shading is used to "shade" a path, what actually happens is that the path is temporarily used for clipping and then the rectangular shading is drawn, scaled and shifted such that all parts of the path are filled.

The default shading is a smooth transition from gray to white and from top to bottom. However, other shadings are also possible, for example a shading that will sweep a color from the center to the corners outward. To choose the shading, you can use the shading= option, which will also automatically invoke the shade option. Note that this does *not* change the shading color, only the way the colors sweep. For changing the colors, other options are needed, which are explained below.

```
/\text{tikz/shading} = \langle name \rangle (no default)
```

This selects a shading named $\langle name \rangle$. The following shadings are predefined: axis, radial, and ball.

```
\tikz \shadedraw [shading=axis] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=radial] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=ball] (0,0) circle (.5cm);
```

The shadings as well as additional shadings are described in more detail in Section 47.

To change the color of a shading, special options are needed like left color, which sets the color of an axis shading from left to right. These options implicitly also select the correct shading type, see the following example

```
\tikz \shadedraw [left color=red,right color=blue]
  (0,0) rectangle (1,1);
```

For a complete list of the possible options see Section 47 once more.

```
/tikz/shading angle=(degrees)
```

(no default, initially 0)

This option rotates the shading (not the path!) by the given angle. For example, we can turn a top-to-bottom axis shading into a left-to-right shading by rotating it by 90°.



```
\tikz \shadedraw [shading=axis,shading angle=90] (0,0) rectangle (1,1);
```

You can also define new shading types yourself. However, for this, you need to use the basic layer directly, which is, well, more basic and harder to use. Details on how to create a shading appropriate for filling paths are given in Section 91.3.

15.7 Establishing a Bounding Box

PGF is reasonably good at keeping track of the size of your picture and reserving just the right amount of space for it in the main document. However, in some cases you may want to say things like "do not count this for the picture size" or "the picture is actually a little large." For this you can use the option use as bounding box or the command \useasboundingbox, which is just a shorthand for \path[use as bounding box].

/tikz/use as bounding box

(no value

Normally, when this option is given on a path, the bounding box of the present path is used to determine the size of the picture and the size of all *subsequent* paths are ignored. However, if there were previous path operations that have already established a larger bounding box, it will not be made smaller by this operation (consider the \pgfresetboundingbox command to reset the previous bounding box).

In a sense, use as bounding box has the same effect as clipping all subsequent drawing against the current path—without actually doing the clipping, only making PGF treat everything as if it were clipped.

The first application of this option is to have a {tikzpicture} overlap with the main text:

```
Left of pieture right of picture.

Left of picture\begin{tikzpicture} \draw[use as bounding box] (2,0) rectangle (3,1); \draw (1,0) -- (4,.75); \end{tikzpicture}right of picture.
```

In a second application this option can be used to get better control over the white space around the picture:

```
Left of picture

Left of picture

begin{tikzpicture}

\useasboundingbox (0,0) rectangle (3,1);

\fill (.75,.25) circle (.5cm);

\end{tikzpicture}

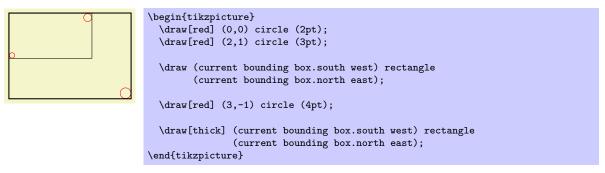
right of picture.
```

Note: If this option is used on a path inside a TeX group (scope), the effect "lasts" only till the end of the scope. Again, this behavior is the same as for clipping.

Consider using \useasboundingbox together with \pgfresetboundingbox in order to replace the bounding box with a new one.

There is a node that allows you to get the size of the current bounding box. The current bounding box node has the rectangle shape and its size is always the size of the current bounding box.

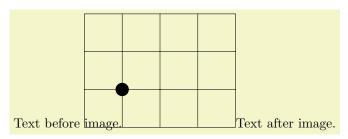
Similarly, the current path bounding box node has the rectangle shape and the size of the bounding box of the current path.



Occasionally, you may want to align multiple tikzpicture environments horizontally and/or vertically at some prescribed position. The vertical alignment can be realized by means of the baseline option since TeX supports the concept of box depth natively. For horizontal alignment, things are slightly more involved. The following approach is realized by means of negative \hspaces before and/or after the picture, thereby removing parts of the picture. However, the actual amount of negative horizontal space is provided by means of image coordinates using the trim left and trim right keys:

```
/tikz/trim left=\langle dimension \ or \ coordinate \ or \ default \rangle (default Opt)
```

The trim left key tells PGF to discard everything which is left of the provided $\langle dimension \ or \ coordinate \rangle$. Here, $\langle dimension \rangle$ is a single x coordinate of the picture and $\langle coordinate \rangle$ is a point with x and y coordinates (but only its x coordinate will be used). The effect is the same as if you issue $\hspace{-s}$ where s is the difference of the picture's bounding box lower left x coordinate and the x coordinate specified as $\langle dimension \ or \ coordinate \rangle$:



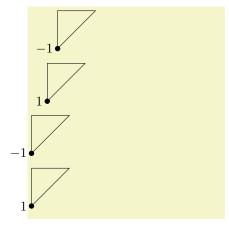
```
Text before image.%

\begin{tikzpicture}[trim left]
\draw (-1,-1) grid (3,2);
\fill (0,0) circle (5pt);
\end{tikzpicture}%

Text after image.
```

Since trim left uses the default trim left=0pt, everything left of x = 0 is removed from the bounding box.

The following example has once the relative long label -1 and once the shorter label 1. Horizontal alignment is established with trim left:



```
\begin{tikzpicture}
\draw (0,1) -- (0,0) -- (1,1) -- cycle;
\fill (0,0) circle (2pt);
\node[left] at (0,0) {\$-1\$};
\end{tikzpicture}
\begin{tikzpicture}
\draw (0,1) -- (0,0) -- (1,1) -- cycle;
\fill (0,0) circle (2pt);
\node[left] at (0,0) {$1$};
\end{tikzpicture}
\par
\begin{tikzpicture}[trim left]
\draw (0,1) -- (0,0) -- (1,1) -- cycle;
\fill (0,0) circle (2pt);
\node[left] at (0,0) {\$-1\$};
\end{tikzpicture}
\par
\begin{tikzpicture}[trim left]
\draw (0,1) -- (0,0) -- (1,1) -- cycle;
\fill (0,0) circle (2pt);
\node[left] at (0,0) {$1$};
\end{tikzpicture}
```

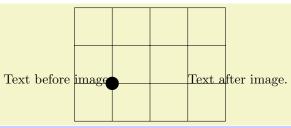
Use trim left=default to reset the value.

```
/tikz/trim right=\langle dimension or coordinate or default \rangle
```

(no default)

This key is similar to trim left: it discards everything which is right of the provided $\langle dimension \ or \ coordinate \rangle$. As for trim left, $\langle dimension \rangle$ denotes a single x coordinate of the picture and $\langle coordinate \rangle$ a coordinate with x and y value (although only its x component will be used).

We use the same example from above and add trim right:



```
Text before image.%

\begin{tikzpicture}[trim left, trim right=2cm, baseline]
\draw (-1,-1) grid (3,2);
\fill (0,0) circle (5pt);
\end{tikzpicture}%

Text after image.
```

In addition to trim left=0pt, we also discard everything which is right of x=2cm. Furthermore, the baseline key supports vertical alignment as well (using the y=0cm baseline).

Use trim right=default to reset the value.

Note that baseline, trim left and trim right are currently the *only* supported way of truncated bounding boxes which are compatible with image externalization (see the external library for details).

```
/pgf/trim lowlevel=true|false
```

(no default, initially false)

This affects only the basic level image externalization: the initial configuration trim lowlevel=false stores the normal image, without trimming, and the trimming into a separate file. This allows reduced bounding boxes without clipping the rest away. The trim lowlevel=true information causes the image externalization to store the trimmed image, possibly resulting in clipping.

15.8 Clipping and Fading (Soft Clipping)

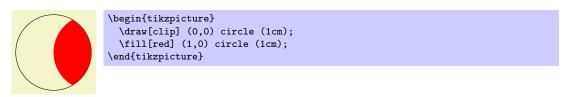
Clipping path means that all painting on the page is restricted to a certain area. This area need not be rectangular, rather an arbitrary path can be used to specify this area. The clip option, explained below, is used to specify the region that is to be used for clipping.

A fading (a term that I propose, fadings are commonly known as soft masks, transparency masks, opacity masks or soft clips) is similar to clipping, but a fading allows parts of the picture to be only "half clipped." This means that a fading can specify that newly painted pixels should be partly transparent. The specification and handling of fadings is a bit complex and it is detailed in Section 21, which is devoted to transparency in general.

/tikz/clip (no value)

This option causes all subsequent drawings to be clipped against the current path and the size of subsequent paths will not be important for the picture size. If you clip against a self-intersecting path, the even-odd rule or the nonzero winding number rule is used to determine whether a point is inside or outside the clipping region.

The clipping path is a graphic state parameter, so it will be reset at the end of the current scope. Multiple clippings accumulate, that is, clipping is always done against the intersection of all clipping areas that have been specified inside the current scopes. The only way of enlarging the clipping area is to end a {scope}.



It is usually a *very* good idea to apply the clip option only to the first path command in a scope. If you "only wish to clip" and do not wish to draw anything, you can use the \clip command, which is a shorthand for \path[clip].

```
\begin{tikzpicture}
  \clip (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
  \end{tikzpicture}
```

To keep clipping local, use {scope} environments as in the following example:



```
\begin{tikzpicture}
\draw (0,0) -- ( 0:1cm);
\draw (0,0) -- (10:1cm);
\draw (0,0) -- (20:1cm);
\draw (0,0) -- (30:1cm);
\begin{scope}[fill=red]
  \fill[clip] (0.2,0.2) rectangle (0.5,0.5);

\draw (0,0) -- (40:1cm);
  \draw (0,0) -- (50:1cm);
  \draw (0,0) -- (60:1cm);
  \draw (0,0) -- (60:1cm);
\draw (0,0) -- (70:1cm);
\draw (0,0) -- (80:1cm);
\draw (0,0) -- (90:1cm);
```

There is a slightly annoying catch: You cannot specify certain graphic options for the command used for clipping. For example, in the above code we could not have moved the fill=red to the \fill command. The reasons for this have to do with the internals of the PDF specification. You do not want to know the details. It is best simply not to specify any options for these commands.

15.9 Doing Multiple Actions on a Path

If more than one of the basic actions like drawing, clipping and filling are requested, they are automatically applied in a sensible order: First, a path is filled, then drawn, and then clipped (although it took Apple two mayor revisions of their operating system to get this right...). Sometimes, however, you need finer control over what is done with a path. For instance, you might wish to first fill a path with a color, then repaint the

path with a pattern and then repaint it with yet another pattern. In such cases you can use the following two options:

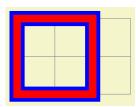
```
\texttt{/tikz/preactions=}\langle options \rangle
```

(no default)

This option can be given to a \path command (or to derived commands like \draw which internally call \path). Similarly to options like draw, this option only has an effect when given to a \path or as part of the options of a node; as an option to a {scope} it has no effect.

When this option is used on a **\path**, the effect is the following: When the path has been completely constructed and is about to be used, a scope is created. Inside this scope, the path is used but not with the original path options, but with $\langle options \rangle$ instead. Then, the path is used in the usual manner. In other words, the path is used twice: Once with $\langle options \rangle$ in force and then again with the normal path options in force.

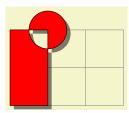
Here is an example in which the path consists of a rectangle. The main action is to draw this path in red (which is why we see a red rectangle). However, the preaction is to draw the path in blue, which is why we see a blue rectangle behind the red rectangle.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\draw
  [preaction={draw,line width=4mm,blue}]
  [line width=2mm,red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

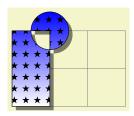
Note that when the preactions are preformed, then the path is already "finished." In particular, applying a coordinate transformation to the path has no effect. By comparison, applying a canvas transformation does have an effect. Let us use this to add a "shadow" to a path. For this, we use the preaction to fill the path in gray, shifted a bit to the right and down:



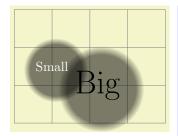
Naturally, you would normally create a style **shadow** that contains the above code. The shadow library, see Section 48, contains predefined shadows of this kind.

It is possible to use the **preaction** option multiple times. In this case, for each use of the **preaction** option, the path is used again (thus, the $\langle options \rangle$ do not accumulate in a single usage of the path). The path is used in the order of **preaction** options given.

In the following example, we use one **preaction** to add a shadow and another to provide a shading, while the main action is to use a pattern.



A complicated application is shown in the following example, where the path is used several times with different fadings and shadings to create a special visual effect:



```
\begin{tikzpicture}
   % Define an interesting style
   button/.style={
     % First preaction: Fuzzy shadow
     preaction={fill=black,path fading=circle with fuzzy edge 20 percent,
                opacity=.5,transform canvas={xshift=1mm,yshift=-1mm}},
     \% Second preaction: Background pattern
     preaction={pattern=#1,
                 path fading=circle with fuzzy edge 15 percent},
     % Third preaction: Make background shiny
     preaction={top color=white,
                 bottom color=black!50,
                 shading angle=45,
                 path fading=circle with fuzzy edge 15 percent,
                 opacity=0.2},
     % Fourth preaction: Make edge especially shiny
     preaction={path fading=fuzzy ring 15 percent,
                 top color=black!5,
                 bottom color=black!80,
                 shading angle=45},
     inner sep=2ex
   },
   button/.default=horizontal lines light blue,
 ]
 \draw [help lines] (0,0) grid (4,3);
 \node [button] at (2.2,1) {\Huge Big};
 \node [button=crosshatch dots light steel blue,
        text=white] at (1,1.5) {Small};
\end{tikzpicture}
```

/tikz/postaction=\langle options\rangle

(no default)

The postactions work in the same way as the preactions, only they are applied *after* the main action has been taken. Like preactions, multiple **postaction** options may be given to a \path command, in which case the path is reused several times, each time with a different set of options in force.

If both pre- and postactions are specified, then the preactions are taken first, then the main action, and then the post actions.

In the first example, we use a postaction to draw the path, after it has already been drawn:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\draw
  [postaction={draw,line width=2mm,blue}]
  [line width=4mm,red,fill=white] (0,0) rectangle (2,2);
\end{tikzpicture}
```

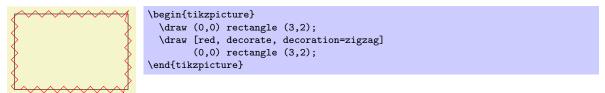
In another example, we use a postaction to "colorize" a path:



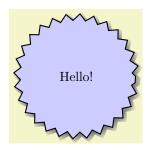
15.10 Decorating and Morphing a Path

Before a path is used, it is possible to first "decorate" and/or "morph" it. Morphing means that the path is replaced by another path that is slightly varied. Such morphings are a special case of the more general

"decorations" described in detail in Section 22. For instance, in the following example the path is drawn twice: Once normally and then in a morphed (=decorated) manner.



Naturally, we could have combined this into a single command using pre- or postaction. It is also possible to deform shapes:



16 Nodes and Edges

16.1 Overview

In the present section, the usage of nodes in TikZ is explained. A node is typically a rectangle or circle or another simple shape with some text on it.

Nodes are added to paths using the special path operation node. Nodes are not part of the path itself. Rather, they are added to the picture after the path has been drawn.

In Section 16.2 the basic syntax of the node operation is explained, followed in Section 16.3 by the syntax for multi-part nodes, which are nodes that contain several different text parts. After this, the different options for the text in nodes are explained. In Section 16.5 the concept of *anchors* is introduced along with their usage. In Section 16.7 the different ways transformations affect nodes are studied. Sections 16.8 and 16.9 are about placing nodes on or next to straight lines and curves. In Section 16.11 it is explained how a node can be used as a "pseudo-coordinate." Section 16.12 introduces the edge operation, which works similar to the to operation and also similar to the node operation.

16.2 Nodes and Their Shapes

In the simplest case, a node is just some text that is placed at some coordinate. However, a node can also have a border drawn around it or have a more complex background and foreground. Indeed, some nodes do not have a text at all, but consist solely of the background. You can name nodes so that you can reference their coordinates later in the same picture or, if certain precautions are taken as explained in Section 16.13, also in different pictures.

There are no special T_EX commands for adding a node to a picture; rather, there is path operation called node for this. Nodes are created whenever TikZ encounters node or coordinate at a point on a path where it would expect a normal path operation (like -- (1,1) or sin (1,1)). It is also possible to give node specifications *inside* certain path operations as explained later.

The node operation is typically followed by some options, which apply only to the node. Then, you can optionally *name* the node by providing a name in round braces. Lastly, for the node operation you must provide some label text for the node in curly braces, while for the coordinate operation you may not. The node is placed at the current position of the path *after the path has been drawn*. Thus, all nodes are drawn "on top" of the path and retained until the path is complete. If there are several nodes on a path, they are drawn on top of the path in the order they are encountered.

```
third node

(0,0) node {first node}

-- (1,1) node {second node}

-- (0,2) node {third node};
```

The syntax for specifying nodes is the following:

```
\path ... node[\langle options \rangle](\langle name \rangle) at(\langle coordinate \rangle) \{\langle text \rangle\} ...;
```

The effect of at is to place the node at the coordinate given after at and not, as would normally be the case, at the last position. The at syntax is not available when a node is given inside a path operation (it would not make any sense there).

The $(\langle name \rangle)$ is a name for later reference and it is optional. You may also add the option name= $\langle name \rangle$ to the $\langle option \rangle$ list; it has the same effect.

```
/tikz/name=⟨node name⟩ (no default)
```

Assigns a name to the node for later reference. Since this is a "high-level" name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a node. Thus, you can name a node just 1 or perhaps start of chart or even y_1. Your node name should not contain any punctuation like a dot, a comma, or a colon since these are used to detect what kind of coordinate you mean when you reference a node.

```
\text{/tikz/alias} = \langle another \ node \ name \rangle (no default)
```

This option allows you to provide another name for the node. Giving this option multiple times will allow you to access the node via several aliases. Using the late options option, you can also assign an alias name to a node at a later point.

```
\text{/tikz/at} = \langle coordinate \rangle (no default)
```

This is another way of specifying the at coordinate. Note that, typically, you will have to enclose the $\langle coordinate \rangle$ in curly braces so that a comma inside the $\langle coordinate \rangle$ does not confuse T_FX.

The $\langle options \rangle$ is an optional list of options that apply only to the node and have no effect outside. The other way round, most "outside" options also apply to the node, but not all. For example, the "outside" rotation does not apply to nodes (unless some special options are used, sigh). Also, the outside path action, like draw or fill, never applies to the node and must be given in the node (unless some special other options are used, deep sigh).

As mentioned before, we can add a border and even a background to a node:

```
third node

\tikz \fill[fill=examplefill]
(0,0) node {first node}
-- (1,1) node[draw] {second node}
-- (0,2) node[fill=red!20,draw,double,rounded corners] {third node};
```

The "border" is actually just a special case of a much more general mechanism. Each node has a certain *shape* which, by default, is a rectangle. However, we can also ask TikZ to use a circle shape instead or an ellipse shape (you have to include one of the **shapes.geometric** library for the latter shape):

```
third node
third node
third node
third node

third node

third node

third node

third node

third node

first node

third node

third node

first node

third node

third node

second node

first node

third node

thi
```

In the future, there might be much more complicated shapes available such as, say, a shape for a resistor or a shape for a UML class. Unfortunately, creating new shapes is a bit tricky and makes it necessary to use the basic layer directly. Life is hard.

To select the shape of a node, the following option is used:

```
\texttt{/tikz/shape=}\langle shape \ name \rangle (no default, initially rectangle)
```

Select the shape either of the current node or, when this option is not given inside a node but somewhere outside, the shape of all nodes in the current scope.

Since this option is used often, you can leave out the shape=. When TikZ encounters an option like circle that it does not know, it will, after everything else has failed, check whether this option is the name of some shape. If so, that shape is selected as if you had said shape= $\langle shape \ name \rangle$.

By default, the following shapes are available: rectangle, circle, coordinate, and, when the package pgflibraryshapes is loaded, also ellipse. Details of these shapes, like their anchors and size options, are discussed in Section 16.2.1.

The following styles influence how nodes are rendered:

```
/tikz/every node (style, initially empty)
```

This style is installed at the beginning of every node.

```
/tikz/every (shape) node
```

(style, initially empty)

These styles are installed at the beginning of a node of a given $\langle shape \rangle$. For example, every rectangle node is used for rectangle nodes, and so on.



```
\begin{tikzpicture}
  [every rectangle node/.style={draw},
   every circle node/.style={draw,double}]
  \draw (0,0) node[rectangle] {A} -- (1,1) node[circle] {B};
  \end{tikzpicture}
```

There is a special syntax for specifying "light-weight" nodes:

```
\path ... coordinate [\langle options \rangle] (\langle name \rangle) at (\langle coordinate \rangle) ...;
This has the same effect as \node [shape=coordinate] [\langle options \rangle] (\langle name \rangle) at (\langle coordinate \rangle) {}, where the at part may be omitted.
```

Since nodes are often the only path operation on paths, there are two special commands for creating paths containing only a node:

\node

Inside {tikzpicture} this is an abbreviation for \path node.

\coordinate

Inside {tikzpicture} this is an abbreviation for \path coordinate.

16.2.1 Predefined Shapes

PGF and TikZ define three shapes, by default:

- rectangle,
- circle, and
- coordinate.

By loading library packages, you can define more shapes like ellipses or diamonds; see Section 49 for the complete list of shapes.

The coordinate shape is handled in a special way by TikZ. When a node x whose shape is coordinate is used as a coordinate (x), this has the same effect as if you had said (x.center). None of the special "line shortening rules" apply in this case. This can be useful since, normally, the line shortening causes paths to be segmented and they cannot be used for filling. Here is an example that demonstrates the difference:

16.2.2 Common Options: Separations, Margins, Padding and Border Rotation

The exact behaviour of shapes differs, shapes defined for more special purposes (like a, say, transistor shape) will have even more custom behaviors. However, there are some options that apply to most shapes:

```
/pgf/inner sep=\langle dimension \rangle (no default, initially .3333em)
```

alias /tikz/inner sep

An additional (invisible) separation space of $\langle dimension \rangle$ will be added inside the shape, between the text and the shape's background path. The effect is as if you had added appropriate horizontal and vertical skips at the beginning and end of the text to make it a bit "larger."

For those familiar with CSS, this is the same as padding.

```
/pgf/inner xsep=\langle dimension \rangle
```

(no default, initially .3333em)

alias /tikz/inner xsep

Specifies the inner separation in the x-direction, only.

```
/pgf/inner ysep=\langle dimension \rangle
```

(no default, initially .3333em)

alias /tikz/inner ysep

Specifies the inner separation in the y-direction, only.

```
/pgf/outer sep=\langle dimension \rangle
```

(no default, initially .5\pgflinewidth)

alias /tikz/outer sep

This option adds an additional (invisible) separation space of $\langle dimension \rangle$ outside the background path. The main effect of this option is that all anchors will move a little "to the outside."

For those familiar with CSS, this is same as margin.

The default for this option is half the line width. When the default is used and when the background path is draw, the anchors will lie exactly on the "outside border" of the path (not on the path itself). When the shape is filled, but not drawn, this may not be desirable. In this case, the outer sep should be set to zero point.

```
filled drawn
```

$/pgf/outer xsep=\langle dimension \rangle$

(no default, initially .5\pgflinewidth)

alias /tikz/outer xsep

Specifies the outer separation in the x-direction, only.

$/pgf/outer ysep=\langle dimension \rangle$

(no default, initially .5\pgflinewidth)

alias /tikz/outer ysep

Specifies the outer separation in the y-direction, only.

```
/pgf/minimum height=\langle dimension \rangle
```

(no default, initially Opt)

alias /tikz/minimum height

This option ensures that the height of the shape (including the inner, but ignoring the outer separation) will be at least $\langle dimension \rangle$. Thus, if the text plus the inner separation is not at least as large as $\langle dimension \rangle$, the shape will be enlarged appropriately. However, if the text is already larger than $\langle dimension \rangle$, the shape will not be shrunk.

```
lcm | Ocm |
```

 $/pgf/minimum width=\langle dimension \rangle$

(no default, initially Opt)

alias /tikz/minimum width

Same as minimum height, only for the width.

```
\begin{tikzpicture} \draw (0,0) node[minimum height=2cm,minimum width=3cm,draw] {$3 \times 2$}; \end{tikzpicture}
```

```
/pgf/minimum size=\langle dimension\rangle
```

(no default)

alias /tikz/minimum size

Sets both the minimum height and width at the same time.

```
\text{begin{tikzpicture}} \draw (0,0) node[minimum size=2cm,draw] {square}; \draw (0,-2) node[minimum size=2cm,draw,circle] {circle}; \end{tikzpicture}
```

```
/pgf/shape aspect = \langle aspect \ ratio \rangle
```

(no default)

alias /tikz/shape aspect

Sets a desired aspect ratio for the shape. For the diamond shape, this option sets the ratio between width and height of the shape.

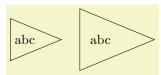
```
\lambda \text{begin{tikzpicture}} \\ \draw (0,0) \ \text{node[shape aspect=1,diamond,draw] {aspect 1};} \\ \draw (0,-2) \ \text{node[shape aspect=2,diamond,draw] {aspect 2};} \\ \text{end{tikzpicture}} \end{tikzpicture}
```

Some shapes (but not all), support a special kind of rotation. This rotation affects only the border of a shape and is independent of the node contents, but *in addition* to any other transformations.

```
\tikzstyle{every node}=[dart, shape border uses incircle,
inner sep=1pt, draw]
\text{begin{tikzpicture}}
\foreach \a/\b/\c in {A/0/0, B/45/0, C/0/45, D/45/45}
\node [shape border rotate=\b, rotate=\c] at (\b/36,-\c/36) {\a};
\end{tikzpicture}
```

There are two types of rotation: restricted and unrestricted. Which type of rotation is applied is determined by on how the shape border is constructed. If the shape border is constructed using an incircle, that is, a circle that tightly fits the node contents (including the inner sep), then the rotation can be unrestricted. If, however, the border is constructed using the natural dimensions of the node contents, the rotation is restricted to integer multiples of 90 degrees.

Why should there be two kinds of rotation and border construction? Borders constructed using the natural dimensions of the node contents provide a much tighter fit to the node contents, but to maintain this tight fit, the border rotation must be restricted to integer multiples of 90 degrees. By using an incircle, unrestricted rotation is possible, but the border will not make a very tight fit to the node contents.



```
\tikzstyle{every node}=[isosceles triangle, draw]
\begin{tikzpicture}
 \node {abc};
 \node [shape border uses incircle] at (2,0) {abc};
\end{tikzpicture}
```

There are PGF keys that determine how a shape border is constructed, and to specify its rotation. It should be noted that not all shapes support these keys, so reference should be made to the documentation for individual shapes.

```
/pgf/shape border uses incircle=\langle boolean \rangle (default true) alias /tikz/shape border uses incircle
```

Determines if the border of a shape is constructed using the incircle. If no value is given $\langle boolean \rangle$ will take the default value true.

```
\label{eq:pgf/shape border rotate} $$ \protect\ protect\ protect
```

Rotates the border of a shape independently of the node contents, but in addition to any other transformations. If the shape border is not constructed using the incircle, the rotation will be rounded to the nearest integer multiple of 90 degrees when the shape is drawn.

Note that if the border of the shape is rotated, the compass point anchors, and 'text box' anchors (including mid east, base west, and so on), do not rotate, but the other anchors do:

Finally, a somewhat unfortunate side-effect of rotating shape borders is that the supporting shapes do not distinguish between outer xsep and outer ysep, and typically, the larger of the two values will be used.

16.3 Multi-Part Nodes

Most nodes just have a single simple text label. However, nodes of a more complicated shape might be made up from several *node parts*. For example, in automata theory a so-called Moore state has a state name, drawn in the upper part of the state circle, and an output text, drawn in the lower part of the state circle. These two parts are quite independent. Similarly, a UML class shape would have a name part, a method part, and an attributes part. Different molecule shapes might use parts for the different atoms to be drawn at the different positions, and so on.

Both PGF and TikZ support such multipart nodes. On the lower level, PGF provides a system for specifying that a shape consists of several parts. On the TikZ level, you specify the different node parts by using the following command:

```
\nodepart[\langle options \rangle] \{\langle part \ name \rangle\}
```

This command can only be used inside the $\langle text \rangle$ argument of a **node** path operation. It works a little bit like a **\part** command in LATEX. It will stop the typesetting of whatever node part was typeset until now and then start putting all following text into the node part named $\langle part \ name \rangle$ —until another **\partname** is encountered or until the node $\langle text \rangle$ ends. The $\langle options \rangle$ will be local to this part.



```
\begin{tikzpicture}
\node [circle split,draw,double,fill=red!20]
{
    % No \nodepart has been used, yet. So, the following is put in the
    % ''text'' node part by default.
    $q_1$
    \nodepart{lower} % Ok, end ''text'' part, start ''output'' part
    $00$
}; % output part ended.
\end{tikzpicture}
```

You will have to lookup which parts are defined by a shape.

The following styles influences node parts:

```
/tikz/every (part name) node part
```

(style, initially empty)

This style is installed at the beginning of every node part named $\langle part \ name \rangle$.



```
\tikz [every lower node part/.style={red}]
\node [circle split,draw] {$q_1$ \nodepart{lower} $00$};
```

16.4 The Node Text

16.4.1 Text Parameters: Color and Opacity

The simplest option for the text in nodes is its color. Normally, this color is just the last color installed using color=, possibly inherited from another scope. However, it is possible to specifically set the color used for text using the following option:

```
/\text{tikz/text} = \langle color \rangle (no default)
```

Sets the color to be used for text labels. A color= option will immediately override this option.

Just like the color itself, you may also wish to set the opacity of the text only. For this, use the text opacity option, which is detailed in Section 21.

16.4.2 Text Parameters: Font

Next, you may wish to adjust the font used for the text. Use the following option for this:

```
/tikz/font=\( font commands \)
```

(no default)

Sets the font used for text labels.

```
italic
\text{begin{tikzpicture}}
\draw[font=\itshape] (1,0) -- +(1,1) node[above] {italic};
\end{tikzpicture}
```

A perhaps more useful example is the following:

16.4.3 Text Parameters: Alignment and Width for Multi-Line Text

Normally, when a node is typeset, all the text you give in the braces is put in one long line (in an \hbox, to be precise) and the node will become as wide as necessary.

From time to time you may wish to create nodes that contain multiple lines of text. There are three different ways of achieving this:

1. Inside the node, you can put some standard environment that produces multi-line, aligned text. For instance, you can use a {tabular} inside a node:

```
upper left upper right lower left lower right
```

```
\tikz \node [draw] {
  \begin{tabular}{cc}
  upper left & upper right\\
  lower left & lower right
  \end{tabular}
};
```

This approach offers the most flexibility in the sense that it allows you to use all of the alignment commands offered by your format of choice.

2. You use $\$ inside your node to mark the end of lines and then request TikZ to arrange these lines in some manner. This will only be done, however, if the align option has been given.

```
This is a demonstration.

This is a demonstration.

This is a demonstration.

A tikz[align=left] \node[draw] {This is a \demonstration.};

This is a demonstration.
```

The \\ command takes an optional extra space as an argument in square brackets.

```
This is a demonstration text for alignments. \tikz \node[fill=examplefill,align=right] {This is a\\[-2pt] demonstration text for\\[1ex] alignments.};
```

3. You can request that TikZ does an automatic line-breaking for you inside the node by specifying a fixed text width for the node. In this case, you can still use $\$ to enforce a line-break. Note that when you specify a text width, the node will have this width, independently of whether the text actually "reaches the end" of the node.

Let us now first have a look at the text width command.

```
/tikz/text width=\(dimension\) (no default)
```

This option will put the text of a node in a box of the given width (something akin to a $\{minipage\}$ of this width, only portable across formats). If the node text is not as wide as $\langle dimension \rangle$, it will nevertheless be put in a box of this width. If it is larger, line breaking will be done.

By default, when this option is given, a ragged right border will be used (align=left). This is sensible since, typically, these boxes are narrow and justifying the text looks ugly. You can, however, change the alignment using align or directly using commands line \centering.

```
This is a demonstration text for showing how line breaking works. 

\tikz \draw (0,0) node[fill=examplefill,text width=3cm] {This is a demonstration text for showing how line breaking works.};
```

Setting $\langle dimension \rangle$ to an empty string causes the automatic line breaking to be disabled.

```
/\text{tikz/align} = \langle alignment\ option \rangle (no default)
```

This key is used to set up an alignment for multi-line text inside a node. If text width is set to some width (let us call this alignment with line breaking), the align key will setup the \leftskip and the \rightskip in such a way that the text is broken and aligned according to \(\lambda \lightgar \text{ing} \). If text width is not set (that is, set to the empty string; let us call this alignment without line breaking), then a different mechanism is used internally, namely the key node halign header, is set to an

appropriate value. While this key, which is documented below, is not to be used by beginners, the net effect is simple: When text width is not set, you can use $\$ to break lines and align them according to $\langle alignment\ option\rangle$ and the resulting node's width will be minimal to encompass the resulting lines. In detail, you can set $\langle alignment\ option\rangle$ to one of the following values:

align=left For alignment without line breaking, the different lines are simply aligned such that their left borders are below one another.

This is a demonstration text for alignments.

```
\tikz \node[fill=examplefill,align=left]
{This is a\\ demonstration text for\\ alignments.};
```

For alignment with line breaking, the same will happen only the lines will now, additionally, be broken automatically:

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=examplefill,text width=3cm,align=left]
{This is a demonstration text for showing how line breaking works.};
```

align=flush left For alignment without line breaking this option has exactly the same effect as left. However, for alignment with line breaking, there is a difference: While left uses the original plain TEX definition of a ragged right border, in which TEX will try to balance the right border as well as possible, flush left causes the right border to be ragged in the LATEX-style, in which no balancing occurs. This looks ugly, but it may be useful for very narrow boxes and when you wish to avoid hyphenations.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=examplefill,text width=3cm,align=flush left]
{This is a demonstration text for showing how line breaking works.};
```

align=right Works like left, only for right alignment.

This is a demonstration text for alignments.

```
\tikz \node[fill=examplefill,align=right]
{This is a\\ demonstration text for\\ alignments.};
```

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=examplefill,text width=3cm,align=right]
{This is a demonstration text for showing how line breaking works.};
```

align=flush right Works like flush left, only for right alignment.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=examplefill,text width=3cm,align=flush right]
{This is a demonstration text for showing how line breaking works.};
```

align=center Works like left or right, only for centered alignment.

This is a demonstration text for alignments.

```
\tikz \node[fill=examplefill,align=center]
{This is a\\ demonstration text for\\ alignments.};
```

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=examplefill,text width=3cm,align=center]
{This is a demonstration text for showing how line breaking works.};
```

align=flush center Works like flush left or flush right, only for center alignment.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=examplefill,text width=3cm,align=flush center]
{This is a demonstration text for showing how line breaking works.};
```

align=justify For alignment without line breaking, this has the same effect as left. For alignment with line breaking, this causes the text to be "justified." Use this only with rather broad nodes.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=examplefill,text width=3cm,align=justify]
{This is a demonstration text for showing how line breaking works.};
```

In the above example, TeX complains (rightfully) about three very badly typeset lines. (For this manual I asked TeX to stop complaining by using \hbadness=10000, but this is a foul deed, indeed.)

align=none Disables all alignments and \\ will not be redefined.

```
/tikz/node halign header=\langle macro storing a header\rangle
```

(no default, initially empty)

This is the key that is used by align internally for alignment without line breaking. Read the following only if you are familiar with the \halign command.

This key only has an effect if text width is empty, otherwise it is ignored. Furthermore, if $\langle macrostoring\ a\ header \rangle$ is empty, then this key also has no effect. So, suppose text width is empty, but $\langle header \rangle$ is not. In this case the following happens:

When the node text is parsed, the command \\ is redefined internally. This redefinition is done in such a way that the text from the start of the node to the first occurrence of \\ is put in an \hbox. Then the text following \\ up to the next \\ is put in another \hbox. This goes on until the text between the last \\ and the closing \} is also put in an \hbox.

The $\langle macro\ storing\ a\ header \rangle$ should be a macro that contains some text suitable for use as a header for the **\halign** command. For instance, you might define

```
\def\myheader{\hfil\hfil##\hfil\cr}
\tikz [node halign header=\myheader] ...
```

You cannot just say node halign header=\hfil\hfil#\hfil\cr because this confuses TEX inside matrices, so this detour via a macro is needed.

Next, conceptually, all these boxes are recursively put inside an $\ \$ command. Assuming that $\langle first \rangle$ is the first of the above boxes, the command $\ \$ is the first of the above boxes, the command $\ \$ Then, the following box is created, where $\langle second \rangle$ is the second input box: $\$ halign{ $\langle header \rangle \setminus box \langle previous\ box \rangle \setminus cr \setminus box \langle second \rangle \setminus cr}$. Let us call the resulting box the $\langle previous\ box \rangle$ once more. Then the next box that is created is $\$ halign{ $\langle header \rangle \setminus box \langle previous\ box \rangle \setminus cr \setminus box \langle third \rangle \setminus cr}$.

All of this means that if $\langle header \rangle$ is an \halign header like \hfil#\hfil\cr, then all boxes will be centered relative to one another. Similarly, a $\langle header \rangle$ of \hfil#\cr causes the text to be flushed right.

Note that this mechanism is not flexible enough to all multiple columns inside $\langle header \rangle$. You will have to use a tabular or a matrix in such cases.

One further note: Since the text of each line is placed in a box, settings will be local to each "line." This is very similar to the way a cell in a tabular or a matrix behaves.

16.4.4 Text Parameters: Height and Depth of Text

In addition to changing the width of nodes, you can also change the height of nodes. This can be done in two ways: First, you can use the option minimum height, which ensures that the height of the whole node is at least the given height (this option is described in more detail later). Second, you can use the option text height, which sets the height of the text itself, more precisely, of the TEX text box of the text. Note that the text height typically is not the height of the shape's box: In addition to the text height, an internal inner sep is added as extra space and the text depth is also taken into account.

I recommend using minimum size instead of text height except for special situations.

/tikz/text height=\langle dimension\rangle

(no default)

Sets the height of the text boxes in shapes. Thus, when you write something like node {text}, the text is first typeset, resulting in some box of a certain height. This height is then replaced by the height text height. The resulting box is then used to determine the size of the shape, which will typically be larger. When you write text height= without specifying anything, the "natural" size of the text box remains unchanged.

```
\tikz \node[draw] {y};
\tikz \node[draw,text height=10pt] {y};
```

```
/tikz/text depth=\langle dimension \rangle
```

(no default)

This option works like text height, only for the depth of the text box. This option is mostly useful when you need to ensure a uniform depth of text boxes that need to be aligned.

16.5 Positioning Nodes

When you place a node at some coordinate, the node is centered on this coordinate by default. This is often undesirable and it would be better to have the node to the right or above the actual coordinate.

16.5.1 Positioning Nodes Using Anchors

PGF uses a so-called anchoring mechanism to give you a very fine control over the placement. The idea is simple: Imagine a node of rectangular shape of a certain size. PGF defines numerous anchor positions in the shape. For example to upper right corner is called, well, not "upper right anchor," but the north east anchor of the shape. The center of the shape has an anchor called center on top of it, and so on. Here are some examples (a complete list is given in Section 16.2.1).



Now, when you place a node at a certain coordinate, you can ask TikZ to place the node shifted around in such a way that a certain anchor is at the coordinate. In the following example, we ask TikZ to shift the first node such that its **north east** anchor is at coordinate (0,0) and that the **west** anchor of the second node is at coordinate (1,1).

```
first node

\tikz \draw (0,0) node[anchor=north east] {first node}
rectangle (1,1) node[anchor=west] {second node};
```

Since the default anchor is center, the default behaviour is to shift the node in such a way that it is centered on the current position.

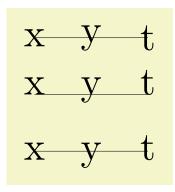
```
/tikz/anchor=⟨anchor name⟩
```

(no default)

Causes the node to be shifted such that it's anchor (anchor name) lies on the current coordinate.

The only anchor that is present in all shapes is **center**. However, most shapes will at least define anchors in all "compass directions." Furthermore, the standard shapes also define a **base** anchor, as well as **base west** and **base east**, for placing things on the baseline of the text.

The standard shapes also define a mid anchor (and mid west and mid east). This anchor is half the height of the character "x" above the base line. This anchor is useful for vertically centering multiple nodes that have different heights and depth. Here is an example:



```
\begin{tikzpicture}[scale=3,transform shape]
    % First, center alignment -> wobbles
    \draw[anchor=center] (0,1) node{x} -- (0.5,1) node{y} -- (1,1) node{t};
    % Second, base alignment -> no wobble, but too high
    \draw[anchor=base] (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t};
    % Third, mid alignment
    \draw[anchor=mid] (0,0) node{x} -- (0.5,0) node{y} -- (1,0) node{t};
    \end{tikzpicture}
```

16.5.2 Basic Placement Options

Unfortunately, while perfectly logical, it is often rather counter-intuitive that in order to place a node *above* a given point, you need to specify the **south** anchor. For this reason, there are some useful options that allow you to select the standard anchors more intuitively:

```
tikz/above = \langle offset \rangle (default Opt)
```

Does the same as anchor=south. If the $\langle offset \rangle$ is specified, the node is additionally shifted upwards by the given $\langle offset \rangle$.

```
above \tikz \fill (0,0) circle (2pt) node[above] {above};

above \tikz \fill (0,0) circle (2pt) node[above=2pt] {above};
```

```
\texttt{/tikz/below=}\langle offset \rangle (default Opt)
```

Similar to above.

```
/tikz/left=\langle offset \rangle (default Opt)
```

Similar to above.

```
/tikz/right = \langle offset \rangle (default Opt)
```

Similar to above.

/tikz/above left (no value)

Does the same as anchor=south east. Note that giving both above and left options does not have the same effect as above left, rather only the last left "wins." Actually, this option also takes an

(offset) parameter, but using this parameter without using the positioning library is deprecated. (The positioning library changes the meaning of this parameter to something more sensible.)

```
above left

/tikz/above right
Similar to above left.

/tikz/below left
Similar to above left.

/tikz/below left
Similar to above left.

/tikz/below right
Similar to above left.

/tikz/below right
Similar to above left.

/tikz/below right
Similar to above left.
```

16.5.3 Advanced Placement Options

While the standard placement options suffice for simple cases, the **positioning** library offers more convenient placement options.

```
\usetikzlibrary{positioning} % MEX and plain EX \usetikzlibrary[positioning] % ConTEXt
```

The library defines additional options for placing nodes conveniently. It also redefines the standard options like above so that they give you better control of node placement.

When this library is loaded, the options like above or above left behave differently.

```
/tikz/above=\specification\ (default Opt)
```

With the positioning library loaded, the above option does not take a simple $\langle dimension \rangle$ as its parameter. Rather, it can (also) take a more elaborate $\langle specification \rangle$ as parameter. This $\langle specification \rangle$ has the following general form: It starts with an optional $\langle shifting\ part \rangle$ and is followed by an optional $\langle of-part \rangle$. Let us start with the $\langle shifting\ part \rangle$, which can have three forms:

1. It can simply be a \(\langle \frac{dimension}{\} \) (or a mathematical expression that evaluates to a dimension) like 2cm or 3cm/2+4cm. In this case, the following happens: the node's anchor is set to south and the node is vertically shifted upwards by \(\langle \frac{dimension}{\} \rangle \).

```
\lambda begin{tikzpicture}
\draw[help lines] (0,0) grid (2,2);
\node at (1,1) [above=2pt+3pt,draw] {above};
\end{tikzpicture}
```

This use of the above option is the same as if the positioning library were not loaded.

2. It can be a \(\langle number \rangle \) (that is, any mathematical expression that does not include a unit like pt or cm). Examples are 2 or 3+sin(60). In this case, the anchor is also set to south and the node is vertically shifted by the vertical component of the coordinate (0, \(\langle number \rangle \)).

```
\lambda begin{tikzpicture}
\draw[help lines] (0,0) grid (2,2);
\node at (1,1) [above=.2,draw] {above};
% south border of the node is now 2mm above (1,1)
\end{tikzpicture}
```

3. It can be of the form (number or dimension 1) and (number or dimension 2). This specification does not make particular sense for the above option, it is much more useful for options like above left. The reason it is allowed for the above option is that it is sometimes automatically used, as explained later.

The effect of this option is the following. First, the point ($\langle number\ of\ dimension\ 2\rangle$, $\langle number\ or\ dimension\ 1\rangle$) is computed (note the inversed order), using the normal rules for evaluating such a coordinate, yielding some position. Then, the node is shifted by the vertical component of this point. The anchor is set to south.

```
above \text{\lambda begin{tikzpicture}} \draw[help lines] (0,0) grid (2,2); \node at (1,1) [above=.2 and 3mm,draw] {above}; \text{\gamma south border of the node is also 2mm above (1,1) \end{tikzpicture}}
```

The $\langle shifting \ part \rangle$ can optionally be followed by a $\langle of-part \rangle$, which has one of the following forms:

1. The $\langle of\text{-}part \rangle$ can be of $\langle coordinate \rangle$, where $\langle coordinate \rangle$ is not in parentheses and it is not just a node name. An example would be of somenode.north or of 2,3. In this case, the following happens: First, the node's at parameter is set to the $\langle coordinate \rangle$. Second, the node is shifted according to the $\langle shift\text{-}part \rangle$. Third, the anchor is set to south.

Here is a basic example:

```
\text{\lambda} \text{
```

As can be seen the above=5mm of somenode.north east option does, indeed, place the node 5mm above the north east anchor of somenode. The same effect could have been achieved writing above=5mm followed by at=(somenode.north east).

If the $\langle shifting\text{-}part \rangle$ is missing, the shift is not zero, but rather the value of the node distance key is used, see below.

- 2. The $\langle of\text{-}part \rangle$ can be of $\langle node\ name \rangle$. An example would be of somenode. In this case, the following usually happens:
 - The anchor is set to south.
 - The node is shifted according to the \(\shifting \ part \) or, if it is missing, according to the value of node distance.
 - The node's at parameter is set to $\langle node \ name \rangle$.north.

The net effect of all this is that the new node will be placed in such a way that the distance between is south border and $\langle node\ name \rangle$'s north border is exactly the given distance.

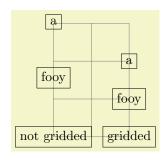
```
| above=1cm of some node | above=1cm of some n
```

It is possible to change the behaviour of this $\langle specification \rangle$ rather drastically, using the following key:

```
/tikz/on grid=\langle boolean \rangle (no default, initially false)
```

When this key is set to true, an $\langle of\text{-}part \rangle$ of the current form behaves differently: The anchors set for the current node as well as the anchor used for the other $\langle node\ name \rangle$ are set to center.

This has the following effect: When you say above=1cm of somenode with on grid set to true, the new node will be placed in such a way that its center is 1cm above the center of somenode. Repeatedly placing nodes in this way will result in nodes that are centered on "grid coordinate," hence the name of the option.



```
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,3);

% Not gridded
  \node (a1) at (0,0) {not gridded};
  \node (b1) [above=1cm of a1] {fooy};
  \node (c1) [above=1cm of b1] {a};

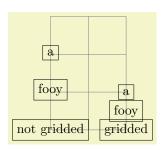
% gridded
  \node (a2) at (2,0) {gridded};
  \node (b2) [on grid,above=1cm of a2] {fooy};
  \node (c2) [on grid,above=1cm of b2] {a};

\end{tikzpicture}
```

/tikz/node distance=⟨shifting part⟩

(no default, initially 1cm and 1cm)

The value of this key is used as $\langle shifting \ part \rangle$ is used if and only if a $\langle of\text{-}part \rangle$ is present, but no $\langle shifting \ part \rangle$.



/tikz/below=\langle specification\rangle

(no default)

This key is redefined in the same manner as above.

/tikz/left=\langle specification \rangle

(no default)

This key is redefined in the same manner as above, only all vertical shifts are replaced by horizontal shifts.

/tikz/right=\langle specification \rangle

(no default)

This key is redefined in the same manner as left.

/tikz/above left=\langle specification \rangle

(no default)

This key is also redefined in a manner similar to the above, but behaviour of the $\langle shifting \; part \rangle$ is more complicated:

- 1. When the $\langle shifting\ part \rangle$ is of the form $\langle number\ or\ dimension \rangle$ and $\langle number\ or\ dimension \rangle$, it has (essentially) the effect of shifting the node vertically upwards by the first $\langle number\ or\ dimension \rangle$ and to the left by the second. To be more precise, the coordinate ($\langle second\ number\ or\ dimension \rangle$, $\langle first\ number\ or\ dimension \rangle$) is computed and then the node is shifted vertically by the y-part of the resulting coordinate and horizontally be the negated x-part of the result. (This is exactly what you expect, except possibly when you have used the x and y options to modify the xy-coordinate system so that the unit vectors no longer point in the expected directions.)
- 2. When the $\langle shifting\ part \rangle$ is of the form $\langle number\ or\ dimension \rangle$, the node is shifted by this $\langle number\ or\ dimension \rangle$ in the direction of 135°. This means that there is a difference between a $\langle shifting\ part \rangle$ of 1cm and of 1cm and 1cm: In the second case, the node is shifted by 1cm upward and 1cm

to the left; in the first case it is shifted by $\frac{1}{2}\sqrt{2}$ cm upward and by the same amount to the left. A more mathematical way of phrasing this is the following: A plain $\langle dimension \rangle$ is measured in the l_2 -norm, while a $\langle dimension \rangle$ and $\langle dimension \rangle$ is measured in the l_1 -norm.

The following example should help to illustrate the difference:

```
\begin{tikzpicture}[every node/.style={draw,circle}]
  \draw[help lines] (0,0) grid (2,5);
 \begin{scope}[node distance=5mm and 5mm]
   \node (b) at (1,4) {b};
   \node [left=of b] {1};
                                 \node [right=of b] {2};
   \node [above=of b] {3};
                                 \node [below=of b] {4};
   \node [above left=of b] {5}; \node [above right=of b] {6};
   \node [below left=of b] {7}; \node [below right=of b] {8};
  \end{scope}
  \begin{scope}[node distance=5mm]
    \node (a) at (1,1) {a};
   \node [left=of a] {1};
                                 \node [right=of a] {2};
   \node [above=of a] {3};
                                 \node [below=of a] {4};
   \node [above left=of a] {5}; \node [above right=of a] {6};
   \node [below left=of a] {7}; \node [below right=of a] {8};
 \end{scope}
\end{tikzpicture}
```

```
\begin{tikzpicture}[every node/.style={draw,rectangle}]
              \draw[help lines] (0,0) grid (2,5);
             \begin{scope}[node distance=5mm and 5mm]
               \node (b) at (1,4) {b};
b
               \node [left=of b] {1};
                                             \node [right=of b] {2};
               \node [above=of b] {3};
                                             \node [below=of b] {4}:
               \node [above left=of b] {5}; \node [above right=of b] {6};
               \node [below left=of b] {7}; \node [below right=of b] {8};
              \end{scope}
              \begin{scope}[node distance=5mm]
               \node (a) at (1,1) {a};
               \node [left=of a] {1};
                                             \node [right=of a] {2};
               \node [above=of a] {3};
                                             \node [below=of a] {4};
               \node [above left=of a] {5}; \node [above right=of a] {6};
               \node [below left=of a] {7}; \node [below right=of a] {8};
             \end{scope}
            \end{tikzpicture}
```

```
\begin{tikzpicture}[every node/.style={draw,rectangle},on grid]
                          \draw[help lines] (0,0) grid (4,4);
           6
                          \begin{scope}[node distance=1]
                            \node (a) at (2,3) {a};
                            \node [left=of a] {1};
                                                          \node [right=of a] {2};
                                                          \node [below=of a] {4};
                            \node [above=of a] {3}:
                            \node [above left=of a] {5}; \node [above right=of a] {6};
                            \node [below left=of a] {7}; \node [below right=of a] {8};
                          \end{scope}
                          \begin{scope}[node distance=1 and 1]
5
                            \node (b) at (2,0) {b};
                            \node [left=of b] {1};
                                                          \node [right=of b] {2}:
                            \node [above=of b] {3};
                                                          \node [below=of b] {4};
                            \node [above left=of b] {5}; \node [above right=of b] {6};
                            \node [below left=of b] {7}; \node [below right=of b] {8};
                          \end{scope}
             8
                        \end{tikzpicture}
```

```
/tikz/below left=\langle specification \rangle
    Works similar to above left.

/tikz/above right=\langle specification \rangle
    Works similar to above left.

/tikz/below right=\langle specification \rangle
    Works similar to above left.
(no default)

Works similar to above left.
```

The positioning package also introduces the following new placement keys:

```
/tikz/base left=\langle specification\rangle
```

(no default)

This key works like the left key, only instead of the east anchor, the base east anchor is used and, when the second form of an $\langle of\text{-}part \rangle$ is used, the corresponding base west anchor.

This key is useful for chaining together nodes so that their base lines are aligned.

```
X a y X a y
```

```
/tikz/base right=\langle specification \rangle
```

(no default)

Works like base left.

```
/tikz/mid left=\langle specification \rangle
```

(no default)

Works like base left, but with mid east and mid west anchors instead of base east and base west.

```
/tikz/mid right=\langle specification \rangle
```

(no default)

Works like mid left.

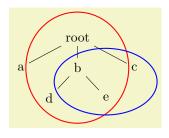
16.5.4 Arranging Nodes Using Chains and Matrices

The simple above and right options may not always suffice for arranging a large number of nodes. For such situations TikZ offers two libraries that make positioning easier: The chains library and the matrix library. The first is mostly useful for creating "chains of nodes" and, more generally, "flows." The second allows you to arrange multiple nodes in rows and columns. These methods for positioning nodes are described in two separate Sections 18 and 29.

16.6 Fitting Nodes to a Set of Coordinates

It is sometimes desirable that the size and position of a node is not given using anchors and size parameters, rather one would sometimes have a box be placed and be sized such that it "is just large enough to contain this, that, and that point." This situation typically arises when a picture has been drawn and, afterwards, parts of the picture are supposed to be encircled or highlighted.

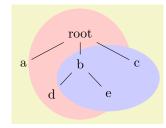
In this situation the fit option from the fit library is useful, see Section 35 for the details. The idea is that you may give the fit option to a node. The fit option expects a list of coordinates (one after the other without commas) as its parameter. The effect will be that the node's text area has exactly the necessary size so that it contains all the given coordinates. Here is an example:



```
\begin{tikzpicture}[level distance=8mm]
\node (root) {root}
  child { node (a) {a} }
  child { node (b) {b}
    child { node (d) {d} }
    child { node (e) {e} } }
  child { node (c) {c} };

\node[draw=red,inner sep=0pt,thick,ellipse,fit=(root) (b) (d) (e)] {};
\node[draw=blue,inner sep=0pt,thick,ellipse,fit=(b) (c) (e)] {};
\end{tikzpicture}
```

If you want to fill the fitted node you will usually have to place it on a background layer.



```
\begin{tikzpicture}[level distance=8mm]
\node (root) {root}
child { node (a) {a} }
child { node (b) {b}
child { node (d) {d} }
child { node (e) {e} } }
child { node (c) {c} };
\begin{pgfonlayer}{background}
\node[fill=red!20,inner sep=0pt,ellipse,fit=(root) (b) (d) (e)] {};
\node[fill=blue!20,inner sep=0pt,ellipse,fit=(b) (c) (e)] {};
\end{pgfonlayer}
\end{tikzpicture}
```

16.7 Transformations

It is possible to transform nodes, but, by default, transformations do not apply to nodes. The reason is that you usually do *not* want your text to be scaled or rotated even if the main graphic is transformed. Scaling text is evil, rotating slightly less so.

However, sometimes you do wish to transform a node, for example, it certainly sometimes makes sense to rotate a node by 90 degrees. There are two ways to achieve this:

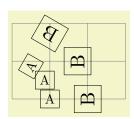
1. You can use the following option:

/tikz/transform shape

(no value)

Causes the current "external" transformation matrix to be applied to the shape. For example, if you said \tikz[scale=3] and then say node[transform shape] {X}, you will get a "huge" X in your graphic.

2. You can give transformation option *inside* the option list of the node. *These* transformations always apply to the node.



16.8 Placing Nodes on a Line or Curve Explicitly

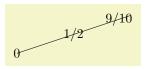
Until now, we always placed node on a coordinate that is mentioned in the path. Often, however, we wish to place nodes on "the middle" of a line and we do not wish to compute these coordinates "by hand." To facilitate such placements, TikZ allows you to specify that a certain node should be somewhere "on" a line. There are two ways of specifying this: Either explicitly by using the **pos** option or implicitly by placing the node "inside" a path operation. These two ways are described in the following.

```
/\text{tikz/pos} = \langle fraction \rangle (no default)
```

When this option is given, the node is not anchored on the last coordinate. Rather, it is anchored on some point on the line from the previous coordinate to the current point. The $\langle fraction \rangle$ dictates how "far" on the line the point should be. A $\langle fraction \rangle$ of 0 is the previous coordinate, 1 is the current one, everything else is in between. In particular, 0.5 is the middle.

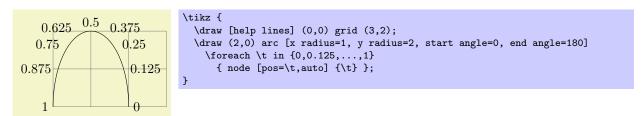
Now, what is "the previous line"? This depends on the previous path construction operation.

In the simplest case, the previous path operation was a "line-to" operation, that is, a $--\langle coordinate \rangle$ operation:



```
\tikz \draw (0,0) -- (3,1)
node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

For the arc operation, the position is simply the corresponding position on the arc:



The next case is the curve-to operation (the .. operation). In this case, the "middle" of the curve, that is, the position 0.5 is not necessarily the point at the exact half distance on the line. Rather, it is some point at "time" 0.5 of a point traveling from the start of the curve, where it is at time 0, to the end of the curve, which it reaches at time 0.5. The "speed" of the point depends on the length of the support vectors (the vectors that connect the start and end points to the control points). The exact math is a bit complicated (depending on your point of view, of course); you may wish to consult a good book on computer graphics and Bézier curves if you are intrigued.

```
\tikz \draw (0,0) .. controls +(right:3.5cm) and +(right:3.5cm) .. (0,3) \foreach \p in \{0,0.125,...,1\} \{node[pos=\p]\{\p}\};

0.875
0.625
0.375
0.25
```

Another interesting case are the horizontal/vertical line-to operations |- and -|. For them, the position (or time) 0.5 is exactly the corner point.

For all other path construction operations, the position placement does not work, currently.

```
/tikz/auto=\direction\ (default is scope's setting)
```

This option causes an anchor positions to be calculated automatically according to the following rule. Consider a line between to points. If the $\langle direction \rangle$ is left, then the anchor is chosen such that the node is to the left of this line. If the $\langle direction \rangle$ is right, then the node is to the right of this line. Leaving out $\langle direction \rangle$ causes automatic placement to be enabled with the last value of left or right used. A $\langle direction \rangle$ of false disables automatic placement. This happens also whenever an anchor is given explicitly by the anchor option or by one of the above, below, etc. options.

This option only has an effect for nodes that are placed on lines or curves.

```
\begin{array}{c} f \longleftrightarrow e \\ g & f-g & e-f \\ \downarrow g-h & c-d \\ h & h-a \\ a \longrightarrow b \end{array}
```

/tikz/swap (no value)

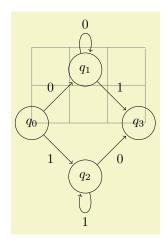
This option exchanges the roles of left and right in automatic placement. That is, if left is the current auto placement, right is set instead and the other way round.

```
0.5
   0.6
                   0.4
        0.6
              0.4
0.7
                       0.3
                  0.2
     0.8
0.8
                       0.2
      0.9
                 0.1
0.025
                    0.975
   0
```

```
\begin{tikzpicture}[auto]
  \draw[help lines, use as bounding box] (0,-.5) grid (4,5);

\draw (0.5,0) .. controls (9,6) and (-5,6) .. (3.5,0)
  \foreach \pos in {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
    {node [pos=\pos,swap,fill=red!20] {\pos}}
  \foreach \pos in {0.025,0.2,0.4,0.6,0.8,0.975}
    {node [pos=\pos,fill=blue!20] {\pos}};

\end{tikzpicture}
```



```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,auto]
  \draw[help lines] (0,0) grid (3,2);
  \node[state] (q_0)
                                              {$q_0$};
  \node[state] (q_1) [above right of=q_0] {$q_1$};
  \label{lower} $$ \ (q_2) \ [below right of=q_0] {$q_2$};
  \label{low-right-of-q_1} $$\substack{q_3$ [below right of=q_1] {$q_3$};}
                                                     {0} (q_1)
  \path[->] (q_0) edge
                                       node
                                       node [swap] {1} (q_2)
                   edge
             (q_1) edge
                                       node
                                                     {1} (q_3)
                   edge [loop above] node
                                                     {0} ()
             (q_2) edge
                                                    {0} (q_3)
                                       node
                   edge [loop below] node
                                                     {1} ();
\end{tikzpicture}
```

/tikz/' (no value)

This is a very short alias for swap.

/tikz/sloped (no value)

This option causes the node to be rotated such that a horizontal line becomes a tangent to the curve. The rotation is normally done in such a way that text is never "upside down." To get upside-down text, use can use [rotate=180] or [allow upside down], see below.

```
0.75 1 0.25
```

```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
\foreach \p in {0,0.25,...,1} {node[sloped,above,pos=\p]{\p}};
```

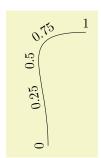


```
\begin{tikzpicture}[->]
\draw (0,0) -- (2,0.5) node[midway,sloped,above] {$x$};
\draw (2,-.5) -- (0,0) node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

/tikz/allow upside down=\langle boolean \rangle

(default true, initially false)

If set to true, TikZ will not "righten" upside down text.



```
\tikz [allow upside down]
  \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
  \foreach \p in {0,0.25,...,1} {node[sloped,above,pos=\p]{\p}};
```

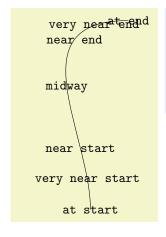


```
\begin{tikzpicture}[->,allow upside down]
  \draw (0,0) -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0) node[midway,sloped,below] {$y$};
  \end{tikzpicture}
```

There exist styles for specifying positions a bit less "technically":

/tikz/midway (style, no value)

This has the same effect as pos=0.5.



/tikz/near start

(style, no value)

Set to pos=0.25.

/tikz/near end

(style, no value)

Set to pos=0.75.

```
/tikz/very near start
Set to pos=0.125.

/tikz/very near end
Set to pos=0.875.

/tikz/at start
Set to pos=0.

/tikz/at end
Set to pos=1.

(style, no value)
(style, no value)
(style, no value)
```

16.9 Placing Nodes on a Line or Curve Implicitly

When you wish to place a node on the line (0,0) -- (1,1), it is natural to specify the node not following the (1,1), but "somewhere in the middle." This is, indeed, possible and you can write (0,0) -- node{a} (1,1) to place a node midway between (0,0) and (1,1).

What happens is the following: The syntax of the line-to path operation is actually -- node $\langle node specification \rangle \langle coordinate \rangle$. (It is even possible to give multiple nodes in this way.) When the optional node is encountered, that is, when the -- is directly followed by node, then the specification(s) are read and "stored away." Then, after the $\langle coordinate \rangle$ has finally been reached, they are inserted again, but with the pos option set.

There are two things to note about this: When a node specification is "stored," its catcodes become fixed. This means that you cannot use overly complicated verbatim text in them. If you really need, say, a verbatim text, you will have to put it in a normal node following the coordinate and add the pos option.

Second, which pos is chosen for the node? The position is inherited from the surrounding scope. However, this holds only for nodes specified in this implicit way. Thus, if you add the option [near end] to a scope, this does not mean that *all* nodes given in this scope will be put on near the end of lines. Only the nodes for which an implicit pos is added will be placed near the end. Typically, this is what you want. Here are some examples that should make this clearer:

Like the line-to operation, the curve-to operation . . also allows you to specify nodes "inside" the operation. After both the first . . and also after the second . . you can place node specifications. Like for the --operation, these will be collected and then reinserted after the operation with the pos option set.

16.10 The Label and Pin Options

16.10.1 Overview

In addition to the node path operation, the two options label and pin can be used to "add a node next to another node". As an example, suppose we want to draw a graph in which the nodes are small circles:

```
\tikz [circle] {
    \node [draw] (s) {};
    \node [draw] (a) [right=of s] {} edge (s);
    \node [draw] (b) [right=of a] {} edge (a);
    \node [draw] (t) [right=of b] {} edge (b);
}
```

Now, in the above example, suppose we wish to indicate that the first node is the start node and the last node is the target node. We could write \node (s) {\$s\$};, but this would enlarge the first node. Rather, we want the "s" to be placed next to the node. For this, we need to create *another* node, but next to the existing node. The label and pin option allow us to do exactly this without having to use the cumbersome node syntax:

```
\overset{s}{\bigcirc} - - \bigcirc - - \bigcirc
```

```
\tikz [circle] {
  \node [draw] (s) [label=$s$] {};
  \node [draw] (a) [right=of s] {} edge (s);
  \node [draw] (b) [right=of a] {} edge (a);
  \node [draw] (t) [right=of b, label=$t$] {} edge (b);
}
```

16.10.2 The Label Option

```
\texttt{/tikz/label=[}\langle options \rangle ] \langle angle \rangle : \langle text \rangle
```

(no default)

When this option is given to a **node** operation, it causes *another* node to be added to the path after the current node has been finished. This extra node will have the text $\langle text \rangle$. It is placed, in principle, in the direction $\langle angle \rangle$ relative to the main node, but the exact rules are a bit complex. Suppose the **node** currently under construction is called **main node** and let us call the label node **label node**. Then the following happens:

1. The $\langle angle \rangle$ is used to determine a position on the border of the main node. If the $\langle angle \rangle$ is missing, the value of the following key is used instead:

```
/tikz/label position=\langle angle \rangle
```

(no default, initially above)

Sets the default position for labels.

The $\langle angle \rangle$ determines the position on the border of the shape in two different ways. Normally, the border position is given by main node. $\langle angle \rangle$. This means that the $\langle angle \rangle$ can either be a number like 0 or -340, but it can also be an anchor like north. Additionally, the special angles above, below, left, right, above left, and so on are automatically replaced by the corresponding angles 90, 270, 180, 0, 135, and so on.

A special case arises when the following key is set:

```
/tikz/absolute=⟨true or false⟩
```

(default true)

When this key is set, the \(\langle \rangle \) is interpreted differently: We still use a point on the border of the main node, but the angle is measured "absolutely," that is, an angle of 0 refers to the point on the border that lies on a straight line from the main node's center to the right (relative to the paper, not relative to the local coordinate system or either the node or the scope).

The difference can be seen in the following example:



```
\tikz [rotate=-80,every label/.style={draw,red}]
\node [transform shape,rectangle,draw,label=right:label] {main node};
```



```
\tikz [rotate=-80,every label/.style={draw,red},absolute]
\node [transform shape,rectangle,draw,label=right:label] {main node};
```

2. Then, an anchor point for the label node. It is determined in such a way that the label node will "face away" from the border of the main node. The anchor that is chosen depends on the position of the border point that is chosen and its position relative to the center of the main node and on whether the transform shape option is set. In general, the choice should be what you would expect, but you may have to set the anchor yourself in difficult situations.

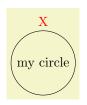
```
\underbrace{\left(\begin{array}{c} \text{default} 60^{\circ} \\ \text{my circle} \end{array}\right)}_{-90^{\circ}}
```

```
\tikz
\node [circle,draw,
    label=default,
    label=60:\$60^\circ\$,
    label=below:\$-90^\circ\$] {my circle};
```

3. One $\langle angle \rangle$ is special: If you set the $\langle angle \rangle$ to center, then the label will be placed on the center of the main node. This is mainly useful for adding a label text to an existing node, especially if it has been rotated.



You can pass $\langle options \rangle$ to the node label node. For this, you provide the options in square brackets before the $\langle angle \rangle$. If you do so, you need to add braces around the whole argument of the label option and this is also the case if you have brackets or commas or semicolons or anything special in the $\langle text \rangle$.



```
\tikz \node [circle,draw,label={[red]above:X}] {my circle};
```



```
\begin{tikzpicture}
  \node [circle,draw,label={[name=label node]above left:$a,b$}] {};
  \draw (label node) -- +(1,1);
  \end{tikzpicture}
```

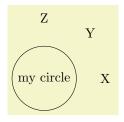
If you provide multiple label options, then multiple extra label nodes are added in the order they are given.

The following styles influence how labels are drawn:

/tikz/label distance= $\langle distance \rangle$

(no default, initially Opt)

The $\langle distance \rangle$ is additionally inserted between the main node and the label node.



/tikz/every label

(style, initially empty)

This style is used in every node created by the label option. The default is draw=none, fill=none.

16.10.3 The Pin Option

```
\tikz/pin=[\langle options \rangle] \langle angle \rangle : \langle text \rangle
```

(no default)

This option is quite similar to the label option, but there is one difference: In addition to adding an extra node to the picture, it also adds an edge from this node to the main node. This causes the node to look like a pin that has been added to the main node:



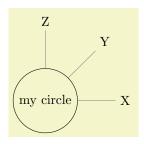
\tikz \node [circle,fill=blue!50,minimum size=1cm,pin=60:\$q_0\$] {};

The meaning of the $\langle options \rangle$ and the $\langle angle \rangle$ and the $\langle text \rangle$ is exactly the same as for the node option. Only, the options and styles the influence the way pins look are different:

/tikz/pin distance=⟨distance⟩

(no default, initially 3ex)

This $\langle distance \rangle$ is used instead of the label distance for the distance between the main node and the label node.



/tikz/every pin

(style, initially draw=none,fill=none)

This style is used in every node created by the pin option.

/tikz/pin position= $\langle angle \rangle$

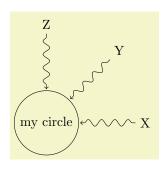
(no default, initially above)

The default pin position. Works like label position.

/tikz/every pin edge

(style, initially help lines)

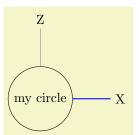
This style is used in every edge created by the pin options.

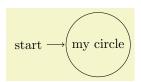


/tikz/pin edge= $\langle options \rangle$

(no default, initially empty)

This option can be used to set the options that are to be used in the edge created by the pin option.





16.10.4 The Quotes Syntax

The label and pin option provide a concise syntax for creating nodes next to existing nodes, but even this syntax is often a bit too verbose. By including the following library, you get access to an even more concise syntax:

```
\usetikzlibrary{quotes} % MEX and plain TeX \usetikzlibrary[quotes] % ConTeXt
```

Enables the quotes syntax for labels, pins and edge nodes.

Let us start with the basics of what this library does: Once loaded, inside the options of a **node** command, instead of the usual $\langle key \rangle = \langle value \rangle$ pairs, you may also provide strings of the following form (the actual syntax slightly more general, see the detailed descriptions later on):

```
"\langle text \rangle "\langle options \rangle
```

The $\langle options \rangle$ must be surrounded in curly braces when they contain a comma, otherwise the curly braces are optional. The may be preceded by an optional space.

When a $\langle string \rangle$ of the above form is encountered inside the options of a node, then it is internally transformed to

```
label=\{[\langle options \rangle] \langle text \rangle\}
```

Let us have a look at an example:

The above has the same effect as the following:

```
my label \tikz \node [label={[red]my label}, draw] {my node};

my node
```

Here are further examples, one where no $\langle options \rangle$ are added to the label, one where a position is specified, and examples with more complicated options in curly braces:

```
\begin{tikzpicture}
     label
                 \matrix [row sep=5mm] {
                   \node [draw, "label"]
                                                           {A}; \\
      Α
                   \node [draw, "label" left]
                                                           {B}; \\
                   \node [draw, "label" color=red]
                                                            {C}; \\
                   \node [draw, "label" {red,draw,thick}] {D}; \\
label B
              \end{tikzpicture}
     label
      С
     label
      D
```

Let us now have a more detailed look at what which commands this library provides:

```
/tikz/quotes mean label
```

(no value)

When this option is used (which is the default when this library is loaded), then, as described above, inside the options of a node a special syntax check is done.

The syntax. For each string in the list of options it is tested whether it starts with a quotation mark (note that this will never happen for normal keys since the normal keys of TikZ do not start with quotation marks). When this happens, the $\langle string \rangle$ should not be a key-value pair, but, rather, must have the form:

```
"\langle text \rangle", \langle options \rangle
```

(We will discuss the optional apostrophe in a moment. It is not really important for the current option, but only for edge labels, which are discussed later).

Transformation to a label option. When a $\langle string \rangle$ has the above form, it is treated (almost) as if you had written

```
label=\{[\langle options \rangle] \langle text \rangle\}
```

instead. The "almost" refers to the following additional feature: In reality, before the $\langle options \rangle$ are executed inside the label command, the direction keys above, left, below right and so on are redefined so that above is a shorthand for label position=90 and similarly for the other keys. The net effect is that in order to specify the position of the $\langle text \rangle$ relative to the main node you can just put something like left or above right inside the $\langle options \rangle$:

```
90° \tikz \node ["$90^\circ$" above, "$180^\circ$" left, circle, draw] {circle};
```

Alternatively, you can also use $\langle direction \rangle$: $\langle actual \ text \rangle$ as your $\langle text \rangle$. This works since the label command allows you to specify a direction at the beginning when it is separated by a colon:

```
90° \tikz \node ["90:$90^\circ$", "left:$180^\circ$", circle, draw] {circle};
```

Arguably, placing above or left behind the $\langle text \rangle$ seems more natural than having it inside the $\langle text \rangle$. In addition to the above, before the $\langle options \rangle$ are executed, the following style is also executed:

```
/tikz/every label quotes
```

(style, no value)

```
\tikz [every label quotes/.style=red] \node ["90:$90^\circ$", "left:$180^\circ$", circle, draw] {circle};
```

Handling commas and colons inside the text. The $\langle text \rangle$ may not contain a comma, unless it is inside curly braces. The reason is that the key handler separates the total options of a node along the commas it finds. So, in order to have text containing a comma, just add curly braces around either the comma or just around the whole $\langle text \rangle$:

```
yes, we can tikz \node ["{yes, we can}", draw] {foo};

foo
```

The same is true for a colon, only in this case you may need to surround specifically the colon by curly braces to stop the label option from interpreting everything before the colon as a direction:

```
yes: we can tikz \node ["yes{:} we can", draw] {foo};

foo
```

The optional apostrophe. Following the closing quotation marks in a $\langle string \rangle$ there may (but need not) be a single quotation mark (an apostrophe), possibly surrounded by whitespaces. If it is present, it is simply added to the $\langle options \rangle$ as another option (and, indeed, a single apostrophe is a legal option in TikZ, it is a shorthand for swap):

```
has the same effect as
String
"foo"'
                "foo" {'}
                "foo" {',red}
"foo", red
"foo" '{red}
                "foo" {',red}
"foo"{',red}
                "foo" {',red}
"foo"{red,'}
                "foo" {red,'}
"foo"{'red}
                "foo" {'red} (illegal; there is no key 'red)
"foo" red'
                "foo" {red'} (illegal; there is no key red')
```

/tikz/quotes mean pin

(no value)

This option has exactly the same effect as quotes mean label, only instead of transforming quoted text to the label option, they get transformed to the pin option:

```
\tikz [quotes mean pin]
\node ["$90^\circ$" above, "$180^\circ$" left, circle, draw] {circle};
```

Instead of every label quotes, the following style is executed with each such pin:

```
/tikz/every pin quotes
```

(style, no value)

If instead of labels or pins you would like quoted strings to be interpreted in a different manner, you can also define your own handlers:

```
/tikz/node quotes mean=\langle replacement\rangle
```

(no default)

This key allows you to define your own handler for quotes options. Inside the options of a node, whenever a key-value pair with the syntax

```
"\langle text \rangle", \langle options \rangle
```

is encountered, the following happens: The above string gets replaced by $\langle replacement \rangle$ where inside the $\langle replacement \rangle$ the parameter #1 is $\langle text \rangle$ and #2 is $\langle options \rangle$. If the apostrophe is present (see also the discussion of quotes mean label), the $\langle options \rangle$ start with ',.

The \(\lambda replacement \rangle \) is then parsed normally as options (using \pgfkeys).

Here is an example, where the quotes are used to define labels that are automatically named according to the text:

Some further options provided by the quotes library concern labels next to edges rather than nodes and they are described in Section 16.12.2.

16.11 Connecting Nodes: Using Nodes as Coordinates

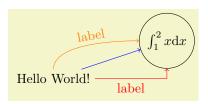
Once you have defined a node and given it a name, you can use this name to reference it. This can be done in two ways, see also Section 13.2.3. Suppose you have said \path(0,0) node(x) {Hello World!}; in order to define a node named x.

1. Once the node x has been defined, you can use (x.\(\langle anchor\)) wherever you would normally use a normal coordinate. This will yield the position at which the given \(\langle anchor\) is in the picture. Note that transformations do not apply to this coordinate, that is, (x.north) will be the northern anchor of x even if you have said scale=3 or xshift=4cm. This is usually what you would expect.

2. You can also just use (x) as a coordinate. In most cases, this gives the same coordinate as (x.center). Indeed, if the shape of x is coordinate, then (x) and (x.center) have exactly the same effect.

However, for most other shapes, some path construction operations like -- try to be "clever" when they are asked to draw a line from such a coordinate or to such a coordinate. When you say (x)--(1,1), the -- path operation will not draw a line from the center of x, but from the border of x in the direction going towards (1,1). Likewise, (1,1)--(x) will also have the line end on the border in the direction coming from (1,1).

In addition to --, the curve-to path operation .. and the path operations -| and |- will also handle nodes without anchors correctly. Here is an example, see also Section 13.2.3:



16.12 Connecting Nodes: Using the Edge Operation

16.12.1 Basic Syntax of the Edge Operation

The edge operation works like a to operation that is added after the main path has been drawn, much like a node is added after the main path has been drawn. This allows each edge to have a different appearance. As the node operation, an edge temporarily suspends the construction of the current path and a new path p is constructed. This new path p will be drawn after the main path has been drawn. Note that p can be totally different from the main path with respect to its options. Also note that if there are several to and/or node operations in the main path, each creates its own path(s) and they are drawn in the order that they are encountered on the path.

```
\path ... edge[\langle options \rangle] \langle nodes \rangle (\langle coordinate \rangle) ...;
```

The effect of the edge operation is that after the main path the following path is added to the picture:

```
\path[every edge, \langle options \rangle] (\tikztostart) \langle path \rangle;
```

Here, $\langle path \rangle$ is the to path. Note that, unlike the path added by the to operation, the (\tikztostart) is added before the $\langle path \rangle$ (which is unnecessary for the to operation, since this coordinate is already part of the main path).

The \tikztostart is the last coordinate on the path just before the edge operation, just as for the node or to operations. However, there is one exception to this rule: If the edge operation is directly preceded by a node operation, then this just-declared node is the start coordinate (and not, as would normally be the case, the coordinate where this just-declared node is placed – a small, but subtle difference). In this regard, edge differs from both node and to.

If there are several edge operations in a row, the start coordinate is the same for all of them as their target coordinates are not, after all, part of the main path. The start coordinate is, thus, the coordinate preceding the first edge operation. This is similar to nodes insofar as the edge operation does not modify the current path at all. In particular, it does not change the last coordinate visited, see the following example:

```
\begin{tikzpicture}
  \node (a) at
                (0:1) {$a$};
  \node (b) at (90:1) {$b$} edge [->]
                                             (a);
  \node (c) at (180:1) {$c$} edge [->]
                                             (a)
                                             (b);
                              edge
                                   [<-1
  \node (d) at (270:1) {$d$}
                                   [->]
                                             (a)
                              edge
                              edge [dotted]
                                             (b)
                              edge [<-]
                                             (c);
\end{tikzpicture}
```

A different way of specifying the above graph using the edge operation is the following:

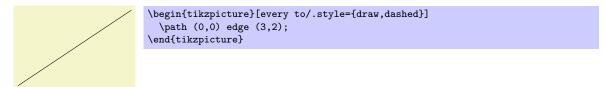
As can be seen, the path of the edge operation inherits the options from the main path, but you can locally overrule them.

```
\begin{tikzpicture}
  foreach \name/\angle in {a/0,b/90,c/180,d/270}
    \node (\nome) at (\angle:1.5) {<math>\nomes};
  \path[->] (b) edge
                                 node[above right] {$5$}
                                                                 (a)
                edge
                                                                 (c)
                      [-,dotted] node[below,sloped] {missing}
                edge
                                                                (d)
             (c) edge
                                                                 (a)
                                                                 (d)
                 edge
             (d) edge [red]
                                 node[above,sloped] {very}
                                 node[below,sloped] {bad}
                                                                 (a);
\end{tikzpicture}
```

Instead of every to, the style every edge is installed at the beginning of the main path.

```
/tikz/every edge (style, initially draw)
```

Executed for each edge.



16.12.2 Nodes on Edges: Quotes Syntax

The standard way of specifying nodes that are placed "on" an edge (or on a to-path; all of the following is also true for to-paths) is to put node specifications after the edge keyword, but before the target coordinate. Another way is to use the edge node option and its friends. Yet another way is to use the quotes syntax.

The syntax is essentially the same as for labels added to nodes as described in Section 16.10.4 and you also need to load the quotes library.

In detail, when the quotes library is loaded, each time a key-value pair in a list of options passed to an edge or a to path command starts with ", the key-value pair must actually be a string of the following form:

```
"\langle text \rangle", \langle options \rangle
```

This string is transformed into the following:

```
edge node=node [every edge quotes,\langle options \rangle]{\langle text \rangle}
```

As described in Section 16.10.4, the apostrophe becomes part of the $\langle options \rangle$, when present. The following style is important for the placement of the labels:

```
/tikz/every edge quotes
```

(style, initially auto)

This style is auto by default, which causes labels specified using the quotes-syntax to be placed next to the edges. Unless the setting of auto has been changed, they will be placed to the left.

```
left \tikz \draw (0,0) edge ["left", ->] (2,0);
```

In order to place all labels to the right by default, change this style to auto=right:

```
right \tikz [every edge quotes/.style={auto=right}] \draw (0,0) edge ["right", ->] (2,0);
```

To place all nodes "on" the edge, just make this style empty (and, possibly, make your labels opaque):

```
\tikz [every edge quotes/.style={fill=white,font=\footnotesize}] \draw (0,0) edge ["mid", ->] (2,1);
```

You may often wish to place some edge nodes to the right of edges and some to the left. For this, the special treatment of the apostrophe is particularly convenient: Recall that in TikZ there is an option just called ', which is a shorthand for swap. Now, following the closing quotation mark come the options of an edge node. Thus, if the closing quotation mark is followed by an apostrophe, the swap option will be added to the edge label, causing it is be placed on the other side. Because of the special treatment, you can even add another option like near end after the apostrophe without having to add curly braces and commas:

```
start left
right end

tikz
\draw (0,0) edge ["left", "right"',
"start" near start,
"end"' near end] (4,0);
```

In order to modify the distance between the edge labels and the edge, you should consider introducing some styles:

16.13 Referencing Nodes Outside the Current Picture

16.13.1 Referencing a Node in a Different Picture

It is possible (but not quite trivial) to reference nodes in pictures other than the current one. This means that you can create a picture and a node therein and, later, you can draw a line from some other position to this node.

To reference nodes in different pictures, proceed as follows:

- 1. You need to add the remember picture option to all pictures that contain nodes that you wish to reference and also to all pictures from which you wish to reference a node in another picture.
- 2. You need to add the overlay option to paths or to whole pictures that contain references to nodes in different pictures. (This option switches the computation of the bounding box off.)
- 3. You need to use a driver that supports picture remembering and you need to run TEX twice.

(For more details on what is going on behind the scenes, see Section 83.3.2.) Let us have a look at the effect of these options.

```
/tikz/remember picture=\langle boolean\rangle
```

(no default, initially false)

This option tells TikZ that it should attempt to remember the position of the current picture on the page. This attempt may fail depending on which backend driver is used. Also, even if remembering works, the position may only be available on a second run of T_EX .

Provided that remembering works, you may consider saying

```
\tikzstyle{every picture}+=[remember picture]
```

to make TikZ remember all pictures. This will add one line in the .aux file for each picture in your document – which typically is not very much. Then, you do not have to worry about remembered pictures at all.

/tikz/overlay (no value)

This option is mainly intended for use when nodes in other pictures are referenced, but you can also use it in other situations. The effect of this option is that everything within the current scope is not taken into consideration when the bounding box of the current picture is computed.

You need to specify this option on all paths (or at least on all parts of paths) that contain a reference to a node in another picture. The reason is that, otherwise, TikZ will attempt to make the current picture large enough to encompass the node in the other picture. However, on a second run of TeX this will create an even bigger picture, leading to larger and larger pictures. Unless you know what you are doing, I suggest specifying the overlay option with all pictures that contain references to other pictures.

Let us now have a look at a few examples. These examples work only if this document is processed with a driver that supports picture remembering.

Inside the current text we place two pictures, containing nodes named n1 and n2, using

```
\tikz[remember picture] \node[circle,fill=red!50] (n1) {};
```

which yields

```
\tikz[remember p\cture] \node[fill=blue!50] (n2) {};
```

, and

yielding the node . To connect these nodes, we create another picture using the overlay option and also the remember picture option.

```
\begin{tikzpicture}[remember picture,overlay]
  \draw[->,very thick] (n1) -- (n2);
\end{tikzpicture}
```

Note that the last picture is seemingly empty. What happens is that it has zero size and contains an arrow that lies well outside its bounds. As a last example, we connect a node in another picture to the first two nodes. Here, we provide the **overlay** option only with the line that we do not wish to count as part of the picture.

```
\begin{tikzpicture}[remember picture]
\node (c) [circle,draw] {Big circle};

\draw [overlay,->,very thick,red,opacity=.5]
(c) to[bend left] (n1) (n1) -| (n2);
\end{tikzpicture}
```

16.13.2 Referencing the Current Page Node – Absolute Positioning

There is a special node called current page that can be used to access the current page. It is a node of shape rectangle whose south west anchor is the lower left corner of the page and whose north east anchor is the upper right corner of the page. While this node is handled in a special way internally, you can reference it as if it were defined in some remembered picture other than the current one. Thus, by giving the remember picture and the overlay options to a picture, you can position nodes absolutely on a page.

The first example places some text in the lower left corner of the current page:

```
\begin{tikzpicture}[remember picture,overlay]
  \node [xshift=1cm,yshift=1cm] at (current page.south west)
        [text width=7cm,fill=red!20,rounded corners,above right]
  {
    This is an absolutely positioned text in the
    lower left corner. No shipout-hackery is used.
  };
  \end{tikzpicture}
```

The next example adds a circle in the middle of the page.

```
\begin{tikzpicture}[remember picture,overlay]
  \draw [line width=1mm,opacity=.25]
  (current page.center) circle (3cm);
  \end{tikzpicture}
```

The final example overlays some text over the page (depending on where this example is found on the page, the text may also be behind the page).

```
\begin{tikzpicture}[remember picture,overlay]
  \node [rotate=60,scale=10,text opacity=0.2]
  at (current page.center) {Example};
  \end{tikzpicture}
```

16.14 Late Code and Late Options

All options given to a node only locally affect this one node. While this is a blessing in most cases, you may sometimes want to cause options to have effects "later" on. The other way round, you may sometimes note "only later" that some options should be added to the options of a node. For this, the following version of the node path command can be used:

```
\path ... node also[\langle late options \rangle] (\langle name \rangle) ...
```

Note that the $\langle name \rangle$ is compulsory and that no text may be given. Also, the ordering of options and node label must be as above.

The effect of the above is the following effect: The node $\langle name \rangle$ must already be existing. Now, the $\langle late\ options \rangle$ are executed in a local scope. Most of these options will have no effect since you cannot change the appearance of the node, that is, you cannot change a red node into a green node using these "late" options. However, giving the append after command and prefix after command options inside the $\langle late\ options \rangle$ (directly or indirectly) does have the desired effect: The given path gets executed with the \tikzlastnode set to the determined node.

The net effect of all this is that you can provide, say, the label option inside the $\langle options \rangle$ to a add a label to a node that has already been constructed.

As explained in Section 14, you can use the options append after command and prefix after command to add a path after a node. The following macro may be useful there:

\tikzlastnode

Expands to the last node on the path.

Instead of the node also syntax, you can also the following option:

```
\texttt{/tikz/late options=}\langle options \rangle (no default)
```

This option can be given on a path (but not as an argument to a node path command) and has the same effect as the node also path command. Inside the $\langle options \rangle$, you should use the name option to specify the node for which you wish to add late options:



17 Specifying Graphs

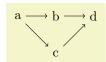
17.1 Overview

TikZ offers a powerful path command for specifying how the nodes in a graph are connected by edges and arcs: The graph path command, which becomes available when you load the graphs library.

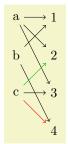
```
\usetikzlibrary{graphs} % MEX and plain TEX \usetikzlibrary[graphs] % ConTEXt
```

The package must be loaded to use the graph path command.

In this section, by graph we refer to a set of nodes together with some edges (sometimes also called arcs, in case they are directed) such as the following:

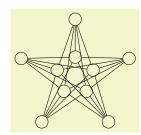


```
\tikz \graph { a -> {b, c} -> d };
```

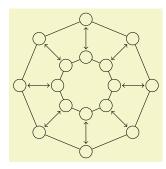


```
\tikz \graph {
  subgraph I_nm [V={a, b, c}, W={1,...,4}];

a -> { 1, 2, 3 };
  b -> { 1, 4 };
  c -> { 2 [>green!75!black], 3, 4 [>red]}
};
```



```
\tikz
\graph [nodes={draw, circle}, clockwise, radius=.5cm, empty nodes, n=5] {
   subgraph I_n [name=inner] --[complete bipartite]
   subgraph I_n [name=outer]
};
```



```
\tikz
\graph [nodes={draw, circle}, clockwise, radius=.75cm, empty nodes, n=8] {
   subgraph C_n [name=inner] <->[shorten <=1pt, shorten >=1pt]
   subgraph C_n [name=outer]
};
```

The nodes of a graph are normal TikZ nodes, the edges are normal lines drawn between nodes. There is nothing in the graph library that you cannot do using the normal \node and the edge command. Rather, its purpose is to offer a concise and powerful way of *specifying* which nodes are present and how they are connected. The graph library only offers simple methods for specifying where the nodes should be shown, its main strength is in specifying which nodes and edges are present in principle. The problem of finding "good positions on the canvas" for the nodes of a graph is left to *graph drawing algorithms*, which are covered in Section ?? and which are not part of the graphs library; indeed, these algorithms can be used also with graphs specified using node and edge commands. Again, the purpose of the graph command is mainly to specify which nodes and edges are present, nothing more and nothing less.

The graph library uses a syntax that is quite different from the normal TikZ syntax for specifying nodes. The reason for this is that for many medium-sized graphs it can become quite cumbersome to specify all the

nodes using \node repeatedly and then using a great number of edge command; possibly with complicated \foreach statements. Instead, the syntax of the graph library is loosely inspired by the DOT format, which is quite useful for specifying medium-sized graphs, with some extensions on top.

17.2 Concepts

The present section aims at giving a quick overview of the main concepts behind the graph command. The exact syntax is explained in more detail in later sections.

17.2.1 Concept: Node Chains

The basic way of specifying a graph is to write down a node chain as in the following example:

```
foo bar blub \tikz [every node/.style = draw] \graph { foo -> bar -> blub };
```

As can be seen, the text foo -> bar -> my node creates three nodes, one with the text foo, one with bar and one with the text blub. These nodes are connected by arrows, which are caused by the -> between the node texts. Such a sequence of node texts and arrows between them is called a *chain* in the following. Inside a graph there can be more than one chain:

```
\begin{array}{c} a \longrightarrow b \longrightarrow c \\ \\ d \longrightarrow e \longrightarrow f \\ \end{array} \begin{array}{c} \text{\tikz } graph \ \{\\ \\ a \rightarrow b \rightarrow c;\\ \\ d \rightarrow e \rightarrow f;\\ \\ g \rightarrow f;\\ \}; \end{array}
```

Multiple chains are separated by a semicolon or a comma (both have exactly the same effect). As the example shows, when a node text is seen for the second time, instead of creating a new node, a connection is created to the already existing node.

When a node like f is created, both the node name and the node text are identical by default. This is not always desirable and can be changed by using the as key or by providing another text after a slash:

When you wish to use a node name that contains special symbols like commas or dashes, you must surround the node name by quotes. This allows you to use quite arbitrary text as a "node name":

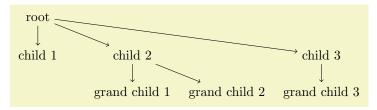
```
\tikz \graph {
    "$x_1$" -> "$x_2$"[red] -> "$x_3,x_4$";
    "$x_1$" -> [bend left] "$x_3,x_4$";
};
```

17.2.2 Concept: Chain Groups

Multiple chains that are separated by a semicolon or a comma and that are surrounded by curly braces form what will be called a *chain group* or just a *group*. A group in itself has no special effect. However, things get interesting when you write down a node or even a whole group and connect it to another group. In this case, the "exit points" of the first node or group get connected to the "entry points" of the second node or group:

```
\begin{array}{c} a \longrightarrow b \longrightarrow c \longrightarrow f \\ d \longrightarrow e \end{array} \begin{array}{c} \text{\tikz \graph } \{\\ a \rightarrow \{\\ b \rightarrow c,\\ d \rightarrow e\\ \} \rightarrow f\\ \}; \end{array}
```

Chain groups make it easy to create tree structures:



```
\tikz \graph [grow down, branch right=2.5cm] {
  root -> {
    child 1,
    child 2 -> {
      grand child 1,
      grand child 2
    },
    child 3 -> {
      grand child 3
    }
}
```

As can be seen, the placement is not particularly nice by default, use the algorithms from the graph drawing libraries to get a better layout.

17.2.3 Concept: Edge Labels and Styles

When connectors like -> or -- are used to connect nodes or whole chain groups, one or more edges will typically be created. These edges can be styles easily by providing options in square brackets directly after these connectors:

```
a \longrightarrow b \longrightarrow c
\begin{array}{c} \text{d} \\ \text{d} \\ \text{d} \\ \text{d} \\ \text{tikz } \text{ } \text{graph } \{\\ \text{a } -> [\text{red}] \text{ } \text{b } -- [\text{thick}] \text{ } \{\text{c, d}\};\\ \text{} \}; \\ \end{array}
```

Using the quotes syntax, see Section 16.10.4, you can even add labels to the edges easily by putting the labels in quotes:

```
a foo b bar c bar d tikz \graph {
    a ->[red, "foo"] b --[thick, "bar"] {c, d};
};
```

For the first edge, the effect is as desired, however between b and the group {c,d} two edges are inserted and the options thick and the label option "bar" is applied to both of them. While this is the correct and consistent behaviour, we typically might wish to specify different labels for the edge going from b to c and the edge going from b to d. To achieve this effect, we can no longer specify the label as part of the options of --. Rather, we must pass the desired label to the nodes c and d, but we must somehow also indicate that these options actually "belong" to the edge "leading to" to nodes. This is achieved by preceeding the options with a greater-than sign:

```
a → b foo c | \tikz \graph { | a → b -- {c [> "foo"], d [> "bar"']}; | }; | bar d |
```

Symmetrically, preceeding the options by < causes the options and labels to apply to the "outgoing" edges of the node:

This syntax allows you to easily create trees with special edge labels as in the following example of a treap:

```
\tikz
\graph [edge quotes={fill=white,inner sep=1pt},
grow down, branch right, nodes={circle,draw}] {

"" -> h [>"9"] -> {

c [>"4"] -> {

a [>"2"],
e [>"0"]
},
j [>"7"]
};

};
```

17.2.4 Concept: Node Sets

When you write down some node text inside a graph command, a new node is created by default unless this node has already been created inside the same graph command. In particular, if a node has already been declared outside of the current graph command, a new node of the same name gets created.

This is not always the desired behaviour. Often, you may wish to make nodes part of a graph than have already been defined prior to the use of the graph command. For this, simply surround a node name by parentheses. This will cause a reference to be created to an already existing node:

```
A → B → C

\tikz {
\node (a) at (0,0) {A};
\node (b) at (1,0) {B};
\node (c) at (2,0) {C};

\graph { (a) → (b) → (c) };
}
```

You can even go a step further: A whole collection of nodes can all be flagged to belong to a *node set* by adding the option $set=\langle node\ set\ name \rangle$. Then, inside a graph command, you can collectively refer to these nodes by surrounding the node set name in parentheses:

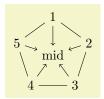
```
\tikz [new set=my nodes] {
    \node [set=my nodes, circle, draw] at (1,1) {A};
    \node [set=my nodes, rectangle, draw] at (1.5,0) {B};
    \node [set=my nodes, diamond, draw] at (1,-1) {C};
    \node (d) [star, draw] at (3,0) {D};

    \graph { X -> (my nodes) -> (d) };
}
```

17.2.5 Concept: Graph Macros

Often, a graph will consist – at least in parts – of standard parts. For instance, a graph might contain a cycle of certain size or a path or a clique. To facilitate specifying such graphs, you can define a *graph macro*. Once a graph macro has been defined, you can use the name of the graph to make a copy of the graph part of the graph currently being specified:

```
\tikz \graph { subgraph K_n [n=6, clockwise] };
```



```
\tikz \graph { subgraph C_n [n=5, clockwise] -> mid };
```

The library graphs.standard defines a number of such graphs, including the complete clique K_n on n nodes, the complete bipartite graph $K_{n,m}$ with shores sized n and m, the cycle C_n on n nodes, the path P_n on n nodes, and the independent set I_n on n nodes.

17.2.6 Concept: Graph Expressions and Color Classes

When a graph is being constructed using the graph command, it is constructed recursively by uniting smaller graphs to larger graphs. During this recursive union process the nodes of the graph get implicitly *colored* (conceptually) and you can also explicitly assign colors to individual nodes and even change the colors as the graph is being specified. All nodes having the same color form what is called a *color class*.

The power of color class is that special *connector operators* allow you to add edges between nodes having certain colors. For instance, saying clique=red at the beginning of a group will cause all nodes that have been flagged as being (conceptually) "red" to be connected as a clique. Similarly, saying complete bipartite={red}{green} will cause edges to be added between all red and all green nodes. More advanced connectors, like the butterfly connector, allow you to add edges between color classes in a fancy manner.

```
\tikz [x=8mm, y=6mm, font=\footnotesize, circle]
\graph [nodes={fill=blue!70, text=white}, empty nodes, n=8] {
    subgraph I_n [name=A] --[butterfly={level=4}]
    subgraph I_n [name=B] --[butterfly={level=2}]
    subgraph I_n [name=C] --[butterfly]
    subgraph I_n [name=D] --
    subgraph I_n [name=E]
};
```

17.3 Syntax of the Graph Path Command

17.3.1 The Graph Command

In order to construct a graph, you should use the **graph** path command, which can be used anywhere on a path at any place where you could also use a command like, say, **plot** or --.

\graph

Inside a {tikzpicture} this is an abbreviation for \path graph.

```
\path ... graph[\langle options \rangle] \langle group \ specification \rangle ...;
```

When this command is encountered on a path, the construction of the current path is suspended (similarly to an edge command or a node command). In a local scope, the $\langle options \rangle$ are first executed with the key path /tikz/graphs using the following command:

```
\texttt{\tikzgraphsset}\{\langle options \rangle\}
```

Executes the *(options)* with the path prefix /tikz/graphs.

Apart from the keys explained in the following, further permissible keys will be listed during the course of the rest of this section.

/tikz/graphs/every graph

(style, no value)

This style is executed at the beginning of every graph path command prior to the \(options \).

Once the scope has been set up and once the $\langle options \rangle$ have been executed, a parser starts to parse the $\langle group\ specification \rangle$. The exact syntax of such a group specification in explained in detail in Section 17.3.2. Basically, a group specification is a list of chain specifications, separated by commas or semicolons.

Depending on the content of the $\langle group\ specification \rangle$, two things will happen:

- 1. A number of new nodes may be created. These will be inserted into the picture in the same order as if they had been created using multiple node path commands at the place where the graph path command was used. In other words, all nodes created in a graph path command will be painted on top of any nodes created earlier in the path and behind any nodes created later in the path. Like normal nodes, the newly created nodes always lie on top of the path that is currently being created (which is often empty, for instance when the \graph command is used).
- 2. Edges between the nodes may be added. They are added in the same order as if the edge command had been used at the position where the graph command is being used.

Let us now have a look at some common keys that may be used inside the $\langle options \rangle$:

/tikz/graphs/nodes=\langle options\rangle

(no default)

This option causes the $\langle options \rangle$ to be applied to each newly created node inside the $\langle group specification \rangle$.

```
a \longrightarrow b \longrightarrow c \tikz \graph [nodes=red] { a -> b -> c };
```

Multiple uses of this key accumulate.

/tikz/graphs/edges=\langle options\rangle

(no default)

This option causes the $\langle options \rangle$ to be applied to each newly created edge inside the $\langle group specification \rangle$.

```
a \longrightarrow b \longrightarrow c \qquad \text{$$ \text{ (edges=\{red,thick}) { a -> b -> c };}
```

Again, multiple uses of this key accumulate.

```
/tikz/graphs/edge=\langle options\rangle
```

(no default)

This is an alias for edges.

```
/tikz/graphs/edge node=\langle node specification\rangle
```

(no default)

This key specifies that the $\langle node\ specification \rangle$ should be added to each newly created edge as an implicitly placed node.

```
a \longrightarrow b \longrightarrow c \tikz \graph [edge node={node [red, near end] {X}}] { a -> b -> c };
```

Again, multiple uses of this key accumulate.

```
a YX b YX c | \tikz \graph [edge node={node [near end] {X}}, edge node={node [near start] {Y}}] { a -> b -> c };
```

/tikz/graphs/edge label= $\langle text \rangle$

(no default)

This key is an abbreviation for edge node=node[auto]{ $\langle text \rangle$ }. The net effect is that the text is placed next to the newly created edges.

```
a \xrightarrow{X} b \xrightarrow{X} c
d
\tikz \graph [edge label=x] { a -> b -> {c,d} };
```

```
/tikz/graphs/edge label'=\langle text \rangle
```

(no default)

This key is an abbreviation for edge node=node[auto,swap] $\{\langle text \rangle\}$.

```
\tikz \graph [edge label=out, edge label'=in]
{ subgraph C_n [clockwise, n=5] };
```

17.3.2 Syntax of Group Specifications

A (group specification) inside a graph path command has the following syntax:

```
\{ [\langle options \rangle] \langle list \ of \ chain \ specifications \rangle \}
```

The $\langle chain\ specifications \rangle$ must contain chain specifications, whose syntax is detailed in the next section, separated by either commas or semicolons; you can freely mix them. It is permissible to use empty lines (which are mapped to \par commands internally) to structure the chains visually, they are simply ignored by the parser.

In the following example, the group specification consists of three chain specifications, namely of $a \rightarrow b$, then c alone, and finally $d \rightarrow e \rightarrow f$:

```
a → b

\tikz \graph {
    a → b,
    c;

c

d → e → f
```

The above has the same effect as the more compact group specification {a->b,c,d->e->f}.

Commas are used to detect where chain specifications end. However, you will often wish to use a comma also inside the options of a single node like in the following example:

Note that the above example works as expected: The first comma inside the option list of **a** is not interpreted as the end of the chain specification "**a** [red". Rather, commas inside square brackets are "protected" against being interpreted as separators of group specifications.

The $\langle options \rangle$ that can be given at the beginning of a group specification are local to the group. They are executed with the path prefix /tikz/graphs. Note that for the outermost group specification of a graph it makes no difference whether the options are passed to the graph command or whether they are given at the beginning of this group. However, for groups nested inside other groups, it does make a difference:

```
a b d

a -> { [nodes=red] % the option is local to these nodes:
b, c
} ->
d
};
```

Using foreach. There is special support for the \foreach statement inside groups: You may use the statement inside a group specification at any place where a \(\chin \) specification \(\chi \) would normally go. In this case, the \foreach statement is executed and for each iteration the content of the statement's body is treated and parsed as a new chain specification.

```
\begin{array}{c} a \longrightarrow a_1 \longrightarrow x_1 \\ y_1 \\ a_2 \longrightarrow x_2 \\ y_2 \\ a_3 \longrightarrow x_3 \\ b \end{array} \begin{array}{c} \text{ \thin } \{x_1, x_2, x_3\} \\ x_4 \longrightarrow x_2 \\ x_5 \longrightarrow x_4 \\ x_5 \longrightarrow x_4 \\ x_5 \longrightarrow x_5 \\ x_7 \longrightarrow x_8 \longrightarrow x_8 \\ x_7 \longrightarrow x_8 \longrightarrow x_8 \\ x_8 \longrightarrow x_8 \longrightarrow x_8 \\ x_9 \longrightarrow x_8 \longrightarrow x_8 \longrightarrow x_8 \\ x_9 \longrightarrow x_8 \longrightarrow x_8 \longrightarrow x_8 \longrightarrow x_8 \\ x_9 \longrightarrow x_9 \longrightarrow
```

Using macros. In some cases you may wish to use macros and T_EX code to compute which nodes and edges are present in a group. You cannot use macros in the normal way inside a graph specification since the parser does not expand macros as it scans for the start and end of groups and node names. Rather, only after commas, semicolons, and hyphens have already been detected and only after all other parsing decisions have been made will macros be expanded. At this point, when a macro expands to, say a,b, this will not result in two nodes to be created since the parsing is already done. For these reasons, a special key is needed to make it possible to "compute" which nodes should be present in a group.

```
tikz/graph/parse=\langle text \rangle (no default)
```

This key can only be used inside the $\langle options \rangle$ of a $\langle group\ specification \rangle$. Its effect is that the $\langle text \rangle$ is inserted at the beginning of the current group as if you had entered it there. Naturally, it makes little sense to just write down some static $\langle text \rangle$ since you could just as well directly place it at the beginning of the group. The real power of this command stems from the fact that the keys mechanism allows you to say, for instance, parse/.expand once to insert the text stored in some macro into the group.

```
a \longrightarrow b \longrightarrow c \qquad \begin{array}{c} \text{ $def\mychain{ a -> b -> c; }} \\ \text{tikz $graph { [parse/.expand once=\mychain] d -> e };} \\ d \longrightarrow e \end{array}
```

In the following, more fancy example we use a loop to create a chain of dynamic length.

```
1 ---> 2 ---> 3 ---> 4

\def\mytext{1}
\foreach \i in \{2,...,#1} \{
\xdef\mytext\{\mytext -> \i}
\}

\tikzgraphsset\{my \chain/.\style=\{
\utils/\exec=\my\chain\{#1\},
\uparse/.\expand \once=\my\text\}
\tikz \graph \{ \[my \chain=4] \};
\end{array}
```

Multiple uses of this key accumulate, that is, all the texts given in the different uses is inserted in the order it is given.

17.3.3 Syntax of Chain Specifications

A $\langle chain\ specification \rangle$ has the following syntax: It consists of a sequence of $\langle node\ specifications \rangle$, where subsequent node specifications are separated by $\langle edge\ specifications \rangle$. Node specifications, which typically consist of some text, are discussed in the next section in more detail. They normally represent a single node that is either newly created or exists already, but they may also specify a whole set of nodes.

An $\langle edge\ specification \rangle$ specifies which of the node(s) to the left of the edge specification should be connected to which node(s) to the right of it and it also specifies in which direction the connections go. In the following, we only discuss how the direction is chosen, the powerful mechanism behind choosing which nodes should be connect is detailed in Section 17.6.

The syntax of an edge specification is always one of the following five possibilities:

```
-> [⟨options⟩]
-- [⟨options⟩]
```

 $\leftarrow [\langle options \rangle]$

```
\leftarrow [\langle options \rangle] -!- [\langle options \rangle]
```

The first four correspond to a directed edge, an undirected edge, a "backward" directed edge, and a bidirected edge, respectively. The fifth edge specification means that there should be no edge (this specification can be used together with the simple option to remove edges that have previously been added, see Section 17.4).

Suppose the nodes $\langle left \ nodes \rangle$ are to the left of the $\langle edge \ specification \rangle$ and $\langle right \ nodes \rangle$ are to the right and suppose we have written -> between them. Then the following happens:

- 1. The *(options)* are executed (inside a local scope) with the path /tikz/graphs. These options may setup the connector algorithm (see below) and may also use keys like edge or edge label to specify how the edge should look like. As a convenience, whenever an unknown key is encountered for the path /tikz/graphs, the key is passed to the edge key. This means that you can directly use options like thick or red inside the *(options)* and they will apply to the edge as expected.
- 2. The chosen connector algorithm, see Section 17.6, is used to compute from which of the $\langle left \ nodes \rangle$ an edge should lead to which of the $\langle right \ nodes \rangle$. Suppose that $(l_1, r_1), \ldots, (l_n, r_n)$ is the list of node pairs that result (so there should be an edge between l_1 and r_1 and another edge between l_2 and r_2 and so on).
- 3. For each pair (l_i, r_i) an edge is created. This is done by calling the following key (for the edge specification ->, other keys are executed for the other kinds of specifications):

```
\tikz/graphs/new ->= {\langle left\ node \rangle} {\langle right\ node \rangle} {\langle edge\ options \rangle} {\langle edge\ nodes \rangle}  (no default)
```

This key will be called for a -> edge specification with the following four parameters:

- (a) $\langle left \ node \rangle$ is the name of the "left" node, that is, the name of l_i .
- (b) $\langle right \ node \rangle$ is the name of the right node.
- (c) \(\langle edge \ options \rangle \) are the accumulated options from all calls of \(\forall \text{tikz/graph/edges} \) in groups that surround the edge specification.
- (d) $\langle edge\ nodes \rangle$ is text like node {A} node {B} that specifies some nodes that should be put as labels on the edge using TikZ's implicit positioning mechanism.

By default, the key executes the following code:

```
\path [draw,->,every new ->,\langle edge\ options \rangle] (\langle left\ node \rangle\tikzgraphleftanchor) to \langle edge\ nodes \rangle (\langle right\ node \rangle\tikzgraphrightanchor);
```

You are welcome to change the code underlying the key.

```
/tikz/every new ->
```

This key gets executed by default for a new ->.

/tikz/graphs/left anchor=\(anchor\)

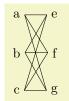
(no default)

(style, no value)

This anchor is used for the node that is to the left of an edge specification. Setting this anchor to the empty string means that no special anchor is used (which is the default). The $\langle anchor \rangle$ is stored in the macro \tikzgraphleftanchor with a leading dot.

```
\begin{array}{ccc}
a \longrightarrow e \\
b \longrightarrow f \\
c \longrightarrow g
\end{array}
```

```
\tikz \graph {
    {a,b,c} -> [complete bipartite] {e,f,g}
};
```



```
\tikz \graph [left anchor=east, right anchor=west] {
    {a,b,c} -- [complete bipartite] {e,f,g}
};
```

```
/tikz/graphs/right anchor=\lanchor\rangle
                                                                                                              (no default)
     Works like left anchor, only for \tikzgraphrightanchor.
For the other three kinds of edge specifications, the following keys will be called:
\tikz/graphs/new --= {\langle left\ node \rangle} {\langle right\ node \rangle} {\langle edge\ options \rangle} {\langle edge\ nodes \rangle}
                                                                                                             (no default)
     This key is called for -- with the same parameters as above. The only difference in the definition
     is that in the \path command the -> gets replaced by -.
     /tikz/every new --
                                                                                                         (style, no value)
\tikz/graphs/new <->=\{\langle left\ node \rangle\}\{\langle right\ node \rangle\}\{\langle edge\ options \rangle\}\{\langle edge\ nodes \rangle\}
                                                                                                              (no default)
     Called for <-> with the same parameters as above. The -> is replaced by <-
     /tikz/every new <->
                                                                                                         (style, no value)
\tikz/graphs/new \leftarrow = {\langle left\ node \rangle} {\langle right\ node \rangle} {\langle edge\ options \rangle} {\langle edge\ nodes \rangle}
                                                                                                              (no default)
     Called for <- with the same parameters as above.<sup>9</sup>
     /tikz/every new <-
                                                                                                         (style, no value)
```

Here is an example that shows the default rendering of the different edge specifications:

 $\tikz/graphs/new -!-=\{\langle left\ node \rangle\}\{\langle right\ node \rangle\}\{\langle edge\ options \rangle\}\{\langle edge\ nodes \rangle\}$

Called for -!- with the same parameters as above. Does nothing by default.

(no default)

17.3.4 Syntax of Node Specifications

Node specifications are the basic building blocks of a graph specification. There are three different possible kinds of node specifications, each of which has a different syntax:

Direct Node Specification

```
"\langle node\ name \rangle"/"\langle text \rangle" [\langle options \rangle] (note that the quotation marks are optional and only needed when the \langle node\ name \rangle contains special symbols)
```

Reference Node Specification

($\langle node \ name \ or \ node \ set \ name \rangle$)

Group Node Specification

 $\langle group \ specification \rangle$

The rule for determining which of the possible kinds is meant is as follows: If the node specification starts with an opening parenthesis, a reference node specification is meant; if it starts with an opening curly brace, a group specification is meant; and in all other cases a direct node specification is meant.

Direct Node Specifications. If after reading the first symbol of a node specification is has been detected to be *direct*, TikZ will collect all text up to the next edge specification and store it as the $\langle node name \rangle$; however, square brackets are used to indicate options and a slash ends the $\langle node name \rangle$ and start a special $\langle text \rangle$ that is used as a rendering text instead of the original $\langle node name \rangle$.

Due to the way the parsing works and due to the restrictions on node names, most special characters are forbidding inside the $\langle node\ name \rangle$, including commas, semicolons, hyphens, braces, dots, parentheses,

⁹You might wonder why this key is needed: It seems more logical at first sight to just call **new edge directed** with swapped first parameters. However, a positioning algorithm might wish to take the fact into account that an edge is "backward" rather than "forward" in order to improve the layout. Also, different arrow heads might be used.

slashes, dashes, and more (but spaces, single underscores, and the hat character are allowed). To use special characters in the name of a node, you can optionally surround the $\langle node\ name \rangle$ and/or the $\langle text \rangle$ by quotation marks. In this case, you can use all of the special symbols once more. The details of what happens, exactly, when the $\langle node\ name \rangle$ is surrounded by quotation marks is explained later; surrounding the $\langle text \rangle$ by quotation marks has essentially the same effect as surrounding it by curly braces.

Once the node name has been determined, it is checked whether the same node name was already used inside the current graph. If this is the case, then we say that the already existing node is *referenced*; otherwise we say that the node is *fresh*.

```
a → b \tikz \graph {
    a → b; % both are fresh
    c → a; % only c is fresh, a is referenced
};
```

This behaviour of deciding whether a node is fresh or referenced can, however, be modified by using the following keys:

```
/tikz/graphs/use existing node=⟨true or false⟩ (default true)
```

When this key is set to true, all nodes will be considered to the referenced, no node will be fresh. This option is useful if you have already created all the nodes of a graph prior to using the graph command and you now only wish to connect the nodes.

```
/tikz/graphs/fresh nodes=\(\text{true or false}\) (default true)
```

When this key is set to true, all nodes will be considered to be fresh. This option is useful when you create for instance a tree with many identical nodes.

When a node name is encountered that was already used previously, a new name is chosen is follows: An apostrophe (') is appended repeatedly until a node name is found that has not yet been used:

When a fresh node has been detected, a new node is created in the inside a protecting scope. For this, the current placement strategy is asked to compute a default position for the node, see Section 17.8 for details. Then, the command

```
\node (\langle full\ node\ name \rangle) [\langle node\ options \rangle] {\langle text \rangle};
```

is called. The different parameters are as follows:

• The $\langle full\ node\ name \rangle$ is normally the $\langle node\ name \rangle$ that has been determined as described before. However, there are two exceptions:

First, if the $\langle node\ name \rangle$ is empty (which happens when there is no $\langle node\ name \rangle$ before the slash), then a fresh internal node name is created and used as $\langle full\ node\ name \rangle$. This name is guaranteed to be different from all node names used in this or any other graph. Thus, a direct node starting with a slash represents an anonymous fresh node.

Second, you can use the following key to prefix the $\langle node\ name \rangle$ inside the $\langle full\ node\ name \rangle$:

```
/\text{tikz/graphs/name} = \langle text \rangle (no default)
```

This key prepends the $\langle text \rangle$, followed by a separating symbol (a space by default), to all $\langle node \ name \rangle$ s inside a $\langle full \ node \ name \rangle$. Repeated calls of this key accumulate, leading to ever-longer "name paths":

Note that, indeed, in the above example six nodes are created even though the first and second set of nodes have the same $\langle node\ name \rangle$. The reason is that the full names of the six nodes are all different. Also note that only the $\langle node\ name \rangle$ is used as the node text, not the full name. This can be changed as described later on.

This key can be used repeately, leading to ever longer node names.

```
/tikz/graphs/name separator=\langle symbols \rangle
```

(no default, initially \space)

Changes the symbol that is used to separate the $\langle text \rangle$ from the $\langle node\ name \rangle$. The default is \space, resulting in a space.

- The $\langle node \ options \rangle$ are
 - 1. The options that have accumulated in calls to nodes from the surrounding scopes.
 - 2. The local $\langle options \rangle$.

The options are executed with the path prefix /tikz/graphs, but any unknown key is executed with the prefix /tikz. This means, in essence, that some esoteric keys are more difficult to use inside the options and that any key with the prefix /tikz/graphs will take precedence over a key with the prefix /tikz.

• The $\langle text \rangle$ that is passed to the \node command is computed as follows: First, you can use the following key to directly set the $\langle text \rangle$:

```
/\text{tikz/graphs/as}=\langle text \rangle (no default)
```

The $\langle text \rangle$ is used as the text of the node. This allows you to provide a text for the node that differs arbitrarily from the name of the node.

```
x \longrightarrow y_5 \rightarrow a-b \tikz \graph { a [as=$x$] -- b [as=$y_5$] -> c [red, as={a--b}] };
```

This key always takes precedence over all of the mechanisms described below.

In case the as key is not used, a default text is chosen as follows: First, when a direct node specification contains a slash (or, for historical reasons, a double underscore), the text to the right of the slash (or double underscore) is stored in the macro \tikzgraphnodetext; if there is no slash, the $\langle node\ name \rangle$ is stored in \tikzgraphnodetext, instead. Then, the current value of the following key is used as $\langle text \rangle$:

/tikz/graphs/typeset= $\langle code \rangle$

(no default)

The macro or code stored in this key is used as the $\langle text \rangle$ if the node. Inside the $\langle code \rangle$, the following macros are available:

\tikzgraphnodetext

This macro expands to the $\langle text \rangle$ to the right of the double underscore or slash in a direct node specification or, if there is no slash, to the $\langle node\ name \rangle$.

\tikzgraphnodename

This macro expands to the name of the current node with the path.

\tikzgraphnodepath

This macro expands to the current path of the node. These paths result from the use of the name key as described above.

\tikzgraphnodefullname

This macro contains the concatenation of the above two.

By default, the typesetter is just set to \tikzgraphnodetext, which means that the default text of a node is its name. However, it may be useful to change this: For instance, you might wish that the text of all graph nodes is, say, surrounded by parentheses:

A more advanced macro might take apart the node text and render it differently:

The following styles install useful predefined typesetting macros:

/tikz/graphs/empty nodes

(no value)

Just sets typeset to nothing, which causes all nodes to have an empty text (unless, of course, the as option is used):



```
\tikz \graph [empty nodes, nodes={circle, draw}] { a -> {b, c} };
```

/tikz/graphs/math nodes

(no value)

Sets typeset to \$\tikzgraphnodetext\$, which causes all nodes names to be typeset in math mode:



```
\tikz \graph [math nodes, nodes={circle, draw}] { a_1 -> {b^2, c_3^n} };
```

If a node is referenced instead of fresh, then this node becomes the node that will be connected by the preceding or following edge specification to other nodes. The $\langle options \rangle$ are executed even for a referenced node, but they cannot be used to change the appearance of the node (because the node exists already). Rather, the $\langle options \rangle$ can only be used to change the logical coloring of the node, see Section 17.6 for details.

Quoted Node Names. When the $\langle node \ name \rangle$ and/or the $\langle text \rangle$ of a node is surrounded by quotation marks, you can use all sorts of special symbols as part of the text that are normally forbidden:

```
| The last text | The last tex
```

In detail, for the following happens when qutation marks are encountered at the beginning of a node name or its text:

- Everthing following the quotation mark up to the next single quotation mark is collected into a macro $\langle collected \rangle$. All sorts of special characters, including commas, square brackets, dashes, and even backslashes are allowed here. Basically, the only restriction is that braces must be balanced.
- A double quotation mark ("") does not count as the "next single quotation mark." Rather, it is replaced by a single quotation mark. For instance, "He said, ""Hello world."" would be stored inside (collected) as He said, "Hello world." However, this rule applies only on the outer-most level of braces. Thus, in

```
"He {said, ""Hello world.""}"
```

we would get He {said, ""Hello world.""} as $\langle collected \rangle$.

• "The next single quotation mark" refers to the next quotation mark on the current level of braces, so in "hello {"} world", the next quotation mark would be the one following world.

Now, once the $\langle collected \rangle$ text has been gather, it is used as follows: When used as $\langle text \rangle$ (what is acutally displayed), it is just used "as is". When it is used as $\langle node\ name \rangle$, however, the following happens: Every "special character" in $\langle collected \rangle$ is replaced by its Unicode name, surrounded by Q-signs. For instance, if $\langle collected \rangle$ is Hello, world!, the $\langle node\ name \rangle$ is the somewhat longer text HelloQCOMMAQ worldQEXCLAMATION MARKQ. Admittely, referencing such a node from outside the graph is cumbersome, but when you use exactly the same $\langle collected \rangle$ text once more, the same $\langle node\ name \rangle$ will result. The following characters are considered "special":

```
|$&^~_[](){}/.-,+*','!":;<=>?@#%\{}
```

These are exactly the Unicode character with a decimal code number between 33 and 126 that are neither digits nor letters.

Reference Node Specifications. A reference node specification is a node specification that starts with an opening parenthesis. In this case, parentheses must surround a $\langle name \rangle$ as in (foo), where foo is the $\langle name \rangle$. The following will now happen:

- 1. It is tested whether $\langle name \rangle$ is the name of a currently active *node set*. This case will be discussed in a moment.
- 2. Otherwise, the $\langle name \rangle$ is interpreted and treated as a referenced node, but independently of whether the node has already been fresh in the current graph or not. In other words, the node must have been defined either already inside the graph (in which case the parenthesis are more or less superfluous) or it must have been defined outside the current picture.

The way the referenced node is handled is the same way as for a direct node that is a referenced node. If the node does not already exist, an error message is printed.

Let us now have a look at node sets. Inside a {tikzpicture} you can locally define a *node set* by using the following key:

```
/\text{tikz/new set} = \langle set \ name \rangle (no default)
```

This will setup a node set named $\langle set\ name \rangle$ within the current scope. Inside the scope, you can add nodes to the node set using the **set** key. If a node set of the same name already exists in the current scope, it will be reset and made empty for the current scope.

Note that this command has the path /tikz and is normally used outside the graph command.

 $\text{/tikz/set} = \langle set \ name \rangle$ (no default)

This key can be used as an option with a **node** command. The $\langle set \ name \rangle$ must be the name of a node set that has previously been created inside some enclosing scope via the **new set** key. The effect is that the current node is added to the node set.

When you use a graph command inside a scope where some node set called $\langle set \ name \rangle$ is defined, then inside this graph command you use ($\langle set \ name \rangle$) to reference all of the nodes in the node set. The effect is the same as if instead of the reference to the set name you had created a group specification containing a list of references to all the nodes that are part of the node set.

```
begin{tikzpicture}[new set=red, new set=green, shorten >=2pt]

\foreach \i in \{1,2,3\} \{
    \node [draw, red!80, set=red] (r\i) at (\i,1) \{\fr_\i\};
    \node [draw, green!50!black, set=green] (g\i) at (\i,2) \{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\f
```

There is an interesting caveat with referencing node sets: Suppose that at the beginning of a graph you just say (foo); where foo is a set name. Unless you have specified special options, this will cause the following to happen: A group is created whose members are all the nodes of the node set foo. These nodes become referenced nodes, but otherwise nothing happens since, by default, the nodes of a group are not connected automatically. However, the referenced nodes have now been referenced inside the graph, you can thus subsequently access them as if they had been defined inside the graph. Here is an example showing how you can create nodes outside a graph command and then connect them inside as if they had been declared inside:

Group Node Specifications. At a place where a node specification should go, you can also instead provide a group specification. Since nodes specifications are part of chain specifications, which in turn are part of group specifications, this is a recursive definition.

As can be seen in the above example, when two groups of nodes are connected via an edge specification, it is not immediately obvious which connecting edges are added. This is detailed in Section 17.6.

17.3.5 Specifying Tries

In computer science, a *trie* is a special kind of tree, where for each node and each symbol of an alphabet, there is at most one child of the node labeled with this symbol.

The trie key is useful for drawing tries, but it can also be used in other situations. What it does, essentially, is to prepend the node names of all nodes before the current node of the current chain to the

node's name. This will often make it easier or more natural to specify graphs in which several nodes have the same label.

```
/tikz/graphs/trie=(true or false) (default true, initially false)
```

If this key is set to true, after a node has been created on a chain, the name key is executed with the node's (node name). Thus, all nodes later on this chain have the "path" of nodes leading to this node as their name. This means, in particular, that

- 1. two nodes of the same name but in different parts of a chain will be different,
- 2. while if another chain starts with the same nodes, no new nodes get created.

In total, this is exactly the behaviour you would expect of a trie:

You can even "reiterate" over a path in conjunction with the simple option. However, in this case, the default placement strategies will not work and you will need options like layered layout from the graph drawing libraries. You can also use the trie key locally and later reference nodes using their full name:

17.4 Simple Versus Multi-Graphs

The graph library allows you to construct both simple graphs and multi-graphs. In a simple graph there can be at most one edge between any two vertices, while in a multi-graph there can be multiple edges (hence the name). The two keys multi and simple allow you to switch (even locally inside on of the graph's scopes) between which kind of graph is being constructed. By default, the graph command produces a multi-graph since these are faster to construct.

```
/tikz/graphs/multi (no value)
```

When this edge is set for a whole graph (which is the default) or just for a group (which is useful if the whole graph is simple in general, but a part is a multi-graph), then when you specify an edge between two nodes several times, several such edges get created:

```
a b \tikz \graph [multi] { % "multi" is not really necessary here
a ->[bend left, red] b;
a ->[bend right, blue] b;
};
```

In case multi is used for a scope inside a larger scope where the simple option is specified, then inside the local multi scope edges are immediately created and they are completely ignored when it comes to deciding which kind of edges should be present in the surrounding simple graph. From the surrounding scope's point of view it is as if the local multi graph contained no edges at all.

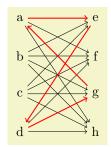
This means, in particular, that you can use the multi option with a single edge to "enforce" this edge to be present in a simple graph.

In contrast a multi-graph, in a simple graph, at most one edge gets created for every pair of vertices:

```
a b \tikz \graph [simple] {
    a -> [bend left, red] b;
    a -> [bend right, blue] b;
};
```

As can be seen, the second edge "wins" over the first edge. The general rule is as follows: In a simple graph, whenever an edge between two vertices is specified multiple times, only the very last specification and its options will actually be executed.

The real power of the simple option lies in the fact that you can first create a complicated graph and then later redirect and otherwise modify edges easily:



```
\tikz \graph [simple, grow right=2cm] {
    {a,b,c,d} ->[complete bipartite] {e,f,g,h};
    { [edges={red,thick}] a -> e -> d -> g -> a };
};
```

One particularly interesting kind of edge specification for a simple graph is -!-. Recall that this is used to indicate that "no edge" should be added between certain nodes. In a multi-graph, this key usually has no effect (unless the key new -!- has been redefined) and is pretty superfluous. In a simple graph, however, it counts as an edge kind and you can thus use it to remove an edge that been added previously:

```
\tikz \graph [simple] {
subgraph K_n [n=8, clockwise];
% Get rid of the following edges:
1 -!- 2;
3 -!- 4;
6 -!- 8;
% And make one edge red:
1 --[red] 3;
```

Creating a graph such as the above in other fashions is pretty awkward.

For every unordered pair $\{u, v\}$ of vertices at most one edge will be created in a simple graph. In particular, when you say $a \rightarrow b$ and later also $a \leftarrow b$, then only the edge $a \leftarrow b$ will be created. Similarly, when you say $a \rightarrow b$ and later $b \rightarrow a$, then only the edge $b \rightarrow a$ will be created.

The power of the simple command comes at a certain cost: As the graph is being constructed, a (sparse) array is created that keeps track for each edge of the last edge being specified. Then, at the end of the scope containing the simple command, for every pair of vertices the edge is created. This is implemented by two nested loops iterating over all possible pairs of vertices – which may take quite a while in a graph of, say, 1000 vertices. Internally, the simple command is implemented as an operator that adds the edges when it is called, but this should be unimportant in normal situations.

17.5 Graph Edges: Labeling and Styling

When the graph library creates an edge between two nodes in a graph, the appearance (called "styling" in TikZ) can be specified in different ways. Sometimes you will simply wish to say "the edges between these two groups of node should be red," but sometimes you may wish to say "this particular edge going into this node should be red." In the following, different ways of specifying such styling requirements are discussed. Note that adding labels to edges is, from TikZ's piont of view, almost the same as styling edges, since they are also specified using options.

17.5.1 Options For All Edges Between Two Groups

When you write ... -> [options] ... somewhere inside your graph specification, this typically cause one or more edges to be created between the nodes in the chain group before the -> and the nodes in the chain group following it. The options are applied to all of them. In particular, if you use the quotes library and you write some text in quotes inside the options, this text will be added as a label to each edge:

As documented in the quotes library in more detail, you can easily modify the appearance of edge labels created using the quotes syntax by adding options after the closing quotes:

```
a \xrightarrow{X} b \xrightarrow{y} c \xrightarrow{Z} d \tikz \graph {
   a ->["x"] b ->["y"'] c ->["z" red] d;
};
```

The following options make it easy to setup the styling of nodes created in this way:

```
/tikz/graphs/edge quotes=(options) (no default)
```

A shorthand for setting the style every edge quotes to $\langle options \rangle$.

```
a \xrightarrow{X} b \xrightarrow{y} c \xrightarrow{b} d \tikz \graph [edge quotes={blue,auto}] {
   a ->["x"] b ->["y"'] c ->["b" red] d;
};
```

/tikz/graphs/edge quotes center

(no value)

A shorthand for edge quotes to anchor=center.

```
\begin{array}{c} a \xrightarrow{} b \xrightarrow{} y \xrightarrow{} c \xrightarrow{} d \\ \begin{array}{c} a \xrightarrow{} ["x"] \ b \xrightarrow{} ["y"] \ c \xrightarrow{} ["z" \ red] \ d; \\ \end{array} \\ \}; \end{array}
```

/tikz/graphs/edge quotes mid

(no value)

A shorthand for edge quotes to anchor=mid.

```
\begin{array}{c} a \xrightarrow{} b \xrightarrow{} y \xrightarrow{} c \xrightarrow{} d \\ \begin{array}{c} a \xrightarrow{} ["x"] \ b \xrightarrow{} ["y"] \ c \xrightarrow{} ["z" \ red] \ d; \\ \end{array} \\ \}; \end{array}
```

17.5.2 Changing Options For Certain Edges

Conside the following tree-like graph:

```
a b \tikz \graph { a -> {b,c} };
```

Suppose we wish to specify that the edge from a to b should be red, while the edge from a to c should be blue. The difficulty lies in the fact that *both* edges are created by the single -> operator and we can only add one of these option red or blue to the operator.

There are several ways to solve this problem. First, we can simply split up the specification and specify the two edges separately:

While this works quite well, we can no longer use the nice chain group syntax of the graphs library. For the rather simple graph a->{b,c} this is not a big problem, but if you specify a tree with, say, 30 nodes it

is really worthwhile being able to specify the tree "in its natural form in the TEX code" rather than having to list all of the edges explicitly. Also, as can be seen in the above example, the node placement is changed, which is not always desirable.

One can sidestep this problem using the **simple** option: This option allows you to first specify a graph and then, later on, replace edges by other edges and, thereby, provide new options:

The first line is the original specification of the tree, while the following two lines replace some edges of the tree (in this case, all of them) by edges with special options. While this method is slower and in the above example creates even longer code, it is very useful if you wish to, say, highlight a path in a larger tree: First specify the tree normally and, then, "respecify" the path or paths with some other edge options in force. In the following example, we use this to highlight a whole subtree of a larger tree:

17.5.3 Options For Incoming and Outgoing Edges

When you use the syntax ... ->[options] ... to specify options, you specify options for the "connections between two sets of nodes". In many cases, however, it will be more natural to specify options "for the edges lead to or coming from a certain node" and you will want to specify these options "at the node". Returning to the example of the graph a->{b,c} where we want a red edge between a and b and a blue edge between a and c, this could also be phrased as follows: "Make the edge leading to b red and make the edge leading to c blue."

For this situation, the graph library offers a number of special keys, which are documented in the following. However, most of the time you will not use these keys directly, but, rather, use a special syntax explained in Section 17.5.4.

```
/tikz/graphs/target edge style=\langle options\rangle (no default)
```

This key can (only) be used with a *node* inside a graph specification. When used, the $\langle options \rangle$ will be added to every edge that is created by a connector like \rightarrow in which the node is a *target*. Consider the following example:

```
\begin{array}{c} a \longrightarrow c \longrightarrow e \\ \\ \{a,b\} \rightarrow \\ \{c \text{ [target edge style=red], d } \} \rightarrow \\ \\ \{e,f\} \\ \}; \end{array}
```

In the example, only when the edge from a to c is created, c is the "target" of the edge. Thus, only this edge becomes red.

When an edge already has options set directly, the $\langle options \rangle$ are executed after these direct options, thus, they "overrule" them:

```
\begin{array}{c} a \longrightarrow c \longrightarrow e \\ & \{ \text{a, b} \} \rightarrow \text{[blue, thick]} \\ & \{ \text{c [target edge style=red], d} \} \rightarrow \\ & \{ \text{e, f} \} \\ \}; \end{array}
```

The $\langle options \rangle$ set in this way will stay attached to the node, so also for edges created later on that lead to the node will have these options set:

Multiple uses of this key accumulate. However, you may sometimes also wish to "clear" these options for a key since at some later point you no longer wish the $\langle options \rangle$ to be added when some further edges are added. This can be achieved using the following key:

/tikz/graphs/target edge clear

(no value)

Clears all $\langle options \rangle$ for edges with the node as a target and also edge labels (see below) for this node.

```
a \rightarrow c \tikz \graph {
    {a, b} ->
    {c [target edge style=red], d},
    b -> c[target edge clear]
};
```

/tikz/graphs/target edge node=\langle node specification\rangle

(no default)

This key works like target edge style, only the $\langle node\ specification \rangle$ will not be added as options to any newly created edges with the current node as their target, but rather it will be added as a node specification.

As for target edge style multiple uses of this key accumulate and the key target edge clear will (also) clear all target edge nodes that have been set for a node earlier on.

```
/tikz/graphs/source edge style=\langle options\rangle
```

(no default)

Works exactly like target edge style, only now the $\langle options \rangle$ are only added when the node is a source of a newly created edge:

```
\begin{array}{c} a \longrightarrow c \longrightarrow e \\ \\ \{a,b\} -> \\ \{c \text{ [source edge style=red], d} \} -> \\ \\ \{e,f\} \\ \}; \end{array}
```

If both for the source and also for the target of an edge $\langle options \rangle$ have been specified, the options are applied in the following order:

- 1. First come the options from the edge itself.
- 2. Then come the options contributed by the source node using this key.
- 3. Then come the options contributed by the target node using target node style.

```
/tikz/graphs/source edge node=\langle node specification\rangle
```

(no default)

Works like source edge style and target edge node.

```
/tikz/graphs/source edge clear=\langle node specification\rangle
```

(no default)

Works like target edge clear.

17.5.4 Special Syntax for Options For Incoming and Outgoing Edges

The keys target node style and its friends are powerful, but a bit cumbersome to write down. For this reason, the graphs library introduces a special syntax that is based on what I call the "first-char syntax" of keys. Inside the options of a node inside a graph, the following special rules apply:

1. Whenever an option starts with >, the rest of the options are passed to target edge style. For instance, when you write a [>red], then this has the same effect as if you had written

```
a[target edge style={red}]
```

- 2. Whenever an options starts with <, the rest of the options are passed to source edge style.
- 3. In both of the above case, in case the options following the > or < sign start with a quote, the created edge label is passed to source edge node or target edge node, respectively.

This is exactly what you want to happen.

Additionally, the following styles provide shorthands for "clearing" the target and source options:

```
/tikz/graphs/clear > (no value)
```

A more easy-to-remember shorthand for target edge clear.

```
/tikz/graphs/clear < (no value)
```

A more easy-to-remember shorthand for source edge clear.

These mechanisms make it especially easy to create trees in which the edges are labeled in some special way:

17.5.5 Placing Node Texts on Incoming Edges

Normally, the text of a node is shown (only) inside the node. In some case, for instance when drawing certain kind of trees, the nodes themselves should not get any text, but rather the edge leading to the node should be labeled as in the following example:

As the example shows, it is a bit cumbersome that we have to label the nodes and then specify the same text once more using the incoming edge syntax.

For these cases, it would be better if the text of the node where not used with the node but, rather, be passed directly to the incoming or the outgoing edge. The following styles do exactly this:

```
\label{eq:continuous_continuous_continuous} \slashed text on incoming edges = \slashed options \slashed text on incoming edges = \slashed options \slashed op
```

When this key is used with a node or a group, the following happens:

- 1. The command target edge node={node[\(options \)] {\tikzgraphnodetext}} is executed. This means that all incoming edges of the node get a label with the text that would usually be displayed in the node. You can use keys like math nodes normally.
- 2. The command as={} is executed. This means that the node itself will display nothing.

Here is an example that show how this command is used.

```
/tikz/graphs/put node text on outgoing edges=(options)
```

(no default)

Works like the previous key, only with target replaced by source.

17.6 Graph Operators, Color Classes, and Graph Expressions

TikZ's graph command employs a powerful mechanism for adding edges between nodes and sets of nodes. To a graph theorist, this mechanism may be known as a $graph\ expression$: A graph is specified by starting with small graphs and then applying operators to them that form larger graphs and that connect and recolor colored subsets of the graph's node in different ways.

17.6.1 Color Classes

TikZ keeps track of a (multi)coloring of the graph as it is being constructed. This does not mean that the actual color of the nodes on the page will be different, rather, in the following we refer to "logical" colors in the way graph theoreticians do. These "logical" colors are only important while the graph is being constructed and they are "thrown away" at the end of the construction. The actual ("physical") colors of the nodes are set independently of these logical colors.

As a graph is being constructed, each node can be part of one or more overlapping *color classes*. So, unlike what is sometimes called a *legal coloring*, the logical colorings that TikZ keeps track of may assign multiple colors to the same node and two nodes connected by an edge may well have the same color.

Color classes must be declared prior to use. This is done using the following key:

```
/tikz/graphs/color class=\langle color class name\rangle
```

(no default

This sets up a new color class called $\langle color \ class \ name \rangle$. Nodes and whole groups of nodes can now be colored with $\langle color \ class \ name \rangle$. This is done using the following keys, which become available inside the current scope:

```
/tikz/graphs/\(\langle color class name \rangle \)
```

(no value)

This key internally uses the operator command to setup an operator that will cause all nodes of the current group to get the "logical color" $\langle color\ class\ name \rangle$. Nodes retain this color in all encompassing scopes, unless it is explicitly changed (see below) or unset (again, see below).

```
a
b
c
d
```

```
g_3 \xrightarrow{r_1} r_2
g_2 \qquad g_1
```

```
\tikz \graph [color class=red, color class=green, math nodes, clockwise, n=5] {
   [complete bipartite={red}{green}]
   { [red]    r_1, r_2 },
   { [green]    g_1, g_2, g_3 }
};
```

```
/tikz/graphs/not \langle color class name \rangle
```

(no value)

Sets up an operator for the current scope so that all nodes in it loose the color $\langle color\ class\ name \rangle$. You can also use ! $\langle color\ class\ name \rangle$ as an alias for this key.

```
\( \frac{r_1}{g_3} \) \( \frac{r_2}{r_2} \) \( \frac{[complete bipartite={red}{green}]}{{ [red] r_1, r_2}, { [green] g_1, g_2, g_3}, { g_2 [recolor green by=red]}} \)
```

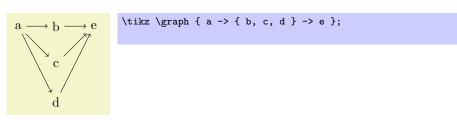
```
/tikz/graphs/recolor \langle color \ class \ name \rangle by=\langle new \ color \rangle
```

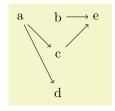
(no default)

Causes all keys having color $\langle color \ class \ name \rangle$ to get $\langle new \ color \rangle$ instead. They loose having color $\langle color \ class \ name \rangle$, but other colors are not affected.

The following color classes are available by default:

- Color class all. Every node is part of this class by default. This is useful to access all nodes of a (sub)graph, since you can simply access all nodes of this color class.
- Color classes source and target. These classes are used to identify nodes that lead "into" a group of nodes and nodes from which paths should "leave" the group. Details on how these colors are assigned are explained in Section 17.6.3. By saying not source or not target with a node, you can influence how it is connected:





\tikz \graph { a -> { b[not source], c, d[not target] } -> e };

• Color classes source' and target'. These are temporary colors that are also explained in Section 17.6.3.

17.6.2 Graph Operators on Groups of Nodes

Recall that the **graph** command constructs graphs recursively from nested $\langle group\ specifications \rangle$. Each such $\langle group\ specification \rangle$ describes a subset of the nodes of the final graph. A $graph\ operator$ is an algorithm that gets the nodes of a group as input and (typically) adds edges between these nodes in some sensible way. For instance, the **clique** operator will simply add edges between all nodes of the group.

```
/tikz/graphs/operator=\langle code \rangle
```

(no default)

This key has an effect in three places:

- 1. It can be used in the $\langle options \rangle$ of a $\langle direct\ node\ specification \rangle$.
- 2. It can be used in the $\langle options \rangle$ of a $\langle group\ specification \rangle$.
- 3. It can be used in the $\langle options \rangle$ of an $\langle edge\ specification \rangle$.

The first case is a special case of the second, since it is treated like a group specification containing a single node. The last case is more complicated and discussed in the next section. So, let us focus on the second case.

Even though the $\langle options \rangle$ of a group are given at the beginning of the $\langle group\ specification \rangle$, the $\langle code \rangle$ is only executed when the group has been parsed completely and all its nodes have been identified. If you use the operator multiple times in the $\langle options \rangle$, the effect accumulates, that is, all code passed to the different calls of operator gets executed in the order it is encountered.

The $\langle code \rangle$ can do "whatever it wants," but it will typically add edges between certain nodes. You can configure what kind of edges (directed, undirected, etc.) are created by using the following keys:

/tikz/graphs/default edge kind= $\langle value \rangle$

(no default, initially --)

This key stores one of the five edge kinds --, <-, and -!-. When an operator wishes to create a new edge, it should typically set

```
\tikzgraphsset{new \pfkeysvalueof{/tikz/graphs/default edge kind}=...}
```

While this key can be set explicitly, it may be more convenient to use the abbreviating keys listed below. Also, this key is automatically set to the current value of $\langle edge\ specification \rangle$ when a joining operator is called, see the discussion of joining operators in Section 17.6.3.

/tikz/graphs/-- (no value)

Sets the default edge kind to --.



```
\tikz \graph { subgraph K_n [--, n=5, clockwise, radius=6mm] };
```

/tikz/graphs/->

(no value)

Sets the default edge kind to ->.



```
\tikz \graph { subgraph K_n [->, n=5, clockwise, radius=6mm] };
```

/tikz/graphs/<-

(no value)

Sets the default edge kind to <-.



```
\tikz \graph { subgraph K_n [<-, n=5, clockwise, radius=6mm] };</pre>
```

/tikz/graphs/<-> (no value)

Sets the default edge kind to <->.



```
\tikz \graph { subgraph K_n [<->, n=5, clockwise, radius=6mm] };
```

```
/tikz/graphs/-!- (no value)
```

Sets the default edge kind to -!-.

When the $\langle code \rangle$ of an operator is executed, the following commands can be used to find the nodes that should be connected:

$\tikzgraphforeachcolorednode{\langle color name \rangle} {\langle macro \rangle}$

When this command is called inside $\langle code \rangle$, the following will happen: TikZ will iterate over all nodes inside the just-specified group that have the color $\langle color \ name \rangle$. The order in which they are iterated over is the order in which they appear inside the group specification (if a node is encountered several times inside the specification, only the first occurrence counts). Then, for each node the $\langle macro \rangle$ is executed with the node's name as the only argument.

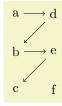
In the following example we use an operator to connect every node colored all inside the subgroup to he node root.

$\tikzgraphpreparecolor{\langle color name \rangle}{\langle counter \rangle}{\langle counter \rangle}$

This command is used to "prepare" the nodes of a certain color for random access. The effect is the following: It is counted how many nodes there are having color $\langle color \ name \rangle$ in the current group and the result is stored in $\langle counter \rangle$. Next, macros named $\langle prefix \rangle 1$, $\langle prefix \rangle 2$, and so on are defined, that store the names of the first, second, third, and so on node having the color $\langle color \ name \rangle$.

The net effect is that after you have prepared a color, you can quickly iterate over them. This is especially useful when you iterate over several color at the same time.

As an example, let us create an operator then adds a zig-zag path between two color classes:



```
\newcount\leftshorecount
                           \newcount\rightshorecount
                           \newcount\myothercount
\newcount\mvcount
\def\zigzag{
  \tikzgraphpreparecolor{left shore}\leftshorecount{left shore prefix}
  \tikzgraphpreparecolor{right shore}\rightshorecount{right shore prefix}
    \advance\mycount\ by\ 1\relax\%
    % Add the "forward" edge
    \tikzgraphsset{new ->=
      {\csname left shore prefix\the\mycount\endcsname}
      {\csname right shore prefix\the\mycount\endcsname}{}{}}
    \myothercount=\mycount\relax%
    \advance\myothercount by1\relax%
    \tikzgraphsset{new <-=
      {\csname left shore prefix\the\myothercount\endcsname}
      {\csname right shore prefix\the\mycount\endcsname}{}{}}
  \ifnum\mvothercount<\leftshorecount\relax
  \repeat
\begin{tikzpicture}
  \graph [color class=left shore, color class=right shore]
  { [operator=\zigzag]
    { [left shore, Cartesian placement]
                                                              a, b, c },
    { [right shore, Cartesian placement, nodes={xshift=1cm}] d, e, f }
 };
\end{tikzpicture}
```

Naturally, in order to turn the above code into a usable operator, some more code would be needed (like default values and taking care of shores of different sizes).

There are a number of predefined operators, like clique or cycle, see the reference Section 17.9 for a complete list.

17.6.3 Graph Operators for Joining Groups

When you join two nodes foo and bar by the edge specification ->, it is fairly obvious, what should happen: An edge from (foo) to (bar) should be created. However, suppose we use an edge specification between two node sets like {a,b,c} and {d,e,f}. In this case, it is not so clear which edges should be created. One might argue that all possible edges from any node in the first set to any node in the second set should be added. On the other hand, one might also argue that only a matching between these two sets should be created. Things get even more muddy when a longer chain of node sets are joined.

Instead of fixing how edges are created between two node sets, TikZ takes a somewhat more general, but also more complicated approach, which can be broken into two parts. In the following, assume that the following chain specification is given:

```
\langle spec_1 \rangle \langle edge \ specification \rangle \langle spec_2 \rangle
```

An example might be $\{a,b,c\} \rightarrow \{d, e\rightarrow f\}$.

The source and target vertices. Let us start with the question of which vertices of the first node set should be connected to vertices in the second node set.

There are two predefined special color classes that are used for this: source and target. For every group specification, some vertices are colored as source vertices and some vertices are target vertices (a node can both be a target and a source). Initially, every vertex is both a source and a target, but that can change as we will see in a moment.

The intuition behind source and target vertices is that, in some sense, edges "from the outside" lead into the group via the source vertices and lead out of the group via the target vertices. To be more precise, the following happens:

- 1. The target vertices of the first group are connected to the source vertices of the second group.
- 2. In the group resulting from the union of the nodes from $\langle spec_1 \rangle$ and $\langle spec_2 \rangle$, the source vertices are only those from the first group, and the target vertices are only those from the second group.

Let us go over the effect of these rules for the example {a,b,c} -> {d, e->f}. First, each individual node is initially both a source and a target vertex. Then, in {a,b,c} all nodes are still both source and

target vertices since just grouping vertices does not change their colors. Now, in e->f something interesting happens for the first time: the target vertices of the "group" e (which is just the node e) are connected to the source vertices of the "group" f. This means, that an edge is added from e to f. Then, in the resulting group e->f the only source vertex is e and the only target vertex is f. This implies that in the group {d,e->f} the sources are d and e and the targets are d and f.

Now, in {a,b,c} -> {d,e->f} the targets of {a,b,c} (which are all three of them) are connected to the sources of {d,e->f} (which are just d and e). Finally, in the whole graph only a, b, and c are sources while only d and f are targets.

The next objective is to make more precise what it means that "the targets of the first graph" and the "sources of the second graph" should be connected. We know already of a general way of connecting nodes of a graph: operators! Thus, we use an operator for this job. For instance, the complete bipartite operator adds an edge from every node having a certain color to every node have a certain other color. This is exactly what we need here: The first color is "the color target restricted to the nodes of the first graph" and the second color is "the color source restricted to the nodes of the second graph."

However, we cannot really specify that only nodes from a certain subgraph are meant – the operator machinery only operates on all nodes of the current graph. For this reason, what really happens is the following: When the graph command encounters $\langle spec_1 \rangle$ $\langle edge\ specification \rangle\ \langle spec_2 \rangle$, it first computes and colors the nodes of the first and the second specification independently. Then, the target nodes of the first graph are recolored to target' and the source nodes of the second graph are recolored to source'. Then, the two graphs are united into one graph and a *joining operator* is executed, which should add edges between target' and source'. Once this is done, the colors target' and source' get erased. Note that in the resulting graph only the source nodes from the first graph are still source nodes and likewise for the target nodes of the second graph.

The joining operators. The job of a joining operator is to add edges between nodes colored target, and source. The following rule is used to determine which operator should be chosen for performing this job:

- 2. Otherwise, the current value of the following key is used:

```
\label{tikz/graphs/default edge operator=} \langle key \rangle \hspace{1cm} \text{(no default)}
```

This key stores the name of a $\langle key \rangle$ that is executed for every $\langle edge\ specification \rangle$ whose $\langle options \rangle$ do not contain the operator key.

A typical joining operator is complete bipartite. It takes the names of two color classes as input and adds edges from all vertices of the first class to all vertices of the second class. Now, the trick is that the default value for the complete bipartite key is {target'}{source'}. Thus, if you just write ->[complete bipartite], the same happens as if you had written

```
->[complete bipartite={target'}{source'}]
```

This is exactly what we want to happen. The same default values are also set for other joining operators like matching or butterfly.

Even though an operator like complete bipartite is typically used together with an edge specification, it can also be used as a normal operator together with a group specification. In this case, however, the color classes must be named explicitly:

A list of predefined joining operators can be found in the reference Section 17.9.

The fact that joining operators can also be used as normal operators leads to a subtle problem: A normal operator will typically use the current value of default edge kind to decide which kind of edges should be put between the identified vertices, while a joining operator should, naturally, use the kind of edge specified by the $\langle edge\ specification \rangle$. This problem is solved as follows: Like a normal operator, a joining operator should also use the current value of default edge kind for the edges it produces. The trick is that this will automatically be set to the current $\langle edge\ specification \rangle$ when the operator explicitly in the $\langle options \rangle$ of the edge specification or implicitly in the default edge operator.

17.7 Graph Macros

A graph macro is a small graph that is inserted at some point into the graph that is currently being constructed. There is special support for such graph macros in TikZ. You might wonder why this is necessary – can't one use TeX's normal macro mechanism? The answer is "no": one cannot insert new nodes into a graph using normal macros because the chains, groups, and nodes are determined prior to macro expansion. Thus, any macro encountered where some node text should go will only be expanded when this node is being named and typeset.

A graph macro is declared using the following key:

This key declares that $\langle graph \ name \rangle$ can subsequently be used as a replacement for a $\langle node \ name \rangle$. Whenever the $\langle graph \ name \rangle$ is used in the following, a graph group will be inserted instead whose content is exactly $\langle specification \rangle$. In case $\langle graph \ name \rangle$ is used together with some $\langle options \rangle$, they are executed prior to inserting the $\langle specification \rangle$.

In the next example, we use a key to configure a subgraph:

Actually, the n key is already defined internally for a similar purpose.

As a last example, let us define a somewhat more complicated graph macro.

```
\newcount\mycount
\tikzgraphsset{
  levels/.store in=\tikzgraphlevel,
  levels=1.
  declare={bintree}{%
    [/utils/exec={%
      \ifnum\tikzgraphlevel=1\relax%
        \def\childtrees{ / }%
      \else%
        \mycount=\tikzgraphlevel%
        \advance\mycount by-1\relax%
        \edef\childtrees{
          / -> {
            bintree[levels=\the\mycount],
            bintree[levels=\the\mycount]
          }}
      \fi%
    parse/.expand once=\childtrees
    % Everything is inside the \childtrees...
  }
\tikz \graph [grow down=5mm, branch right=5mm] { bintree [levels=5] };
```

Note that when you use a graph macro several time inside the same graph, you will typically have to use the name option so that different copies of the subgraph are created:

You will find a list of useful graph macros in the reference section, Section 17.9.1.

17.8 Online Placement Strategies

The main job of the graph library is to make it easy to specify which nodes are present in a graph and how they are connected. In contrast, it is *not* the primary job of the library to compute good positions for nodes in a graph and in many cases you will have to use for instance a \matrix or specify good positions "by hand." Nevertheless, some basic support for automatic node placement is provided for simple cases. The graph library will provide you with information about the position of nodes inside their groups and chains.

As a graph is begin constructed, a placement strategy is used to determine a (reasonably good) position for the nodes as they are created. These placement strategies get some information about what TikZ has already seen concerning the already constructed nodes, but it gets no information concerning the upcoming nodes. Because of this lack of information concerning the future, the strategies need to be what is called an *online strategy* in computer science. (The opposite are *offline strategies*, which get information about the whole graph and all the sizes of the nodes in it. A library that allows such offline strategies is currently under development.)

Strategies are selected using keys like no placement or Cartesian placement. It is permissible to use different strategies inside different parts of a graph, even though the different strategies do not always work together in perfect harmony.

17.8.1 Manual Placement

/tikz/graphs/no placement

(no value)

This strategy simply "switches off" the whole placement mechanism, causing all nodes to be placed at the origin by default. You need to use this strategy if you position nodes "by hand". For this, you can use the at key, the shift keys:

Since the syntax and the many braces and parentheses are a bit cumbersome, the following two keys might also be useful:

```
\text{/tikz/graphs/x=}\langle x \ dimension \rangle
```

(no default)

When you use this key, it will have the same effect as if you had written $at=\{(\langle x \ dimension \rangle, \langle y \ dimension \rangle)\}$, where $\langle y \ dimension \rangle$ is a value set using the y key:

```
c
    \tikz \graph [no placement]
{
    a[x=0,y=0] -> b[x=1,y=0] -> c[x=0,y=1];
};
```

Note that you can specify an x or a y key for a whole scope and then vary only the other key:

Note that these keys have the path /tikz/graphs/, so they will be available inside graphs and will not clash with the usual x and y keys of TikZ, which are used to specify the basic lengths of vectors

```
\t / tikz/graphs/y = \langle y | dimension \rangle (no default)
See above.
```

17.8.2 Placement on a Grid

/tikz/graphs/Cartesian placement

(no value)

This strategy is the default strategy. It works, roughly, as follows: For each new node on a chain, advance a "logical width" counter and for each new node in a group, advance a "logical depth" counter. When a chain contains a whole group, then the "logical width" taken up by the group is the maximum over the logical widths taken up by the chains inside the group; and symmetrically the logical depth of a chain is the maximum of the depths of the groups inside it.

This slightly confusing explanation is perhaps best exemplified. In the below example, the two numbers indicate the two logical width and depth of each node as computed by the graph library. Just ignore the arcane code that is used to print these numbers.

You will find a detailed description of how these logical units are computed, exactly, in Section 17.8.6.

Now, even though we talk about "widths" and "depths" and even though by default a graph "grows" to the right and down, this is by no means fixed. Instead, you can use the following keys to change how widths and heights are interpreted:

```
/tikz/graphs/chain shift=\langle coordinate \rangle
```

(no default, initially (1,0))

Under the regime of the Cartesian placement strategy, each node is shifted by the current logical width times this $\langle coordinate \rangle$.

```
c
b
a
e
h
d
g
f
```

```
\tikz \graph [chain shift=(45:1)] {
  a -> b -> c;
  d -> e;
  f -> g -> h;
};
```

/tikz/graphs/group shift=\langle coordinate\rangle

(no default, initially (0,-1))

Like for chain shift, each node is shifted by the current logical depth times this (coordinate).

```
c
a b e
d g h
```

```
\tikz \graph [chain shift=(45:7mm), group shift=(-45:7mm)] {
  a -> b -> c;
  d -> e;
  f -> g -> h;
};
```

/tikz/graphs/grow up=⟨distance⟩

(default 1)

Sets the chain shift to ($\langle distance \rangle$,0), so that chains "grow upward." The distance by which the center of each new element is removed from the center of the previous one is $\langle distance \rangle$.

/tikz/graphs/grow down=⟨distance⟩

(default 1)

Like grow up.

/tikz/graphs/grow left= $\langle distance \rangle$

(default 1)

Like grow up.

```
c \leftarrow b \leftarrow a \tikz \graph [grow left=7mm] { a -> b -> c};
```

/tikz/graphs/grow right=⟨distance⟩

(default 1)

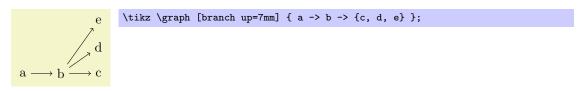
Like grow up.

```
a \rightarrow b \rightarrow c \tikz \graph [grow right=7mm] { a -> b -> c};
```

/tikz/graphs/branch up= $\langle distance \rangle$

(default 1)

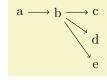
Sets the group shift so that groups "branch upward." The distance by which the center of each new element is removed from the center of the previous one is $\langle distance \rangle$.



Note that when you draw a tree, the **branch** ... keys specify how siblings (or adjacent branches) are arranged, while the **grow** ... keys specify in which direction the branches "grow".

/tikz/graphs/branch down= $\langle distance \rangle$

(default 1)



\tikz \graph [branch down=7mm] { a -> b -> {c, d, e}};

/tikz/graphs/branch left=\langle distance \rangle

(default 1)



 $\label{left=7mm, grow down=7mm} { a -> b -> {c, d, e}};$

/tikz/graphs/branch right= $\langle distance \rangle$

(default 1)



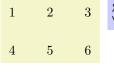
\tikz \graph [branch right=7mm, grow down=7mm] { a -> b -> {c, d, e}};

The following keys place nodes in a $N \times M$ grid.

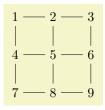
/tikz/graphs/grid placement

(no value)

This key works similar to Cartesian placement. As for that placement strategy, a node has logical width and depth 1. However, the computed total width and depth are mapped to a $N \times M$ grid. The values of N and M depend on the size of the graph and the value of wrap after. The number of columns M is either set to wrap after explicitly or computed automatically as $\sqrt{|V|}$. N is the number of rows needed to lay out the graph in a grid with M columns.



```
% An example with 6 nodes, 3 colums and therefor 2 rows
\tikz \graph [grid placement] { subgraph I_n[n=6, wrap after=3] };
```



% An example with 9 nodes with columns and rows computed automatically \tikz \graph [grid placement] { subgraph Grid_n [n=9] };

In case a user-defined graph instead of a pre-defined subgraph is to be layed out using grid placement, n has to be specified explicitely:

```
a b c \tikz \graph [grid placement] {
    [n=6, wrap after=3]
    a -- b -- c -- d -- e -- f
};
```

17.8.3 Placement Taking Node Sizes Into Account

Options like grow up or branch right do not take the sizes of the to-be-positioned nodes into account – all nodes are placed quite "dumbly" at grid positions. It turns out that the Cartesian placement can also be used to place notes in such a way that their height and/or width is taken into account. Note, however, that while the following options may yield an adequate placement in many situations, when you need advanced alignments you should use a matrix or advanced offline strategies to place the nodes.

```
/\text{tikz/graphs/grow right sep=}\langle \textit{distance}\rangle (default 1em)
```

This key has several effects, but let us start with the bottom line: Nodes along a chain are placed in such a way that the left end of a new node is $\langle distance \rangle$ from the right end of the previous node:

```
start long text short more text end
very long text
long longer longest
```

```
\tikz \graph [grow right sep, left anchor=east, right anchor=west] {
   start -- {
     long text -- {short, very long text} -- more text,
     long -- longer -- longest
   } -- end
};
```

What happens internally is the following: First, the anchor of the nodes is set to west (or north west or south west, see below). Second, the logical width of a node is no longer 1, but set to the actual width of the node (which we define as the horizontal difference between the west anchor and the east anchor) in points. Third, the chain shift is set to (1pt,0pt).

```
/tikz/graphs/grow left sep=\langle distance \rangle (default 1em)

longest — longer — long

\tikz \graph [grow left sep] { long -- longer -- longest };

/tikz/graphs/grow up sep=\langle distance \rangle (default 1em)
```

```
\tikz \graph [grow up sep] {
    a / \$a=x\$ --
    b / \{\$b=\displaystyle \int_0^1 x dx\$\} --
    c [draw, circle, inner sep=7mm]
};

b = \int_0^1 x dx
a = x
```

```
/tikz/graphs/grow down sep=\langle distance \rangle
```

(default 1em)

As above.

/tikz/graphs/branch right sep= $\langle distance \rangle$

(default 1em)

This key works like grow right sep, only it affects groups rather than chains.

```
an even longer text long some text

short very long text longer a

more text longest b

end
```

```
\tikz \graph [grow down, branch right sep] {
  start -- {
    an even longer text -- {short, very long text} -- more text,
    long -- longer -- longest,
    some text -- a -- b
  } -- end
};
```

When both this key and, say, grow down sep are set, instead of the west anchor, the north west anchor will be selected automatically.

/tikz/graphs/branch left sep= $\langle distance \rangle$

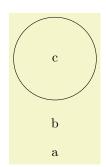
(default 1em)

```
some text long an even longer text

a longer very long text short
b more text
end
```

```
\tikz \graph [grow down sep, branch left sep] {
   start -- {
     an even longer text -- {short, very long text} -- more text,
     long -- longer,
     some text -- a -- b
   } -- end
};
```

(default 1em)



\tikz \graph [branch up sep] { a, b, c[draw, circle, inner sep=7mm] };

/tikz/graphs/branch down sep= $\langle distance \rangle$

(default 1em)

17.8.4 Placement On a Circle

The following keys place nodes on circles. Note that, typically, you do not use circular placement directly, but rather use one of the two keys clockwise or counterclockwise.

/tikz/graphs/circular placement

(no value

This key works quite similar to Cartesian placement. As for that placement strategy, a node has logical width and depth 1. However, the computed total width and depth are mapped to polar coordinates rather than Cartesian coordinates.

```
/tikz/graphs/chain polar shift=(\langle angle \rangle : \langle radius \rangle)
```

(no default, initially (0:1))

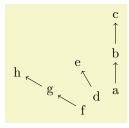
Under the regime of the circular placement strategy, each node on a chain is shifted by ($\langle logical \ width \rangle \langle angle \rangle$: $\langle logical \ width \rangle \langle angle \rangle$).

```
\begin{array}{c} c \\ \uparrow \\ b \\ \uparrow \\ a \end{array}
```

```
\tikz \graph [circular placement] {
  a -> b -> c;
  d -> e;
  f -> g -> h;
};
```

 $\label{eq:condition} $$ \text{tikz/graphs/group polar shift=(} angle : \langle radius \rangle) $$ (no default, initially (45:0)) $$$

Like for group shift, each node on a chain is shifted by $(\langle logical\ depth\rangle \langle angle\rangle : \langle logical\ depth\rangle \langle angle\rangle)$.



```
\tikz \graph [circular placement, group polar shift=(30:0)] {
  a -> b -> c;
  d -> e;
  f -> g -> h;
};
```

/tikz/graphs/radius=⟨dimension⟩

(no default, initially 1cm)

This is an initial value that is added to the total computed radius when the polar shift of a node has been calculated. Essentially, this key allows you to set the $\langle radius \rangle$ of the innermost circle.

```
a b c d \tikz \graph [circular placement, radius=5mm] { a, b, c, d };

a \tikz \graph [circular placement, radius=1cm] { a, b, c, d };

b c d
```

/tikz/graphs/phase= $\langle angle \rangle$

(no default, initially 90)

This is an initial value that is added to the total computed angle when the polar shift of a node has been calculated.

```
a \tikz \graph [circular placement] { a, b, c, d };

b c d \tikz \graph [circular placement, phase=0] { a, b, c, d };

d a
```

/tikz/graphs/clockwise= $\langle number \rangle$

(default \tikzgraphVnum)

This key sets the group shift so that if there are exactly $\langle number \rangle$ many nodes in a group, they will form a complete circle. If you do not provide a $\langle number \rangle$, the current value of tikzgraphVnum is used, which is exactly what you want when you use predefined graph macros like subgraph $\texttt{K_n}$.

```
a \tikz \graph [clockwise=4] { a, b, c, d };

d b

c
```

```
\tikz \graph [clockwise] { subgraph K_n [n=5] };
```



/tikz/graphs/counterclockwise= $\langle number \rangle$

(default \tikzgraphVnum)

Works like clockwise, only the direction is inverted.

17.8.5 Levels and Level Styles

As a graph is being parsed, the **graph** command keeps track of a parameter called the *level* of a node. Provided that the graph is actually constructed in a tree-like manner, the level is exactly equal to the level of the node inside this tree.

/tikz/graphs/placement/level

(no value)

This key stores a number that is increased for each element on a chain, but gets reset at the end of a group:

```
\begin{array}{c} a:1 \xrightarrow{\rightarrow} b:2 \\ c:2 \xrightarrow{\rightarrow} d:3 \\ e:3 \xrightarrow{\rightarrow} f:4 \\ g:4 \\ h:3 \\ j:2 \end{array}
```

```
\tikz \graph [ branch down=5mm, typeset=
    \tikzgraphnodetext:\pgfkeysvalueof{/tikz/graphs/placement/level}]
{
    a -> {
        b,
        c -> {
            d,
            e -> {f,g},
            h
        },
        j
    }
};
```

Unlike the parameters depth and width described in the next section, the key level is always available.

In addition to keeping track of the value of the level key, the graph command also executes the following keys whenever it creates a node:

```
\tikz/graph/level=\langle level \rangle
```

(style, no default)

This key gets executed for each newly created node with $\langle level \rangle$ set to the current level of the node. You can use this key to, say, reconfigure the node distance or the node color.

/tikz/graph/level \langle level \rangle

(style, no value)

This key also gets executed for each newly created node with $\langle level \rangle$ set to the current level of the node.

```
\begin{array}{c}
\mathbf{a} \longrightarrow \mathbf{b} \\
c \longrightarrow \mathbf{d} \\
e \longrightarrow \mathbf{f} \\
g
\end{array}
```

```
\tikz \graph [
   branch down=5mm,
   level 1/.style={nodes=red},
   level 2/.style={nodes=green!50!black},
   level 3/.style={nodes=blue}]
{
   a -> {
      b,
      c -> {
       d,
      e -> {f,g},
      h
      },
      j
   }
};
```

```
\begin{array}{c} a \longrightarrow b \\ \downarrow c \longrightarrow d \\ \downarrow e + f \\ \downarrow g \\ \downarrow h \\ \downarrow j \end{array}
```

```
\tikz \graph [
    branch down=5mm,
    level 1/.style={grow right=2cm},
    level 2/.style={grow right=1cm},
    level 3/.style={grow right=5mm}]
{
    a -> {
       b,
       c -> {
       d,
       e -> {f,g},
       h
      },
      j
    }
};
```

17.8.6 Defining New Online Placement Strategies

In the following the details of how to define a new placement strategy are explained. Most readers may wish to skip this section.

As a graph specification is being parsed, the **graph** library will keep track of different numbers that identify the positions of the nodes. Let us start with what happens on a chain. First, the following counter is increased for each element of the chain:

/tikz/graphs/placement/element count

(no value)

This key stores a number that tells us the position of the node on the current chain. However, you only have access to this value inside the code passed to the macro compute position, explained later on.

As can be seen, each group resets the element counter.

The second value that is computed is more complicated to explain, but it also gives more interesting information:

/tikz/graphs/placement/width

(no value)

This key stores the "logical width" of the nodes parsed up to now in the current group or chain (more precisely, parsed since the last call of place in an enclosing group). This is not necessarily the "total physical width" of the nodes, but rather a number representing how "big" the elements prior to the current element were. This may be their width, but it may also be their height or even their number (which, incidentally, is the default). You can use the width to perform shifts or rotations of to-be-created nodes (to be explained later).

The logical width is defined recursively as follows. First, the width of a single node is computed by calling the following key:

```
\label{logical_node_name} $$ / tikz/graphs/placement/logical node width = & \langle full\ node\ name \rangle $$ (no\ default)
```

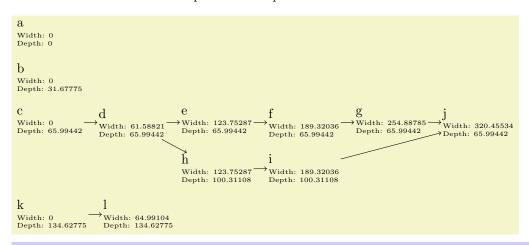
This key is called to compute a physical or logical width of the node $\langle full\ node\ name \rangle$. You can change the code of this key. The code should return the computed value in the macro $\protect\operatorname{\sc hom}$ by default, this key returns 1.

The width of a chain is the sum of the widths of its elements. The width of a group is the maximum of the widths of its elements.

To get a feeling what the above rules imply in practice, let us first have a look at an example where each node has logical width and height 1 (which is the default). The arcane options at the beginning of the code just setup things so that the computed width and depth of each node is displayed at the bottom of each node.

```
\tikz
                                   \graph [nodes={align=center, inner sep=1pt}, grow right=7mm,
                                            typeset={\tikzgraphnodetext\\[-4pt]
b
                                                      \tiny\mywidth\\[-6pt]\tiny\mydepth},
                                            placement/compute position/.append code=
_{1.0}^{0}
                                               \pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
\mathbf{c}
     d
                f
                                              \pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}]
               3.0
2.0
          2.0
          h 🏖
                                   b.
               3.0
3.0
                                   c \rightarrow d \rightarrow \{
                                     e -> f -> g,
                                     h -> i
k
                                  } -> j,
                                  k -> 1
```

In the next example the "logical" width and depth actually match the "physical" width and height. This is caused by the grow right sep option, which internally sets the logical node width key so that it returns the width of its parameter in points.



Symmetrically to chains, as a group is being constructed, counters are available for the number of chains encountered so far in the current group and for the logical depth of the current group:

/tikz/graphs/placement/element count

(no value)

This key stores a number that tells us the sequence number of the chain in the current group.

```
a:0 \rightarrow b:0 \rightarrow c:0
d:1
e:2
f:1
g:2 \rightarrow h:2

\text{tikz \graph [ grow right sep, branch down=5mm, typeset=\tikzgraphnodetext:\mynum, placement/compute position/.append code= \pgfkeysgetvalue{\tikz/graphs/placement/chain count}{\mynum}]

{
a -> b -> {c,d,e},
f,
g -> h
};
```

/tikz/graphs/placement/depth

(no value)

Similarly to the width key, this key stores the "logical depth" of the nodes parsed up to now in the current group or chain and, also similarly, this key may or may not be related to the actual depth/height of the current node. As for the width, the exact definition is as follows: For a single node, the depth is computed by the following key:

```
/tikz/graphs/placement/logical node depth=\(\frac{full node name}{\}\) \((\text{no default})\)
```

The code behind this key should return the "logical height" of the node $\langle full\ node\ name \rangle$ in the macro \pgfmathresult.

Second, the depth of a group is the sum of the depths of its elements. Third, the depth of a chain is the maximum of the depth of its elements.

The width, depth, element count, and chain count keys get updated automatically, but do not have an effect by themselves. This is to the following two keys:

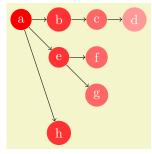
/tikz/graphs/placement/compute position= $\langle code \rangle$

(no default)

The $\langle code \rangle$ is called by the graph command just prior to creating a new node (the exact moment when this key is called is detailed in the description of the place key). When the $\langle code \rangle$ is called, all of the keys described above will hold numbers computed in the way described above.

The job of the $\langle code \rangle$ is to setup node options appropriately so that the to-be-created node will be placed correctly. Thus, the $\langle code \rangle$ should typically set the key nodes={shift= $\langle coordinate \rangle$ } where $\langle coordinate \rangle$ is the computed position for the node. The $\langle code \rangle$ could also set other options like, say, the color of a node depending on its depth.

The following example appends some code to the standard code of compute position so that "deeper" nodes of a tree are lighter. (Naturally, the same effect could be achieved much more easily using the level key.)



```
\newcount\mycount
\def\lightendeepernodes{
    \pgfmathsetcount{\mycount}{
       100-20*\pgfkeysvalueof{/tikz/graphs/placement/width}
    }
    \edef\mydepth{\the\mycount}
    \tikzset{nodes={fill=red!\mydepth,circle,text=white}}
}
\tikz
    \graph [placement/compute position/.append code=\lightendeepernodes]
    {
       a -> {
          b -> c -> d,
          e -> {
             f,
             g
        },
        h
       }
    };
}
```

/tikz/graphs/placement/place

(no value)

Executing this key has two effects: First, the key compute position is called to compute a good position for future nodes (usually, these "future nodes" are just a single node that is created immediately). Second, all of the above counters like depth or width are reset (but not level).

There are two places where this key is sensibly called: First, just prior to creating a node, which happens automatically. Second, when you change the online strategy. In this case, the computed width and depth values from one strategy typically make no sense in the other strategy, which is why the new strategy should proceed "from a fresh start." In this case, the implicit call of compute position ensures that the new strategy gets the last place the old strategy would have used as its starting point, while the computation of its positions is now relative to this new starting point.

For these reasons, when an online strategy like Cartesian placement is called, this key gets called implicitly. You will rarely need to call this key directly, except when you define a new online strategy.

17.9 Reference: Predefined Elements

17.9.1 Graph Macros

```
\usetikzlibrary{graph.standard} % MEX and plain TeX \usetikzlibrary[graph.standard] % ConTeXt
```

This library defines a number of graph macros that are often used in the literature. When new graphs are added to this collection, they will follow the definitions in the Mathematica program, see mathworld.wolfram.com/topics/SimpleGraphs.html.

Graph subgraph I_n

This graph consists just of n unconnected vertices. The following key is used to specify the set of these vertices:

/tikz/graphs/V={⟨list of vertices⟩}

(no default)

Sets a list of vertex names for use with graphs like subgraph I_n and also other graphs. This list is available in the macro \tikzgraphV. The number of elements of this list is available in \tikzgraphVnum.

/tikz/graphs/n=\langle number \rangle

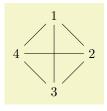
(no default)

This is an abbreviation for $V=\{1,\ldots,\langle number\rangle\}$, name shore $V=\{name=V\}$.



```
\tikz \graph [branch right, nodes={draw, circle}]
{ subgraph I_n [V={a,b,c}] };
```

This graph is not particularly exciting by itself. However, it is often used to introduce nodes into a graph that are then connected as in the following example:



\tikz \graph [clockwise, clique] { subgraph I_n [n=4] };

Graph subgraph I_nm

This graph consists of two sets of once n unconnected vertices and then m unconnected vertices. The first set consists of the vertices set by the key V, the other set consists of the vertices set by the key V.

In order to set the graph path name of the two sets, the following keys get executed:

/tikz/graphs/name shore V

(style, initially empty)

Set this style to, say, name=my V set in order to set a name for the V set.

/tikz/graphs/name shore W

(style, initially empty)

Same as for name shore V.

/tikz/graphs/W={⟨list of vertices⟩}

(no default)

Sets the list of vertices for the W set. The elements and their number are available in the macros \tikzgraphW and \tikzgraphWnum, respectively.

```
/\text{tikz/graphs/m} = \langle number \rangle
```

(no default)

This is an abbreviation for $W=\{1,\ldots,\langle number\rangle\}$, name shore $W=\{name=W\}$.

The main purpose of this subgraph is to setup the nodes in a bipartite graph:

Graph subgraph K_n

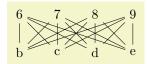
This graph is the complete clique on the vertices from the V key.



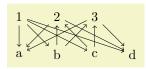
```
\tikz \graph [clockwise] { subgraph K_n [n=7] };
```

Graph subgraph K_nm

This graph is the complete bipartite graph with the two shores V and W as in subgraph I_nm.



```
\tikz \graph [branch right, grow down]
{ subgraph K_nm [V={6,...,9}, W={b,...,e}] };
```



```
\tikz \graph [simple, branch right, grow down]
{
  subgraph K_nm [V={1,2,3}, W={a,b,c,d}, ->];
  subgraph K_nm [V={2,3}, W={b,c}, <-];
};</pre>
```

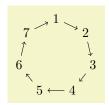
Graph subgraph P_n

This graph is the path on the vertices in V.

```
1 — 2 — 3 \tikz \graph [branch right] { subgraph P_n [n=3] };
```

Graph subgraph C_n

This graph is the cycle on the vertices in V.



```
\tikz \graph [clockwise] { subgraph C_n [n=7, ->] };
```

Graph subgraph Grid_n

This graph is a grid of the vertices in V.

```
/tikz/graphs/wrap after=\langle number \rangle
```

(no default)

Defines the number of nodes placed in a single row of the grid. This value implicitly defines the number of grid columns as well. In the following example a grid placement is used to visualize the edges created between the nodes of a Grid_n subgraph using different values for wrap after.

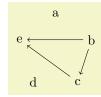
17.9.2 Group Operators

The following keys use the operator key to setup operators that connect the vertices of the current group having a certain color in a specific way.

```
/tikz/graphs/clique=\langle color \rangle
```

(default all)

Adds an edge between all vertices of the current group having the (logical) color $\langle color \rangle$. Since, by default, this color is set to all, which is a color that all nodes get by default, when you do not specify anything, all nodes will be connected.



```
\tikz \graph [color class=red, clockwise, n=5] {
  [clique=red, ->]
  a, b[red], c[red], d, e[red]
};
```

/tikz/graphs/induced independent set= $\langle color \rangle$

(default all)

This key is the "opposite" of a clique: It removes all edges in the current group having belonging to color class $\langle color \rangle$. More precisely, an edge of kind -!- is added for each pair of vertices. This means that edge only get removed if you specify the simple option.

```
\tikz \graph [simple] {
    subgraph K_n [<->, n=7, clockwise]; % create lots of edges
    { [induced independent set] 1, 3, 4, 5, 6 }
};
```

```
/tikz/graphs/cycle=\langle color \rangle
```

(default all)

Connects the nodes colored $\langle color \rangle$ is a cyclic fashion. The ordering is the ordering in which they appear in the whole graph specification.

```
\begin{array}{ccc} f & a \\ & \uparrow \\ e & & \uparrow \\ d & c \end{array}
```

```
\tikz \graph [clockwise, n=6, phase=60] {
    { [cycle, ->] a, b, c },
    { [cycle, <-] d, e, f }
};</pre>
```

/tikz/graphs/induced cycle= $\langle color \rangle$

(default all)

While the cycle command will only add edges, this key will also remove all other edges between the nodes of the cycle, provided we are constructing a simple graph.

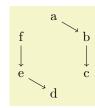
```
\begin{array}{c}
7 \\
7 \\
6 \\
5 \\
4
\end{array}
```

```
\tikz \graph [simple] {
  subgraph K_n [n=7, clockwise]; % create lots of edges
  { [induced cycle, ->, edge=red] 2, 3, 4, 6, 7 },
};
```

$tikz/graphs/path=\langle color \rangle$

(default all)

Works like cycle, only there is no edge from the last to the first vertex.

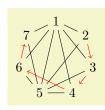


```
\tikz \graph [clockwise, n=6] {
    { [path, ->] a, b, c },
    { [path, <-] d, e, f }
};</pre>
```

/tikz/graphs/induced path= $\langle color \rangle$

(default all)

Works like induced cycle, only there is no edge from the last to the first vertex.



```
\tikz \graph [simple] {
  subgraph K_n [n=7, clockwise]; % create lots of edges
  { [induced path, ->, edges=red] 2, 3, 4, 6, 7 },
};
```

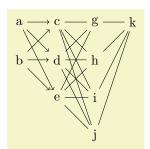
17.9.3 Joining Operators

The following keys are typically used as options of an $\langle edge\ specification \rangle$, but can also be called in a group specification (however, then, the colors need to be set explicitly).

```
tikz/graphs/complete bipartite=\langle from color \rangle \langle to color \rangle
```

(default {source'}{target'})

Adds all possible edges from every node having color $\langle from\ color \rangle$ to every node having color $\langle to\ color \rangle$:



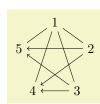
```
f \stackrel{a}{\varprojlim} b
```

```
\tikz \graph [color class=red, color class=green, clockwise, n=6] {
   [complete bipartite={red}{green}, ->]
   a [red], b[red], c[red], d[green], e[green], f[green]
};
```

/tikz/graphs/induced complete bipartite

(no value)

Works like the complete bipartite operator, but in a simple graph any edges between the vertices in either shore are removed (more precisely, they get replaced by -!- edges).



```
\tikz \graph [simple] {
  subgraph K_n [n=5, clockwise]; % Lots of edges
  {2, 3} ->[induced complete bipartite] {4, 5}
};
```

/tikz/graphs/matching= $\langle from \ color \rangle \langle to \ color \rangle$

(default {source'}{target'})

This joining operator forms a maximum matching between the nodes of the two sets of nodes having colors $\langle from\ color \rangle$ and $\langle to\ color \rangle$, respectively. The first node of the from set is connected to the first node of to set, the second node of the from set is connected to the second node of the to set, and so on. If the sets have the same size, what results is what graph theoreticians call a $perfect\ matching$, otherwise only a maximum, but not perfect matching results.

```
/tikz/graphs/matching and star=\langle from \ color \rangle \langle to \ color \rangle
```

(default {source'}{target'})

The matching and star connector works like the matching connector, only it behaves differently when the two to-be-connected sets have different size. In this case, all the surplus nodes get connected to the last node of the other set, resulting what is known as a *star* in graph theory. This simple rule allows for some powerful effects (since this connector is the default, there is no need to add it here):

```
\begin{array}{c} a \longrightarrow b \longrightarrow d \longrightarrow f \\ \\ c \longrightarrow e \end{array}
```

```
\tikz \graph { a -> {b, c} -> {d, e} -- f};
```

The matching and star connector also makes it easy to create trees and series-parallel graphs.

/tikz/graphs/butterfly=\langle options\rangle

(no default)

The butterfly connector is used to create the kind of connections present between layers of a so-called butterfly network. As for other connectors, two sets of nodes are connected, which are the nodes having color target' and source' by default. In a level l connection, the first l nodes of the first set are connected to the second l nodes of the second set, while the second l nodes of the first set get connected to the first l nodes of the second set. Then, for next l nodes of both sets a similar kind of connection is installed. Additionally, each node gets connected to the corresponding node in the other set with the same index (as in a matching):

```
\tikz \graph [left anchor=east, right anchor=west,
                                                   branch down=4mm, grow right=15mm] {
2
                                   subgraph I_n [n=12, name=A] --[butterfly={level=3}]
subgraph I_n [n=12, name=B] --[butterfly={level=2}]
3
4
                                   subgraph I_n [n=12, name=C]
                                 };
5
             5
6
7
8
             9
9
10
            10
11
            11
12
            12
```

Unlike most joining operators, the colors of the nodes in the first and the second set are not passed as parameters to the butterfly key. Rather, they can be set using the *(options)*, which are executed with the path prefix /tikz/graphs/butterfly.

```
/tikz/graphs/butterfly/level=\langle level \rangle
                                                                                          (no default, initially 1)
     Sets the level l for the connections.
/tikz/graphs/butterfly/from=\langle color\rangle
                                                                                  (no default, initially target')
     Sets the color class of the from nodes.
/tikz/graphs/butterfly/to=\langle color \rangle
                                                                                  (no default, initially source')
```

18 Matrices and Alignment

18.1 Overview

When creating pictures, one often faces the problem of correctly aligning parts of the picture. For example, you might wish that the **baselines** of certain nodes should be on the same line and some further nodes should be below these nodes with, say, their centers on a vertical lines. There are different ways of solving such problems. For example, by making clever use of anchors, nearly all such alignment problems can be solved. However, this often leads to complicated code. An often simpler way is to use *matrices*, the use of which is explained in the current section.

A TikZ matrix is similar to LATEX's {tabular} or {array} environment, only instead of text each cell contains a little picture or a node. The sizes of the cells are automatically adjusted such that they are large enough to contain all the cell contents.

Matrices are a powerful tool and they need to be handled with some care. For impatient readers who skip the rest of this section: you must end every row with \setminus . In particular, the last row must be ended with \setminus .

Many of the ideas implemented in TikZ's matrix support are due to Mark Wibrow – many thanks to Mark at this point!

18.2 Matrices are Nodes

Matrices are special in many ways, but for most purposes matrices are treated like nodes. This means, that you use the node path command to create a matrix and you only use a special option, namely the matrix option, to signal that the node will contain a matrix. Instead of the usual TEX-box that makes up the text part of the node's shape, the matrix is used. Thus, in particular, a matrix can have a shape, this shape can be drawn or filled, it can be used in a tree, and so on. Also, you can refer to the different anchors of a matrix.

```
/tikz/matrix=\langle true or false \rangle
```

(default true)

This option can be passed to a node path command. It signals that the node will contain a matrix.

```
Hello
```

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,2);
  \node [matrix,fill=red!20,draw=blue,very thick] (my matrix) at (2,1)
  {
     \draw (0,0) circle (4mm); & \node[rotate=10] {Hello}; \\
     \draw (0.2,0) circle (2mm); & \fill[red] (0,0) circle (3mm); \\
     };
  \draw [very thick,->] (0,0) |- (my matrix.west);
  \end{tikzpicture}
```

The exact syntax of the matrix is explained in the course of this section.

/tikz/every matrix

(style, initially empty)

This style is used in every matrix.

Even more so than nodes, matrices will often be the only object on a path. Because of this, there is a special abbreviation for creating matrices:

\matrix

Inside {tikzpicture} this is an abbreviation for \path node[matrix].

Even though matrices are nodes, some options do not have the same effect as for normal nodes:

- 1. Rotations and scaling have no effect on a matrix as a whole (however, you can still transform the contents of the cells normally). Before the matrix is typeset, the rotational and scaling part of the transformation matrix is reset.
- 2. For multi-part shapes you can only set the text part of the node.
- 3. All options starting with text such as text width have no effect.

18.3 Cell Pictures

A matrix consists of rows of *cells*. Each row (including the last one!) is ended by the command \\. The character & is used to separate cells. Inside each cell, you must place commands for drawing a picture, called the *cell picture* in the following. (However, cell pictures are not enclosed in a complete {pgfpicture} environment, they are a bit more light-weight. The main difference is that cell pictures cannot have layers.) It is not necessary to specify beforehand how many rows or columns there are going to be and if a row contains less cell pictures than another line, empty cells are automatically added as needed.

18.3.1 Alignment of Cell Pictures

For each cell picture a bounding box is computed. These bounding boxes and the origins of the cell pictures determine how the cells are aligned. Let us start with the rows: Consider the cell pictures on the first row. Each has a bounding box and somewhere inside this bounding box the origin of the cell picture can be found (the origin might even lie outside the bounding box, but let us ignore this problem for the moment). The cell pictures are then shifted around such that all origins lie on the same horizontal line. This may make it necessary to shift some cell pictures upwards and others downwards, but it can be done and this yields the vertical alignment of the cell pictures this row. The top of the row is then given by the top of the "highest" cell picture in the row, the bottom of the row is given by the bottom of the lowest cell picture. (To be more precise, the height of the row is the maximum y-value of any of the bounding boxes and the depth of the row is the negated minimum y-value of the bounding boxes).

```
\begin{tikzpicture}
  [every node/.style={draw=black,anchor=base,font=\huge}]

\matrix [draw=red]
  {
      \node {a}; \fill[blue] (0,0) circle (2pt); &
      \node {X}; \fill[blue] (0,0) circle (2pt); &
      \node {g}; \fill[blue] (0,0) circle (2pt); \\
      \node {g};
```

Each row is aligned in this fashion: For each row the cell pictures are vertically aligned such that the origins lie on the same line. Then the second row is placed below the first row such that the bottom of the first row touches the top of the second row (unless a row sep is used to add a bit of space). Then the bottom of the second row touches the top of the third row, and so on. Typically, each row will have an individual height and depth.

```
begin{tikzpicture}
  [every node/.style={draw=black,anchor=base}]

a X g

\matrix [draw=red]
{
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
};

a X g

\matrix [row sep=3mm,draw=red] at (0,-2)
{
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
};
\end{tikzpicture}
```

Let us now have a look at the columns. The rules for how the pictures on any given column are aligned are very similar to the row alignment: Consider all cell pictures in the first column. Each is shifted horizontally such that the origins lie on the same vertical line. Then, the left end of the column is at the left end of the bounding box that protrudes furthest to the left. The right end of the column is at the right end of the bounding box that protrudes furthest to the right. This fixes the horizontal alignment of the cell pictures in the first column and the same happens the cell pictures in the other columns. Then, the right end of the first column touches the left end of the second column (unless column sep is used). The right end of the second column touches the left end of the third column, and so on. (Internally, two columns are actually used to achieve the desired horizontal alignment, but that is only an implementation detail.)

```
\begin{tikzpicture}[every node/.style={draw}]
Hallo
               \matrix [draw=red]
     X
                 \node[left]
                              {Hallo}; \fill[blue] (0,0) circle (2pt); \\
                 \node
                              {X};
                                       fill[blue] (0,0) circle (2pt); \
                 \node[right] {g};
                                       \fill[blue] (0,0) circle (2pt); \\
              };
             \end{tikzpicture}
             \begin{tikzpicture}[every node/.style={draw}]
       6
               \matrix [draw=red,column sep=1cm]
```

```
| Node {8}; & \node {6}; \\
| \node {4}; & \node {2}; \\
| \rac{1}{3}; \\
| \rac{1}{4}; \\
| \rac{1}{5}; \\
```

18.3.2 Setting and Adjusting Column and Row Spacing

There are different ways of setting and adjusting the spacing between columns and rows. First, you can use the options column sep and row sep to set a default spacing for all rows and all columns. Second, you can add options to the & character and the \\ command to adjust the spacing between two specific columns or rows. Additionally, you can specify whether the space between two columns or rows should be considered between the origins of cells in the column or row or between their borders.

```
/\text{tikz/column sep=}\langle spacing \ list \rangle (no default)
```

This option sets a default space that is added between every two columns. This space can be positive or negative and is zero by default. The $\langle spacing \ list \rangle$ normally contains a single dimension like 2pt.

More generally, the $\langle spacing \ list \rangle$ may contain a whole list of numbers, separated by commas, and occurrences of the two key words between origins and between borders. The effect of specifying such a list is the following: First, all numbers occurring in the list are simply added to compute the final spacing. Second, concerning the two keywords, the last occurrence of one of the keywords is important. If the last occurrence is between borders or if neither occurs, then the space is inserted between the two columns normally. However, if the last occurs is between origins, then the following happens: The distance between the columns is adjusted such that the difference between the origins of all the cells in the first column (remember that they all lie on straight line) and the origins of all the cells in the second column is exactly the given distance.

The between origins option can only be used for columns mentioned in the first row, that is, you cannot specify this option for columns introduced only in later rows.

```
    123
    M
    1

    12
    12
    1

    1
    123
    1
```

```
\begin{tikzpicture}
  \matrix [draw,column sep={1cm,between origins},nodes=draw]
  {
      \node(a) {123}; & \node (b) {1}; & \node {1}; \\
      \node {12}; & \node {1}; \\
      \node {1}; \node {1}
```

/tikz/row sep=\langle spacing list\rangle

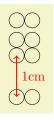
(no default)

This option works like column sep, only for rows. Here, too, you can specify whether the space is added between the lower end of the first row and the upper end of the second row, or whether the space is computed between the origins of the two rows.

```
\begin{tikzpicture}
  \matrix [draw,row sep=1cm,nodes=draw]
  {
      \node (a) {123}; & \node {1}; & \node {1}; \\
      \node (b) {12}; & \node {12}; & \node {1}; \\
      \node {1}; & \node {123}; & \node {1}; \\
      \node {1}; & \node {123}; & \node {1}; \\
      \rightarrow
    };
    \draw [<->,red,thick] (a.south) -- (b.north) node [right,midway] {1cm};
    \end{tikzpicture}
```

```
123 1 1 1 1cm 12 12 1 1 123 1
```

The row-end command \\ allows you to provide an optional argument, which must be a dimension. This dimension will be added to the list in row sep. This means that, firstly, any numbers you list in this argument will be added as an extra row separation between the line being ended and the next line and, secondly, you can use the keywords between origins and between borders to locally overrule the standard setting for this line pair.



```
\begin{tikzpicture}
\matrix [row sep=1mm]
{
    \draw (0,0) circle (2mm); & \draw (0,0) circle (2mm); \\
    \draw (0,0) circle (2mm); & \draw (0,0) circle (2mm); \\[-1mm]
    \draw (0,0) coordinate (a) circle (2mm); &
    \draw (0,0) circle (2mm); \\[1cm, between origins]
    \draw (0,0) coordinate (b) circle (2mm); &
    \draw (0,0) circle (2mm); \\
    \draw (0,0) circle (2mm); \\
};
    \draw [<->,red,thick] (a.center) -- (b.center) node [right,midway] {1cm};
\end{tikzpicture}
```

The cell separation character & also takes an optional argument, which must also be a spacing list. This spacing list is added to the column sep having a similar effect as the option for the \\ command for rows.

This optional spacing list can only be given the first time a new column is started (usually in the first row), subsequent usages of this option in later rows have no effect.

```
    8
    1
    6

    3
    5
    7

    4
    9
    2
```

```
\begin{tikzpicture}
\matrix [draw,nodes=draw,column sep=1mm]
{
    \node {8}; &[2mm] \node{1}; &[-1mm] \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
};
\end{tikzpicture}
```

```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep=1mm]
{
    \node {8}; &[2mm] \node(a){1}; &[1cm,between origins] \node(b){6}; \\
    \node {3}; & \node {5}; & \node {7}; \\
    \node {4}; & \node {9}; & \node {2}; \\
    };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {11mm};
  \end{tikzpicture}
```



```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep={1cm,between origins}]
    \node (a) {8}; & \node (b) {1}; &[between borders] \node (c) {6}; \\
              {3}; & \node
                                                                   {7}; \\
                                                         \node
    \node
                               {5}; &
              {4}; & \node
                                {9}; &
                                                        \node
                                                                   {2}; \\
    \node
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm};
  \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway] {10mm};
\end{tikzpicture}
```

18.3.3 Cell Styles and Options

The following styles and options are useful for changing the appearance of all cell pictures:

```
/tikz/every cell=\{\langle row \rangle\}\{\langle column \rangle\}
```

(style, no default, initially empty)

This style is installed at the beginning of each cell picture with the two parameters being the current $\langle row \rangle$ and $\langle column \rangle$ of the cell. Note that setting this style to draw will not cause all nodes to be drawn since the draw option has to be passed to each node individually.

Inside this style (and inside all cells), the current $\langle row \rangle$ and $\langle column \rangle$ number are also accessible via the counters \pgfmatrixcurrentrow and \pgfmatrixcurrentcolumn.

```
/tikz/cells=\langle options \rangle
```

(no default)

This key adds the $\langle options \rangle$ to the style every cell. It mainly just a shorthand for the code every cell/.append style= $\langle options \rangle$.

```
/tikz/nodes=⟨options⟩
```

(no default)

This key adds the $\langle options \rangle$ to the style every node. It mainly just a shorthand for the code every node/.append style= $\langle options \rangle$.

The main use of this option is the install some options for the nodes *inside* the matrix that should not apply to the matrix *itself*.

```
8 1 6
3 5 7
4 9 2
```

```
begin{tikzpicture}
  \matrix [nodes={fill=blue!20,minimum size=5mm}]
{
      \node {8}; & \node{1}; & \node {6}; \\
      \node {3}; & \node{5}; & \node {7}; \\
      \node {4}; & \node{9}; & \node {2}; \\
    };
\end{tikzpicture}
```

The next set of styles can be used to change the appearance of certain rows, columns, or cells. If more than one of these styles is defined, they are executed in the below order (the every cell style is executed before all of the below).

```
/tikz/column \langle number \rangle
```

(style, no value)

This style is used for every cell in column $\langle number \rangle$.

/tikz/every odd column

(style, no value)

This style is used for every cell in an odd column.

/tikz/every even column

(style, no value)

This style is used for every cell in an even column.

```
/tikz/row (number) (style, no value)
```

This style is used for every cell in row $\langle number \rangle$.

/tikz/every odd row

(style, no value)

This style is used for every cell in an odd row.

/tikz/every even row

(style, no value)

This style is used for every cell in an even row.

```
/tikz/row \langle row number \rangle column \langle column number \rangle
```

(style, no value)

This style is used for the cell in row $\langle row \ number \rangle$ and column $\langle column \ number \rangle$.

```
8 1 6
3 5 7
4 9 2

\text{begin{tikzpicture} \ [row 1/.style={red}, \ column 2/.style={green!50!black}, \ row 3 column 3/.style={blue}]

\text{matrix \ \ \ \node {8}; & \node{1}; & \node {6}; \\ \node {3}; & \node{5}; & \node {7}; \\ \node {4}; & \node{9}; & \node {2}; \\ \};
\end{tikzpicture}
```

You can use the column $\langle number \rangle$ option to change the alignment for different columns.

```
123 456 789
12 45 78
1 4 7
```

```
begin{tikzpicture}
  [column 1/.style={anchor=base west},
    column 2/.style={anchor=base east},
    column 3/.style={anchor=base}]
  \matrix
  {
      \node {123}; & \node{456}; & \node {789}; \\
      \node {12}; & \node{45}; & \node {78}; \\
      \node {1}; & \node{4}; & \node {7}; \\
      \};
  \end{tikzpicture}
```

In many matrices all cell pictures have nearly the same code. For example, cells typically start with \node{ and end };. The following options allow you to execute such code in all cells:

```
/tikz/execute at begin cell=\langle code \rangle
```

(no default)

The code will be executed at the beginning of each nonempty cell.

```
/tikz/execute at end cell=\langle code \rangle
```

(no default)

The code will be executed at the end of each nonempty cell.

```
/tikz/execute at empty cell=\langle code \rangle
```

(no default)

The code will be executed inside each empty cell.

```
\begin{tikzpicture}
  1
8
             [matrix of nodes/.style={
3
     7
                execute at begin cell=\node\bgroup,
                execute at end cell=\egroup;,%
     2
                execute at empty cell=\node{--};%
              }]
             \matrix [matrix of nodes]
               8 & 1 & \\
               3 & & 7 \\
                 Вr.
                     & 2 \\
           \end{tikzpicture}
```

The matrix library defines a number of styles that make use of the above options.

18.4 Anchoring a Matrix

Since matrices are nodes, they can be anchored in the usual fashion using the **anchor** option. However, there are two ways to influence this placement further. First, the following option is often useful:

```
/\text{tikz/matrix anchor} = \langle anchor \rangle (no default)
```

This option has the same effect as anchor, but the option applies only to the matrix itself, not to the cells inside. If you just say anchor=north as an option to the matrix node, all nodes inside matrix will also have this anchor, unless it is explicitly set differently for each node. By comparison, matrix anchor sets the anchor for the matrix, but for the nodes inside the value of anchor remain unchanged.

```
\begin{tikzpicture}
123
          \matrix [matrix anchor=west] at (0,0)
12
            \node {123}; \\ % still center anchor
 1
            \node {12}; \\
            \node {1}; \\
          };
123
          \matrix [anchor=west] at (0,-2)
12
            \node {123}; \\ % inherited west anchor
1
            \node {12}; \\
            \node {1}; \\
          };
        \end{tikzpicture}
```

The second way to anchor a matrix is to use an anchor of a node inside the matrix. For this, the anchor option has a special effect when given as an argument to a matrix:

```
\texttt{/tikz/anchor} = \langle anchor \ or \ node.anchor \rangle (no default)
```

Normally, the argument of this option refers to anchor of the matrix node, which is the node that includes all of the stuff of the matrix. However, you can also provide an argument of the form $\langle node \rangle . \langle anchor \rangle$ where $\langle node \rangle$ must be node defined inside the matrix and $\langle anchor \rangle$ is an anchor of this node. In this case, the whole matrix is shifted around in such a way that this particular anchor of this particular node lies at the at position of the matrix. The same is true for matrix anchor.

```
\text{begin{tikzpicture}}
\draw[help lines] (0,0) grid (3,2);
\matrix[matrix anchor=inner node.south,anchor=base,row sep=3mm] at (1,1)
\{
\text{node {a}; & \node {b}; & \node {c}; & \node {d}; \\
\node {a}; & \node {\text{inde {c}; & \node {d}; \\
\node {a}; & \node {\text{b}; & \node {c}; & \node {d}; \\
\node {a}; & \node {\text{b}; & \node {c}; & \node {d}; \\
\text{node {a}; & \node {c}; & \node {d}; \\
\text{end{tikzpicture}}}
```

18.5 Considerations Concerning Active Characters

Even though TikZ seems to use & to separate cells, PGF actually uses a different command to separate cells, namely the command $\protect\pr$

```
/tikz/ampersand replacement=\langle macro name or empty\rangle
```

(no default)

If a macro name is provided, this macro will be defined to be equal to \pgfmatrixnextcell inside matrices and & will not be made active. For instance, you could say ampersand replacement=\& and then use \& to separate columns as in the following example:



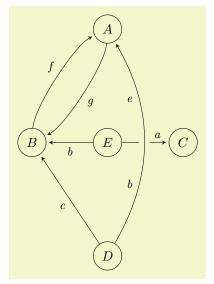
```
\tikz
\matrix [ampersand replacement=\&]
{
    \draw (0,0) circle (4mm); \& \node[rotate=10] {Hello}; \\
    \draw (0.2,0) circle (2mm); \& \fill[red] (0,0) circle (3mm); \\
};
```

18.6 Examples

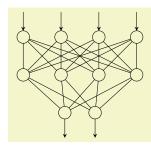
The following examples are adapted from code by Mark Wibrow. The first two redraw pictures from Timothy van Zandt's PStricks documentation:

```
U = \begin{cases} X & X \\ Y & Y \end{cases} 
X \times_{Z} Y \xrightarrow{p} X 
Q = \begin{cases} Y & f \end{cases}
Y \xrightarrow{g} Z
```

```
\begin{tikzpicture}
  \matrix [matrix of math nodes,row sep=1cm]
    |(U)| U &[2mm]
                                          &[8mm]
                    |(XZY)| X \times Z Y &
                                                  |(X)| X \\
            &
                    | (Y) |
                                                  |(Z)| Z \\
  \begin{scope}[every node/.style={midway,auto,font=\scriptsize}]
    \draw [double, dashed] (U)
                                  -- node {$x$} (X);
                                  -- node {$p$} (X -| XZY.east)
                            (X)
    \draw
                                  -- node \{f\}\ (Z)
                                  -- node {$g$} (Y)
                                  -- node {$q$} (XZY)
                                  -- node {$y$} (U);
   \end{scope}
\end{tikzpicture}
```

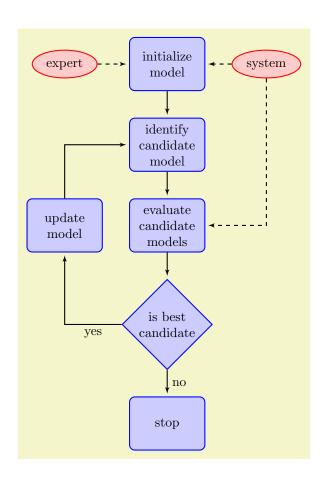


```
\begin{tikzpicture}[>=stealth,->,shorten >=2pt,looseness=.5,auto]
 \matrix [matrix of math nodes,
          column sep={2cm,between origins},
          row sep={3cm,between origins},
          nodes={circle, draw, minimum size=7.5mm}]
           & |(A)| A &
   |(B)| B & |(E)| E & |(C)| C \\
           & |(D)| D
 \begin{scope}[every node/.style={font=\small\itshape}]
    \draw (A) to [bend left] node [midway] {g} (B);
   \draw (B) to [bend left] node [midway]
                                            {f} (A);
    \draw (D) --
                            node [midway]
                                             {c} (B);
                            node [midway]
   \draw (E) --
                                            {b} (B);
    \draw (E) --
                            node [near end] {a} (C);
    \draw [-,line width=8pt,draw=graphicbackground]
        (D) to [bend right, looseness=1] (A);
   \draw (D) to [bend right, looseness=1]
           node [near start] {b} node [near end] {e} (A);
 \end{scope}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \matrix (network)
    [matrix of nodes, %
     nodes in empty cells,
     nodes={outer sep=0pt,circle,minimum size=4pt,draw},
     column sep={1cm,between origins},
     row sep={1cm,between origins}]
                                                          & \\
                   &
                                      &
                                                          & \\
    |[draw=none]| & |[xshift=1mm]| & |[xshift=-1mm]| \\
  \foreach \a in \{1, \ldots, 4\}{
    \draw (network-3-2) -- (network-2-\a);
\draw (network-3-3) -- (network-2-\a);
    \draw [-stealth] ([yshift=5mm]network-1-\a.north) -- (network-1-\a);
    \foreach \b in \{1, \ldots, 4\}
      \draw (network-1-\a) -- (network-2-\b);
  \draw [stealth-] ([yshift=-5mm]network-3-2.south) -- (network-3-2);
  \draw [stealth-] ([yshift=-5mm]network-3-3.south) -- (network-3-3);
\end{tikzpicture}
```

The following example is adapted from code written by Kjell Magne Fauske, which is based on the following paper: K. Bossley, M. Brown, and C. Harris, Neurofuzzy identification of an autonomous underwater vehicle, *International Journal of Systems Science*, 1999, 30, 901–913.



```
\begin{tikzpicture}
  Tauto.
  decision/.style={diamond, draw=blue, thick, fill=blue!20,
                    text width=4.5em,align=flush center,
                    inner sep=1pt},
   block/.style ={rectangle, draw=blue, thick, fill=blue!20,
                    text width=5em,align=center, rounded corners,
                    minimum height=4em},
  line/.style
                  ={draw, thick, -latex', shorten >=2pt},
   cloud/.style
                 ={draw=red, thick, ellipse,fill=red!20,
                    minimum height=2em}]
  \matrix [column sep=5mm,row sep=7mm]
    % row 1
      \node [cloud] (expert)
                              {expert}; &
      \node [block] (init)
                                {initialize model}; &
      \node [cloud] (system) {system}; \\
    % row 2
     & \node [block] (identify) {identify candidate model}; & \\
    % row 3
      \node [block] (update)
                              {update model}; &
      \node [block] (evaluate) {evaluate candidate models}; & \\
    % row 4
     & \node [decision] (decide) {is best candidate}; & \\
    % row 5
     & \node [block] (stop)
                                   {stop}; & \\
  \begin{scope}[every path/.style=line]
    \path
                   (init) -- (identify);
(identify) -- (evaluate);
    \path
                   (evaluate) -- (decide);
    \path
                   (update) |- (identify);
(decide) -| node [near start] {yes} (update);
    \path
    \path
                              -- node [midway] {no} (stop);
-- (init);
                   (decide)
    \path
    \path [dashed] (expert)
                              -- (init);
    \path [dashed] (system)
    \path [dashed] (system)
                             |- (evaluate);
  \end{scope}
\end{tikzpicture}
```

19 Making Trees Grow

19.1 Introduction to the Child Operation

Trees are a common way of visualizing hierarchical structures. A simple tree looks like this:

Admittedly, in reality trees are more likely to grow *upward* and not downward as above. You can tell whether the author of a paper is a mathematician or a computer scientist by looking at the direction their trees grow. A computer scientist's trees will grow downward while a mathematician's tree will grow upward. Naturally, the *correct* way is the mathematician's way, which can be specified as follows:

In TikZ, there are two ways of specifying trees: Using either the graph path operation, which is covered in Section 17, or using the child path operation, which is covered in the present section. Both methods have their advantages.

In TikZ, trees are specified by adding children to a node on a path using the child operation:

```
\path ... child[\langle options \rangle] for each \langle variables \rangle in \{\langle values \rangle\} \{\langle child path \rangle\} ...;
```

This operation should directly follow a completed **node** operation or another **child** operation, although it is permissible that the first **child** operation is preceded by options (we will come to that).

When a node operation like node {X} is followed by child, TikZ starts counting the number of child nodes that follow the original node {X}. For this, it scans the input and stores away each child and its arguments until it reaches a path operation that is not a child. Note that this will fix the character codes of all text inside the child arguments, which means, in essence, that you cannot use verbatim text inside the nodes inside a child. Sorry.

Once the children have been collected and counted, TikZ starts generating the child nodes. For each child of a parent node TikZ computes an appropriate position where the child is placed. For each child, the coordinate system is transformed so that the origin is at this position. Then the $\langle child \; path \rangle$ is drawn. Typically, the child path just consists of a node specification, which results in a node being drawn at the child's position. Finally, an edge is drawn from the first node in the $\langle child \; path \rangle$ to the parent node.

The optional foreach part (note that there is no backslash before foreach) allows you to specify multiple children in a single child command. The idea is the following: A \foreach statement is (internally) used to iterate over the list of $\langle values \rangle$. For each value in this list, a new child is added to the node. The syntax for $\langle variables \rangle$ and for $\langle values \rangle$ is the same as for the \foreach statement, see Section 64. For example, when you say

```
node {root} child [red] foreach \name in {1,2} {node {\name}}
```

the effect will be the same as if you had said

```
node {root} child[red] {node {1}} child[red] {node {2}}
```

When you write

```
node {root} child[\pos] foreach \name/\pos in {1/left,2/right} {node[\pos] {\name}}
```

the effect will be the same as for

```
node {root} child[left] {node[left] {1}} child[right] {node[right] {2}}
```

You can nest things as in the following example:



```
\begin{tikzpicture}
  [level distance=4mm,level/.style={sibling distance=8mm/#1}]
  \coordinate
   child foreach \x in {0,1}
      {child foreach \y in {0,1}}
      {child foreach \z in {0,1}};
\end{tikzpicture}
```

The details and options for this operation are described in the rest of this present section.

19.2 Child Paths and Child Nodes

For each child of a root node, its $\langle child\ path \rangle$ is inserted at a specific location in the picture (the placement rules are discussed in Section 19.5). The first node in the $\langle child\ path \rangle$, if it exists, is special and called the *child node*. If there is no first node in the $\langle child\ path \rangle$, that is, if the $\langle child\ path \rangle$ is missing (including the curly braces) or if it does not start with node or with coordinate, then an empty child node of shape coordinate is automatically added.

Consider the example \node {x} child {node {y}} child;. For the first child, the \(\child \) path\(\) has the child node node {y}. For the second child, no child node is specified and, thus, it is just coordinate.

As for any normal node, you can give the child node a name, shift it around, or use options to influence how it is rendered.

```
root
\text{begin{tikzp} \node[rect \child \{r\child \{r\
```

```
\begin{tikzpicture}
  \node[rectangle,draw] {root}
   child {node[circle,draw] (left node) {left}}
   child {node[ellipse,draw] (right node) {right}};
  \draw[dashed,->] (left node) -- (right node);
\end{tikzpicture}
```

In many cases, the $\langle child\ path \rangle$ will just consist of a specification of a child node and, possibly, children of this child node. However, the node specification may be followed by arbitrary other material that will be added to the picture, transformed to the child's coordinate system. For your convenience, a move-to (0,0) operation is inserted automatically at the beginning of the path. Here is an example:

```
hegin{tikzpicture}
    \node {root}
    child {[fill] circle (2pt)}
    child {[fill] circle (2pt)};
    \end{tikzpicture}
```

At the end of the $\langle child\ path \rangle$ you may add a special path operation called edge from parent. If this operation is not given by yourself somewhere on the path, it will be automatically added at the end. This option causes a connecting edge from the parent node to the child node to be added to the path. By giving options to this operation you can influence how the edge is rendered. Also, nodes following the edge from parent operation will be placed on this edge, see Section 19.6 for details.

To sum up:

- 1. The child path starts with a node specification. If it is not there, it is added automatically.
- 2. The child path ends with a edge from parent operation, possibly followed by nodes to be put on this edge. If the operation is not given at the end, it is added automatically.

19.3 Naming Child Nodes

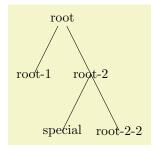
Child nodes can be named like any other node using either the name option or the special syntax in which the name of the node is placed in round parentheses between the node operation and the node's text.

If you do not assign a name to a child node, TikZ will automatically assign a name as follows: Assume that the name of the parent node is, say, parent. (If you did not assign a name to the parent, TikZ will do so itself, but that name will not be user-accessible.) The first child of parent will be named parent-1, the second child is named parent-2, and so on.

This naming convention works recursively. If the second child parent-2 has children, then the first of these children will be called parent-2-1 and the second parent-2-2 and so on.

If you assign a name to a child node yourself, no name is generated automatically (the node does not have two names). However, "counting continues," which means that the third child of parent is called parent-3 independently of whether you have assigned names to the first and/or second child of parent.

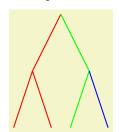
Here is an example:



```
\begin{tikzpicture}[sibling distance=15mm]
  \node (root) {root}
  child
  child {
    child {coordinate (special)}
    child
  };
  \node at (root-1) {root-1};
  \node at (root-2) {root-2};
  \node at (special) {special};
  \node at (root-2-2) {root-2-2};
}
```

19.4 Specifying Options for Trees and Children

Each child may have its own $\langle options \rangle$, which apply to "the whole child," including all of its grandchildren. Here is an example:



```
\begin{tikzpicture}
  [thick,level 2/.style={sibling distance=10mm}]
  \coordinate
    child[red] {child child}
    child[green] {child child[blue]};
  \end{tikzpicture}
```

The options of the root node have no effect on the children since the options of a node are always "local" to that node. Because of this, the edges in the following tree are black, not red.

This raises the problem of how to set options for all children. Naturally, you could always set options for the whole path as in \path [red] node {root} child child; but this is bothersome in some situations. Instead, it is easier to give the options before the first child as follows:

Here is the set of rules:

- 1. Options for the whole tree are given before the root node.
- 2. Options for the root node are given directly to the node operation of the root.

- 3. Options for all children can be given between the root node and the first child.
- 4. Options applying to a specific child path are given as options to the child operation.
- 5. Options applying to the node of a child, but not to the whole child path, are given as options to the node command inside the $\langle child \; path \rangle$.

There are additional styles that influence how children are rendered:

/tikz/every child

(style, initially empty)

This style is used at the beginning of each child, as if you had given the style's contents as options to the child operation.

/tikz/every child node

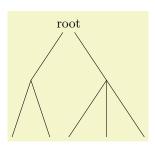
(style, initially empty)

This style is used at the beginning of each child node in addition to the every node style.

/tikz/level= $\langle number \rangle$

(style, no default, initially empty)

This style is executed at the beginning of each set of children, where $\langle number \rangle$ is the current level in the current tree. For example, when you say \node {x} child child;, then level=1 is used before the first child. The style or code of this key will be passed $\langle number \rangle$ as its first parameter. If this first child has children itself, then level=2 would be used for them.

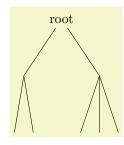


```
\begin{tikzpicture}[level/.style={sibling distance=20mm/#1}]
  \node {root}
  child { child child }
  child { child child child };
\end{tikzpicture}
```

/tikz/level \langle number \rangle

(style, initially empty)

This style is used in addition to the level style. So, when you say \node {x} child child;, then the following key list is executed: level=1,level 1.



```
\begin{tikzpicture}
  [level 1/.style={sibling distance=20mm},
  level 2/.style={sibling distance=5mm}]
  \node {root}
    child { child child }
    child { child child child };
\end{tikzpicture}
```

19.5 Placing Child Nodes

19.5.1 Basic Idea

Perhaps the most difficult part in drawing a tree is the correct layout of the children. Typically, the children have different sizes and it is not easy to arrange them in such a manner that not too much space is wasted, the children do not overlap, and they are either evenly spaced or their centers are evenly distributed. Calculating good positions is especially difficult since a good position for the first child may depend on the size of the last child.

In TikZ, a comparatively simple approach is taken to placing the children. In order to compute a child's position, all that is taken into account is the number of the current child in the list of children and the number of children in this list. Thus, if a node has five children, then there is a fixed position for the first child, a position for the second child, and so on. These positions do not depend on the size of the children and, hence, children can easily overlap. However, since you can use options to shift individual children a bit, this is not as great a problem as it may seem.

Although the placement of the children only depends on their number in the list of children and the total number of children, everything else about the placement is highly configurable. You can change the distance between children (appropriately called the sibling distance) and the distance between levels of the tree. These distances may change from level to level. The direction in which the tree grows can be changed globally and for parts of the tree. You can even specify your own "growth function" to arrange children on a circle or along special lines or curves.

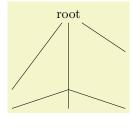
19.5.2 Default Growth Function

The default growth function works as follows: Assume that we are given a node and five children. These children will be placed on a line with their centers (or, more generally, with their anchors) spaced apart by the current sibling distance. The line is orthogonal to the current direction of growth, which is set with the grow and grow' option (the latter option reverses the ordering of the children). The distance from the line to the parent node is given by the level distance.

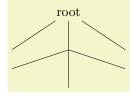
/tikz/level distance= $\langle distance \rangle$

(no default, initially 15mm)

This key determines the distance between different levels of the tree, more precisely, between the parent and the line on which its children are arranged. When given to a single child, this will set the distance for this child only.



```
\begin{tikzpicture}
  \node {root}
  [level distance=20mm]
  child
  child {
     [level distance=5mm]
     child
     child
     child
     child
     child
     child
     }
  child[level distance=10mm];
\end{tikzpicture}
```



```
\begin{tikzpicture}
  [level 1/.style={level distance=10mm},
    level 2/.style={level distance=5mm}]
  \node {root}
    child
    child {
      child child [level distance=10mm]
      child
    }
    child;
}
```

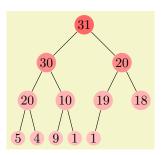
/tikz/sibling distance= $\langle distance \rangle$

(no default, initially 15mm)

This key specifies the distance between the anchors of the children of a parent node.



```
\begin{tikzpicture}
  [level distance=4mm,
  level 1/.style={sibling distance=8mm},
  level 2/.style={sibling distance=4mm},
  level 3/.style={sibling distance=2mm}]
  \text{coordinate}
      child {
         child {child child}
         child {child child}
      }
      child {
         child {child child}
      }
      child {child child}
      child {child child}
    };
  \end{tikzpicture}
```



```
\begin{tikzpicture}
  [level distance=10mm,
   every node/.style={fill=red!60,circle,inner sep=1pt},
   level 1/.style={sibling distance=20mm,nodes={fill=red!45}},
  level 2/.style={sibling distance=10mm,nodes={fill=red!30}},
  level 3/.style={sibling distance=5mm, nodes={fill=red!25}}]
  \node {31}
     child {node {30}
       child {node {20}
         child {node {5}}
         child {node {4}}
       child {node {10}
         child {node {9}}
         child {node {1}}
     }
     child {node {20}
       child {node {19}
         child {node {1}}
         child[missing]
       child {node {18}}
\end{tikzpicture}
```

$\texttt{/tikz/grow=}\langle direction \rangle$

(no default)

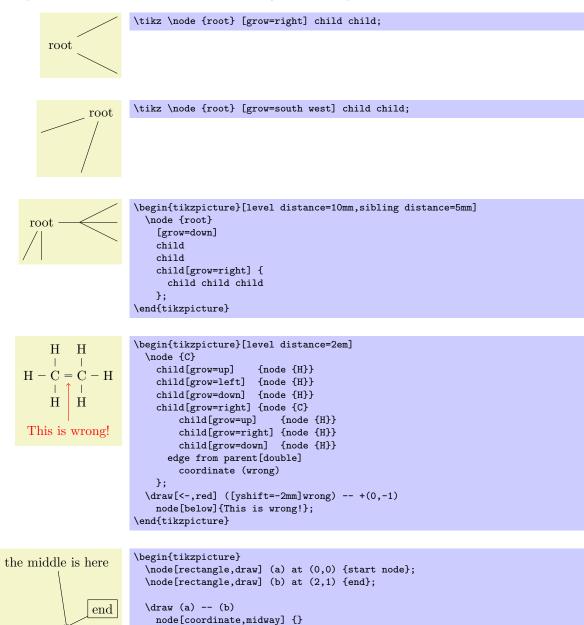
This key is used to define the $\langle direction \rangle$ in which the tree will grow. The $\langle direction \rangle$ can either be an angle in degrees or one of the following special text strings: down, up, left, right, north, south, east, west, north east, north west, south east, and south west. All of these have "their obvious meaning," so, say, south west is the same as the angle -135° .

As a side effect, this option installs the default growth function.

In addition to setting the direction, this option also has a seemingly strange effect: It sets the sibling distance for the current level to Opt, but leaves the sibling distance for later levels unchanged.

This somewhat strange behaviour has a highly desirable effect: If you give this option before the list of children of a node starts, the "current level" is still the parent level. Each child will be on a later level and, hence, the sibling distance will be as specified originally. This will cause the children to be neatly aligned in a line orthogonal to the given $\langle direction \rangle$. However, if you give this option locally to a single child, then "current level" will be the same as the child's level. The zero sibling distance will then cause the child to be placed exactly at a point at distance level distance in the direction $\langle direction \rangle$. However, the children of the child will be placed "normally" on a line orthogonal to the $\langle direction \rangle$.

These placement effects are best demonstrated by some examples:



```
/tikz/grow'=\direction\ (no default)
```

child[grow=100,<-] {node[above] {the middle is here}};</pre>

This key has the same effect as grow, only the children are arranged in the opposite order.

\end{tikzpicture}

19.5.3 Missing Children

start node

Sometimes one or more of the children of a node are "missing." Such a missing child will count as a child with respect to the total number of children and also with respect to the current child count, but it will not be rendered.

```
/tikz/missing=⟨true or false⟩
```

(default true)

If this option is given to a child, the current child counter is increased, but the child is otherwise ignored. In particular, the normal contents of the child is completely ignored.

```
root
1 2 3 5 6
```

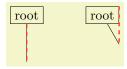
19.5.4 Custom Growth Functions

/tikz/growth parent anchor= $\langle anchor \rangle$

(no default, initially center)

This key allows you to specify which anchor of the parent node is to be used for computing the children's position. For example, when there is only one child and the level distance is 2cm, then the child node will be placed two centimeters below the $\langle anchor \rangle$ of the parent node. "Being placed" means that the child node's anchor (which is the anchor specified using the anchor= option in the node command of the child) is two centimeters below the parent node's $\langle anchor \rangle$.

In the following example, the two red lines both have length 1cm.



```
\begin{tikzpicture}[level distance=1cm]
\node [rectangle,draw] (a) at (0,0) {root}
[growth parent anchor=south] child;

\node [rectangle,draw] (b) at (2,0) {root}
[growth parent anchor=north east] child;

\draw [red,thick,dashed] (a.south) -- (a-1);
\draw [red,thick,dashed] (b.north east) -- (b-1);
\end{tikzpicture}
```

In the next example, the top and bottom nodes are aligned at the top and the bottom, respectively.

```
root big root

|
small big
```

```
\begin{tikzpicture}
  [level distance=2cm,growth parent anchor=north,
    every node/.style={anchor=north,rectangle,draw}
    every child node/.style={anchor=south}]

\node at (0,0) {root} child {node {small}};

\node at (2,0) {big root} child {node {\large big}};
\end{tikzpicture}
```

/tikz/growth function=\langle macro name\rangle

(no default, initially an internal function)

This rather low-level option allows you to set a new growth function. The $\langle macro\ name \rangle$ must be the name of a macro without parameters. This macro will be called for each child of a node. The initial function is an internal function that corresponds to downward growth.

The effect of executing the macro should be the following: It should transform the coordinate system in such a way that the origin becomes the place where the current child should be anchored. When the macro is called, the current coordinate system will be set up such that the anchor of the parent node is in the origin. Thus, in each call, the $\langle macro\ name \rangle$ must essentially do a shift to the child's origin. When the macro is called, the TeX counter \tikznumberofchildren will be set to the total number of children of the parent node and the counter \tikznumberofcurrentchild will be set to the number of the current child.

The macro may, in addition to shifting the coordinate system, also transform the coordinate system further. For example, it could be rotated or scaled.

Additional growth functions are defined in the library, see Section 54.

19.6 Edges From the Parent Node

Every child node is connected to its parent node via a special kind of edge called the edge from parent. This edge is added to the $\langle child \; path \rangle$ when the following path operation is encountered:

```
\path ... edge from parent [\langle options \rangle] ...;
```

This path operation can only be used inside $\langle child\ paths \rangle$ and should be given at the end, possibly followed by $\langle node\ specifications \rangle$ like node {a}. If a $\langle child\ path \rangle$ does not contain this operation, it will be added at the end of the $\langle child\ path \rangle$ automatically.

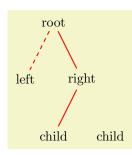
By default, this operation does the following:

1. The following style is executed:

```
/tikz/edge from parent
```

(style, initially draw)

This style is inserted right before the edge from parent path and before the $\langle options \rangle$ are inserted.



```
\begin{tikzpicture}
  [edge from parent/.style={draw,red,thick}]
  \node {root}
  child {node {left} edge from parent[dashed]}
  child {node {right}
    child {node {child}}
    child {node {child} edge from parent[draw=none]}
  };
\end{tikzpicture}
```

- 2. Next, the $\langle options \rangle$ are executed.
- 3. Next, the text stored in the following key is inserted:

```
/tikz/edge from parent path=\langle path \rangle
```

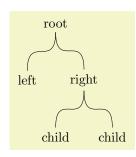
(no default, initially code shown below)

This option allows you to set the edge from parent path to a new path. Initially, this path is the following:

```
(\tikzparentnode\tikzparentanchor) -- (\tikzchildnode\tikzchildanchor)
```

The \tikzparentnode is a macro that will expand to the name of the parent node. This works even when you have not assigned a name to the parent node, in this case an internal name is automatically generated. The \tikzchildnode is a macro that expands to the name of the child node. The two ...anchor macros are empty by default. So, what is essentially inserted is just the path segment (\tikzparentnode) -- (\tikzchildnode); which is exactly an edge from the parent to the child.

You can modify this edge from parent path to achieve all sorts of effects. For example, we could replace the straight line by a curve as follows:



Further useful edge from parent paths are defined in the tree library, see Section 54.

The nodes in a $\langle node\ specification \rangle$ following the edge from parent path command get executed as if the pos option had been added to all these nodes, see also Section 16.8.

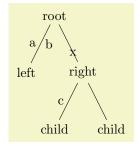
As an example, consider the following code:

```
\node (root) {} child {node (child) {} edge to parent node {label}};
```

The edge to parent operation and the following node operation will, together, have the same effect as if we had said:

```
(root) -- (child) node [pos=0.5] {label}
```

Here is a more complicated example:



```
\begin{tikzpicture}
  \node {root}
    child {
      node {left}
      edge from parent
        node[left] {a}
        node[right] {b}
    child {
      node {right}
        child {
          node {child}
          edge from parent
            node[left] {c}
        child {node {child}}
      edge from parent
        node[near end] {x}
   };
\end{tikzpicture}
```

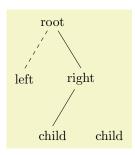
As said before, the anchors in the default edge from parent path are empty. However, you can set them using the following keys:

```
/tikz/child anchor=⟨anchor⟩
```

(no default, initially border)

Specifies the anchor where the edge from parent meets the child node by setting the macro \tikzchild anchor to $.\langle anchor \rangle$.

If you specify border as the $\langle anchor \rangle$, then the macro \tikzchildanchor is set to the empty string. The effect of this is that the edge from the parent will meet the child on the border at an automatically calculated position.



```
\begin{tikzpicture}
  \node {root}
  [child anchor=north]
  child {node {left} edge from parent[dashed]}
  child {node {right}
    child {node {child}}
    child {node {child} edge from parent[draw=none]}
  };
  \end{tikzpicture}
```

```
/tikz/parent anchor=\langle anchor \rangle
```

(no default, initially border)

This option works the same way as the child anchor, only for the parent.

All of the above describes the standard functioning of the edge from parent command. You may, however, sometimes need even more fine-grained control (the graph drawing engine needs it, for instance). In such cases the following key gives you complete control:

```
/tikz/edge from parent macro=\langle macro \rangle
```

(no default)

The $\langle macro \rangle$ gets expanded each time the edge from parent path operation is used. This $\langle macro \rangle$ must take two parameters and must expand to some text that is subsequently parsed by the parser. The first parameter will be the set of $\langle options \rangle$ that where passed to the edge from parent command, the second parameter will be the $\langle node\ specifications \rangle$ that following the command.

The standard behaviour of drawing a straight line from the parent node to the child node could be achieved by setting the $\langle macro \rangle$ to the following:

```
\def\mymacro#1#2{
  [style=edge from parent, #1]
  (\tikzparentnode\tikzparentanchor) -- #2 (\tikzchildnode\tikzchildanchor)
}
```

Note that #2 is placed between -- and the node to ensure that nodes are put "on top" of the line.

20 Plots of Functions

20.1 When Should One Use TikZ for Generating Plots?

There exist many powerful programs that produce plots, examples are GNUPLOT or MATHEMATICA. These programs can produce two different kinds of output: First, they can output a complete plot picture in a certain format (like PDF) that includes all low-level commands necessary for drawing the complete plot (including axes and labels). Second, they can usually also produce "just plain data" in the form of a long list of coordinates. Most of the powerful programs consider it a to be "a bit boring" to just output tabled data and very much prefer to produce fancy pictures. Nevertheless, when coaxed, they can also provide the plain data.

Note that is often not necessary to use TikZ for plots. Programs like GNUPLOT can produce very sophisticated plots and it is usually much easier to simply include these plots as a finished PDF or PostScript graphics.

However, there are a number of reasons why you may wish to invest time and energy into mastering the PGF commands for creating plots:

- Virtually all plots produced by "external programs" use different fonts from the one used in your document.
- Even worse, formulas will look totally different, if they can be rendered at all.
- Line width will usually be too large or too small.
- Scaling effects upon inclusion can create a mismatch between sizes in the plot and sizes in the text.
- The automatic grid generated by most programs is mostly distracting.
- The automatic ticks generated by most programs are cryptic numerics. (Try adding a tick reading " π " at the right point.)
- Most programs make it very easy to create "chart junk" in a most convenient fashion. All show, no content.
- Arrows and plot marks will almost never match the arrows used in the rest of the document.

The above list is not exhaustive, unfortunately.

20.2 The Plot Path Operation

The plot path operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

The syntax of the plot comes in different versions.

```
\path ... --plot\langle further\ arguments\rangle ...;
```

This operation plots the curve through the coordinates specified in the $\langle further\ arguments \rangle$. The current (sub)path is simply continued, that is, a line-to operation to the first point of the curve is implicitly added. The details of the $\langle further\ arguments \rangle$ will be explained in a moment.

```
\path ... plot(further arguments) ...;
```

This operation plots the curve through the coordinates specified in the $\langle further\ arguments \rangle$ by first "moving" to the first coordinate of the curve.

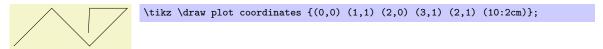
The $\langle further\ arguments \rangle$ are used in three different ways to specifying the coordinates of the points to be plotted:

```
    --plot[⟨local options⟩] coordinates {⟨coordinate 1⟩⟨coordinate 2⟩...⟨coordinate n⟩}
    --plot[⟨local options⟩] file {⟨filename⟩}
    --plot[⟨local options⟩] ⟨coordinate expression⟩
    --plot[⟨local options⟩] function {⟨gnuplot formula⟩}
```

These different ways are explained in the following.

20.3 Plotting Points Given Inline

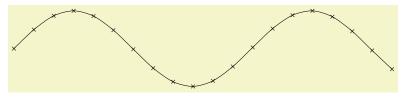
In the first two cases, the points are given directly in the TFX-file as in the following example:



Here is an example showing the difference between plot and --plot:

20.4 Plotting Points Read From an External File

The second way of specifying points is to put them in an external file named $\langle filename \rangle$. Currently, the only file format that TikZ allows is the following: Each line of the $\langle filename \rangle$ should contain one line starting with two numbers, separated by a space. Anything following the two numbers on the line is ignored. Also, lines starting with a % or a # are ignored as well as empty lines. (This is exactly the format that GNUPLOT produces when you say set terminal table.) If necessary, more formats will be supported in the future, but it is usually easy to produce a file containing data in this form.



\tikz \draw plot[mark=x,smooth] file {plots/pgfmanual-sine.table};

The file plots/pgfmanual-sine.table reads:

```
#Curve 0, 20 points

#x y type

0.00000 0.00000 i

0.52632 0.50235 i

1.05263 0.86873 i

1.57895 0.99997 i

...

9.47368 -0.04889 i

10.00000 -0.54402 i
```

It was produced from the following source, using gnuplot:

```
set table "../plots/pgfmanual-sine.table"
set format "%.5f"
set samples 20
plot [x=0:10] sin(x)
```

The (local options) of the plot operation are local to each plot and do not affect other plots "on the same path." For example, plot[yshift=1cm] will locally shift the plot 1cm upward. Remember, however, that most options can only be applied to paths as a whole. For example, plot[red] does not have the effect of making the plot red. After all, you are trying to "locally" make part of the path red, which is not possible.

20.5 Plotting a Function

When you plot a function, the coordinates of the plot data can be computed by evaluating a mathematical expression. Since PGF comes with a mathematical engine, you can specify this expression and then have TikZ produce the desired coordinates for you, automatically.

Since this case is quite common when plotting a function, the syntax is easy: Following the plot command and its local options, you directly provide a $\langle coordinate\ expression \rangle$. It looks like a normal coordinate, but inside you may use a special macro, which is \x by default, but this can be changed using the variable option. The $\langle coordinate\ expression \rangle$ is then evaluated for different values for \x and the resulting coordinates are plotted.

Note that you will often have to put the x- or y-coordinate inside braces, namely whenever you use an expression involving a parenthesis.

The following options influence how the $\langle coordinate\ expression \rangle$ is evaluated:

/tikz/variable=⟨macro⟩

(no default, initially x)

Sets the macro whose value is set to the different values when $\langle coordinate\ expression \rangle$ is evaluated.

$/\text{tikz/samples} = \langle number \rangle$

(no default, initially 25)

Sets the number of samples used in the plot.

```
/tikz/domain=\langle start \rangle : \langle end \rangle
```

(no default, initially -5:5)

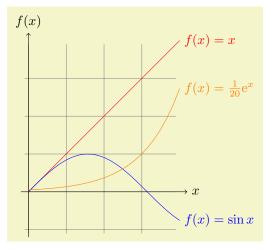
Sets the domain from which the samples are taken.

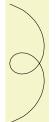
/tikz/samples at= $\langle sample \ list \rangle$

(no default)

This option specifies a list of positions for which the variable should be evaluated. For instance, you can say samples at={1,2,8,9,10} to have the variable evaluated exactly for values 1, 2, 8, 9, and 10. You can use the \foreach syntax, so you can use ... inside the \(sample \) list \(\).

When this option is used, the samples and domain option are overruled. The other way round, setting either samples or domain will overrule this option.





```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth,variable=\t]
plot ({\t*sin(\t r)},{\t*cos(\t r)});
```

20.6 Plotting a Function Using Gnuplot

Often, you will want to plot points that are given via a function like $f(x) = x \sin x$. Unfortunately, T_EX does not really have enough computational power to generate the points on such a function efficiently (it is a text processing program, after all). However, if you allow it, T_EX can try to call external programs that can easily produce the necessary points. Currently, TikZ knows how to call GNUPLOT.

When TikZ encounters your operation $plot[id=\langle id \rangle]$ function{x*sin(x)} for the first time, it will create a file called $\langle prefix \rangle \langle id \rangle$.gnuplot, where $\langle prefix \rangle$ is \jobname. by default, that is, the name of you main .tex file. If no $\langle id \rangle$ is given, it will be empty, which is alright, but it is better when each plot has a unique $\langle id \rangle$ for reasons explained in a moment. Next, TikZ writes some initialization code into this file followed by plot x*sin(x). The initialization code sets up things such that the plot operation will write the coordinates into another file called $\langle prefix \rangle \langle id \rangle$.table. Finally, this table file is read as if you had said plot file{ $\langle prefix \rangle \langle id \rangle$.table}. However, there is just one difference: gnuplot supports a "type" field following the coordinates. If this type field contains 'u' which means unbounded, TikZ will ignore the complete coordinate¹⁰.

For the plotting mechanism to work, two conditions must be met:

- 1. You must have allowed TEX to call external programs. This is often switched off by default since this is a security risk (you might, without knowing, run a TEX file that calls all sorts of "bad" commands). To enable this "calling external programs" a command line option must be given to the TEX program. Usually, it is called something like shell-escape or enable-write18. For example, for my pdflatex the option --shell-escape can be given.
- 2. You must have installed the gnuplot program and TFX must find it when compiling your file.

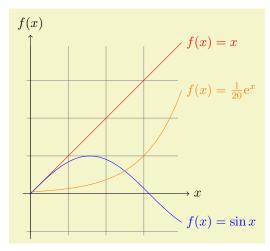
Unfortunately, these conditions will not always be met. Especially if you pass some source to a coauthor and the coauthor does not have GNUPLOT installed, he or she will have trouble compiling your files.

For this reason, TikZ behaves differently when you compile your graphic for the second time: If upon reaching plot[id= $\langle id \rangle$] function{...} the file $\langle prefix \rangle \langle id \rangle$.table already exists and if the $\langle prefix \rangle \langle id \rangle$.gnuplot file contains what TikZ thinks that it "should" contain, the .table file is immediately read without trying to call a gnuplot program. This approach has the following advantages:

- 1. If you pass a bundle of your .tex file and all .gnuplot and .table files to someone else, that person can TEX the .tex file without having to have gnuplot installed.
- 2. If the \write18 feature is switched off for security reasons (a good idea), then, upon the first compilation of the .tex file, the .gnuplot will still be generated, but not the .table file. You can then simply call gnuplot "by hand" for each .gnuplot file, which will produce all necessary .table files.
- 3. If you change the function that you wish to plot or its domain, TikZ will automatically try to regenerate the .table file.
- 4. If, out of laziness, you do not provide an id, the same .gnuplot will be used for different plots, but this is not a problem since the .table will automatically be regenerated for each plot on-the-fly. Note: If you intend to share your files with someone else, always use an id, so that the file can by typeset without having GNUPLOT installed. Also, having unique ids for each plot will improve compilation speed since no external programs need to be called, unless it is really necessary.

When you use plot function{ $\langle gnuplot\ formula \rangle$ }, the $\langle gnuplot\ formula \rangle$ must be given in the gnuplot syntax, whose details are beyond the scope of this manual. Here is the ultra-condensed essence: Use x as the variable and use the C-syntax for normal plots, use t as the variable for parametric plots. Here are some examples:

¹⁰Thanks to Andy Schlaikjer for this patch.

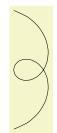


The plot is influenced by the following options: First, the options samples and domain explained earlier. Second, there are some more specialized options.

/tikz/parametric=\langle boolean \rangle

(default true)

Sets whether the plot is a parametric plot. If true, then t must be used instead of x as the parameter and two comma-separated functions must be given in the $\langle gnuplot \ formula \rangle$. An example is the following:



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth]
plot[parametric,id=parametric-example] function{t*sin(t),t*cos(t)};
```

 $/\text{tikz/id} = \langle id \rangle$ (no default)

Sets the identifier of the current plot. This should be a unique identifier for each plot (though things will also work if it is not, but not as well, see the explanations above). The $\langle id \rangle$ will be part of a filename, so it should not contain anything fancy like * or \$.

```
/\text{tikz/prefix} = \langle prefix \rangle (no default)
```

The $\langle prefix \rangle$ is put before each plot file name. The default is $\jobname.$, but if you have many plots, it might be better to use, say plots/ and have all plots placed in a directory. You have to create the directory yourself.

/tikz/raw gnuplot (no value)

This key causes the $\langle gnuplot \ formula \rangle$ to be passed on to GNUPLOT without setting up the samples or the plot operation. Thus, you could write

```
plot[raw gnuplot,id=raw-example] function{set samples 25; plot sin(x)}
```

This can be useful for complicated things that need to be passed to GNUPLOT. However, for really complicated situations you should create a special external generating GNUPLOT file and use the file-syntax to include the table "by hand."

The following styles influence the plot:

/tikz/every plot

This style is installed in each plot, that is, as if you always said

plot[every plot,...]

This is most useful for globally setting a prefix for all plots by saying:

\tikzset{every plot/.style={prefix=plots/}}

20.7 Placing Marks on the Plot

As we saw already, it is possible to add *marks* to a plot using the mark option. When this option is used, a copy of the plot mark is placed on each point of the plot. Note that the marks are placed *after* the whole path has been drawn/filled/shaded. In this respect, they are handled like text nodes.

In detail, the following options govern how marks are drawn:

/tikz/mark=\langle mark mnemonic\rangle

(no default)

(style, initially empty)

Sets the mark to a mnemonic that has previously been defined using the \pgfdeclareplotmark. By default, *, +, and x are available, which draw a filled circle, a plus, and a cross as marks. Many more marks become available when the library plotmarks is loaded. Section 44.6 lists the available plot marks.

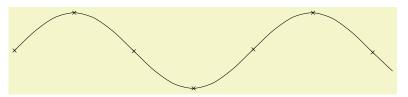
One plot mark is special: the ball plot mark is available only in TikZ. The ball color option determines the balls's color. Do not use this option with a large number of marks since it will take very long to render in PostScript.

Option Effect

/tikz/mark repeat= $\langle r \rangle$

(no default)

This option tells TikZ that only every rth mark should be drawn.



\tikz \draw plot[mark=x,mark repeat=3,smooth] file {plots/pgfmanual-sine.table};

/tikz/mark phase= $\langle p \rangle$

(no default)

This option tells $\text{Ti}k\mathbf{Z}$ that the first mark to be draw should be the pth, followed by the (p+r)th, then the (p+2r)th, and so on.

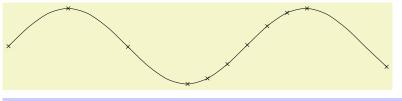


\tikz \draw plot[mark=x,mark repeat=3,mark phase=6,smooth] file {plots/pgfmanual-sine.table};

/tikz/mark indices= $\langle list \rangle$

(no default)

This option allows you to specify explicitly the indices at which a mark should be placed. Counting starts with 1. You can use the \foreach syntax, that is, ... can be used.



\tikz \draw plot[mark=x,mark indices={1,4,...,10,11,12,...,16,20},smooth]
file {plots/pgfmanual-sine.table};

/tikz/mark size=\langle dimension\rangle

(no default)

Sets the size of the plot marks. For circular plot marks, $\langle dimension \rangle$ is the radius, for other plot marks $\langle dimension \rangle$ should be about half the width and height.

This option is not really necessary, since you achieve the same effect by specifying $scale=\langle factor \rangle$ as a local option, where $\langle factor \rangle$ is the quotient of the desired size and the default size. However, using mark size is a bit faster and more natural.

/tikz/every mark

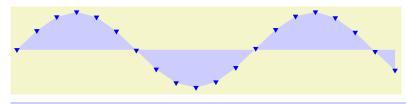
(style, no value)

This style is installed before drawing plot marks. For example, you can scale (or otherwise transform) the plot mark or set its color.

/tikz/mark options=⟨options⟩

(no default)

Redefines every mark such that it sets $\{\langle options \rangle\}$.



\tikz \fill[fill=blue!20]
plot[mark=triangle*,mark options={color=blue,rotate=180}]
file{plots/pgfmanual-sine.table} |- (0,0);

/tikz/no marks

(style, no value)

Disables markers (the same as mark=none).

/tikz/no markers

(style, no value)

Disables markers (the same as mark=none).

20.8 Smooth Plots, Sharp Plots, Jump Plots, Comb Plots and Bar Plots

There are different things the plot operation can do with the points it reads from a file or from the inlined list of points. By default, it will connect these points by straight lines. However, you can also use options to change the behavior of plot.

/tikz/sharp plot

(no value)

This is the default and causes the points to be connected by straight lines. This option is included only so that you can "switch back" if you "globally" install, say, smooth.

/tikz/smooth

(no value)

This option causes the points on the path to be connected using a smooth curve:

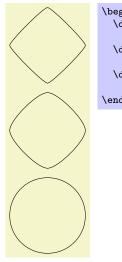


```
\tikz\draw plot[smooth] file{plots/pgfmanual-sine.table};
```

Note that the smoothing algorithm is not very intelligent. You will get the best results if the bending angles are small, that is, less than about 30° and, even more importantly, if the distances between points are about the same all over the plotting path.

```
/\text{tikz/tension} = \langle value \rangle (no default)
```

This option influences how "tight" the smoothing is. A lower value will result in sharper corners, a higher value in more "round" curves. A value of 1 results in a circle if four points at quarter-positions on a circle are given. The default is 0.55. The "correct" value depends on the details of plot.



/tikz/smooth cycle (no value)

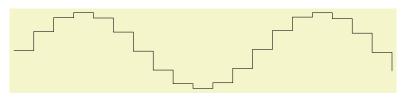
This option causes the points on the path to be connected using a closed smooth curve.



```
\tikz[scale=0.5]
\draw plot[smooth cycle] coordinates{(0,0) (1,0) (2,1) (1,2)}
plot coordinates{(0,0) (1,0) (2,1) (1,2)} -- cycle;
```

/tikz/const plot (no value)

This option causes the points on the path to be connected using piecewise constant series of lines:

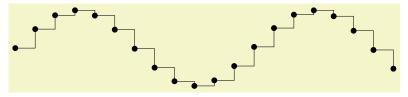


\tikz\draw plot[const plot] file{plots/pgfmanual-sine.table};

/tikz/const plot mark left

(no value)

Just an alias for /tikz/const plot.

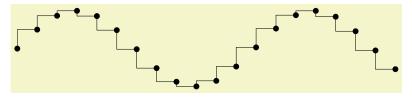


\tikz\draw plot[const plot mark left,mark=*] file{plots/pgfmanual-sine.table};

/tikz/const plot mark right

(no value)

A variant of /tikz/const plot which places its mark on the right ends:

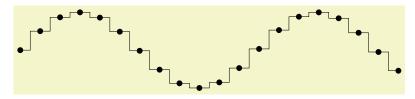


\tikz\draw plot[const plot mark right,mark=*] file{plots/pgfmanual-sine.table};

/tikz/const plot mark mid

(no value)

A variant of /tikz/const plot which places its mark in the middle of the horizontal lines:



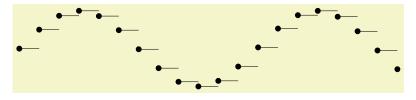
\tikz\draw plot[const plot mark mid,mark=*] file{plots/pgfmanual-sine.table};

More precisely, it generates vertical lines in the middle between each pair of consecutive points. If the mesh width is constant, this leads to symmetrically placed marks ("middle").

/tikz/jump mark left

(no value)

This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed on left open ends:

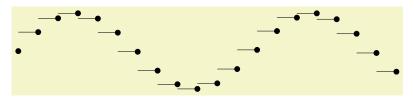


\tikz\draw plot[jump mark left, mark=*] file{plots/pgfmanual-sine.table};

/tikz/jump mark right

(no value)

This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed on right open ends:

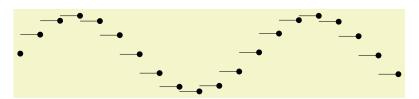


\tikz\draw plot[jump mark right, mark=*] file{plots/pgfmanual-sine.table};

/tikz/jump mark mid

(no value)

This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed in the middle of the horizontal line segments:



```
\tikz\draw plot[jump mark right, mark=*] file{plots/pgfmanual-sine.table};
```

In case of non-constant mesh widths, the same remarks as for const plot mark mid apply.

/tikz/ycomb (no value)

This option causes the plot operation to interpret the plotting points differently. Instead of connecting them, for each point of the plot a straight line is added to the path from the x-axis to the point, resulting in a sort of "comb" or "bar diagram."

\tikz\draw[ultra thick] plot[ycomb,thin,mark=*] file{plots/pgfmanual-sine.table};

```
flut
```

```
\begin{tikzpicture}[ycomb]
\draw[color=red,line width=6pt]

plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
\draw[color=red!50,line width=4pt,xshift=3pt]

plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

/tikz/xcomb (no value)

This option works like ycomb except that the bars are horizontal.

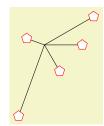


```
\tikz \draw plot[xcomb,mark=x] coordinates{(1,0) (0.8,0.2) (0.6,0.4) (0.2,1)};
```

/tikz/polar comb

(no value)

This option causes a line from the origin to the point to be added to the path for each plot point.



```
\tikz \draw plot[polar comb,
    mark=pentagon*,mark options={fill=white,draw=red},mark size=4pt]
    coordinates {(0:1cm) (30:1.5cm) (160:.5cm) (250:2cm) (-60:.8cm)};
```

/tikz/ybar (no value)

This option produces fillable bar plots. It is thus very similar to ycomb, but it employs rectangular shapes instead of line-to operations. It thus allows to use any fill- or pattern style.

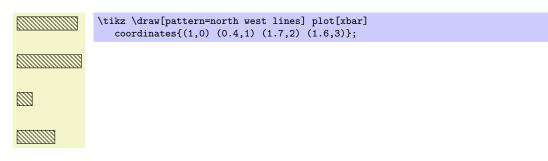


\tikz\draw[draw=blue,fill=blue!60!black] plot[ybar] file{plots/pgfmanual-sine.table};

The use of bar width and bar shift is explained in the plot handler library documentation, section 44.4. Please refer to page 443.

/tikz/xbar (no value)

This option works like ybar except that the bars are horizontal.



/tikz/ybar interval (no value)

As /tikz/ybar, this options produces vertical bars. However, bars are centered at coordinate *intervals* instead of interval edges, and the bar's width is also determined relatively to the interval's length:

```
\begin{tikzpicture}[ybar interval,x=10pt]
   \draw[color=red,fill=red!80]
    plot coordinates{(0,2) (2,1.2) (3,.3) (5,1.7) (8,.9) (9,.9)};
   \end{tikzpicture}
```

Since there are N intervals $[x_i, x_{i+1}]$ for given N+1 coordinates, you will always have one coordinate more than bars. The last y value will be ignored.

You can configure relative shifts and relative bar widths, which is explained in the plot handler library documentation, section 44.4. Please refer to page 444.

/tikz/xbar interval (no value)

Works like ybar interval, but for horizontal bar plots.

```
\begin{tikzpicture}[xbar interval,x=0.5cm,y=0.5cm]
  \draw[color=red,fill=red!80]
    plot coordinates {(3,0) (2,1) (4,1.5) (1,4) (2,6) (2,7)};
  \end{tikzpicture}
```

/tikz/only marks (no value)

This option causes only marks to be shown; no path segments are added to the actual path. This can be useful for quickly adding some marks to a path.

```
\tikz \draw (0,0) sin (1,1) cos (2,0)
plot[only marks,mark=x] coordinates{(0,0) (1,1) (2,0) (3,-1)};

x
```

21 Transparency

21.1 Overview

Normally, when you paint something using any of TikZ's commands (this includes stroking, filling, shading, patterns, and images), the newly painted objects totally obscure whatever was painted earlier in the same area.

You can change this behaviour by using something that can be thought of as "(semi)transparent colors." Such colors do not completely obscure the background, rather they blend the background with the new color. At first sight, using such semitransparent colors might seem quite straightforward, but the math going on in the background is quite involved and the correct handling of transparency fills some 64 pages in the PDF specification.

In the present section, we start with the different ways of specifying "how transparent" newly drawn objects should be. The simplest way is to just specify a percentage like "60% transparent." A much more general way is to use something that I call a *fading*, also known as a soft mask or a mask.

At the end of the section we address the problem of creating so-called *transparency groups*. This problem arises when you paint over a position several times with a semitransparent color. Sometimes you want the effect to accumulate, sometimes you do not.

Note: Transparency is best supported by the pdfTEX driver. The SVG driver also has some support. For PostScript output, opacity is rendered correctly only with the most recent versions of Ghostscript. Printers and other programs will typically ignore the opacity setting.

21.2 Specifying a Uniform Opacity

Specifying a stroke and/or fill opacity is quite easy using the following options.

```
/tikz/draw opacity=\langle value \rangle
```

(no default)

This option sets "how transparent" lines should be. A value of 1 means "fully opaque" or "not transparent at all," a value of 0 means "fully transparent" or "invisible." A value of 0.5 yields lines that are semitransparent.

Note that when you use PostScript as your output format, this option works only with recent versions of Ghostscript.

```
\begin{tikzpicture}[line width=1ex]
  \draw (0,0) -- (3,1);
  \filldraw [fill=examplefill,draw opacity=0.5] (1,0) rectangle (2,1);
  \end{tikzpicture}
```

Note that the draw opacity options only sets the opacity of drawn lines. The opacity of fillings is set using the option fill opacity (documented in Section 15.4.3. The option opacity sets both at the same time.

```
/tikz/opacity=\langle value \rangle
```

(no default)

Sets both the drawing and filling opacity to $\langle value \rangle$.

The following predefined styles make it easier to use this option:

/tikz/transparent

(style, no value)

Makes everything totally transparent and, hence, invisible.

```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[transparent,red] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/ultra nearly transparent

(style, no value)

Makes everything, well, ultra nearly transparent.

/tikz/very nearly transparent

(style, no value)

```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[very nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/nearly transparent

(style, no value)

```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/semitransparent

(style, no value)

```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[semitransparent] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/nearly opaque

(style, no value)

```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/very nearly opaque

(style, no value)

```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[very nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/ultra nearly opaque

(style, no value)

```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[ultra nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/opaque

(style, no value)

This yields completely opaque drawings, which is the default.

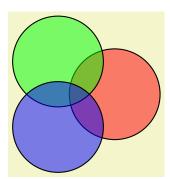
```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[opaque] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/fill opacity= $\langle value \rangle$

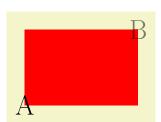
(no default)

This option sets the opacity of fillings. In addition to filling operations, this opacity also applies to text and images.

Note, again, that when you use PostScript as your output format, this option works only with recent versions of Ghostscript.



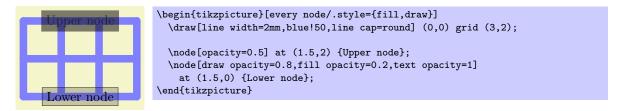
```
\begin{tikzpicture}[thick,fill opacity=0.5]
  \filldraw[fill=red] (0:1cm) circle (12mm);
  \filldraw[fill=green] (120:1cm) circle (12mm);
  \filldraw[fill=blue] (-120:1cm) circle (12mm);
  \end{tikzpicture}
```



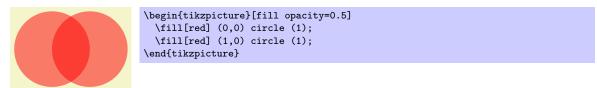
```
/tikz/text opacity=\langle value \rangle
```

(no default)

Sets the opacity of text labels, overriding the fill opacity setting.



Note the following effect: If you set up a certain opacity for stroking or filling and you stroke or fill the same area twice, the effect accumulates:



Often, this is exactly what you intend, but not always. You can use transparency groups, see the end of this section, to change this.

21.3 Fadings

For complicated graphics, uniform transparency settings are not always sufficient. Suppose, for instance, that while you paint a picture, you want the transparency to vary smoothly from completely opaque to completely transparent. This is a "shading-like" transparency. For such a form of transparency I will use the term fading (as a noun). They are also known as soft masks, opacity masks, masks, or soft clips.

21.3.1 Creating Fadings

How do we specify a fading? This is a bit of an art since the underlying mechanism is quite powerful, but a bit difficult to use.

Let us start with a bit of terminology. A *fading* specifies for each point of an area the transparency of that point. This transparency can by any number between 0 and 1. A *fading picture* is a normal graphic that, in a way to be described in a moment, determines the transparency of points inside the fading. Each fading has an underlying fading picture.

The fading picture is a normal graphic drawn using any of the normal graphic drawing commands. A fading and its fading picture are related as follows: Given any point of the fading, the transparency of this point is determined by the luminosity of the fading picture at the same position. The luminosity of a point determines "how bright" the point is. The brighter the point in the fading picture, the more opaque is the point in the fading. In particular, a white point of the fading picture is completely opaque in the fading and a black point of the fading picture is completely transparent in the fading. (The background of the fading picture is always transparent in the fading as if the background were black.)

It is rather counter-intuitive that a *white* pixel of the fading picture will be *opaque* in the fading and a *black* pixel will be *transparent*. For this reason, TikZ defines a color called **transparent** that is the same as **black**. The nice thing about this definition is that the color **transparent**! $\langle percentage \rangle$ in the fading picture yields a pixel that is $\langle percentage \rangle$ percent transparent in the fading.

Turning a fading picture into a normal picture is achieved using the following commands, which are *only defined in the library*, namely the library fadings. So, to use them, you have to say \usetikzlibrary{fadings} first.

```
\label{lem:contents} $$\left(\operatorname{contions}\right) $$ \left(\operatorname{contions}\right) $$ \left(\operatorname{contents}\right) $$ \left(\operatorname{contents}\right) $$ \left(\operatorname{contents}\right) $$
```

This command works like a {tikzpicture}, only the picture is not shown, but instead a fading is defined based on this picture. To set the name of the picture, use the name option (which is normally used to set the name of a node).

```
\ttikz/name=\{\langle name \rangle\} (no default)
```

Use this option with the {tikzfadingfrompicture} environment to set the name of the fading. You must provide this option.

The following shading is 2cm by 2cm and gets more and more transparent from left to right, but is 50% transparent for a large circle in the middle.



In the next example we create a fading picture that contains some text. When the fading is used, we only see the shading "through it."



```
\label{lem:contents} $$ \left( environment\ contents \right) $$ \left( entire \left( environment\ contents \right) \right) $$
```

The plainTEX version of the environment.

```
\label{lem:contents} $$ \arttikzfadingfrompicture [\langle options \rangle] $$ \arrowvert contents $$
```

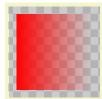
The ConT_EXt version of the environment.

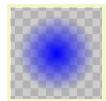
$\texttt{\tikzfading}[\langle options \rangle]$

This command is used to define a fading similarly to the way a shading is defined. In the $\langle options \rangle$ you should

- 1. use the name= $\langle name \rangle$ option to set a name for the fading,
- 2. use the shading option to set the name of the shading that you wish to use,
- 3. extra options for setting the colors of the shading (typically you will set them to the color transparent!(percentage)).

Then, a new fading named $\langle name \rangle$ will be created based on the shading.





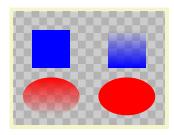
21.3.2 Fading a Path

A fading specifies for each pixel of a certain area how transparent this pixel will be. The following options are used to install such a fading for the current scope or path.

/tikz/path fading= $\langle name \rangle$

(default scope's setting)

This option tells $\text{Ti}k\mathbf{Z}$ that the current path should be faded with the fading $\langle name \rangle$. If no $\langle name \rangle$ is given, the $\langle name \rangle$ set for the whole scope is used. Similarly to options like draw or fill, this option is reset for each path, so you have to add it to each path that should be faded. You can also specify none as $\langle name \rangle$, in which case fading for the path will be switched off in case it has been switched on by previous options or styles.



/tikz/fit fading= $\langle boolean \rangle$

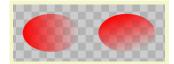
(default true, initially true)

When set to true, the fading is shifted and resized (in exactly the same way as a shading) so that it covers the current path. When set to false, the fading is only shifted so that it is centered on the path's center, but it is not resized. This can be useful for special-purpose fadings, for instance when you use a fading to "punch out" something.

```
/tikz/fading transform=\langle transformation options\rangle
```

(no default)

The $\langle transformation \ options \rangle$ are applied to the fading before it is used. For instance, if $\langle transformation \ options \rangle$ is set to rotate=90, the fading is rotated by 90 degrees.



```
\begin{tikzpicture}[path fading=fade down]
    % Checker board
    \fill [black!20] (0,0) rectangle (4,1.5);
    \path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,1.5);

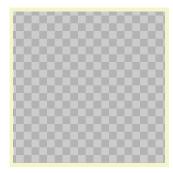
\fill [red,path fading,fading transform={rotate=90}]
    (1,0.75) ellipse (.75 and .5);
\fill [red,path fading,fading transform={rotate=30}]
    (3,0.75) ellipse (.75 and .5);
\end{tikzpicture}
```

```
/tikz/fading angle=\langle degree \rangle
```

(no default)

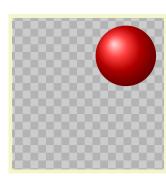
A shortcut for fading transform={rotate= $\langle degree \rangle$ }.

Note that you can "fade just about anything." In particular, you can fade a shading.



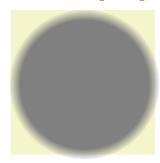
```
\begin{tikzpicture}
  % Checker board
  \fill [black!20] (0,0) rectangle (4,4);
  \path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,4);
  \shade [ball color=blue,path fading=south] (2,2) circle (1.8);
  \end{tikzpicture}
```

The fade inside of the following example is more transparent in the middle than on the outside.



Note that adding the path fading option to a node fades the (background) path, not the text itself. To fade the text, you need to use a scope fading (see below).

Note that using fadings in conjunction with patterns can create visually rather pleasing effects:



```
\tikzfading[name=middle,
            top color=transparent!50,
            bottom color=transparent!50,
            middle color=transparent!20]
\begin{tikzpicture}
             [circle, circular drop shadow,
  \node
              pattern=horizontal lines dark blue,
              path fading=south,
              minimum size=3.6cm] {};
  \pattern
             [path fading=north,
              pattern=horizontal lines dark gray]
    (0,0) circle (1.8cm);
  \pattern
             [path fading=middle,
              pattern=crosshatch dots light steel blue]
    (0,0) circle (1.8cm);
\end{tikzpicture}
```

21.3.3 Fading a Scope

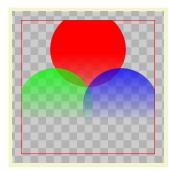
In addition to fading individual paths, you may also wish to "fade a scope," that is, you may wish to install a fading that is used globally to specify the transparency for all objects drawn inside a scope. This effect can also be thought of as a "soft clip" and it works in a similar way: You add the scope fading option to a path in a scope – typically the first one – and then all subsequent drawings in the scope are faded. You will use a transparency group in conjunction, see the end of this section.

```
/tikz/scope fading=\langle fading \rangle
```

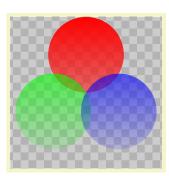
(no default)

In principle, this key works in exactly the same way as the path fading key. The only difference is, that the effect of the fading will persist after the current path till the end of the scope. Thus, the \langle fading \rangle is applied to all subsequent drawings in the current scope, not just to the current path. In this regard, the option works very much like the clip option. (Note, however, that, unlike the clip option, fadings to not accumulate unless a transparency group is used.)

The keys fit fading and fading transform have the same effect as for path fading. Also that, just as for path fading, providing the scope fading option with a {scope} only sets the name of the fading to be used. You have to explicitly provide the scope fading with a path to actually install a fading.



In the following example we resize the fading to the size of the whole picture:



Scope fadings are also needed if you wish to fade a node.

This is some text that will fade out as we go right and down. It is pretty hard to achieve this effect in other ways.

```
\tikz \node [scope fading=south,fading angle=45,text width=3.5cm]
{
   This is some text that will fade out as we go right
   and down. It is pretty hard to achieve this effect in
   other ways.
};
```

21.4 Transparency Groups

Consider the following cross and sign. They "look wrong" because we can see how they were constructed, while this is not really part of the desired effect.



```
\begin{tikzpicture}[opacity=.5]
  \draw [line width=5mm] (0,0) -- (2,2);
  \draw [line width=5mm] (2,0) -- (0,2);
  \end{tikzpicture}
```



```
\begin{tikzpicture}
  \node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};
  \node [opacity=.5]
      at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};
  \end{tikzpicture}
```

Transparency groups are used to render them correctly:



```
\begin{tikzpicture}[opacity=.5]
  \begin{scope}[transparency group]
  \draw [line width=5mm] (0,0) -- (2,2);
  \draw [line width=5mm] (2,0) -- (0,2);
  \end{scope}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};
  \begin{scope}[opacity=.5,transparency group]
  \node at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
    {Smoking};
  \end{scope}
\end{tikzpicture}
```

/tikz/transparency group

(no value)

This option can be given to a scope. It will have the following effect: The scope's contents is stroked/-filled "ignoring any outside transparency." This means, all previous transparency settings are ignored (you can still set transparency inside the group, but never mind). For instance, in the forbidden sign example, the whole sign is first painted (conceptually) like the image on the left hand side. Note that some pixels of the sign are painted multiple times (up to three times), but only the last color "wins."

Then, when the scope is finished, it is painted as a whole. The *fill* transparency settings are now applied to the resulting picture. For instance, the pixel that has been painted three times is just red at the end, so this red color will be blended with whatever is "behind" the group on the page.

Note that, depending on the driver, it is possible to directly put objects in a transparency group that lie outside the picture. This has to do with internal bounding box computations. Section 92 explains how to sidestep this problem.

22 Decorated Paths

22.1 Overview

Decorations are a general concept to make (sub)paths "more interesting." Before we have a look at the details, let us have a look at some examples:

```
\begin{tikzpicture}[thick]
                                                                               (0,3)
                                                                                       -- (3,3);
                                                                               (0,2.5) -- (3,2.5);
                        \draw[decorate.decoration=zigzag]
                        \draw[decorate,decoration=brace]
                                                                               (0,2)
                                                                                       -- (3,2);
                        \draw[decorate,decoration=triangles]
                                                                               (0,1.5) -- (3,1.5);
                        \draw[decorate,decoration={coil,segment length=4pt}] (0,1)
                                                                                       -- (3,1);
                        \draw[decorate,decoration={coil,aspect=0}]
                                                                               (0,.5)
                                                                                       -- (3,.5);
                        \draw[decorate,decoration={expanding waves,angle=7}] (0,0)
                                                                                       -- (3,0);
                      \end{tikzpicture}
                      \begin{tikzpicture}
                        \node [fill=red!20,draw,decorate,decoration={bumps,mirror},
                               minimum height=1cm]
                        {Bumpy};
                       \end{tikzpicture}
                      \begin{tikzpicture}
                        \filldraw[fill=blue!20]
                                                                     (0,3)
                        decorate [decoration=saw]
                                                                    (3,3) }
                        decorate [decoration={coil,aspect=0}] { -- (2,1) }
                        decorate [decoration=bumps]
                                                                \{ -1 (0.3) \}:
                      \end{tikzpicture}
                      \begin{tikzpicture}
                        \node [fill=yellow!50,draw,thick, minimum height=2cm, minimum width=3cm,
                               decorate, decoration={random steps,segment length=3pt,amplitude=1pt}]
Saved from trash
                          {Saved from trash};
                      \end{tikzpicture}
```

The general idea of decorations is the following: First, you construct a path using the usual path construction commands. The resulting path is, in essence, a series of straight and curved lines. Instead of directly using this path for filling or drawing, you can then specify that it should form the basis for a decoration. In this case, depending on which decoration you use, a new path is constructed "along" the path you specified. For instance, with the zigzag decoration, the new path is a zigzagging line that goes along the old path.

Let us have a look at an example: In the first picture, we see a path that consists of a line, an arc, and a line. In the second picture, this path has been used as the basis of a decoration.

```
\tikz \fill
[fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;

\tikz \fill [decorate,decoration={zigzag}]
[fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

It is also possible to decorate only a subpath (the exact syntax will be explained later in this section).

```
\tikz \fill [decoration={zigzag}]
[fill=blue!20,draw=blue,thick] (0,0) -- (2,1)
  decorate { arc (90:-90:.5) } -- cycle;
```

The **zigzag** decoration will be called a *path morphing* decoration because it morphs a path into a different, but topologically equivalent path. Not all decorations are path morphing; rather there are three kinds of decorations.

- 1. The just-mentioned *path morphing* decorations morph the path in the sense that what used to be a straight line might afterwards be a squiggly line or might have bumps. However, a line is still and a line and path deforming decorations do not change the number of subpaths.
 - Examples of such decorations are the snake or the zigzag decoration. Many such decorations are defined in the library decorations.pathmorphing.
- 2. Path replacing decorations completely replace the path by a different path that is only "loosely based" on the original path. For instance, the crosses decoration replaces a path by a path consisting of a sequence of crosses. Note how in the following example filling the path has no effect since the path consist only of (numerous) unconnected straight line subpaths:

```
\tikz \fill [decorate,decoration={crosses}]

[fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

Examples of path replacing decorations are crosses or ticks or shape backgrounds. Such decorations are defined in the library decorations.pathreplacing, but also in decorations.shapes.

3. Path removing decorations completely remove the to-be-decorated path. Thus, they have no effect on the main path that is being constructed. Instead, they typically have numerous side effects. For instance, they might "write some text" along the (removed) path or they might place nodes along this path. Note that for such decorations the path usage command for the main path have no influence on how the decoration looks like.

```
tikz \fill [decorate, decoration={text along path, text=This is a text along a path. Note how the path is lost.}]

MOU PROVE 'U180

(tikz \fill [decorate, decoration={text along path, text=This is a text along a path. Note how the path is lost.}]

[fill=blue!20, draw=blue, thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

Decorations are defined in different decoration libraries, see Section 31 for details. It is also possible to define your own decorations, see Section 80, but you need to use the PGF basic layer and a bit of theory is involved.

Decorations can be used to decorate already decorated paths. In the following three graphics, we start with a simple path, then decorate it once, and then decorate the decorated path once more.

One final word of warning: Decorations can be pretty slow to typeset and they can be inaccurate. The reason is that PGF has to a *lot* of rather difficult computations in the background and TEX is not very good at doing math. Decorations are fastest when applied to straight line segments, but even then they are

much slower than other alternatives. For instance, the ticks decoration can be simulated by clever use of a dashing pattern and the dashing pattern will literally be thousands of times faster to typeset. However, for most decorations there are no real alternatives.

```
\usetikzlibrary{decorations} % MEX and plain TEX \usetikzlibrary[decorations] % ConTEXt
```

In order to use decorations, you first have to load a decoration library. This decoration library defines the basic options described in the following, but it does not define any new decorations. This is done by libraries like decorations.text. Since these more specialized libraries include the decoration library automatically, you usually do not have to bother about it.

22.2 Decorating a Subpath Using the Decorate Path Command

The most general way to decorate a (sub)path is the following path command.

```
\path ... decorate[\langle options \rangle] \{\langle subpath \rangle\} ...;
```

This path operation causes the $\langle subpath \rangle$ to be decorated using the current decoration. Depending on the decoration, this may or may not extend the current path.

```
begin{tikzpicture}
    \draw [help lines] grid (3,2);
    \draw decorate [decoration={name=zigzag}]
        { (0,0) .. controls (0,2) and (3,0) .. (3,2) |- (0,0) };
    \end{tikzpicture}
```

The path can include straight lines, curves, rectangles, arcs, circles, ellipses, and even already decorated paths (that is, you can nest applications of the **decorate** path command, see below).

Due to the limits on the precision in T_EX, some inaccuracies in positioning when crossing input segment boundaries may occasionally be found.

You can use nodes normally inside the $\langle subpath \rangle$.

```
begin{tikzpicture}
    \draw [help lines] grid (3,2);
    \draw decorate [decoration={name=zigzag}]
        { (0,0) -- (2,2) node (hi) [left,draw=red] {Hi!} arc(90:0:1)};
    \draw [blue] decorate [decoration={crosses}] {(3,0) -- (hi)};
    \end{tikzpicture}
```

The following key is used to select the decoration and also to select further "rendering options" for the decoration.

```
/pgf/decoration = \langle decoration \ options \rangle (no default) alias /tikz/decoration
```

This option is used to specify which decoration is used and how it will look like. Note that this key will *not* cause any decorations to be applied, immediately. It takes the decorate path command or the decorate option to actually decorate a path. The decoration option is only used to specify which decoration should be used, in principle. You can also use this option at the beginning of a picture or a scope to specify the decoration to be used with each invocation of the decorate path command. Naturally, any local options of the decorate path command override these "global" options.

The $\langle decoration \ options \rangle$ are special options (which have the path prefix /pgf/decoration/) that determine the properties of the decoration. Which options are appropriate for a decoration strongly

depend on the decoration, you will have to look up the appropriate options in the documentation of the decoration, see Section 31.

There is one option (available only in TikZ) that is special:

```
/pgf/decoration/name = \langle name \rangle
```

(no default, initially none)

Use this key to set which decoration is to be used. The $\langle name \rangle$ can both be a decoration or a meta-decoration (you need to worry about the difference only if you wish to define your own decorations).

If you set $\langle name \rangle$ to none, no decorations are added.

Since this option is used so often, you can also leave out the name= part. Thus, the above example can be rewritten more succinctly:

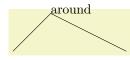


```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw decorate [decoration=zigzag]
      { (0,0) .. controls (0,2) and (3,0) .. (3,2) };
  \end{tikzpicture}
```

In general, when $\langle decoration \ options \rangle$ are parsed, for each unknown key it is checked whether that key happens to be a (meta-)decoration and, if so, the name option is executed for this key.

Further options allow you to adjust the position of decorations relative to the to-be-decorated path. See Section 22.4 below for details.

Recall that some decorations actually completely remove the to-be-decorated path. In such cases, the construction of the main path is resumed after the **decorate** path command ends.



It is permissible to nest **decorate** commands. In this case, the path resulting from the first decoration process is used as the to-be-decorated path for the second decoration process. This is especially useful for drawing fractals. The Koch snowflake decoration replaces a straight line like _____ by _____. Repeatedly applying this transformation to a triangle yields a fractal that looks a bit like a snowflake, hence the name.



```
\begin{tikzpicture}[decoration=Koch snowflake,draw=blue,fill=blue!20,thick]
  \filldraw (0,0) -- ++(60:1) -- ++(-60:1) -- cycle ;
  \filldraw decorate{ (0,-1) -- ++(60:1) -- ++(-60:1) -- cycle };
  \filldraw decorate{ decorate{ (0,-2.5) -- ++(60:1) -- ++(-60:1) -- cycle }};
  \end{tikzpicture}
```

22.3 Decorating a Complete Path

You may sometimes wish to decorate a path over whose construction you have no control. For instance, the path of the background of a node is created without having a chance to issue a decorate path command. In such cases you can use the following option, which allows you to decorate a path "after the fact."

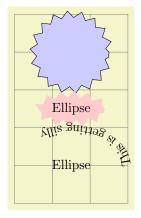
```
/tikz/decorate=\langle boolean \rangle
```

(default true)

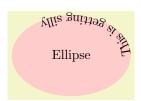
When this key is set, the whole path is decorated after it has been finished. The decoration used for decorating the path is set via the decoration way, in exactly the same way as for the decorate path command. Indeed, the following two commands have the same effect:

- 1. \path decorate $[\langle options \rangle]$ $\{\langle path \rangle\};$
- 2. \path [decorate, $\langle options \rangle$] $\langle path \rangle$;

The main use or the decorate option is the you can also use it with the nodes. It then causes the background path of the node to be decorated. Note that you can decorate a background path only once in this manner. That is, in contrast to the decorate path command you cannot apply this option twice (this would just set it to true, once more).

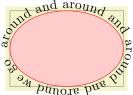


In the last example, the text along path decoration removes the path. In such cases it is useful to use a pre- or postaction to cause the decoration to be applied only before or after the main path has been used. Incidentally, this is another application of the decorate option that you cannot achieve with the decorate path command.



```
\begin{tikzpicture}[decoration=zigzag]
  \node at (1.5,1) [inner sep=6mm,fill=red!20,ellipse,
    postaction={decorate,decoration=
    {text along path,text={This is getting silly}}}] {Ellipse};
  \end{tikzpicture}
```

Here is more useful example, where a postaction is used to add the path after the main path has been drawn.



22.4 Adjusting Decorations

22.4.1 Positioning Decorations Relative to the To-Be-Decorate Path

The following option, which are only available with TikZ, allow you to modify the positioning of decorations relative to the to-be-decorated path.

```
/pgf/decoration/raise=\langle dimension \rangle
```

(no default, initially Opt)

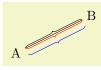
The segments of the decoration are raised by $\langle dimension \rangle$ relative to the to-be-decorated path. More precisely, the segments of the path are offset by this much "to the left" of the path as we travel along the path. This raising is done after and in addition to any transformations set using the transform option (see below).

A negative $\langle dimension \rangle$ will offset the decoration "to the right" of the to-be-decorated path.

/pgf/decoration/mirror=\langle boolean \rangle

(no default)

Causes the segments of the decoration to be mirrored along the to-be-decorated path. This is done after and in addition to any transformations set using the transform and/or raise options.

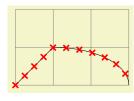


$/pgf/decoration/transform=\langle transformations \rangle$

(no default)

This key allows you to specify general $\langle transformations \rangle$ to be applied to the segments of a decoration. These transformations are applied before and independently of raise and mirror transformations. The $\langle transformations \rangle$ should be normal TikZ transformations like shift or rotate.

In the following example the **shift only** transformation is used to make sure that the crosses are *not* sloped along the path.



22.4.2 Starting and Ending Decorations Early or Late

You sometimes may wish to "end" a decoration a bit early on the path. For instance, you might wish a snake decoration to stop 5mm before the end of the path and to continue in a straight line. There are different ways of achieving this effect, but the easiest may be the pre and post options, which only have an effect in TikZ. Note, however, that they can only be used with decorations, not with meta-decorations.

```
/pgf/decoration/pre=\langle decoration \rangle
```

(no default, initially lineto)

This key sets a decoration that should be used before the main decoration starts. The $\langle decoration \rangle$ will be used for a length of pre length, which Opt by default. Thus, for the pre option to have any effect, you also need to set the pre length option.



```
\begin{tikzpicture}
\tikz [decoration={zigzag,pre=lineto,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
\end{tikzpicture}
```



```
\begin{tikzpicture}
\tikz [decoration={zigzag,pre=moveto,pre length=1cm}]
\draw [decorate] (0,0) -- (2,1) arc (90:0:1);
\end{tikzpicture}
```

```
\tikzpicture}
\tikz [decoration={zigzag,pre=crosses,pre length=1cm}]
\draw [decorate] (0,0) -- (2,1) arc (90:0:1);
\tikzpicture}
```

Note that the default pre option is lineto, not curveto. This means that the default pre decoration will not follow curves (for efficiency reasons). Change the pre key to curveto if you have a curved path.

```
\begin{tikzpicture}
\tikz [decoration={zigzag,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
\end{tikzpicture}

\begin{tikzpicture}
\tikz [decoration={zigzag,pre=curveto,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
\end{tikzpicture}
```

/pgf/decoration/pre length= $\langle dimension \rangle$

(no default, initially Opt)

This key sets the distance along which the pre-decoration should be used. If you do not need/wish a pre-decoration, set this key to Opt (exactly this string, not just to something that evaluates to the same things such as Ocm).

```
/pgf/decorations/post=\langle decoration \rangle
```

(no default, initially lineto)

Works like pre, only for the end of the decoration.

```
/pgf/decorations/post length=\langle dimension \rangle
```

(no default, initially Opt)

Works like pre length, only for the end of the decoration.

Here is a typical example that shows how these keys can be used:

23 Transformations

PGF has a powerful transformation mechanism that is similar to the transformation capabilities of METAFONT. The present section explains how you can access it in TikZ.

23.1 The Different Coordinate Systems

It is a long process from a coordinate like, say, (1, 2) or (1cm, 5pt), to the position a point is finally placed on the display or paper. In order to find out where the point should go, it is constantly "transformed," which means that it is mostly shifted around and possibly rotated, slanted, scaled, and otherwise mutilated.

In detail, (at least) the following transformations are applied to a coordinate like (1,2) before a point on the screen is chosen:

- 1. PGF interprets a coordinate like (1,2) in its xy-coordinate system as "add the current x-vector once and the current y-vector twice to obtain the new point."
- 2. PGF applies its coordinate transformation matrix to the resulting coordinate. This yields the final position of the point inside the picture.
- 3. The backend driver (like dvips or pdftex) adds transformation commands such that the coordinate is shifted to the correct position in T_FX's page coordinate system.
- 4. PDF (or PostScript) apply the canvas transformation matrix to the point, which can once more change the position on the page.
- 5. The viewer application or the printer applies the device transformation matrix to transform the coordinate to its final pixel coordinate on the screen or paper.

In reality, the process is even more involved, but the above should give the idea: A point is constantly transformed by changes of the coordinate system.

In TikZ, you only have access to the first two coordinate systems: The xy-coordinate system and the coordinate transformation matrix (these will be explained later). PGF also allows you to change the canvas transformation matrix, but you have to use commands of the core layer directly to do so and you "better know what you are doing" when you do this. The moment you start modifying the canvas matrix, PGF immediately loses track of all coordinates and shapes, anchors, and bounding box computations will no longer work.

23.2 The XY- and XYZ-Coordinate Systems

The first and easiest coordinate systems are PGF's xy- and xyz-coordinate systems. The idea is very simple: Whenever you specify a coordinate like (2,3) this means $2v_x + 3v_y$, where v_x is the current x-vector and v_y is the current y-vector. Similarly, the coordinate (1,2,3) means $v_x + 2v_y + 3v_z$.

Unlike other packages, PGF does not insist that v_x actually has a y-component of 0, that is, that it is a horizontal vector. Instead, the x-vector can point anywhere you want. Naturally, normally you will want the x-vector to point horizontally.

One undesirable effect of this flexibility is that it is not possible to provide mixed coordinates as in (1,2pt). Life is hard.

To change the x-, y-, and z-vectors, you can use the following options:

```
/tikz/x=(value) (no default, initially 1cm)
```

If $\langle value \rangle$ is a dimension, the x-vector of PGF's xyz-coordinate system is set up to point $\langle value \rangle$ to the right, that is, to $(\langle value \rangle, 0pt)$.

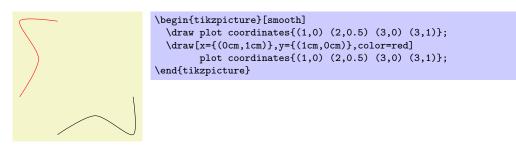
```
\text{\draw} \text
```

The last example shows that the size of steppings in grids, just like all other dimensions, are not affected by the x-vector. After all, the x-vector is only used to determine the coordinate of the upper right corner of the grid.

If $\langle value \rangle$ is a coordinate, the x-vector of PGF's xyz-coordinate system to the specified coordinate. If $\langle value \rangle$ contains a comma, it must be put in braces.

```
\begin{tikzpicture}
\draw (0,0) -- (1,0);
\draw[x={(2cm,0.5cm)},color=red] (0,0) -- (1,0);
\end{tikzpicture}
```

You can use this, for example, to exchange the meaning of the x- and y-coordinate.



```
/tikz/y=\(value\) (no default, initially 1cm)
```

Works like the x= option, only if $\langle value \rangle$ is a dimension, the resulting vector points to $(0, \langle value \rangle)$.

```
\text{/tikz/z} = \langle value \rangle (no default, initially -3.85 \text{mm})
```

Works like the y= option, but now a dimension is the point $(\langle value \rangle, \langle value \rangle)$.

```
\text{\left\( \) \text{
```

23.3 Coordinate Transformations

PGF and TikZ allow you to specify *coordinate transformations*. Whenever you specify a coordinate as in (1,0) or (1cm,1pt) or (30:2cm), this coordinate is first "reduced" to a position of the form "x points to the right and y points upwards." For example, (1in,5pt) is reduced to " $72\frac{72}{100}$ points to the right and 5 points upwards" and (90:100pt) means "0pt to the right and 100 points upwards."

The next step is to apply the current coordinate transformation matrix to the coordinate. For example, the coordinate transformation matrix might currently be set such that it adds a certain constant to the x value. Also, it might be set up such that it, say, exchanges the x and y value. In general, any "standard" transformation like translation, rotation, slanting, or scaling or any combination thereof is possible. (Internally, PGF keeps track of a coordinate transformation matrix very much like the concatenation matrix used by PDF or PostScript.)

The most important aspect of the coordinate transformation matrix is that it applies to coordinates only! In particular, the coordinate transformation has no effect on things like the line width or the dash pattern or the shading angle. In certain cases, it is not immediately clear whether the coordinate transformation matrix should apply to a certain dimension. For example, should the coordinate transformation matrix apply to

grids? (It does.) And what about the size of arced corners? (It does not.) The general rule is "If there is no 'coordinate' involved, even 'indirectly,' the matrix is not applied." However, sometimes, you simply have to try or look it up in the documentation whether the matrix will be applied.

Setting the matrix cannot be done directly. Rather, all you can do is to "add" another transformation to the current matrix. However, all transformations are local to the current TEX-group. All transformations are added using graphic options, which are described below.

Transformations apply immediately when they are encountered "in the middle of a path" and they apply only to the coordinates on the path following the transformation option.

```
\tikz \draw (0,0) rectangle (1,0.5) [xshift=2cm] (0,0) rectangle (1,0.5);
```

A final word of warning: You should refrain from using "aggressive" transformations like a scaling of a factor of 10000. The reason is that all transformations are done using T_EX, which has a fairly low accuracy. Furthermore, in certain situations it is necessary that TikZ *inverts* the current transformation matrix and this will fail if the transformation matrix is badly conditioned or even singular (if you do not know what singular matrices are, you are blessed).

$\texttt{/tikz/shift} = \{\langle coordinate \rangle\}$

(no default)

Adds the $\langle coordinate \rangle$ to all coordinates.



/tikz/shift only

(no value)

This option does not take any parameter. Its effect is to cancel all current transformations except for the shifting. This means that the origin will remain where it is, but any rotation around the origin or scaling relative to the origin or skewing will no longer have an effect.

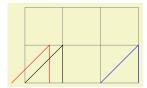
This option is useful in situations where a complicated transformation is used to "get to a position," but you then wish to draw something "normal" at this position.



/tikz/xshift=\langle dimension\rangle

(no default)

Adds $\langle dimension \rangle$ to the x value of all coordinates.



$\texttt{/tikz/yshift=} \langle dimension \rangle$

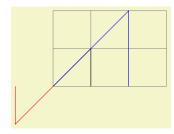
(no default)

Adds $\langle dimension \rangle$ to the y value of all coordinates.

$\texttt{/tikz/scale} = \langle factor \rangle$

(no default)

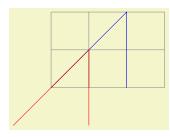
Multiplies all coordinates by the given $\langle factor \rangle$. The $\langle factor \rangle$ should not be excessively large in absolute terms or very close to zero.



$\texttt{/tikz/scale around=}\{\langle factor \rangle : \langle coordinate \rangle\}$

(no default)

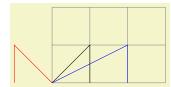
Scales the coordinate system by $\langle factor \rangle$, with the "origin of scaling" centered on $\langle coordinate \rangle$ rather than the origin.



$/tikz/xscale = \langle factor \rangle$

(no default)

Multiplies only the x-value of all coordinates by the given $\langle factor \rangle$.



$\verb|/tikz/yscale=| \langle factor \rangle|$

(no default)

Multiplies only the y-value of all coordinates by $\langle factor \rangle$.

$/tikz/xslant = \langle factor \rangle$

(no default)

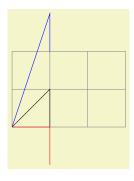
Slants the coordinate horizontally by the given $\langle factor \rangle$:



/tikz/yslant= $\langle factor \rangle$

(no default)

Slants the coordinate vertically by the given $\langle factor \rangle$:



```
/tikz/rotate=\(degree\) (no default)
```

Rotates the coordinate system by $\langle degree \rangle$:

```
\tikz/rotate around=\{\langle degree \rangle: \langle coordinate \rangle\}
```

(no default)

Rotates the coordinate system by $\langle degree \rangle$ around the point $\langle coordinate \rangle$.

```
/\text{tikz/cm}=\{\langle a \rangle, \langle b \rangle, \langle c \rangle, \langle d \rangle, \langle coordinate \rangle\}
```

(no default)

applies the following transformation to all coordinates: Let (x, y) be the coordinate to be transformed and let $\langle coordinate \rangle$ specify the point (t_x, t_y) . Then the new coordinate is given by $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$. Usually, you do not use this option directly.



/tikz/reset cm (no value)

Completely resets the coordinate transformation matrix to the identity matrix. This will destroy not only the transformations applied in the current scope, but also all transformations inherited from surrounding scopes. Do not use this option, unless you really, really know what you are doing.

23.4 Canvas Transformations

A canvas transformation, see Section 76.4 for details, is best thought of as a transformation in which the drawing canvas is stretched or rotated. Imaging writing something on a balloon (the canvas) and then blowing air into the balloon: Not only does the text become larger, the thin lines also become larger. In particular, if you scale the canvas by a factor of two, all lines are twice as thick.

Canvas transformations should be used with great care. In most circumstances you do *not* want line widths to change in a picture as this creates visual inconsistency.

Just as important, when you use canvas transformations PGF loses track of positions of nodes and of picture sizes since it does not take the effect of canvas transformations into account when it computes coordinates of nodes (do not, however, rely on this; it may change in the future).

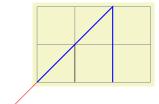
Finally, note that a canvas transformation always applies to a path as a whole, it is not possible (as for coordinate transformations) to use different transformations in different parts of a path.

In short, you should not use canvas transformations unless you really know what you are doing.

```
/tikz/transform canvas=\langle options\rangle
```

(no default)

The $\langle options \rangle$ should contain coordinate transformations options like scale or xshift. Multiple options can be given, their effects accumulate in the usual manner. The effect of these $\langle options \rangle$ (immediately) changes the current canvas transformation matrix. The coordinate transformation matrix is not changed. Tracking of the picture size is (locally) switched off and the node coordinate will no longer be correct.



Part IV

Graph Drawing

by Till Tantau et al.

Graph drawing algorithms do the tough work of computing a layout of a graph for you. TikZ comes with powerful such algorithms, but you can also implement new algorithms in the Lua programming language.

You need to use LuaT_EX to typeset this part of the manual (and, also, to use algorithmic graph drawing).

Part V

Libraries

by Till Tantau

In this part the library packages are documented. They provide additional predefined graphic objects like new arrow heads or new plot marks, but sometimes also extensions of the basic PGF or TikZ system. The libraries are not loaded by default since many users will not need them.









```
\tikzset{
 ld/.style={level distance=#1},lw/.style={line width=#1},
  level 1/.style={ld=4.5mm, trunk,
                                              lw=1ex ,sibling angle=60},
  level 2/.style={ld=3.5mm, trunk!80!leaf a,lw=.8ex,sibling angle=56},
 level 3/.style={ld=2.75mm,trunk!60!leaf a,lw=.6ex,sibling angle=52},
 level 4/.style={ld=2mm, trunk!40!leaf a,lw=.4ex,sibling angle=48},
level 5/.style={ld=1mm, trunk!20!leaf a,lw=.3ex,sibling angle=44},
 level 6/.style={ld=1.75mm,leaf a,
                                              lw=.2ex,sibling angle=40},
\pgfarrowsdeclare{leaf}{leaf}
  {\pgfarrowsleftextend{-2pt} \pgfarrowsrightextend{1pt}}
  \pgfpathmoveto{\pgfpoint{-2pt}{0pt}}
  \pgfpatharc{150}{30}{1.8pt}
  \pgfpatharc{-30}{-150}{1.8pt}
  \pgfusepathqfill
\newcommand{\logo}[5]
  \colorlet{border}{#1}
  \colorlet{trunk}{#2}
  \colorlet{leaf a}{#3}
  \colorlet{leaf b}{#4}
  \begin{tikzpicture}
    \scriptsize\scshape
    \draw[border,line width=1ex,yshift=.3cm,
          yscale=1.45,xscale=1.05,looseness=1.42]
      (1,0) to [out=90, in=0]
                                  (0,1) to [out=180,in=90] (-1,0)
            to [out=-90,in=-180] (0,-1) to [out=0, in=-90] (1,0) -- cycle;
    \coordinate (root) [grow cyclic,rotate=90]
    child {
      child [line cap=round] foreach \a in {0,1} {
        child foreach \b in {0,1} {
          child foreach \c in {0,1} {
            child foreach \d in {0,1} {
              child foreach \leafcolor in {leaf a,leaf b}
                { edge from parent [color=\leafcolor,-#5] }
        1 1 1
      } edge from parent [shorten >=-1pt,serif cm-,line cap=butt]
    \node [align=center,below] at (0pt,-.5ex)
    { \textcolor{border}{T}heoretical \\ \textcolor{border}{C}omputer \\
      \textcolor{border}{S}cience };
  \end{tikzpicture}
\begin{minipage}{3cm}
  \logo{green!80!black}{green!25!black}{green}{green!80}{leaf}\\
  \logo{green!50!black}{black}{green!80!black}{red!80!green}{leaf}\\
  \logo{red!75!black}{red!25!black}{red!75!black}{orange}{leaf}\\
  \label{logo} $$ \log_{black!50}{black!50}{black!25}{} $$
\end{minipage}
```

24 Arrow Tip Library

The package defines additional arrow tips, which are described below. Note that neither the standard packages nor this package defines an arrow name containing > or <. These are left for the user to define as he or she sees fit.

```
\usepgflibrary{arrows.spaced} % MEX and plain TeX and pure pgf \usepgflibrary[arrows.spaced] % ConTeXt and pure pgf \usetikzlibrary{arrows.spaced} % MEX and plain TeX when using TikZ \usetikzlibrary[arrows.spaced] % ConTeXt when using TikZ
```

This library contains all arrows starting with spaced. These are the same are the normal forms, only a bit of, well, space is added at the tip.

The arrow tips to, stealth, latex, space, their reversed forms, and | are predefined, but listed below for completeness, nevertheless.

24.1 Mathematical Arrow Tips

```
to yields thick \( \ldots \) and thin \( \ldots \)

to reversed yields thick \( \ldots \) and thin \( \ldots \)

implies yields thick \( \ldots \) and thin \( \ldots \)

spaced to yields thick \( \ldots \) and thin \( \ldots \)

spaced to reversed yields thick \( \ldots \) and thin \( \ldots \)

spaced implies yields thick \( \ldots \) and thin \( \ldots \)

spaced implies yields thick \( \ldots \) and thin \( \ldots \)
```

24.2 Triangular Arrow Tips

```
yields thick \longrightarrow and thin \longrightarrow , double \longrightarrow and \longrightarrow yields thick \longrightarrow and thin \longrightarrow , double \longrightarrow and \longrightarrow yields thick \longrightarrow and thin \longrightarrow
latex
latex reversed
latex'
                                                                                    yields thick \longleftarrow and thin \longleftarrow yields thick \longleftarrow and thin \longleftarrow , double \longleftarrow and \longleftarrow yields thick \longleftarrow and thin \longleftarrow , double \longleftarrow and \longleftarrow
latex' reversed
stealth
stealth reversed
                                                                                     yields thick |\longleftarrow| and thin |\longleftarrow|
stealth'
                                                                                     yields thick → and thin →
stealth' reversed
                                                                                     yields thick \longrightarrow and thin \longleftarrow
triangle 90
triangle 90 reversed
                                                                                     yields thick and thin
                                                                                     yields thick ◀─▶ and thin ◀─▶
triangle 60
triangle 60 reversed
                                                                                     yields thick and thin
triangle 45
                                                                                     yields thick | \longleftarrow | and thin | \longleftarrow |
                                                                                     yields thick 
ightharpoonup and thin 
ightharpoonup
triangle 45 reversed
                                                                                     yields thick \langle \longrightarrow \rangle and thin \langle \longleftarrow \rangle
open triangle 90
open triangle 90 reversed
                                                                                     yields thick \triangleright and thin \triangleright
                                                                                     yields thick ♦—> and thin ♦—>
open triangle 60
open triangle 60 reversed
                                                                                     yields thick A and thin
                                                                                     yields thick \triangleleft \longrightarrow  and thin \triangleleft \longrightarrow 
open triangle 45
                                                                                     yields thick | and thin | and thin |
open triangle 45 reversed
                                                                                  yields thick \| \longleftrightarrow \| and thin \| \longleftrightarrow \|, double \| \longleftrightarrow \| and \| \longleftrightarrow \|
spaced latex
                                                                                 yields thick |\hspace{-0.1cm}-\hspace{-0.1cm}-\hspace{-0.1cm}| and thin |\hspace{-0.1cm}-\hspace{-0.1cm}-\hspace{-0.1cm}-\hspace{-0.1cm}| and thin |\hspace{-0.1cm}-\hspace{-0.1cm}-\hspace{-0.1cm}-\hspace{-0.1cm}-\hspace{-0.1cm}|
spaced latex reversed
spaced latex'
                                                                                 yields thick and thin
spaced latex' reversed
                                                                                  yields thick |\longleftrightarrow| and thin |\longleftrightarrow|, double |\longleftrightarrow| and |\longleftrightarrow|
spaced stealth
                                                                                  yields thick | And thin | Adouble | And | 
spaced stealth reversed
                                                                                  yields thick ★ and thin ★
spaced stealth'
spaced stealth' reversed
                                                                                 yields thick ► and thin ►
```

```
spaced triangle 90
                                       yields thick and thin
                                       yields thick and thin
spaced triangle 90 reversed
spaced triangle 60
                                       yields thick | and thin |
spaced triangle 60 reversed
                                       yields thick and thin
spaced triangle 45
                                       yields thick \blacksquare \longrightarrow \blacksquare and thin \blacksquare \longrightarrow \blacksquare
spaced triangle 45 reversed
                                       yields thick and thin
                                       yields thick and thin yields thick and thin
spaced open triangle 90
spaced open triangle 90 reversed
                                       spaced open triangle 60
spaced open triangle 60 reversed
                                       yields thick \triangleright and thin \triangleright
spaced open triangle 45
                                       yields thick \triangleleft \longrightarrow and thin \triangleleft \longrightarrow
spaced open triangle 45 reversed
                                       yields thick | and thin | and thin |
```

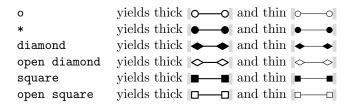
24.3 Barbed Arrow Tips

```
angle 90
                         yields thick \longleftrightarrow and thin \longleftrightarrow
                         yields thick ► and thin ►
angle 90 reversed
                         yields thick \longleftrightarrow and thin \longleftrightarrow
angle 60
                         yields thick \rightarrow and thin \rightarrow
angle 60 reversed
angle 45
                         yields thick |\langle \rangle| and thin |\langle \rangle|
                         yields thick > and thin > yields thick (A) and thin (A)
angle 45 reversed
hooks
                         hooks reversed
spaced angle 90
                                  yields thick \longleftrightarrow and thin \longleftrightarrow
                                  yields thick \( \) and thin \( \)
spaced angle 90 reversed
                                  yields thick \longleftrightarrow and thin \longleftrightarrow
spaced angle 60
                                  yields thick | > and thin | > |
spaced angle 60 reversed
                                  yields thick \parallel \longleftrightarrow \parallel and thin \parallel \longleftrightarrow \parallel
spaced angle 45
                                  yields thick \rightarrow and thin \rightarrow
spaced angle 45 reversed
                                  yields thick \longleftrightarrow and thin \longleftrightarrow
spaced hooks
                                  spaced hooks reversed
```

24.4 Bracket-Like Arrow Tips

```
[-] yields thick [—] and thin [—]
]-[ yields thick ]—[ and thin ]—[
(-) yields thick (—) and thin (—)
)-( yields thick )—( and thin )—(
|-| yields thick |—] and thin |—]
spaced [-spaced ] yields thick [—] and thin [—]
spaced ]-spaced [ yields thick ]—[ and thin |—]
spaced (-spaced ) yields thick |—] and thin (—)
spaced )-spaced ( yields thick )—( and thin )—(
spaced |-spaced | yields thick |—] and thin |—]
```

24.5 Circle, Diamond and Square Arrow Tips



spaced o	yields thick \circ — \circ and thin \circ — \circ
spaced *	yields thick $\blacksquare lacktriangle$ and thin $\blacksquare lacktriangle$
spaced diamond	yields thick $\blacksquare \spadesuit \blacksquare$ and thin $\blacksquare \spadesuit \blacksquare$
spaced open diamond	yields thick $ \Leftrightarrow $ and thin $ \Leftrightarrow $
spaced square	yields thick $\blacksquare \blacksquare \blacksquare \blacksquare$ and thin $\blacksquare \blacksquare \blacksquare \blacksquare$
spaced open square	yields thick \square and thin \square

24.6 Serif-Like Arrow Tips

24.7 Partial Arrow Tips

left to	yields thick \longrightarrow and thin \longrightarrow
left to reversed	yields thick and thin
right to	yields thick \leftarrow and thin \leftarrow
right to reversed	yields thick and thin
left hook	yields thick and thin
left hook reversed	yields thick and thin
right hook	yields thick and thin
right hook reversed	yields thick and thin
spaced left to	yields thick $\!$
spaced left to reversed	yields thick \parallel and thin \parallel
spaced right to	yields thick $\parallel \longleftarrow \parallel$ and thin $\parallel \longleftarrow \parallel$
spaced right to reversed	yields thick and thin
spaced left hook	yields thick and thin
spaced left hook reversed	yields thick and thin
spaced right hook	yields thick and thin
spaced right hook reversed	yields thick and thin

24.8 Line Caps

round cap	yields for line width 1ex
butt cap	yields for line width 1ex
triangle 90 cap	yields for line width 1ex
triangle 90 cap reversed	yields for line width 1ex
fast cap	yields for line width 1ex
fast cap reversed	yields for line width 1ex
spaced round cap	yields for line width 1ex
spaced butt cap	yields for line width 1ex
spaced triangle 90 cap	yields for line width 1ex
spaced triangle 90 cap reversed	yields for line width 1ex
spaced fast cap	yields for line width $1 ex$
spaced fast cap reversed	yields for line width 1ex >

24.9 Spacing Tips

The spacing arrow tips are useful for combining them with other arrows to get arrows that do not touch the end of the line.

space yields thick and thin

25 Automata Drawing Library

```
\usetikzlibrary{automata} % MEX and plain TeX \usetikzlibrary[automata] % ConTeXt
```

This packages provides shapes and styles for drawing finite state automata and Turing machines.

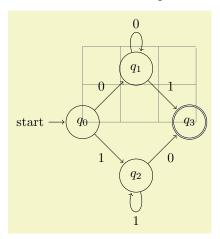
25.1 Drawing Automata

The automata drawing library is intended to make it easy to draw finite automata and Turing machines. It does not cover every situation imaginable, but most finite automata and Turing machines found in text books can be drawn in a nice and convenient fashion using this library.

To draw an automaton, proceed as follows:

- 1. For each state of the automaton, there should be one node with the option state.
- 2. To place the states, you can either use absolute positions or relative positions, using options like above or right.
- 3. Give a unique name to each state node.
- 4. Accepting and initial states are indicated by adding the options accepting and initial, respectively, to the state nodes.
- 5. Once the states are fixed, the edges can be added. For this, the edge operation is most useful. It is, however, also possible to add edges after each node has been placed.
- 6. For loops, use the edge [loop] operation.

Let us now see how this works for a real example. Let us consider a nondeterministic four state automaton that checks whether an input contains the sequence 0*1 or the sequence 1*0.



```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid,auto]
 \draw[help lines] (0,0) grid (3,2);
 \node[state,initial] (q_0)
                                                    {$q_0$};
 \node[state]
                        (q_1) [above right=of q_0] {$q_1$};
 \node[state]
                        (q_2) [below right=of q_0] {$q_2$};
 \node[state,accepting](q_3) [below right=of q_1] {$q_3$};
 \path[->] (q_0) edge
                                                 {0} (q_1)
                                    node
                                    node [swap] \{1\} (q_2)
                  edge
            (q_1) edge
                                    node
                                                 {1} (q_3)
                  edge [loop above] node
                                                 (0)
                                    node [swap] {0} (q_3)
            (q_2) edge
                  edge [loop below] node
                                                 {1} ();
\end{tikzpicture}
```

25.2 States With and Without Output

The state style actually just "selects" a default underlying style. Thus, you can define multiple new complicated state style and then simply set the state style to your given style to get the desired kind of styles.

By default, the following state styles are defined:

/tikz/state without output

(style, no value)

This node style causes nodes to be drawn as circles. Also, this style calls every state.

/tikz/state with output

(style, no value)

This node style causes nodes to be drawn as split circles, that is, using the circle split shape. In the upper part of the shape you have the name of the style, in the lower part the output is placed. To specify the output, use the command \nodepart{lower} inside the node. This style also calls every state.

```
q_0 q_1 q_1 q_2 q_3 q_4 q_5 q_6 q_6
```

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\node[state without output] {$q_0$};

\node[state with output] at (2,0) {$q_1$ \nodepart{lower} $00$};
\end{tikzpicture}
```

/tikz/state

(style, initially state without output)

You should redefine it to something else, if you wish to use states of a different nature.

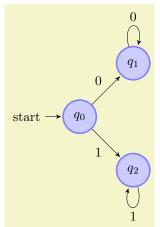
```
\begin{array}{c}
q_0 \\
11
\end{array}

\begin{array}{c}
q_1 \\
00
\end{array}
```

/tikz/every state

(style, initially empty)

This style is used by state with output and also by state without output. By default, it does nothing, but you can use it to make your state look more fancy:



```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid,>=stealth',
    every state/.style={draw=blue!50, very thick, fill=blue!20}]
  \node[state,initial] (q_0)
                                                    {$q_0$};
  \node[state]
                         (q_1) [above right=of q_0] {$q_1$};
                         (q_2) [below right=of q_0] {$q_2$};
  \node[state]
                                                        {0} (q_1)
  \path[->] (q_0) edge
                                     node [above left]
                                                        {1} (q_2)
                  edge
                                     node [below left]
            (q_1) edge [loop above] node
                                                         {0} ()
            (q_2) edge [loop below] node
                                                        {1} ();
\end{tikzpicture}
```

25.3 Initial and Accepting States

The styles initial and accepting are similar to the state style as they also just select an "underlying" style, which installs the actual settings for initial and accepting states.

Let us start with the initial states.

/tikz/initial

(style, initially initial by arrow)

This style is used to draw initial states.

/tikz/initial by arrow

(style, no value)

This style causes an arrow and, possibly, some text to be added to the node. The arrow points from the text to the node. The node text and the direction and the distance can be set using the following key:

/tikz/initial text= $\langle text \rangle$

(no default, initially start)

This key sets the text to be used. Use an empty text to suppress all text.

/tikz/initial where= $\langle direction \rangle$

(no default, initially left)

Set the place where the text should be shown. Allowed values are above, below, left, and right.

/tikz/initial distance= $\langle distance \rangle$

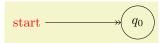
(no default, initially 3ex)

Sets the length of the arrow leading from the text to the state node.

/tikz/every initial by arrow

(style, initially empty)

This style is executed at the beginning of every path that contains the arrow and the text. You can use it to, say, make the text red or whatever.



\begin{tikzpicture}[every initial by arrow/.style={text=red,->>}]
\node[state,initial,initial distance=2cm] {\$q_0\$};
\end{tikzpicture}

/tikz/initial above

(style, no value)

This is a shorthand for initial by arrow, initial where=above.

/tikz/initial below

(style, no value)

Works similarly to the previous option.

/tikz/initial left

(style, no value)

Works similarly to the previous option.

/tikz/initial right

(style, no value)

Works similarly to the previous option.

/tikz/initial by diamond

(style, no value)

This style uses a diamond to indicate an initial node.

For the accepting states, the situation is similar: There is also an accepting style that selects the way accepting states are rendered. There are now two options: First, accepting by arrow, which works the same way as initial by arrow, only with the direction of arrow reversed, and accepting by double, where accepting states get a double line around them.

/tikz/accepting

(style, initially accepting by double)

This style is used to draw accepting states. You can replace this by the style accepting by arrow to get accepting states with an arrow leaving them.

/tikz/accepting by double

(style, no value)

This style causes a double line to be drawn around a state.

/tikz/accepting by arrow

(style, no value)

This style causes an arrow and, possibly, some text to be added to the node. The arrow points to the text from the node.

The same options as for initial states can be used, only with initial replaced by accepting:

/tikz/accepting text= $\langle text \rangle$

(no default, initially empty)

This key sets the text to be used.

/tikz/accepting where= $\langle direction \rangle$

(no default, initially right)

Set the place where the text should be shown. Allowed values are above, below, left, and right.

/tikz/intial distance= $\langle distance \rangle$

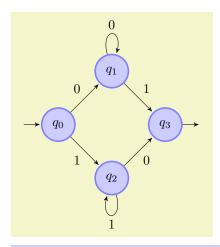
(no default, initially 3ex)

Sets the length of the arrow leading from the text to the state node.

/tikz/every accepting by arrow

(style, initially empty)

Executed at the beginning of every path that contains the arrow and the text.



```
\begin{tikzpicture}
  [shorten >=1pt,node distance=2cm,on grid,>=stealth',initial text=,
   every state/.style={draw=blue!50,very thick,fill=blue!20},
   accepting/.style=accepting by arrow]
  \node[state,initial] (q_0)
                        (q_1) [above right=of q_0] {$q_1$};
  \node[state]
  \node[state]
                        (q_2) [below right=of q_0] {$q_2$};
  \node[state,accepting](q_3) [below right=of q_1] {$q_3$};
  \phi[->] (q_0) edge
                                    node [above left] \{0\} (q_1)
                  edge
                                    node [below left]
                                                        {1} (q_2)
            (q_1) edge
                                    node [above right] {1} (q_3)
                  edge [loop above] node
                                                        {0} ()
            (q_2) edge
                                    node [below right] {0} (q_3)
                  edge [loop below] node
                                                        {1} ();
\end{tikzpicture}
```

/tikz/accepting above

(style, no value)

This is a shorthand for accepting by arrow, accepting where=above.

/tikz/accepting below

(style, no value)

Works similarly to the previous option.

/tikz/accepting left

(style, no value)

Works similarly to the previous option.

/tikz/accepting right

(style, no value)

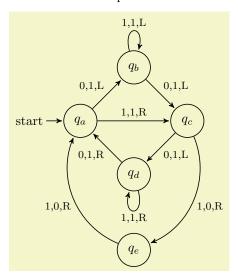
Works similarly to the previous option.

25.4 Examples

In the following example, we once more typeset the automaton presented in the previous sections. This time, we use the following rule for accepting/initial state: Initial states are red, accepting states are green, and normal states are orange. Then, we must find a path from a red state to a green state.

```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid,>=stealth',thick,
    every state/.style={fill,draw=none,orange,text=white,circular drop shadow},
    accepting/.style ={green!50!black,text=white},
    initial/.style
                    ={red,text=white}]
  \node[state,initial] (q_0)
                                                     {$q_0$};
                         (q_1) [above right=of q_0] {$q_1$};
  \node[state]
  \node[state]
                         (q_2) [below right=of q_0] {$q_2$};
  \label{localization} $$ \c [state, accepting](q_3) [below right=of q_1] {$q_3$};
  \phi = (q_0) edge
                                     node [above left] \{0\} (q_1)
                                     node [below left] {1} (q_2)
                  edge
            (q_1) edge
                                     node [above right] {1} (q_3)
                  edge [loop above] node
                                                         () {0}
                                    node [below right] {0} (q_3)
            (q_2) edge
                  edge [loop below] node
                                                         {1} ();
\end{tikzpicture}
```

The next example is the current candidate for the five-state busiest beaver:



```
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,%
                    auto, node distance=2cm, on grid, semithick,
                    inner sep=2pt,bend angle=45]
  \node[initial,state] (A)
                                               {$q_a$};
  \node[state]
                       (B) [above right=of A] {$q_b$};
  \node[state]
                       (D) [below right=of A] {$q_d$};
  \node[state]
                       (C) [below right=of B] {$q_c$};
  \node[state]
                       (E) [below=of D]
                                               {$q_e$};
  \path [every node/.style={font=\footnotesize}]
                              node {0,1,L} (B)
        (A) edge
                              node {1,1,R} (C)
            edge
        (B) edge [loop above] node {1,1,L} (B)
                              node {0,1,L} (C)
            edge
        (C) edge
                              node {0,1,L} (D)
            edge [bend left] node {1,0,R} (E)
        (D) edge [loop below] node {1,1,R} (D)
                              node {0,1,R} (A)
            edge
        (E) edge [bend left] node {1,0,R} (A);
\end{tikzpicture}
```

26 Background Library

```
\usetikzlibrary{backgrounds} % MEX and plain TEX \usetikzlibrary[backgrounds] % ConTEXt
```

This library defines "backgrounds" for pictures. This does not refer to background pictures, but rather to frames drawn around and behind pictures. For example, this package allows you to just add the framed option to a picture to get a rectangular box around your picture or gridded to put a grid behind your picture.

The first use of this library is to make the following key available:

/tikz/on background layer

(no value)

This key can be used with a {scope}. It will cause everything inside the scope to be typeset on a background layer. Note that the scope should not be "deeply nested" inside the picture since changes to the graphic state (like the color or the transformation matrix) "do not survive a layer switch." For more details on layers see Section 90.



```
\begin{tikzpicture}
  % On main layer:
  \fill[blue] (0,0) circle (1cm);

\begin{scope} [on background layer]
    \fill[yellow] (-1,-1) rectangle (1,1);
\end{scope}

\begin{scope} [on background layer]
    \fill[black] (-.8,-.8) rectangle (.8,.8);
\end{scope}

% On main layer again:
  \fill[blue!50] (-.5,-1) rectangle (.5,1);
\end{tikzpicture}
```

When this package is loaded, the following styles become available:

/tikz/show background rectangle

(style, no value)

This style causes a rectangle to be drawn behind your graphic. This style option must be given to the {tikzpicture} environment or to the \tikz command.



```
\begin{tikzpicture}[show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

The size of the background rectangle is determined as follows: We start with the bounding box of the picture. Then, a certain separator distance is added on the sides. This distance can be different for the x- and y-directions and can be set using the following options:

```
/tikz/inner frame xsep=\langle dimension \rangle
```

(no default, initially 1ex)

Sets the additional horizontal separator distance for the background rectangle.

```
/tikz/inner frame ysep=\langle dimension \rangle
```

(no default, initially 1ex)

Same for the vertical separator distance.

```
/tikz/inner frame sep=\langle dimension \rangle
```

(no default)

Sets the horizontal and vertical separator distances simultaneously.

The following two styles make setting the inner separator a bit easier to remember:

/tikz/tight background

(style, no value)

Sets the inner frame separator to 0pt. The background rectangle will have the size of the bounding box.

/tikz/loose background

(style, no value)

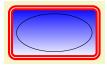
Sets the inner frame separator to 2ex.

You can influence how the background rectangle is rendered by setting the following style:

/tikz/background rectangle

(style, initially draw)

This style dictates how the background rectangle is drawn or filled. The default setting causes the path of the background rectangle to be drawn in the usual way. Setting this style to, say, fill=blue!20 causes a light blue background to be added to the picture. You can also use more fancy settings as shown in the following example:



```
\begin{tikzpicture}
  [background rectangle/.style=
      {double,ultra thick,draw=red,top color=blue,rounded corners},
      show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
  \end{tikzpicture}
```

Naturally, no one in their right mind would use the above, but here is a nice background:



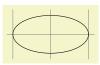
/tikz/framed (style, no value)

This is a shorthand for show background rectangle.

/tikz/show background grid

(style, no value)

This style behaves similarly to the **show background rectangle** style, but it will not use a rectangle path, but a grid. The lower left and upper right corner of the grid is computed in the same way as for the background rectangle:



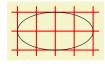
```
\begin{tikzpicture}[show background grid]
 \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

You can influence the background grid by setting the following style:

/tikz/background grid

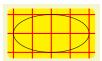
(style, initially draw, help lines)

This style dictates how the background grid path is drawn.



```
\begin{tikzpicture}
  [background grid/.style={thick,draw=red,step=.5cm},
    show background grid]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

This option can be combined with the framed option (use the framed option first):



/tikz/gridded

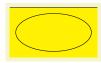
(style, no value)

This is a shorthand for show background grid.

/tikz/show background top

(style, no value)

This style causes a single line to be drawn at the top of the background rectangle. Normally, the line coincides exactly with the top line of the background rectangle:



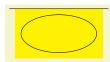
```
\begin{tikzpicture}[
    background rectangle/.style={fill=yellow},
    framed,show background top]
    \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

The following option allows you to lengthen (or shorten) the line:

/tikz/outer frame xsep=⟨dimension⟩

(no default, initially Opt)

The $\langle dimension \rangle$ is added at the left and right side of the line.



```
\begin{tikzpicture}
  [background rectangle/.style={fill=yellow},
  framed,
  show background top,
  outer frame xsep=1ex]
  \draw (0,0) ellipse (10mm and 5mm);
  \end{tikzpicture}
```

/tikz/outer frame ysep= $\langle dimension \rangle$

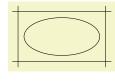
(no default, initially Opt)

This option does not apply to the top line, but to the left and right lines, see below.

/tikz/outer frame $sep=\langle dimension \rangle$

(no default)

Sets both the x- and y-separation.

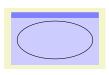


```
\begin{tikzpicture}
  [background rectangle={fill=blue!20},
  outer frame sep=1ex,%
  show background top,%
  show background bottom,%
  show background left,%
  show background right]
  \draw (0,0) ellipse (10mm and 5mm);
  \end{tikzpicture}
```

You can influence how the line is drawn grid by setting the following style:

/tikz/background top

(style, initially draw)



/tikz/show background bottom

(style, no value)

Works like the style for the top line.

/tikz/show background left

(style, no value)

Works similarly.

/tikz/show background right

(style, no value)

Works similarly.

27 Calc Library

```
\label{lineary} $$ \ary{calc} % $$ MEX and plain $$ T_EX \ary[calc] % $$ ConT_EXt$
```

The library allows advanced Coordinate Calculations. It is documented in all detail in Section 13.5 on page 119.

28 Calendar Library

```
\usetikzlibrary{calendar} % MEX and plain TeX \usetikzlibrary[calendar] % ConTeXt
```

The library defines the \calendar command, which can be used to typeset calendars. The command relies on the \pgfcalendar command from the pgfcalendar package, which is loaded automatically.

The \calendar command is quite configurable, allowing you to produce all kinds of different calendars.

28.1 Calendar Command

The core command for creating calendars in TikZ is the \calendar command. It is available only inside {tikzpicture} environments (similar to, say, the \draw command).

The syntax for this command is similar to commands like \node or \mbox{matrix} . However, it has its complete own parser and only those commands described in the following will be recognized, nothing else. Note, furthermore, that a $\langle calendar\ specification \rangle$ is not a path specification, indeed, no path is created for the calendar.

The specification syntax. The $\langle calendar \ specification \rangle$ must be a sequence of elements, each of which has one of the following structures:

• $[\langle options \rangle]$

You provide $\langle options \rangle$ in square brackets as in [red,draw=none]. These $\langle options \rangle$ can be any TikZ option and they apply to the whole calendar. You can provide this element multiple times, the effect accumulates.

(⟨name⟩)

This has the same effect as saying [name= $\langle name \rangle$]. The effect of providing a $\langle name \rangle$ is explained later. Note already that a calendar is not a node and the $\langle name \rangle$ is not the name of a node.

• at (\(\langle coordinate \rangle \))

This has the same effect as saying $[at=(\langle coordinate \rangle)]$.

• if $(\langle date\ condition \rangle)$ $\langle options\ or\ commands \rangle$ else $\langle else\ options\ or\ commands \rangle$ The effect of such an if is explained later.

At the beginning of every calendar, the following style is used:

/tikz/every calendar

(style, initially empty)

This style is used with every calendar.

The date range. The overall effect of the \calendar command is to execute code for each day of a range of dates. This range of dates is set using the following option:

```
\texttt{/tikz/dates} = \langle start \ date \rangle to \langle end \ date \rangle
```

(no default)

This option specifies the date range. Both the start and end date are specified as described on page 662. In short: You can provide ISO-format type dates like 2006-01-02, you can replace the day of month by last to refer to the last day of a month (so 2006-02-last is the same as 2006-02-28), and you can add a plus sign followed by a number to specify an offset (so 2006-01-01+-1 is the same as 2005-12-31).

It will be useful to fix two pieces of terminology for the following descriptions: The \calendar command iterates over the dates in the range. The current date refers to the current date the command is processing as it iterates over the dates. For each current date code is executed, which will be called the current date code. The current date code consists of different parts, to be detailed later.

The central part of the current date code is the execution of the code \tikzdaycode. By default, this code simply produces a node whose text is set to the day of month. This means that unless further action is taken, all days of a calendar will be put on top of each other! To avoid this, you must modify the current date code to shift days around appropriately. Predefined arrangements like day list downward or week list do this for you, but you can define arrangements yourself. Since defining an arrangement

is a bit tricky, it is explained only later on. For the time being, let us use a predefined arrangement to produce our first calendar:

```
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

Changing the spacing. In the above calendar, the spacing between the days is determined by numerous options. Most arrangements do not use all of these options, but only those that apply naturally.

```
\ttikz/day xshift=\tdimension\t (no default, initially 3.5ex)
```

Specifies the horizontal shift between days. This is not the gap between days, but the shift between the anchors of their nodes.

```
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
/tikz/day yshift=\langle dimension \rangle
```

(no default, initially 3ex)

Specifies the vertical shift between days. Again, this is the shift between the anchors of their nodes.

```
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31 \text{\table tikz \calendar[dates=2000-01-01 to 2000-01-31, week list, day yshift=2ex];}
```

```
/\text{tikz/month xshift} = \langle dimension \rangle (no default)
```

Specifies an additional horizontal shift between different months.

```
/\text{tikz/month yshift} = \langle dimension \rangle (no default)
```

Specifies an additional vertical shift between different months.

```
\tikz \calendar[dates=2000-01-01 to 2000-02-last, week list,
                1
                                      month yshift=0pt];
3 4 5 6 7
                8
                  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
   1
      2
         3 4 5 6
   8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29
```

```
\tikz \calendar[dates=2000-01-01 to 2000-02-last, week list,
                                      month yshift=1cm];
3 4 5 6
            7
                8
                   9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
    1
      2 3 4 5
      9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29
```

Changing the position of the calendar. The calendar is placed in such a way that, normally, the anchor of the first day label is at the origin. This can be changed by using the at option. When you say $at=\{(1,1)\}$, this anchor of the first day will lie at coordinate (1,1).

In general, arrangements will not always place the anchor of the first day at the origin. Sometimes, additional spacing rules get in the way. There are different ways of addressing this problem: First, you can just ignore it. Since calendars are often placed in their own {tikzpicture} and since their size if computed automatically, the exact position of the origin often does not matter at all. Second, you can put the calendar inside a node as in ...node {\tikz \calendar...}. This allows you to position the node in the normal ways using the node's anchors. Third, you can be very clever and use a single-cell matrix. The advantage is that a matrix allows you to provide any anchor of any node inside the matrix as an anchor for the whole matrix. For example, the following calendar is placed in such a way the center of 2000-01-20 lies on the position (2,2):

```
\begin{tikzpicture}
                  1
                     2
                             \draw[help lines] (0,0) grid (3,2);
                             \matrix [anchor=cal-2000-01-20.center] at (2,2)
 3
   4
       5
          6
              7
                  8
                     9
                             { \calendar(cal)[dates=2000-01-01 to 2000-01-31, week list]; \\};
10 11 12 13 14 15 16
                           \end{tikzpicture}
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

Unfortunately, the matrix-base positions, which is the cleanest way, isn't as portable as the other approaches (it currently does not work with the SVG backend for instance).

Changing the appearance of days. As mentioned before, each day in the above calendar is produced by an execution of the \tikzdaycode. Each time this code is executed, the coordinate system will have been set up appropriately to place the day of the month correctly. You can change both the code and its appearance using the following options.

```
\mbox{\tt /tikz/day code=}\mbox{\tt code}\mbox{\tt (no default, initially see below)} This option allows you to change the code that is executed for each day. The default is to create a
```

node with an appropriate name, but you can change this:

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31, week list, day code={\fill[blue] (0,0) circle (2pt);}];
```

The default code is the following:

\node[name=\pgfcalendarsuggestedname,every day]{\tikzdaytext};

The first part causes the day nodes to be accessible via the following names: If $\langle name \rangle$ is the name given to the calendar via a name= option or via the specification element ($\langle name \rangle$), then \pgfcalendarsuggestedname will expand to $\langle name \rangle - \langle date \rangle$, where $\langle date \rangle$ is the date of the day that is currently being processed in ISO format.

For example, if January 1, 2006 is being processed and the calendar has been named mycal, then the node containing the 1 for this date will be names mycal-2006-01-01. You can later reference this node.

```
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
\begin{tikzpicture}
  \calendar (mycal) [dates=2000-01-01 to 2000-01-31,week list];
  \draw[red] (mycal-2000-01-20) circle (4pt);
  \end{tikzpicture}
```

/tikz/day text= $\langle text \rangle$

(no default)

This option changes the setting of the \tikzdaytext. By default, this macro simply yields the current day of month, but you can change it arbitrarily. Here is a silly example:

```
х
                               \mathbf{X}
                                    X
х
            X
                   х
X
            X
                  X
                         Х
                               X
                                     X
X
            Х
                  \mathbf{X}
                        Х
                               \mathbf{X}
                                     Х
X
           X \quad X \quad X \quad X
х
```

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31, week list, day text=x];
```

More useful examples are based on using the \% command. This command is redefined inside a \pgfcalendar to mean the same as \pgfcalendarshorthand. (The original meaning of \% is lost inside the calendar, you need to save if before the calendar if you really need it.)

The $\$ inserts the current day/month/year/day of week in a certain format into the text. The first letter following the $\$ selects the type (permissible values are d, m, y, w), the second letter specifies how the value should be displayed (- means numerically, = means numerically with leading space, 0 means numerically with leading zeros, t means textual, and . means textual, abbreviated). For example $\$ 0 gives the day with a leading zero (for more details see the description of $\$ 0 pgfcalendarshorthand on page 667).

Let us redefine the day text so that it yields the day with a leading zero:

```
01 02
03 04 05 06 07 08 09
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31, week list, day text=\%d0];
```

/tikz/every day (initially anchor=base east)

(no default)

This style is executed by the default node code for each day. The every day style is useful for changing the way days look. For example, let us make all days red:

```
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
\tikz[every day/.style=red]
\calendar[dates=2000-01-01 to 2000-01-31,week list];
```

Changing the appearance of month and year labels. In addition to the days of a calendar, labels for the months and even years (for really long calendars) can be added. These labels are only added once per month or year and this is not done by default. Rather, special styles starting with month label place these labels and make them visible:

```
January

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

February

1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29
```

```
\tikz \calendar[dates=2000-01-01 to 2000-02-last, week list, month label above centered];
```

The following options change the appearance of the month and year label:

/tikz/month code= $\langle code \rangle$

(no default, initially see below)

This option allows you to specify what the macro \tikzmonthcode should expand to.

By default, the \tikzmonthcode it is set to

\node[every month]{\tikzmonthtext};

Note that this node is not named by default.

```
/tikz/month text=\langle text \rangle
```

(no default)

This option allows you to change the macro \tikzmonthtext. By default, the month text is a long textual presentation of the current month being typeset.

```
January 2000

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
/tikz/every month
```

(style, initially empty)

This style can be used to change the appearance of month labels.

```
/tikz/year code=\langle code \rangle
```

(no default)

Works like month code, only for years.

```
/\text{tikz/year text} = \langle text \rangle (no default)
```

Works like month text, only for years.

```
/tikz/every year (no value)
```

Works like every month, only for years.

Date ifs. Much of the power of the \calendar command comes from the use of conditionals. There are two equivalent way of specifying such a conditional. First, you can add the text if $(\langle conditions \rangle)$ $\langle code \ or \ options \rangle$ to your $\langle calendar \ specification \rangle$, possibly followed by else $\langle else \ code \ or \ options \rangle$. You can have multiple such conditionals (but you cannot nest them in this simple manner). The second way is to use the following option:

```
\tikz/if = (\langle conditions \rangle) \langle code \ or \ options \rangle = lse \langle else \ code \ or \ options \rangle (no default)
```

This option has the same effect as giving a corresponding if in the $\langle calendar\ specification \rangle$. The option is mostly useful for use in the every calendar style, where you cannot provide if conditionals otherwise.

Now, regardless of how you specify a conditional, it has the following effect (individually and independently for each date in the calendar):

- 1. It is checked whether the current date is one of the possibilities listed in $\langle conditions \rangle$. An example of such a condition is Sunday. Thus, when you write if (Saturday, Sunday) {foo}, then foo will be executed for every day in the calendar that is a Saturday or a Sunday.
 - The command \ifdate and, thereby, \pgfcalendarifdate are used to evaluate the \(\)conditions\), see page 663 for a complete list of possible tests. The most useful tests are: Tests like Monday and so on, workday for the days Monday to Friday, weekend for Saturday and Sunday, equals for testing whether the current date equals a given date, at least and at least for comparing the current date with a given date.
- 2. If the date passes the check, the $\langle code\ or\ options \rangle$ is evaluated in a manner to be described in a moment; if the date fails, the $\langle else\ code\ or\ options \rangle$ is evaluated, if present.
 - The $\langle code\ or\ options \rangle$ can either be some code. This is indicated by surrounding the code with curly braces. It can also be a list of TikZ options. This is indicated by surrounding the options with square brackets. For example in the date test if (Sunday) {\draw...} else {\fill...} there are two pieces of code involved. By comparison, if (Sunday) [red] else [green] involves two options.

If $\langle code\ or\ options \rangle$ is code, it is simply executed (for the current day). If it is a list of options, these options are passed to a scope surrounding the current date.

Let us now have a look at some examples. First, we use a conditional to make all Sundays red.

```
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

Next, let us do something on a specific date:

```
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

You might wonder why the circle seems to be "off" the date. Actually, it is centered on the date, it is just that the date label uses the base east anchor, which shifts the label up and right. To overcome this problem we can change the anchor:

```
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

However, the single day dates are now no longer aligned correctly. For this, we can change the day text to \%d=, which adds a space at the beginning of single day text.

In the following, more technical information is covered. Most readers may wish to skip it.

The current date code. As mentioned earlier, for each date in the calendar the current date code is executed. It is the job of this code to shift around date nodes, to render the date nodes, to draw the month labels and to do all other stuff that is necessary to draw a calendar.

The current date code consists of the following parts, in this order:

- 1. The before-scope code.
- 2. A scope is opened.
- 3. The at-begin-scope code.
- 4. All date-ifs from the $\langle calendar \ specification \rangle$ are executed.
- 5. The at-end-scope code.
- 6. The scope is closed.
- 7. The after-scope code.

All of the codes mentioned above can be changed using appropriate options, see below. In case you wonder why so many are needed, the reason is that the current date code as a whole is not surrounded by a scope or TEX group. This means that code executed in the before-scope code and in the after-scope code has an effect on all following days. For example, if the after-scope code modifies the transformation matrix by shifting everything downward, all following days will be shifted downward. If each day does this, you get a list of days, one below the other.

However, you do not always want code to have an effect on everything that follows. For instance, if a day has the date-if if (Sunday) [red], we only want this Sunday to red, not all following days also. Similarly, sometimes it is easier to compute the position of a day relative to a fixed origin and we do not want any modifications of the transformation matrix to have an effect outside the scope.

By cleverly adjusting the different codes, all sorts of different day arrangements are possible.

/tikz/execute before day scope= $\langle code \rangle$

(no default)

The $\langle code \rangle$ is executed before everything else for each date. Multiple calls of this option have an accumulative effect. Thus, if you use this option twice, the code from the first use is used first for each day, followed by the code given the second time.

/tikz/execute at begin day scope= $\langle code \rangle$

(no default)

This code is execute before everything else inside the scope of the current date. Again, the effect is accumulative.

/tikz/execute at end day scope= $\langle code \rangle$

(no default)

This code is executed just before the day scope is closed. The effect is also accumulative, however, in reverse order. This is useful to pair, say, \scope and \endscope commands in at-begin- and at-end-code.

/tikz/execute after day scope= $\langle code \rangle$

(no default)

This is executed at the very end of the current date, outside the scope. The accumulation is also in reverse.

In the rest of the following subsections we have a look at how the different scope codes can be used to create different calendar arrangements.

28.1.1 Creating a Simple List of Days

We start with a list of the days of the calendar, one day below the other. For this, we simply shift the coordinate system downward at the end of the code for each day. This shift must be *outside* the day scope as we want day shifts to accumulate. Thus, we use the following code:

Clearly, we can use this approach to create day lists going up, down, right, left, or even diagonally.

28.1.2 Adding a Month Label

We now want to add a month label to the left of the beginning of each month. The idea is to do two things:

- 1. We add code that is executed only on the first of each month.
- 2. The code is executed before the actual day is rendered. This ensures that options applying to the days do not affect the month rendering.

We have two options where we should add the month code: Either we add it at the beginning of the day scope or before. Either will work fine, but it might be safer to put the code inside the scope to ensure that settings to not inadvertently "leak outside."

```
January

1
2
3
4
4
5
6
7
8

\tikz
\calendar
[dates=2000-01-01 to 2000-01-08,
execute after day scope={\pgftransformyshift{-1em}},
execute at begin day scope=
{\ifdate{\day of month=1}{\tikzmonthcode}{\f}},
every month/.append style={anchor=base east,xshift=-2em}];
```

28.1.3 Creating a Week List Arrangement

Let us now address a more complicated arrangement: A week list. In this arrangement there is line for each week. The horizontal placement of the days is thus that all Mondays lie below each other, likewise for all Tuesdays, and so on.

In order to typeset this arrangement, we can use the following approach: The origin of the coordinate system rests at the anchor for the Monday of each week. That means that at the end of each week the origin is moved downward one line. On all other days, the origin at the end of the day code is the same as at the beginning. To position each day correctly, we use code inside and at the beginning of the day scope to horizontally shift the day according to its day of week.

28.1.4 Creating a Month List Arrangement

For another example, let us create an arrangement that contains one line for each month. This is easy enough to do as for weeks, unless we add the following requirement: Again, we want all days in a column to have the same day of week. Since months start on different days of week, this means that each row has to have an individual offset.

One possible way is to use the following approach: After each month (or at the beginning of each month) we advance the vertical position of the offset by one line. For horizontal placement, inside the day scope we locally shift the day by its day of month. Furthermore, we must additionally shift the day to ensure that the first day of the month lies on the correct day of week column. For this, we remember this day of week the first time we see it.


```
\newcount\mycount
\tikz
  \calendar
    [dates=2000-01-01 to 2000-02-last,
    execute before day scope=
       \ifdate{day of month=1} {
         % Remember the weekday of first day of month
         \mycount=\pgfcalendarcurrentweekday
         % Shift downward
         \pgftransformyshift{-1em}
      }{}
    execute at begin day scope=
       % each day is shifted right according to the day of month
       \pgftransformxshift{\pgfcalendarcurrentday em}
       \% and additionally according to the weekday of the first
       \pgftransformxshift{\the\mycount em}
```

28.2 Arrangements

An arrangement specifies how the days of calendar are arranged on the page. The calendar library defines a number of predefined arrangements.

We start with arrangements in which the days are listed in a long line.

/tikz/day list downward

(style, no value)

This style causes the days of a month to be typeset one below the other. The shift between days is given by day yshift. Between month an additional shift of month yshift is added.

/tikz/day list upward

(style, no value)

works as above, only the list grows upward instead of downward.

/tikz/day list right

(style, no value)

This style also works as before, but the list of days grows to the right. Instead of day yshift and month yshift, the values of day xshift and month xshift are used.

```
28 29 30 31 1 2 3
```

/tikz/day list left

(style, no value)

As above, but the list grows left.

The next arrangement lists days by the week.

/tikz/week list (style, no value)

This style creates one row for each week in the range. The value of day xshift is used for the distance between days in each week row, the value of day yshift is used for the distance between rows. In both cases, "distance" refers to the distance between the anchors of the nodes of the days (or, more generally, the distance between the origins of the little pictures created for each day).

The days inside each week are shifted such that Monday is always at the first position (to change this, you need to copy and then modify the code appropriately). If the date range does not start on a Monday, the first line will not start in the first column, but rather in the column appropriate for the first date in the range.

At the beginning of each month (except for the first month in the range) an additional vertical space of month yshift is added. If this is set to Opt you get a continuous list of days.

```
1 2

3 4 5 6 7 8 9

10 11 12 13 14 15 16

17 18 19 20 21 22 23

24 25 26 27 28 29 30

31

1 2 3 4 5 6

7 8 9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29
```

```
tikz \calendar [dates=2000-01-01 to 2000-02-last,week list];
```

```
\tikz
                 1
                          \calendar [dates=2000-01-01 to 2000-02-last.week list.
3 4 5 6
            7
                8
                   9
                                   month yshift=0pt];
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
       2
         3
            4 5
   8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29
```

The following arrangement gives a very compact view of a whole year.

/tikz/month list (style, no value)

In this arrangement there is a row for each month. As for the week list, the day xshift is used for the horizontal distance. For the vertical shift, month yshift is used.

In each row, all days of the month are listed alongside each other. However, it is once more ensured that days in each column lie on the same day of week. Thus, the very first column contains only Mondays. If a month does not start with a Monday, its days are shifted to the right such that the days lie on the correct columns.

```
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 January
 February
              1 2 3 4
                             6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
   March
                                 6 \quad 7 \quad 8 \quad 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ 16 \ 17 \ 18 \ 19 \ 20 \ 21 \ 22 \ 23 \ 24 \ 25 \ 26 \ 27 \ 28 \ 29 \ 30 \ 31
                                3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
    April
             2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ 16 \ 17 \ 18 \ 19 \ 20 \ 21 \ 22 \ 23 \ 24 \ 25 \ 26 \ 27 \ 28 \ 29 \ 30 \ 31
     May
                                 5 \quad 6 \quad 7 \quad 8 \quad 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15 \ 16 \ 17 \ 18 \ 19 \ 20 \ 21 \ 22 \ 23 \ 24 \ 25 \ 26 \ 27 \ 28 \ 29 \ 30
    June
                    1
                       2
                          July
  August
              1 2 3 4
                          5
                             6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
                                    5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
                              1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 October
November
                 1 2 3
                          4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
                                         7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
December
```

```
\sffamily\scriptsize
\tikz
\calendar [dates=2000-01-01 to 2000-12-31,
month list,month label left,month yshift=1.25em]
if (Sunday) [black!50];
```

28.3 Month Labels

For many calendars you may wish to add a label to each month. We have already covered how month nodes are created and rendered in the description of the \calendar command: use month text, every month, and also month code (if necessary) to change the appearance of the month labels.

What we have not yet covered is where these labels are placed. By default, they are not placed at all as there is no good "default position" for them. Instead, you can use one of the following options to specify a position for the labels:

```
/tikz/month label left (style, no value)
```

Places the month label to the left of the first day of the month. (For week list and month list where a month does not start on a Monday, the position is chosen "as if" the month had started on a Monday – which is usually exactly what you want.)

```
28
29
30
31

February 1
2
3

Tebruary 1
2
3
```

/tikz/month label left vertical

(style, no value)

This style works like the above style, only the label is rotated counterclockwise by 90 degrees.

```
28
29
30
31

Ling 1
20
31

Arg 1
20
31

Arg 2
31

Arg 2
32

Arg 2
33

Arg 2
34

Arg 2
35

Arg 2
36

Arg 2
37

Arg 2
38

Arg 2
48

Arg 2
```

/tikz/month label right

(style, no value)

This style places the month label to the right of the row in which the first day of the month lies. This means that for a day list the label is to the right of the first day, for a week list it is to the right of the first week, and for a month list it is to the right of the whole month.

```
28
29
30
31

1 February
2
3
```

/tikz/month label right vertical

(style, no value)

Works as above, only the label is rotated clockwise by 90 degrees.

```
28
29
30
31

1 For 2 pru 3
28
29
30
31
```

/tikz/month label above left

(style, no value)

This style places the month label above of the row of the first day, flushed left to the leftmost column. The amount by which the label is raised is fixed to 1.25em; use the yshift option with the month node to modify this.

```
February 28 29 30 31 1 2 3
```

```
20 21 22 23
24 25 26 27 28 29 30
31
February
1 2 3 4 5 6
```

8 9 10

```
\tikz
\calendar [dates=2000-01-20 to 2000-02-10,
    week list,month label above left];
```

/tikz/month label above centered

(style, no value)

works as above, only the label is centered above the row containing the first day.

```
February
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
```

```
20 21 22 23
24 25 26 27 28 29 30
31
February
1 2 3 4 5 6
7 8 9 10
```

/tikz/month label above right

(style, no value)

works as above, but flushed right

```
20 21 22 23
24 25 26 27 28 29 30
31 February
1 2 3 4 5 6
7 8 9 10
```

```
\tikz
\calendar [dates=2000-01-20 to 2000-02-10,
week list,month label above right];
```

/tikz/month label below left

(style, no value)

Works like month label above left, only the label is placed below the row. This placement is not really useful with the week list arrangement, but rather with the day list right or month list arrangement.

/tikz/month label below centered

(style, no value)

Works like month label above centered, only below.

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

February
```

```
\calendar [dates=2000-02-01 to 2000-02-last, day list right,month label below centered];
```

28.4 Examples

In the following, some example calendars are shown that come either from real applications or are just nice to look at.

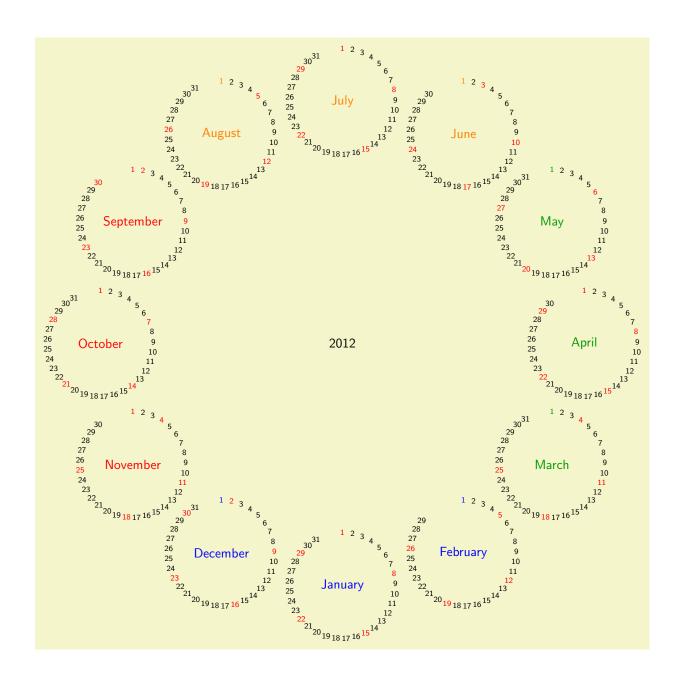
Let us start with a year-2100-countdown, in which we cross out dates as we approach the big celebration. For this, we set the shape to strike out for these dates.

```
\begin{tikzpicture}
    December 2099
                             \calendar
    1 2 3 A 5 6
                               dates=2099-12-01 to 2100-01-last,
 7 8 8 10 11 12 13
                               week list, inner sep=2pt, month label above centered,
14 15 16 17 18 19 20
                               month text=\%mt \%y0
21 22 23 24 25 26 27
                             1
                             if (at most=2099-12-29) [nodes={strike out,draw}]
28 29 30 31
                             if (weekend)
                                                     [black!50, nodes={draw=none}]
     January 2100
                           \end{tikzpicture}
 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

The next calendar shows a deadline, which is 10 days in the future from the current date. The last three days before the deadline are in red, because we really should be done by then. All days on which we can no longer work on the project are crossed out.

```
\begin{tikzpicture}
      10 11 12 13 14
                            \calendar
15 16 17 18 19 20 21
22 23 24 25 26 27 28
                              dates=\year-\month-\day+-25 to \year-\month-\day+25,
                              week list,inner sep=2pt,month label above centered,
29 30 31
                              month text=\textit{\%mt \%y0}
                            ٦
    November 2012
                            if (at least=\year-\month-\day) {}
                              else [nodes={strike out,draw}]
           1 2 3 4
                            if (at most=\sqrt{month-\sqrt{ay+7}})
 5 6 7 8 9 10 11
                              [green!50!black]
                            if (between=\year-\month-\day+8 and \year-\month-\day+10)
12 13 14 15 16 17 18
                              [red]
19 20 21 22 23 24 25
                            if (Sunday)
26 27 28 29
                              [gray,nodes={draw=none}]
                          \end{tikzpicture}
```

The following example is a futuristic calendar that is all about circles:



```
\sffamily
\colorlet{winter}{blue}
\colorlet{spring}{green!60!black}
\colorlet{summer}{orange}
\colorlet{fall}{red}
\mbox{\ensuremath{\it \%}} A counter, since TikZ is not clever enough (yet) to handle
% arbitrary angle systems.
\newcount\mycount
\begin{tikzpicture}
  [transform shape,
   every day/.style={anchor=mid,font=\fontsize{6}{6}\selectfont}]
  \node{\normalsize\the\year};
  \foreach \month/\monthcolor in
    {1/winter,2/winter,3/spring,4/spring,5/spring,6/summer,
     7/summer,8/summer,9/fall,10/fall,11/fall,12/winter}
  {
    % Compute angle:
    \mycount=\month
    \advance\mycount by -1
    \multiply\mycount by 30
    \advance\mycount by -90
    % The actual calendar
    \calendar at (\the\mycount:6.4cm)
      {\tt dates=\the\year-\month-01\ to\ \the\year-\month-last,}
    if (day of month=1) {\color{\monthcolor}\tikzmonthcode}
    if (Sunday) [red]
    if (all)
      % Again, compute angle
      \advance\mycount by -\pgfcalendarcurrentday
      \multiply\mycount by 11
      \advance\mycount by 90
      \pgftransformshift{\pgfpointpolar{\mycount}{1.4cm}}
\end{tikzpicture}
```

Next, let's us have a whole year in a tight column:

```
01
   2 3 4 5 6 7 8
   9 10 11 12 13 14 15
   16 17 18 19 20 21 22
   23 24 25 26 27 28 29
02 30 31 1 2 3 4 5
   6 7 8 9 10 11 12
   13 14 15 16 17 18 19
  20 21 22 23 24 25 26
03 27 28 29 1 2 3 4
   5 6 7 8 9 10 11
   12 13 14 15 16 17 18
  19 20 21 22 23 24 25
04 26 27 28 29 30 31 1
   2 3 4 5 6 7 8
   9 10 11 12 13 14 15
  16 17 18 19 20 21 22
   23 24 25 26 27 28 29
05 30 1 2 3 4 5 6
   7 8 9 10 11 12 13
  14 15 16 17 18 19 20
  21 22 23 24 25 26 27
06 28 29 30 31 1 2 3
   4 5 6 7 8 9 10
  11 12 13 14 15 16 17
  18 19 20 21 22 23 24
07 25 26 27 28 29 30 1
   2 3 4 5 6 7 8
   9 10 11 12 13 14 15
  16 17 18 19 20 21 22
   23 24 25 26 27 28 29
08 30 31 1 2 3 4 5
   6 7 8 9 10 11 12
   13 14 15 16 17 18 19
   20 21 22 23 24 25 26
09 27 28 29 30 31 1
   3 4 5 6 7 8 9
  10 11 12 13 14 15 16
  17 18 19 20 21 22 23
   24 25 26 27 28 29 30
10 1 2 3 4 5 6 7
   8 9 10 11 12 13 14
   15 16 17 18 19 20 21
   22 23 24 25 26 27 28
11 29 30 31 1 2 3 4
   5 6 7 8 9 10 11
  12 13 14 15 16 17 18
  19 20 21 22 23 24 25
12 26 27 28 29 30 1 2
   3 4 5 6 7 8 9
  10 11 12 13 14 15 16
  17 18 19 20 21 22 23
   24 25 26 27 28 29 30
   31
```

29 Chains

```
\usetikzlibrary{chains} % MEX and plain TeX \usetikzlibrary[chains] % ConTeXt
```

This library defines options for creating chains.

29.1 Overview

Chains are sequences of nodes that are – typically – arranged in a row or a column and that are – typically – connected by edges. More generally, they can be used to position nodes of a branching network in a systematic manner. For the positioning of nodes in rows and columns you can also use matrices, see Section 18, but chains can also be used to describe the connections between nodes that have already been connected using, say, matrices. Thus, it often makes sense to use matrices for the positioning of elements and chains to describe the connections.

29.2 Starting and Continuing a Chain

Typically, you construct one chain at a time, but it is permissible to construct multiple chains simultaneously. In this case, the chains must be named differently and you must specify for each node which chain it belongs to.

The first step toward creating a chain is to use the start chain option.

```
\ttikz/start chain=\cchain name\cdirection\c (no default)
```

This key should, but need not, be given as an option to a scope enclosing all nodes of the chain. Typically, this will be a **scope** or the whole **tikzpicture**, but it might just be a path on which all nodes of the chain are found. If no *chain name* is given, the default value **chain** will be used instead.

The key starts a chain named $\langle chain \; name \rangle$ and makes it *active*, which means that it is currently being constructed. The **start** chain can be issued only once to activate a chain, inside a scope in which a chain is active you cannot use this option once more (for the same chain name). The chain stops being active at the end of the scope in which the **start** chain command was given.

Although chains are only locally active (that is, active inside the scope the start chain command was issued), the information concerning the chains is stored globally and it is possible to *continue* a chain after a scope has ended. For this, the continue chain option can be used, which allows you to reactivate an existing chain in another scope.

The $\langle direction \rangle$ is used to determine the placement rule for nodes on the chain. If it is omitted, the current value of the following key is used:

```
/tikz/chain default direction=\langle direction \rangle (no default, initially going right)
This \langle direction \rangle is used in a chain option, if no other \langle direction \rangle is specified.
```

This \aircciton, is used in a chain option, it no other \aircciton, is specified.

The $\langle direction \rangle$ can have two different forms: going $\langle options \rangle$ or placed $\langle options \rangle$. The effect of these rules will be explained in the description of the on chain option. Right now, just remember that the $\langle direction \rangle$ you provide with the chain option applies to the whole chain.

Other than this, this key has no further effect. In particular, to place nodes on the chain, you must use the on chain option, described next.

```
\begin{tikzpicture}[start chain]
          В
Α
                            % The chain is called just "chain"
                            \node [on chain] {A};
                            \node [on chain] {B};
                            \node [on chain] {C};
                          \end{tikzpicture}
                          \begin{tikzpicture}
          В
Α
                            % Same as above, using the scope shorthand
                            { [start chain]
                              \node [on chain] {A};
                              \node [on chain]
                              \node [on chain] {C};
                          \end{tikzpicture}
```

/tikz/continue chain= $\langle chain \ name \rangle \langle direction \rangle$

(no default)

This option allows you to (re)activate an existing chain and to possibly change the default direction. If the chain name is missing, the name of the innermost activated chain is used. If no chain is activated, chain is used.

Let us have a look at the two different applications of this option. The first is to change the direction of a chain as it is being constructed. For this, just give this option somewhere inside the scope of the chain.

The second application is to reactivate a chain after it "has already been closed down."

```
\begin{tikzpicture}[node distance=5mm,
   В
          С
                D
                                            every node/.style=draw]
0
                         { [start chain=1]
                           \node [on chain] {A};
                           \node [on chain] {B};
                           \node [on chain] {C};
                         { [start chain=2 going below]
                           \node [on chain=2] at (0.5,-.5) {0};
                           \node [on chain=2] {1};
                           \node [on chain=2] \{2\};
                         { [continue chain=1]
                           \node [on chain] {D};
                       \end{tikzpicture}
```

29.3 Nodes on a Chain

```
/tikz/on chain=\langle chain \ name \rangle \langle direction \rangle
```

(no default)

This key should be given as an option to a node. When the option is used, the $\langle chain \; name \rangle$ must be the name of a chain that has been started using the start chain option. If $\langle chain \; name \rangle$ is the empty string, the current value of the innermost activated chain is used. If this option is used several times for a node, only the last invocation "wins." (To place a node on several chains, use the \chainin command repeatedly.)

The $\langle direction \rangle$ part is optional. If present, it sets the direction used for this node, otherwise the $\langle direction \rangle$ that was given to the original start chain option is used (or of the last continue chain option, which allows you to change this default).

The effects of this option are the following:

1. An internal counter (there is one local counter for each chain) is increased. This counter reflects the current number of the node in the chain, where the first node is node 1, the second is node 2, and so on.

The value of this internal counter is globally stored in the macro \tikzchaincount.

- 2. If the node does not yet have a name, (having been given using the name option or the name-syntax), the name of the node is set to \(\chin name \) -\(\chin value of the internal chain counter \). For instance, if the chain is called nums, the first node would be named nums-1, the second nums-2, and so on. For the default chain name chain, the first node is named chain-1, the second chain-2, and so on.
- 3. Independently of whether the name has been provided automatically or via the name option, the name of the node is globally stored in the macro \tikzchaincurrent.
- 4. Except for the first node, the macro \tikzchainprevious is now globally set to the name of the node of the previous node on the chain. For the first node of the chain, this macro is globally set to the empty string.
- 5. Except possibly for the first node of the chain, the placement rule is now executed. The placement rule is just a TikZ option that is applied automatically to each node on the chain. Depending on the form of the $\langle direction \rangle$ parameter (either the locally given one or the one given to the start chain option), different things happen.

First, it makes a difference whether the *\(\direction\)* starts with going or with placed. The difference is that in the first case, the placement rule is not applied to the first node of the chain, while in the second case the placement rule is applied also to this first node. The idea is that a going-direction indicates that we are "going somewhere relative to the previous node" whereas a placed indicates that we are "placing nodes according to their number."

Independently of which form is used, the $\langle text \rangle$ inside $\langle direction \rangle$ that follows going or placed (separated by a compulsory space) can have two different effects:

- (a) If it contains an equal sign, then this $\langle text \rangle$ is used as the placement rule, that is, it is simply executed.
- (b) If it does not contain an equal sign, then $\langle text \rangle = of \tikzchainprevious$ is used as the placement rule.

Note that in the first case, inside the $\langle text \rangle$ you have access to \tikzchainprevious and \tikzchaincount for doing your positioning calculations.

6. The following style is executed:

```
/tikz/every on chain
```

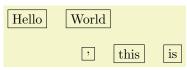
(style, no value)

This key is executed for every node on a chain, including the first one.

Recall that the standard placement rule has a form like right=of (\tikzchainprevious). This means that each new node is placed to the right of the previous one, spaced by the current value of node distance.

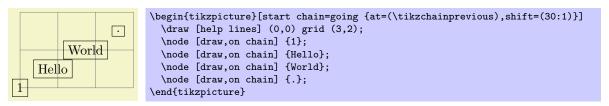
```
Hallo Welt | \begin{tikzpicture}[start chain,node distance=5mm] \ node [draw,on chain] {}; \ node [draw,on chain] {Hallo}; \ node [draw,on chain] {Welt}; \ end{tikzpicture}
```

The optional $\langle direction \rangle$ allows us to temporarily change the direction in the middle of a chain:



```
\begin{tikzpicture}[start chain,node distance=5mm]
  \node [draw,on chain] {Hello};
  \node [draw,on chain] {World};
  \node [draw,on chain=going below] {,};
  \node [draw,on chain] {this};
  \node [draw,on chain] {is};
  \end{tikzpicture}
```

You can also use more complicated computations in the $\langle direction \rangle$:



For each chain, two special "pseudo nodes" are created.

Predefined node (chain name)-begin

This node is the same as the first node on the chain. It is only defined after a first node has been defined.

Predefined node (chain name)-end

This node is the same as the (currently) last node on the chain. As the chain is extended, this node changes.

The on chain option can also be used, in conjunction with late options, to add an already existing node to a chain. The following command, which is only defined inside scopes where a start chain option is present, simplifies this process.

```
\chainin(\langle existing \ name \rangle) [\langle options \rangle]
```

This command makes it easy to add a node to chain that has already been constructed. This node may even be part of a another chain.

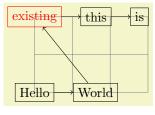
When you say \chainin (some node);, the node some node must already exist. It will then be made part of the current chain. This does not mean that the node can be changed (it is already constructed, after all), but the join option can be used to join some node to the previous last node on the chain and subsequent nodes will be placed relative to some node.

It is permissible to give the on chain option inside the $\langle options \rangle$ in order to specify on which chain the node should be put.

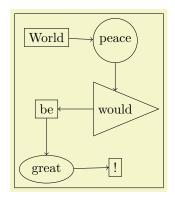
This command is just a shortcut for

```
\path (\langle existing \ name \rangle) [late options={on chain, every chain in, \langle options \rangle}]
```

In particular, it is possible to continue to path after a **\chainin** command, though that does not seem very useful.



Here is an example where nodes are positioned using a matrix and then connected using a chain



```
\begin{tikzpicture}[every node/.style=draw]
  \matrix [matrix of nodes, column sep=5mm, row sep=5mm]
                    World & |(b) [circle]|
    |(a)|
                                                         peace \\
    |(c)|
                        & |(d) [isosceles triangle]| would \\
                    be
    |(e) [ellipse]| great & |(f)|
  { [start chain, every on chain/.style={join=by ->}]
    \chainin (a);
    \chainin (b);
    \chainin (d);
    \chainin (c);
    \chainin (e):
    \chainin (f);
\end{tikzpicture}
```

29.4 Joining Nodes on a Chain

```
/tikz/join=with \langle with \rangle by \langle options \rangle
```

(no default)

When this key is given to any node on a chain (except possibly for the first node), an edge command is added after the node. The with part specifies which node should be used for the start point of the edge; if the with part is omitted, the \tikzchainprevious is used. This edge command gets the \(options \) as parameter and the current node as its target. If there is no previous node and no with is given, no edge command gets executed.

```
/tikz/every join (style, no value)
```

This style is executed each time this command is used.

Note that it makes sense to call this option several times for a node, in order to connect it to several nodes. This is especially useful for joining in branches, see the next section.

```
Hallo Welt foo
```

29.5 Branches

A branch is a chain that (typically only temporarily) extends an existing chain. The idea is the following: Suppose we are constructing a chain and at some node x there is a fork. In this case, one (or even more) branches starts at this fork. For each branch a chain is created, but the first node on this chain should be x. For this, it is useful to use \chainin on the node x to make it part of the different branch chains and to name the branch chains in some way that reflects the name of the main chain.

The start branch option provides a shorthand for doing exactly what was just described.

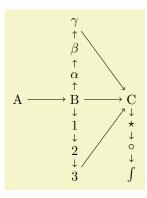
```
\texttt{/tikz/start branch=}\langle branch\ name \rangle \langle direction \rangle
```

(no default)

This key is used in the same manner as the start chain command, however, the effect is slightly different:

- This option may only be used if some chain is already active and there is a (last) node on this chain. Let us call this node the $\langle fork \; node \rangle$.
- The chain is not just called $\langle branch\ name \rangle$, but $\langle current\ chain \rangle / \langle branch\ name \rangle$. For instance, if the $\langle fork\ node \rangle$ is part of the chain called trunk and the $\langle branch\ name \rangle$ is set to left, the complete chain name of the branch is trunk/left. The $\langle branch\ name \rangle$ must be given, there is no default value.

• The \(\langle fork node \rangle \) is automatically "chained into" the branch chain as its first node. Thus, for the first node on the branch that you provide, the join option will cause it to be connected to the fork node.

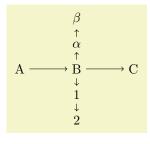


```
\begin{tikzpicture}[every on chain/.style=join,every join/.style=->,
                    node distance=2mm and 1cm]
  { [start chain=trunk]
    \node [on chain] {A};
    \node [on chain] {B};
    { [start branch=numbers going below]
      \node [on chain] {1};
      \node [on chain] {2};
      \node [on chain] {3};
    { [start branch=greek going above]
      \node [on chain] {$\alpha$};
      \node [on chain] {$\beta$};
      \node [on chain] {$\gamma$};
    \node [on chain,join=with trunk/numbers-end,join=with trunk/greek-end] {C};
    { [start branch=symbols going below]
      \node [on chain] {$\star$};
      \node [on chain] {$\circ$};
      \node [on chain] {$\int$};
 }
\end{tikzpicture}
```

/tikz/continue branch= $\langle branch \ name \rangle \langle direction \rangle$

(no default)

This option works like the continue chain option, only $\langle current \ chain \rangle / \langle branch \ name \rangle$ is used as the chain name, rather than just $\langle branch \ name \rangle$.



```
\begin{tikzpicture}[every on chain/.style=join,every join/.style=->,
                    node distance=2mm and 1cm]
  { [start chain=trunk]
    \node [on chain] {A};
    \node [on chain] {B};
    { [start branch=numbers going below] } % just a declaration,
                           going above] } % we will come back later
    { [start branch=greek
    \node [on chain] {C};
    % Now come the branches...
      [continue branch=numbers]
      \node [on chain] {1};
      \node [on chain] {2};
    { [continue branch=greek]
      \node [on chain] {$\alpha$};
      \node [on chain] {$\beta$};
 }
\end{tikzpicture}
```

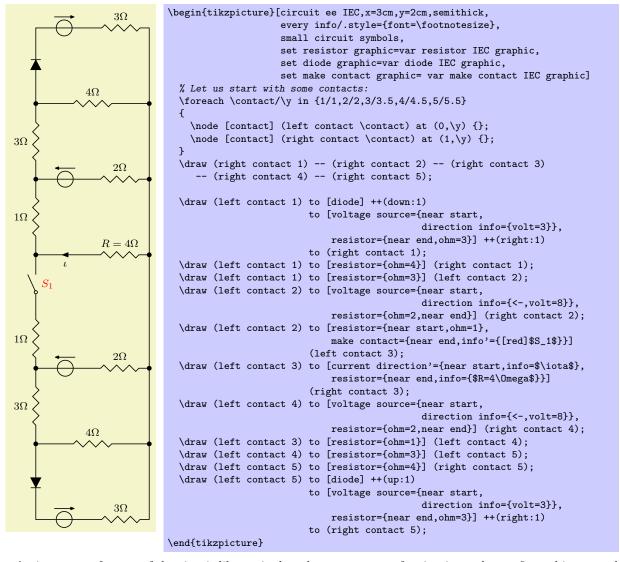
30 Circuit Libraries

Written and documented by Till Tantau, and Mark Wibrow. Inspired by the work of Massimo Redaelli.

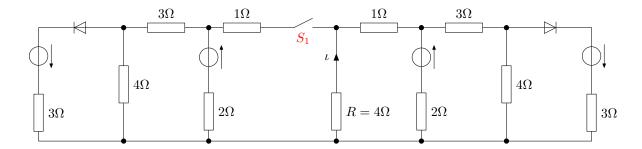
30.1 Introduction

The circuit libraries can be used to draw different kinds of electrical or logical circuits. There is not a single library for this, but a whole hierarchy of libraries that work in concert. The main design goal was to create a balance between ease-of-use and ease-of-extending, while creating high-quality graphical representations of circuits.

30.1.1 A First Example



An important feature of the circuit library is that the appearance of a circuit can be configured in general ways and that the labels are placed automatically by default. Here is the graphic once more, generated from exactly the same source code, with only the options of the {tikzpicture} environment replaced by [rotate=-90,circuit ee IEC,x=3.25cm,y=2.25cm]:



30.1.2 Symbols

A circuit typically consists of numerous electronic elements like logical gates or resistors or diodes that are connected by wires. In PGF/TikZ, we use nodes for the electronic elements and normal lines for the wires. TikZ offers a large number of different ways of positioning and connecting nodes in general, all of which can be used here. Additionally, the circuits library defines an additional useful to-path that is particularly useful for elements like a resistor on a line.

There are many different names that are used to refer to electrical "elements," so a bit of terminology standardization is useful: We will call such elements *symbols*. A *symbol shape* is a PGF shape declared using the \pgfdeclareshape command. A *symbol node* is a node whose shape is a symbol shape.

30.1.3 Symbol Graphics

Symbols can be created by \node[shape=some symbol shape]. However, in order to represent some symbols correctly, just using standard PGF shapes is not sufficient. For instance, most symbols have a visually appealing "default size," but the size of a symbol shape depends only on the current values of parameters like minimum height or inner xsep.

For these reasons, the circuit libraries introduce the concept of a *symbol graphic*. This is a style that causes a \node to not only have the correct shape, but also the correct size and the correct path usage. More generally, this style may set up things in any way so that the "symbol looks correct". When you write, for instance, \node[diode], then the style called diode graphic is used, which in turn is set to something like shape=diode IEC,draw,minimum height=....

Here is an overview of the different kinds of circuit libraries:

• The TikZ-library circuits defines general keys for creating circuits. Mostly, these keys are useful for defining more specialized libraries.

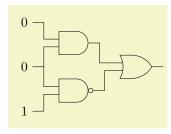
You normally do not use this library directly since it does not define any symbol graphics.

- The TikZ-library circuits.logic defines keys for creating logical gates like and-gates or xor-gates. However, this library also does not actually define any symbol graphics; this is done by two sublibraries:
 - The library circuits.logic.US defines symbol graphics that cause the logical gates to be rendered in the "US-style." It includes all of the above libraries and you can use this library directly.
 - The library circuits.logic.IEC also defines symbol graphics for logical gates, but it uses rectangular gates rather that the round US-gates. This library can coexist peacefully with the above library, you can change which symbol graphics are used "on the fly."
- The TikZ-library cirucits.ee defines keys for symbols from electrical engineering like resistors or capacitors. Again, sublibraries define the actual symbol graphics.
 - The library circuits.ee.IEC defines symbol shapes that follow the IEC norm.
- The PGF-libraries shapes.gates.* define (circuit) symbol shapes. However, you normally do not use these shapes directly, rather you use a style that uses an appropriate symbol graphic, which in turn uses one of these shapes.

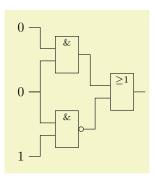
Let us have a look at a simple example. Suppose we wish to create a logical circuit. Then we first have to decide which symbol graphics we would like to use. Suppose we wish to use the US-style, then we would include the library circuits.logic.US. If you wish to use IEC-style symbols, use circuits.logic.IEC. If you cannot decide, include both:

```
\usetikzlibrary{circuits.logic.US,circuits.logic.IEC}
```

To create a picture that contains a US-style circuit you can now use the option circuit logic US. This will set up keys like and gate to create use an appropriate symbol graphic for rendering an and gate. Using the circuit logic IEC instead will set up and gate to use another symbol graphic.



```
\begin{tikzpicture}[circuit logic US]
  \matrix[column sep=7mm]
    \node (i0) {0}; &
                    & \node [and gate] (a1) {}; & \\
    \node (i1) {0}; &
                                                 & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {}; & \\
    \node (i2) {1}; &
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
 \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
 \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
 \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

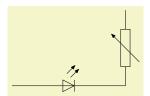


```
\begin{tikzpicture}[circuit logic IEC]
 \matrix[column sep=7mm]
   \node (i0) {0}; &
                   & \node [and gate] (a1) {}; & \\
   \node (i1) {0}; &
                                                 & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {}; & \\
   \node (i2) {1}; &
 \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
 \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
 \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
 \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
 \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
 \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
 \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

30.1.4 Annotations

An annotation is a little extra drawing that can be added to a symbol. For instance, when you add two little parallel arrows pointing away from some electrical element, this usually means that the element is light emitting.

Instead of having one symbol for "diode" and another for "light emitting diode," there is just one diode symbol, but you can add the light emitting annotation to it. This is done by passing the annotation as a parameter to the symbol as in the following example:



30.2 The Base Circuit Library

```
\usetikzlibrary{circuits} % MEX and plain TEX \usetikzlibrary[circuits] % ConTEXt
```

This library is a base library that is included by other circuit libraries. You do not include it directly, but you will typically use some of the general keys, described below.

/tikz/circuits (no value)

This key should be passed as an option to a picture or a scope that contains a circuit. It will do some internal setups. This key is normally called by more specialized keys like circuit ee IEC.

30.2.1 Symbol Size

/tikz/circuit symbol unit= $\langle dimension \rangle$

(no default, initially 7pt)

This dimension is a "unit" for the size of symbols. The libraries generally define the sizes of symbols relative to this dimension. For instance, the longer side of an inductor is, by default, in the IEC library equal to five times this $\langle dimension \rangle$. When you change this $\langle dimension \rangle$, the size of all symbols will automatically change accordingly.

Note, that it is still possible to overwrite the size of any particular symbol. These settings apply only to the default sizes.

/tikz/huge circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 10pt.

/tikz/large circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 8pt.

/tikz/medium circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 7pt.

/tikz/small circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 6pt.

/tikz/tiny circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 5pt.

```
/tikz/circuit symbol size=width \langle width \rangle height \langle height \rangle
```

(no default)

This key sets minimum height to $\langle height \rangle$ times the current value of the circuit symbol unit and the minimum width to $\langle width \rangle$ times this value. Thus, this option can be used with a node command to set the size of the node as a multiple of the circuit symbol unit.

30.2.2 Declaring New Symbols

/tikz/circuit declare symbol= $\langle name \rangle$

(no default)

This key is used to declare a symbol. It does not cause this symbol to be shown nor does it set a graphic to be used for the symbol, it simply "prepares" several keys that can later be used to draw a symbol and to configure it.

In detail, the first key that is defined is just called $\langle name \rangle$. This key should be given as an option to a node or on a to path, as explained below. The key will take options, which can be used to influence the way the symbol graphic is rendered.

Let us have a look at an example. Suppose we want to define a symbol called foo, which just looks like a simple rectangle. We could then say

\tikzset{circuit declare symbol=foo}

The symbol could now be used like this:

```
\node [foo] at (1,1) {};
\node [foo={red}] at (2,1) {};
```

However, in the above example we would not actually see anything since we have not yet set up the graphic to be used by foo. For this, we must use a key called set foo graphic or, generally, set $\langle name \rangle$ graphic. This key gets graphic options as parameter that will be set when a symbol foo should be shown:

```
\begin{tikzpicture}
  [circuit declare symbol=foo,
    set foo graphic={draw,shape=rectangle,minimum size=5mm}]

\node [foo] at (1,1) {};
  \node [foo={red}] at (2,1) {};
\end{tikzpicture}
```

In detail, when you use the key $\langle name \rangle = \langle options \rangle$ with a node, the following happens:

- 1. The inner sep is set to 0.5pt.
- 2. The following style is executed:

```
/tikz/every circuit symbol
```

(style, no value)

Use this style to set up things in general.

- 3. The graphic options that have been set using set $\langle name \rangle$ graphic are set.
- 4. The style every $\langle name \rangle$ is executed. You can use it to configure the symbol further.
- 5. The $\langle options \rangle$ are executed.

The key $\langle name \rangle$ will have a different effect when it is used on a to path command inside a circuit environment (the circuit environment sets up to paths in such a way that the use of a key declared using circuit declare symbol is automatically detected). When $\langle name \rangle$ is used on a to path, the above actions also happen (setting the inner separation, using the symbol graphic, and so on), but they are passed to the key circuit handle symbol, which is explained next.

/tikz/circuit handle symbol= $\langle options \rangle$

(no default)

This key is mostly used internally. Its purpose is to render a symbol. The effect of this key differs, depending on whether it is used as the optional argument of a to path command or elsewhere.

If the key is not used as an argument of a to path command, the $\langle options \rangle$ are simply executed.

The more interesting case happens when the key is given on a to path command. In this case, several things happen:

- 1. The to path is locally changed and set to an internal path (which you should not try to change) that consists mostly of a single straight line.
- 2. The \(\langle options \rangle\) are tentatively executed with filtering switched on. Everything is filtered out, except for the key pos and also the styles at start, very near start, near start, midway, near end, very near end, and at end. If none of them is found, midway is used.
- 3. The filtered option is used to determine a position for the symbol on the path. At the given position (with pos=0 representing the start and pos=1 representing the end), a node will be added to the path (in a manner to be described presently).
- 4. This node gets $\langle options \rangle$ as its option list.
- 5. The node is added by virtue of a special markings decoration. This means that a mark command is executed that causes the node to be placed as a mark on the path.
- 6. The marking decoration will automatically subdivide the path and cause a line to be drawn from the start of the path to the node's border (at the position that lies on a line from the node's center to the start of the path) and then from the node's border (at a position on the other side of the node) to the end of the path.

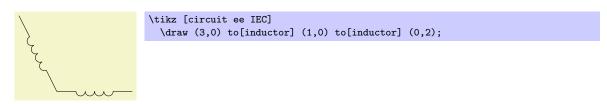
- 7. The marking decoration will also take care of the case that multiple marks are present on a path, in this case the lines from and to the borders of the nodes are only between consecutive nodes.
- 8. The marking decoration will also rotate the coordinate system in such a way that the x-axis points along the path. Thus, if you use the transform shape option, the node will "point along" the path.
- 9. In case a node is at pos=0 or at pos=1 some special code will suppress the superfluous lines to the start or end of the path.

The net effect of all of the above is that a node will be placed "on the path" and the path will have a "gap" just large enough to encompass the node. Another effect is that you can use this key multiple times on a path to add several node to a path, provided they do not overlap.

30.2.3 Pointing Symbols in the Right Direction

Unlike normal nodes, which generally should not be rotated since this will make their text hard to read, symbols often need to be rotated. There are two ways of achieving such rotations:

1. When you place a symbol on a to path, the graphic symbol is automatically rotated such that it "points along the path." Here is an examples that shows how the inductor shape (which looks, unrotated, like this: \(\cop_{\infty}\)) is automatically rotated around:



2. Many shapes cannot be placed "on" a path in this way, namely whenever there are more than two possible inputs. Also, you may wish to place the nodes first, possibly using a matrix, and connect them afterwards. In this case, you can simply add rotations like rotate=90 to the shapes to rotate them. The following four keys make this slightly more convenient:

```
/tikz/point up (no value)
```

This is the same as rotate=90.

```
\tikz [circuit ee IEC] \node [diode,point up] {};
```

/tikz/point down (no value)

This is the same as rotate=-90.

```
\tikz [circuit ee IEC] \node [diode,point down] {};
```

/tikz/point left (no value)

This is the same as rotate=-180.

```
\tikz [circuit ee IEC] \node [diode,point left] {};
```

/tikz/point right (no value)

This key has no effect.

```
\tikz [circuit ee IEC] \node [diode,point right] {};
```

30.2.4 Info Labels

Info labels are used to add text to a circuit symbol. Unlike normal nodes like a rectangle, circuit symbols typically do not have text "on" them, but the text is placed next to them (like the text " 3Ω " next to a resistor).

TikZ already provides the label option for this purpose. The info option is built on top of this option, but it comes in some predefined variants that are especially useful in conjunction with circuits.

```
/\texttt{tikz/info} = [\langle options \rangle] \langle angle \rangle : \langle text \rangle  (no default)
```

This key has nearly the same effect as the label key, only the following style is used additionally automatically:

```
/tikz/every info (style, no value
```

Set this style to configure the styling of info labels. Since this key is *not* used with normal labels, it provides an easy way of changing the way info labels look without changing other labels.

The $\langle options \rangle$ and $\langle angle \rangle$ are passed directly to the label command.

```
\lambda \begin{tikzpicture}[circuit ee IEC,every info/.style=red] \node [resistor,info=$3\Omega$] \{\}; \end{tikzpicture}
```

You will find a detailed discussion of the label option on page 180.

Hint: To place some text on the main node, use center as the $\langle angle \rangle$:

```
\begin{tikzpicture}[circuit ee IEC,every info/.style=red] \node [resistor,info=center:$3\Omega$] {}; \node [resistor,point up,info=center:$R_1$] at (2,0) {}; \end{tikzpicture}
```

```
\texttt{/tikz/info'} = [\langle options \rangle] \langle angle \rangle : \langle text \rangle (no default)
```

This key works exactly like the info key, only in case the $\langle angle \rangle$ is missing, it defaults to below instead of the current value of label position, which is usually above. This means that when you use info, you get a label above the node, while when you use the info' key you get a label below the node. In case the node has been rotated, the positions of the info nodes are rotated accordingly.

```
4\Omega = R_2 \\ R_2 \\ R_1 \\ \begin{tikzpicture}[circuit ee IEC, every info/.style=red] \\ \draw (0,0) to[resistor={info={$3\\0mega$},info'={$R_1$}}] (3,0) \\ to[resistor={info={$4\\0mega$},info'={$R_2$}}] (3,2); \\ \end{tikzpicture}
```

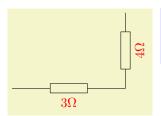
```
\label{likzinfo} $$ \text{loped=}[\langle options \rangle] \langle angle \rangle : \langle text \rangle $$ (no default)
```

This key works like info, only the transform shape option is set when the label is drawn, causing it to follow the sloping of the main node.

```
\begin{tikzpicture}[circu:\draw(0,0) to[resistor:\to[resistor:\to]]
\end{tikzpicture}
```

/tikz/info' sloped= (no default)

This is a combination of info' and info sloped.



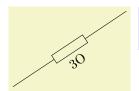
/tikz/circuit declare unit= $\{\langle name \rangle\}\{\langle unit \rangle\}$

(no default)

This key is used to declare keys that make it easy to attach physical units to nodes. The idea is that instead of info=\$3\Omega\$ you can write ohm=3 or instead of info'=\$5\mathrm{S}\$ you can write siemens'=5.

In detail, four keys are defined, namely $/\text{tikz}/\langle name \rangle$, $/\text{tikz}/\langle name \rangle$, $/\text{tikz}/\langle name \rangle$ sloped, and $/\text{tikz}/\langle name \rangle$ sloped. The arguments of all of these keys are of the form $[\langle options \rangle] \langle angle \rangle : \langle value \rangle$ and it is passed (slightly modified) to the corresponding key info, info', info sloped, or info' sloped. The "slight modification" is the following: The text that is passed to the, say, info key is not $\langle value \rangle$, but rather $\alpha \langle value \rangle \langle unit \rangle$ \$

This means that after you said circuit declare unit={ohm}{\Omega}, then ohm=5k will have the same effect as info={[every ohm]\$\mathrm{5k\Omega}\$}. Here, every ohm is a style that allows you to configure the appearance of this unit. Since the info key is used internally, by changing the every info style, you can change the appearance of all units infos.



```
\begin{tikzpicture}[circuit ee IEC,circuit declare unit={my ohm}{0}] \draw (0,0) to[resistor={my ohm' sloped=3}] (3,2); \end{tikzpicture}
```

30.2.5 Declaring and Using Annotations

Annotations are quite similar to info labels. The main difference is that they generally cause something to be drawn by default rather than some text to be added (although an annotation might also add some text). Annotations can be declared using the following key:

```
\text{tikz/circuit declare annotation} = \{\langle name \rangle\} \{\langle distance \rangle\} \{\langle path \rangle\} (no default)
```

This key is used to declare an annotation named $\langle name \rangle$. Once declared, it can be used as an argument of a symbol and will add the drawing in $\langle path \rangle$ to the symbol. In detail, the following happens:

The Main Keys. Two keys called $\langle name \rangle$ and $\langle name \rangle$, are defined. The second causes the annotation to be "mirrored and placed on the other side" of the symbol. Both of these keys may also take further keys as parameter like info keys. Whenever the $\langle name \rangle$ key is used, a local scope is opened and in this scope the following things are done:

- 1. The style every $\langle name \rangle$ is executed.
- 2. The following style is executed and then arrows=->:

```
/tikz/annotation arrow
```

(style, no value)

This style should set the > key to some desirable arrow tip.

- 3. The coordinate system is shifted such that the origin is at the north anchor of the symbol. (For the $\langle name \rangle$) key the coordinate system is flipped and shifted such that the origin is at the south anchor of the symbol.)
- 4. The label distance is locally set to $\langle distance \rangle$.
- 5. The parameter options given to the $\langle name \rangle$ key are executed.
- 6. The $\langle path \rangle$ is executed.

Usage. What all of the above amounts to is best explained by an example. Suppose we wish to create an annotation that looks like a little circular arrow (like \circlearrowleft). We could then say:

```
\tikzset{circuit declare annotation=
    {circular annotation}
    {9pt}
    {(0pt,8pt) arc (-270:80:3.5pt)}
}
```

We can then use it like this:

```
\tikz[circuit ee IEC] \draw (0,0) to [resistor={circular annotation}] (3,0);
```

Well, not very impressive since we do not see anything. This is due to the fact that the $\langle path \rangle$ becomes part of a path that contains the symbol node an nothing else. This path is not drawn or filled, so we do not see anything. What we must do is to use an edge path operation:

The $\langle distance \rangle$ is important for the correct placement of additional **info** labels. When an annotation is present, the info labels may need to be moved further away from the symbol, but not always. For this reason, an annotation defines an additional $\langle distance \rangle$ that is applied to all info labels given as parameters to the annotation. Here is an example, that shows the difference:

```
\tikz[circuit ee IEC]
\draw (0,0) to [resistor={circular annotation,ohm=5}] (2,0)
to [resistor={circular annotation=6hm=5}] (4,0);
```

30.2.6 Theming Symbols

For each symbol, a certain graphical representation is chosen to actually show the symbol. You can modify this graphical representation in several ways:

- You can select a different library and use a different circuit ... key. This will change all graphics used for the symbols.
- You can generally change the size of graphic symbols by setting circuit size unit to a different value or using a key like small circuit symbols.
- You can add options to the graphics used by symbols either globally by setting the every circuit symbol style or locally by setting the every $\langle name \rangle$ style, where $\langle name \rangle$ is the name of a symbol. For instance, in the following picture the symbols are ridiculously thick and resistors are red.

```
begin{tikzpicture}
    [circuit ee IEC,
        every circuit symbol/.style={ultra thick},
        every resistor/.style={red}]

    \draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
    \end{tikzpicture}
```

- You can selectively change the graphic used for a symbol by saying set resistor graphic=.
- You can change one or more of the following styles:

/tikz/circuit symbol open

(style, initially draw)

This style is used with symbols that consist of lines that surround some area. For instance, the IEC version of a resistor is an open symbol.

/tikz/circuit symbol filled

(style, initially draw,fill=black)

This style is used with symbols that are completely filled. For instance, the variant IEC version of an inductor is a filled, black rectangle.

/tikz/circuit symbol lines

(style, initially draw)

This style is used with symbols that consist only of lines that do not surround anything. Examples are a capacitor.

/tikz/circuit symbol wires

(style, initially draw)

This style is used for symbols that consist only of "wires." The difference to the previous style is that a symbol consisting of wires will look strange when the lines are thicker than the lines of normal wires, while for symbols consisting of lines (but not wires) it may look nice to make them thicker. An example is the make contact symbol.

Compare

All circuit environments like circuit logic IEC mainly use options like set and gate graphic=... to set up the graphics used for a certain symbol. It turns out that graphic hidden in the "..." part is also always available as a separate style, whose name contains the library's initials. For instance, the circuit logic IEC option actually contains the following command:

```
set and gate graphic = and gate IEC graphic,
```

The and gate IEC graphic style, in turn, is defined as follows:

```
\tikzset{and gate IEC graphic/.style=
    {
      circuit symbol open,
      circuit symbol size=width 2.5 height 4,
      shape=and gate IEC,
      inner sep=.5ex
    }
}
```

Normally, you do not need to worry about this, since you will not need to access a style like and gate IEC graphic directly; you will only use the and gate key. However, sometimes libraries define *variants* of a graphic; for instance, there are two variants for the resistor graphic in the IEC library. In this case you can set the graphic for the resistor to this variant (or back to the original) by saying set resistor graphic yourself:

30.3 Logical Circuits

30.3.1 Overview

A logical circuit is a circuit that contains what we call logical gates like an and gate or an xor gate. The logical libraries are intended to make it easy to draw such circuits.

In the following, we first have a look at the different libraries that can be used in principle and how the symbols look like. Then we have a more detailed look at how the symbols are used. Finally, we discuss the implementation details.

There are different ways of depicting logical gates, which is why there are different (sub-)libraries for drawing them. They provide the necessary graphical representations of the symbols declared in the following library:

```
\usetikzlibrary{circuits.logic} % MEX and plain EX \usetikzlibrary[circuits.logic] % ConExt
```

This library declares the logical gate symbols, but does not provide the symbol graphics. The library also defines the following key which, however, is also only used indirectly, namely by other libraries:

```
/tikz/circuit logic (no value)
```

This style calls the keys circuit (which internally calls every circuit, then it defines the inputs key and it calls the every circuit logic key.

```
/\text{tikz/inputs} = \langle inputs \rangle (no default)
```

This key is defined only inside the scope of a circuit logic. There, it has the same effect as logic gate inputs, described on page 342.

```
/tikz/every circuit logic (style, no value)
```

Use this key to configure the appearance of logical circuits.

Since the circuit.logic library does not define any actual graphics, you need to use one of the following libraries, instead:

```
\usepgflibrary{circuits.logic.IEC} % MEX and plain TeX and pure pgf \usepgflibrary[circuits.logic.IEC] % ConTeXt and pure pgf \usetikzlibrary{circuits.logic.IEC} % MEX and plain TeX when using TikZ \usetikzlibrary[circuits.logic.IEC] % ConTeXt when using TikZ
```

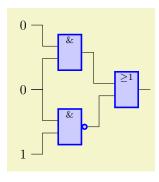
This library provides graphics based on gates recommended by the International Electrotechnical Commission. When you include this library, you can use the following key to set up a scope that contains a logical circuit where the gates are shown in this style.

```
/tikz/circuit logic IEC
```

(no value)

This key calls circuit logic and installs the IEC-like graphics for the logical symbols like and gate.

As explained in Section 30.2.6, for each graphic symbol of the library there is also a style that stores this particular appearance. These keys are called and gate IEC graphic, or gate IEC graphic, and so on.



```
\begin{tikzpicture}[circuit logic IEC,
                    every circuit symbol/.style={
                      logic gate IEC symbol color=black,
                      fill=blue!20,draw=blue,very thick}]
  \matrix[column sep=7mm]
    \node (i0) {0}; &
                    & \node [and gate] (a1) \{\}; & \\
    \node (i1) {0}; &
                                                 & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {}; & \\
    \node (i2) {1}; &
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
 \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
 \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

```
\usepgflibrary{circuits.logic.US} % MEX and plain TeX and pure pgf \usepgflibrary[circuits.logic.US] % ConTeXt and pure pgf \usetikzlibrary{circuits.logic.US} % MEX and plain TeX when using TikZ \usetikzlibrary[circuits.logic.US] % ConTeXt when using TikZ
```

This library provides graphics showing "American" logic gates. It defines the following key:

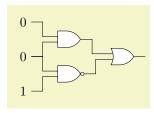
```
/tikz/circuit logic US
```

(no value)

This style calls circuit logic and installs US-like graphics for the logical symbols like and gate. For instance, it says

```
set and gate graphic = and gate US graphic
```

Here is an example:



```
\begin{tikzpicture}[circuit logic CDH,
                    tiny circuit symbols,
                    every circuit symbol/.style={
                      fill=white,draw}]
  \matrix[column sep=7mm]
    \node (i0) {0}; &
                    & \node [and gate] (a1) {}; & \\
    \node (i1) {0}; &
                                                 & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {}; & \\
    \node (i2) {1}; &
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
 \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
 \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
 \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
 \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

```
\usepgflibrary{circuits.logic.CDH} % MEX and plain TeX and pure pgf \usepgflibrary[circuits.logic.CDH] % ConTeXt and pure pgf \usetikzlibrary{circuits.logic.CDH} % MEX and plain TeX when using TikZ \usetikzlibrary[circuits.logic.CDH] % ConTeXt when using TikZ
```

This library provides graphics based on the logic symbols used in A. Croft, R. Davidson, and M. Hargreaves (1992), *Engineering Mathematics*, Addison-Wesley, 82–95. They are identical to the US-style symbols, except for the and- and nand-gates.

/tikz/circuit logic CDH

(no value)

This key calls circuit logic US and installs the two special and- and nand-gates, that is, it uses set and gate graphic with and gate CDH graphic and likewise for nand-gates.

Inside circuit logic XYZ scopes, you can now use the keys shown in Section 30.3.2. We have a more detailed look at one of them, all the other work the same way:

```
/tikz/and gate (no value)
```

This key should be passed to a node command. It will cause the node to "look like" an and gate, where the exact appearance of the gate is dictated by the which circuit environment is used. To further configure the appearance of the and gate, see Section 30.2.6.

```
\tikz [circuit logic IEC] \node [and gate] {$A$};

\tikz [circuit logic US]
{
    \node [and gate,point down] {$A$};
    \node [and gate,point down,info=center:$A$] at (1,0) {};
}
```

Inputs. Multiple inputs can be specified for a logic gate (provided they support multiple inputs: a not gate—also known as an inverter—does not). However, there is an upper limit for the number of inputs which has been set to 1024, which should be way more than would ever be needed.

The following key is used to configure the inputs. It is available only inside a circuit logic environment.

```
/tikz/inputs=⟨input list⟩
```

(no default, initially {normal, normal})

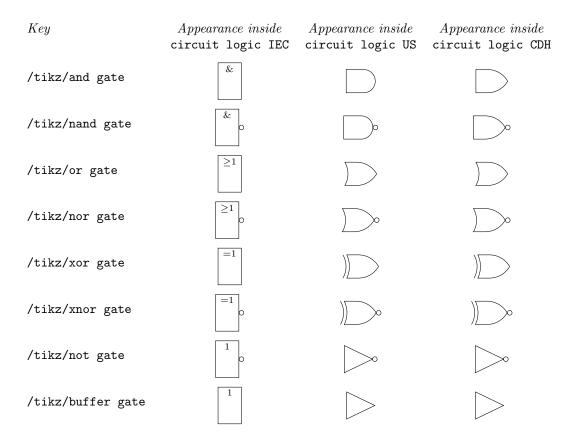
If a gate has n inputs, the $\langle input \ list \rangle$ should consist of n letters, each being i for "inverted" or n for "normal." Inverted gates will be indicated by a little circle. In any case the anchors for the inputs will be set up appropriately, numbered from top to bottom input 1, input 2, ... and so on. If the gate only supports one input the anchor is simply called input with no numerical index.

(This key is just a shorthand for logic gate inputs, described in detail on page 342. There you will also find descriptions of how to configure the size of the inverted circles and the way the symbol size increases when there are too many inputs.)

Output. Every logic gate has one anchor called output.

30.3.2 Symbols: The Gates

The following table shows which symbols are declared by the main circuits.logic library and their appearance in the different sublibraries.



30.3.3 Implementation: The Logic Gates Shape Library

The previous sections described the TikZ interface for creating logical circuits. In this section we take a closer look at the underlying PGF libraries.

Just as there are several TikZ circuit libraries, there are two underlying PGF shape libraries, one for creating US-style gates and one for IEC-style gates. These libraries define *shapes* only. It is the job of the circuit libraries to "theme" them so that they "look nice." However, in principle, you can also use these shapes directly.

Let us begin with the base library that defines the handling of inputs.

```
\usepgflibrary{shapes.gates.logic} % MEX and plain TeX and pure pgf \usepgflibrary[shapes.gates.logic] % ConTeXt and pure pgf \usetikzlibrary{shapes.gates.logic} % MEX and plain TeX when using TikZ \usetikzlibrary[shapes.gates.logic] % ConTeXt when using TikZ
```

This library defines common keys used by all logical gate shapes.

```
/pgf/logic gate inputs=\(\int input list\) (no default, initially \(\normal\), normal\)
```

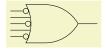
Specify the inputs for the logic gate. The keyword inverted indicates an inverted input which will mean PGF will draw a circle attached to the main shape of the logic gate. Any keyword that is not inverted will be treated as a "normal" or "non-inverted" input (however, for readability, you may wish to use normal or non-inverted), and PGF will not draw the circle. In both cases the anchors for the inputs will be set up appropriately, numbered from top to bottom input 1, input 2, ... and so on. If the gate only supports one input the anchor is simply called input with no numerical index.

```
\begin{tikzpicture}[minimum height=0.75cm]
\node[and gate IEC, draw, logic gate inputs={inverted, normal, inverted}]

(A) {};
\foreach \a in {1,...,3}
\draw (A.input \a -| -1,0) -- (A.input \a);
\draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

For multiple inputs it may be somewhat unwieldy to specify a long list, thus, the following "short-hand" is permitted (this is an extension of ideas due to Juergen Werber and Christoph Bartoschek):

Using i for inverted and n for normal inputs, $\langle input \ list \rangle$ can be specified without the commas. So, for example, ini is equivalent to inverted, normal, inverted.



```
\begin{tikzpicture}[minimum height=0.75cm]
\node[or gate US, draw,logic gate inputs=inini] (A) {};
\foreach \a in {1,...,5}
\draw (A.input \a -| -1,0) -- (A.input \a);
\draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

The height of the gate may be increased to accommodate the number of inputs. In fact, it depends on three variables: n, the number of inputs, r, the radius of the circle used to indicate an inverted input and s, the distance between the centers of the inputs. The default height is then calculated according to the expression $(n+1) \times \max(2r, s)$. This then may be increased to accommodate the node contents or any minimum size specifications.

The radius of the inverted input circle and the distance between the centers of the inputs can be customized using the following keys:

```
/pgf/logic gate inverted radius=(length) (no default, initially 2pt)
```

Set the radius of the circle that is used to indicate inverted inputs. This is also the radius of the circle used for the inverted output of the nand, nor, xnor and not gates.

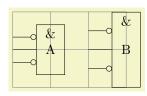


```
\begin{tikzpicture}[minimum height=0.75cm]
  \tikzset{every node/.style={shape=nand gate CDH, draw, logic gate inputs=ii}}
  \node[logic gate inverted radius=2pt] {A};
  \node[logic gate inverted radius=4pt] at (0,-1) {B};
  \end{tikzpicture}
```

/pgf/logic gate input sep= $\langle length \rangle$

(no default, initially .125cm)

Set the distance between the *centers* of the inputs to the logic gate.



PGF will increase the size of the logic gate to accommodate the number of inputs, and the size of the inverted radius and the separation between the inputs. However with all shapes in this library, any increase in size (including any minimum size requirements) will be applied so that the default aspect ratio is unaltered. This means that changing the height will change the width and vice versa.

30.3.4 Implementation: The US-Style Logic Gates Shape Library

```
\usepgflibrary{shapes.gates.logic.US} % MEX and plain TeX and pure pgf \usepgflibrary[shapes.gates.logic.US] % ConTeXt and pure pgf \usetikzlibrary{shapes.gates.logic.US} % MEX and plain TeX when using TikZ \usetikzlibrary[shapes.gates.logic.US] % ConTeXt when using TikZ
```

This library provides "American" logic gate shapes whose names are suffixed with the identifier US. Additionally, alternative and and nand gates are provided which are based on the logic symbols used in A. Croft, R. Davidson, and M. Hargreaves (1992), *Engineering Mathematics*, Addison-Wesley, 82–95. These two shapes are suffixed with CDH.

The "compass point" anchors apply to the main part of the shape and do not include any inverted inputs or outputs. This library provides an additional feature to facilitate the relative positioning of logic gates:

/pgf/logic gate anchors use bounding box=\langle boolean \rangle

(no default, initially false)

When set to true this key will ensure that the compass point anchors use the bounding rectangle of the main shape, which, ignore any inverted inputs or outputs, but includes any outer sep. This only affects the compass point anchors and is not set on a shape by shape basis: whether the bounding box is used is determined by value of this key when the anchor is accessed.



```
\begin{tikzpicture}[minimum height=1.5cm]
\node[xnor gate US, draw, gray!50,line width=2pt] (A) {};
\foreach \x/\y/\z in {false/blue/1pt, true/red/2pt}
\foreach \a in {north, south, east, west, north east,
    south east, north west, south west}
\draw[logic gate anchors use bounding box=\x, color=\y]
    (A.\a) circle(\z);
\end{tikzpicture}
```

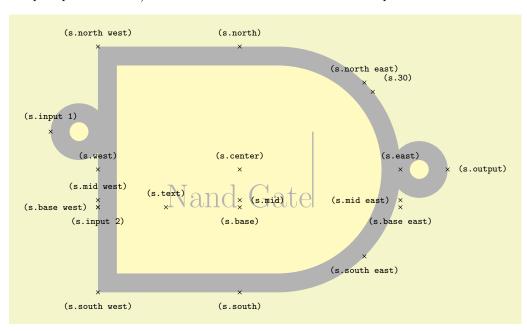
The library defines a number of shapes. For each shape the allowed number of inputs is also shown:

- and gate US, two or more inputs
- and gate CDH, two or more inputs
- nand gate US, two or more inputs
- nand gate CDH, two or more inputs
- or gate US, two or more inputs
- nor gate US, two or more Inputs
- xor gate US, two inputs
- xnor gate US, two inputs
- not gate US, one input
- buffer gate US, one input

In the following, we only have a detailed look at the anchors defined by one of them. We choose the nand gate US because it shows all the "interesting" anchors.

Shape nand gate US

This shape is a nand gate, which supports two or more inputs. If less than two inputs are specified an error will result. The anchors for this gate with two non-inverted inputs (using the normal compass point anchors) are shown below. Anchor 30 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
\node[name=s,shape=nand gate US,shape example, inner sep=0cm,
logic gate inputs={in},
logic gate inverted radius=.5cm] {Nand Gate\vrule width1pt height2cm};
\foreach \anchor/\placement in
    {center/above, text/above, 30/above right,
    mid/right, mid east/left, mid west/above,
    base/below, base east/below, base west/left,
    north/above, south/below, east/above, west/above,
    north east/above, south east/below, south west/below, north west/above,
    output/right, input 1/above, input 2/below}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

30.3.5 Implementation: The IEC-Style Logic Gates Shape Library

```
\usepgflibrary{shapes.gates.logic.IEC} % MEX and plain TeX and pure pgf \usepgflibrary[shapes.gates.logic.IEC] % ConTeXt and pure pgf \usetikzlibrary{shapes.gates.logic.IEC} % MEX and plain TeX when using TikZ \usetikzlibrary[shapes.gates.logic.IEC] % ConTeXt when using TikZ
```

This library provides rectangular logic gate shapes. These shapes are suffixed with IEC as they are based on gates recommended by the International Electrotechnical Commission.

By default each gate is drawn with a symbol, & for and and nand gates, ≥ 1 for or and nor gates, 1 for not and buffer gates, and = 1 for xor and xnor gates. These symbols are drawn automatically (internally they are drawn using the "foreground" path), and are not strictly speaking part of the node contents. However, the gate is enlarged to make sure the symbols are within the border of the node. It is possible to change the symbols and their position within the node using the following keys:

```
/pgf/and gate IEC symbol=\langle text \rangle
```

(no default, initially \char'\&)

Set the symbol for the and gate. Note that if the node is filled, this color will be used for the symbol, making it invisible, so it will be necessary set \(\lambda text\rangle\) to something like \color{black}\char'\&. Alternatively, the logic gate IEC symbol color key can be used to set the color of all symbols simultaneously.

In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by and gate symbol.

```
/pgf/nand gate IEC symbol=\langle text \rangle
```

(no default, initially \char'\&)

Set the symbol for the nand gate. In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by nand gate symbol.

```
/pgf/or gate IEC symbol=\langle text \rangle
```

(no default, initially \$\geq1\$)

Set the symbol for the or gate. In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by or gate symbol.

```
/pgf/nor gate IEC symbol=\langle text \rangle
```

(no default, initially \$\geq1\$)

Set the symbol for the nor gate. In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by nor gate symbol.

```
/pgf/xor gate IEC symbol=\langle text \rangle
```

(no default, initially $\{\$=1\$\}$)

Set the symbol for the xor gate. Note the necessity for braces, as the symbol contains =. In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by xor gate symbol.

```
/pgf/xnor gate IEC symbol=\langle text \rangle
```

(no default, initially {\$=1\$})

Set the symbol for the xnor gate. In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by xnor gate symbol.

```
/pgf/not gate IEC symbol=\langle text \rangle
```

(no default, initially 1)

Set the symbol for the not gate. In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by not gate symbol.

/pgf/buffer gate IEC symbol= $\langle text \rangle$

(no default, initially 1)

Set the symbol for the buffer gate. In TikZ, when the use IEC style logic gates key has been used, this key can be replaced by buffer gate symbol.

```
/pgf/logic gate IEC symbol align=\langle align \rangle
```

(no default, initially top)

Set the alignment of the logic gate symbol (in TikZ, when the use IEC style logic gates key has been used, IEC can be omitted. The specification in $\langle align \rangle$ is a comma separated list from top, bottom, left or right. The distance between the border of the node and the outer edge of the symbol is determined by the values of the inner xsep and inner ysep.

```
begin{tikzpicture}[minimum size=1cm, use IEC style logic gates]

\tikzset{every node/.style={nor gate, draw}}

\node (A) at (0,1.5) {};

\node [logic gate symbol align={bottom, right}] (B) at (0,0) {};

\foreach \g in {A, B}{

\foreach \i in {1,2}

\draw ([xshift=-0.5cm]\g.input \i) -- (\g.input \i);

\draw (\g.output) -- ([xshift=0.5cm]\g.output);

}

\end{tikzpicture}
```

/pgf/logic gate IEC symbol color= $\langle color \rangle$

(no default)

This key sets the color for all symbols simultaneously. This color can be overridden on a case by case basis by specifying a color when setting the symbol text.

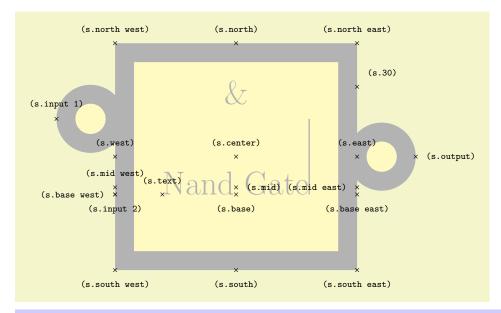
The library defines the following shapes:

- and gate IEC, two or more inputs
- nand gate IEC, two or more inputs
- or gate IEC, two or more inputs
- nor gate IEC, two or more inputs
- xor gate IEC, two inputs
- xnor gate IEC, two inputs
- not gate IEC, one input
- buffer gate IEC, one input

Again, we only have a look at the nand-gate in more detail:

Shape nand gate IEC

This shape is a nand gate. It supports two or more inputs. If less than two inputs are specified an error will result. The anchors for this gate with two inverted inputs are shown below. Anchor 30 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
 \node[name=s,shape=nand gate IEC ,shape example, inner xsep=1cm, inner ysep=1cm,
   minimum height=6cm, nand gate IEC symbol=\color{black!30}\char'\&,
   logic gate inputs={in},
   logic gate inverted radius=0.65cm]
  {Nand Gate\vrule width1pt height2cm};
 \foreach \anchor/\placement in
    {center/above, text/above, 30/above right,
    mid/right, mid east/left, mid west/above,
    base/below, base east/below, base west/left,
    north/above, south/below, east/above, west/above,
    north east/above, south east/below, south west/below, north west/above,
    output/right, input 1/above, input 2/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

30.4 Electrical Engineering Circuits

30.4.1 Overview

An electrical engineering circuit contains symbols like resistors or capacitors or voltage sources and annotations like the two arrows pointing toward an element whose behaviour is light dependent. The electrical engineering libraries, abbreviated ee-libraries, provide such symbols and annotations.

Just as for logical gates, there are different ways of drawing ee-symbols. Currently, there is one main library for drawing circuits, which uses the graphics from the International Electrotechnical Commission, but you can add your own libs. This is why, just as for logical gates, there is a base library and more specific libraries.

```
\usetikzlibrary{circuits.ee} % MFX and plain TFX \usetikzlibrary[circuits.ee] % ConTFXt
```

This library declares the ee symbols, but (mostly) does not provide the symbol graphics, which is left to the sublibraries. Just like the logical gates library, a key is defined that is normally only used internally:

```
/tikz/circuit ee (no value)
```

This style calls the keys circuit (which internally calls every circuit and the following style:

```
/tikz/every circuit ee (style, no value)
```

Use this key to configure the appearance of logical circuits.

The library also declares some standard annotations and units.

As for logical circuits, to draw a circuit the first step is to include a library containing the symbols graphics. Currently, you have to include circuits.ee.IEC.

```
\usetikzlibrary{circuit.ee.IEC} % MTEX and plain TEX \usetikzlibrary[circuit.ee.IEC] % ConTEXt
```

When this library is loaded, you can use the following style:

```
/tikz/circuit ee IEC (no value)
```

This style calls circuit ee and installs the IEC-like graphics for the logical symbols like resistor.

Inside the circuit ee IEC scope, you can now use the keys for symbols, units, and annotations listed in the later sections. We have a more detailed look at one of each of them, all the others work the same way.

Let us start with an example of a symbol: the resistor symbol. The other predefined symbols are listed in Section 30.4.2 and later sections.

```
/\text{tikz/resistor} = \langle options \rangle (no default)
```

This key should be used with a node path command or with the to path command.

Using the Key with Normal Nodes. When used with a node, it will cause this node to "look like" a resistor (by default, in the IEC library, this is just a simple rectangle).

```
\tikz [circuit ee IEC]
\node [resistor] {};
```

Unlike normal nodes, a resistor node generally should not take any text (as in node [resistor] {foo}). Instead, the labeling of resistors should be done using the label, info and ohm options.

```
5Ω \tikz [circuit ee IEC] \node [resistor,ohm=5] {};
```

The $\langle options \rangle$ make no real sense when the **resistor** option is used with a normal node, you can just as well given them to the **node** itself. Thus, the following has the same effect as the above example:

In a circuit, you will often wish to rotate elements. For this, the options point up, point down, point left or point right may be especially useful. They are just shorthands for appropriate rotations like rotate=90.

Using the Key on a To Path. When the resistor key is used on a to path inside a circuit ee IEC, the circuit handle symbol key is called internally. This has a whole bunch of effects:

- 1. The path currently being constructed is cut up to make place for a node.
- 2. This node will be a resistor node that is rotated so that it points "along" the path (unless an option like shift only or an extra rotation is used to change this).
- 3. The $\langle options \rangle$ passed to the resistor key are passed on to the node.
- 4. The $\langle options \rangle$ are pre-parsed to identify a pos key or a key like at start or midway. These keys are used to determine where on the to path the node will lie.

Since the $\langle options \rangle$ of the resistor key are passed on to the resistor node on the path, you can use it to add labels to the node. Here is a simple example:

```
2\mu\Omega \label{eq:local_property} $$ \begin{array}{c} \text{\circuit ee IEC]}\\ \text{\draw (0,0) to [resistor=red]} & (3,0)\\ \text{\to [resistor={ohm=2\mu}]} & (3,2); \end{array} $$
```

You can add multiple labels to a resistor and you can have multiple resistors (or other elements) on a single path.

Inputs, Outputs, and Anchors. Like the logical gates, all ee-symbols have an input and an output anchor. Special-purpose-nodes may have even more anchors of this type. Furthermore, the ee-symbols-nodes also have four standard compass direction anchors.

Changing the Appearance. To configure the appearance of all resistors, see Section 30.2.6. You can use the $\langle options \rangle$ to locally change the appearance of a single resistor.

Let us now have a look at an example of a unit: the Ohm unit. The other predefined units are listed in Section 30.4.6.

 $/\text{tikz/ohm} = \langle value \rangle$ (no default

This key is used to add an info label to a node with a special text: $\mbox{mathrm}{\langle value \rangle \Omega ega}$. In other words, the ohm key can only be used with the options of a node and, when used, it will cause the $\langle value \rangle$ to be placed next to the node, followed by Ω . Since the $\langle value \rangle$ is typeset inside a \mathrm command, when you write ohm=5k you get $5k\Omega$, ohm=5p yields $5p\Omega$, and ohm=5.6\cdot 10^{2}\mu yields $5.6 \cdot 10^{2} \mu\Omega$.

Instead of ohm you can also use ohm', which places the label on the other side.

Finally, there are also keys ohm sloped and ohm' sloped for having the info label rotate together with the main node.

```
\tikz [circuit ee IEC]
\draw (0,0) to [resistor={ohm sloped=5M}] (0,2)
(2,0) to [resistor={ohm' sloped=6f}] (2,2);
```

You can configure the appearance of an Ohm info label using the key every ohm.

Finally, let us have a look at an annotation: the light emitting annotation. The other predefined units are listed in Section 30.4.7.

```
/\text{tikz/light emitting=}\langle options \rangle (no default)
```

Like a unit, an annotation should be given as an additional option to a node. It causes some drawings (in this case, two parallel lines) to be placed next to the node.

```
\tikz [circuit ee IEC] \draw (0,0) to [diode=light emitting] (2,0);
```

The $\langle options \rangle$ can be used for three different things:

- 1. You can use keys like red to change the appearance of this annotation, locally.
- 2. You can use keys like <- or -latex to change the direction and kinds of arrows used in the annotation.

3. You can use info labels like ohm=5 or info=foo inside the \(\lambda options \rangle \). These info labels will be added to the main node (not to the annotation itself), but the label distance will have been changed to accommodate for the space taken up by the annotation.

In addition to light emitting there is also a key called light emitting', which simply places the annotation on the other side of the node.

You can configure the appearance of annotations in three ways:

- You can set the every circuit annotation style.
- You can set the every light emitting style.
- You can set the following key:

/tikz/annotation arrow

(style, no value)

This style should set the default > arrow to some nice value.

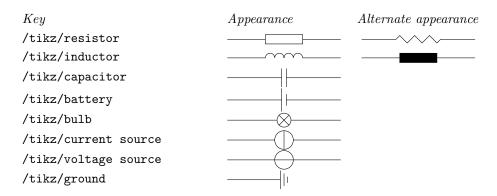
30.4.2 Symbols: Indicating Current Directions

There are two symbols for indicating current directions. These symbols are defined directly inside circuit ee.

The examples have been produced by (in essence) $\langle draw (0,0) to [\langle symbol name \rangle] (3,0)$;

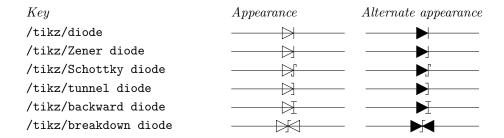
30.4.3 Symbols: Basic Elements

The following table show basic symbols as they are depicted inside the circuit ee IEC environment. To install one of alternate graphics, you have to say set $\langle symbol\ name \rangle$ graphic=var $\langle symbol\ name \rangle$ IEC graphic.



30.4.4 Symbols: Diodes

The following table shows diodes as they are depicted inside the circuit ee IEC environment.



30.4.5 Symbols: Contacts

The following table shows contacts as they are depicted inside the circuit ee IEC environment.

Key	Appearance	$Alternate\ appearance$
/tikz/contact	•	
/tikz/make contact	/	
/tikz/break contact		

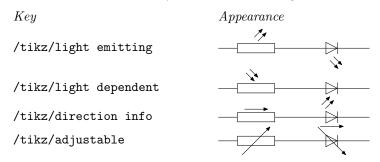
30.4.6 Units

The circuit.ee library predefines the following unit keys:

Key	Appearance of 1 unit
/tikz/ampere	1A
/tikz/volt	1V
/tikz/ohm	1Ω
/tikz/siemens	1S
/tikz/henry	1H
/tikz/farad	$1\mathrm{F}$
/tikz/coulomb	1C
/tikz/voltampere	1VA
/tikz/watt	$1 \mathrm{W}$
/tikz/hertz	$1 \mathrm{Hz}$

30.4.7 Annotations

The circuit.ee.IEC library defines the following annotations:



The lines have been produced using, in essence,

```
\draw (0,0) to [resistor=light emitting] (2,0) to [diode=light emitting'] (4,0);
```

and similarly for the other annotations.

30.4.8 Implementation: The EE-Symbols Shape Library

The TikZ libraries depend on two shape libraries, which are included automatically. Usually, you will not need to use these shapes directly.

```
\usepgflibrary{shapes.gates.ee} % MEX and plain TEX and pure pgf
\usepgflibrary[shapes.gates.ee] % ConTEXt and pure pgf
\usetikzlibrary{shapes.gates.ee} % MEX and plain TEX when using TikZ
```

\usetikzlibrary[shapes.gates.ee] % ConTeXt when using TikZ

This library defines basic shapes that can be used by all ee-circuit libraries. Currently, it defines the following shapes:

- rectangle ee
- circle ee
- direction ee

Additionally, the library defines the following arrow tip: The direction ee arrow tip is basically the same as a triangle 45 arrow tip with rounded joins.

```
direction ee yields thick ←→ and thin ←→
```

However, unlike normal arrow tips, its size does *not* depend on the current line width. Rather, it depends on the value of its arrow options, which should be set to the desired size. Thus, you should say something like \pgfsetarrowoptions{direction ee}{5pt} to set the size of the arrow.

Shape rectangle ee

This shape is completely identical to a normal rectangle, only there are two additional anchors: The input anchor is an alias for the west anchor, while the output anchor is an alias for the east anchor.

Shape circle ee

Like the rectangle ee shape, only for circles.

Shape direction ee

This shape is rather special. It is intended to be used to "turn an arrow tip into a shape." First, you should set the following key to the name of an arrow tip:

```
/pgf/direction ee arrow=(right arrow tip name)
```

(no default)

The value of this key will be used for the arrow tip depicted in an direction ee shape.

When a node of shape direction ee is created, several things happen:

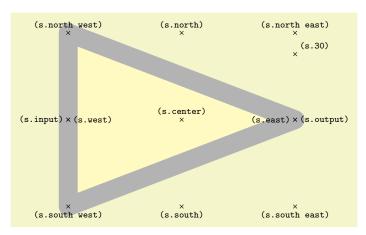
1. The size of the shape is computed according to the following rules: The width of the shape is set up so that the left border of the shape is at the left end of the arrow tip and the right border is at the right end of the arrow tip. These left and right "ends" of the arrow are the left and right extents specified by the arrow itself (see the documentation of the \pgfarrowsdeclare for details). You usually need not worry about this width setting.

By comparison, the height of the arrow is given by the current setting of minimum height. Thus, this key must have been set up correctly to reflect the "real" height of the arrow tip. The reason is that the height of an arrow is not specified when arrows are declared and is, thus, not available, here.

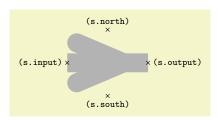
Possibly, the height computation will change in the future to reflect the real height of the arrow, so you should generally set up the minimum height to be the same as the real height.

- 2. A straight line from left to right inside the shape's boundaries is added to the background path.
- 3. The arrow tip, pointing right, is drawn before the background path.

The anchors of this shape are just the compass anchors, which lie on a rectangle whose width and height are the above-computed height and width.



```
\begin{tikzpicture}
  \pgfsetarrowoptions{direction ee}{6cm}
  \node[name=s,shape=direction ee,shape example,minimum height=0.7654*6cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
    north/above, south/below, east/left, west/right,
    north east/above, south east/below, south west/below, north west/above,
    input/left,output/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
  \end{tikzpicture}
```



30.4.9 Implementation: The IEC-Style EE-Symbols Shape Library

```
\usepgflibrary{shapes.gates.ee.IEC} % MEX and plain TeX and pure pgf \usepgflibrary[shapes.gates.ee.IEC] % ConTeXt and pure pgf \usetikzlibrary{shapes.gates.ee.IEC} % MEX and plain TeX when using TikZ \usetikzlibrary[shapes.gates.ee.IEC] % ConTeXt when using TikZ
```

This library defines shapes for depicting ee symbols according to the IEC recommendations. These shapes will typically be used in conjunction with the graphic mechanism detailed earlier, but you can also used them directly.

Shape generic circle IEC

This shape inherits from circle ee, which in turn is just a normal circle with additional input and output anchors at the left and right ends. However, additionally, this shape allows you to specify a path that should be added before the background path using the following key:

```
/pgf/generic circle IEC/before background=\langle code \rangle (no default)
```

When a node of shape generic circle IEC is created, the current setting of this key is used as the "before background path." This means that after the circle's background has been drawn/filled/whatever, the $\langle code \rangle$ is executed.

When the $\langle code \rangle$ is executed, the coordinate system will have been transformed in such a way that

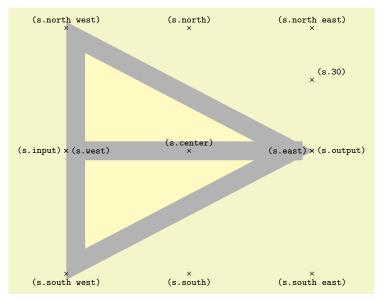
the point (1pt, 0pt) lies at the right end of the circle and (0pt, 1pt) lies at the top of the circle. (More precisely, these points will lie exactly on the middle of the radial line.)

Here is an examples of how to use this shape:



Shape generic diode IEC

This shape is used to depict diodes. The main shape is taken up by a "right pointing" triangle. The anchors are positioned on the border of a rectangle around the diode, see the below example. The diode's size is based on the current settings of minimum width and minimum height.



```
\begin{tikzpicture}
  \node[name=s,shape=generic diode IEC,shape example,minimum size=6cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
    north/above, south/below, east/left, west/right,
    north east/above, south east/below, south west/below, north west/above,
    input/left,output/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
  \end{tikzpicture}
```

This shape, like the generic circle IEC shape, is generic in the sense that there is a special key that is used for the before background drawings:

```
/pgf/generic diode IEC/before background=\langle code \rangle (no default)
```

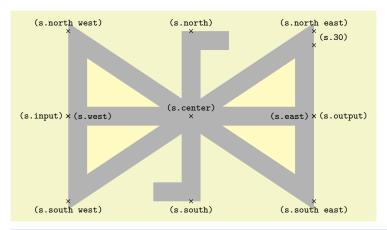
Similarly to the generic circle IEC shape, when a node of shape generic diode IEC is created, the current setting of this key is used as the "before background path." When the $\langle code \rangle$ is executed, the coordinate system will have been transformed in such a way that the origin is at the "tip" of the diode's triangle, the point (0pt, 1pt) is exactly half the diode's height above this origin, and the point (1pt, 0pt) is half the diode's height to the right of the origin.

The idea is that you use this key to draw different kinds of diode endings.



Shape breakdown diode IEC

This shape is used to depict a bidirectional breakdown diode. The diode's size is based on the current settings of minimum width and minimum height.

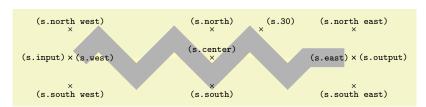


```
\begin{tikzpicture}
\node[name=s,shape=breakdown diode IEC,shape example,minimum width=6cm,minimum height=4cm] {};
\foreach \anchor/\placement in
    {center/above, 30/above right,
    north/above, south/below, east/left, west/right,
    north east/above, south east/below, south west/below, north west/above,
    input/left,output/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape var resistor IEC

This shape is used to depict a variant version of a resistor. Its size is computed as for a rectangle (thus, its size depends things like the minimum height). Then, inside this rectangle, a background path is set up according to the following rule: Starting from the left end, zigzag segments are added to the path. Each segment consists of a line at a 45 degree angle going up to the top of the rectangle, then going down to the bottom, then going up to mid height of the node. As many segments as possible are put inside as possible. The last segment is then connected to the output anchor via a straight line.

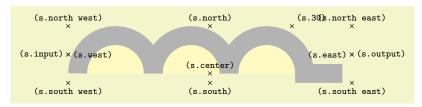
All of this means that, in general, the shape should be much wider than high.



```
\begin{tikzpicture}
  \node[name=s,shape=var resistor IEC,shape example,minimum width=7cm,minimum height=1cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
    north/above, south/below, east/left, west/right,
    north east/above, south east/below, south west/below, north west/above,
    input/left,output/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
  \end{tikzpicture}
```

Shape inductor IEC

This shape is used to depict an inductor, using a bumpy line. Its size is computed as follows: Any text and inner sep are ignored (and should normally not be given). The minimum height plus (twice) the outer ysep specify the distance between the north and south anchors, similarly for the minimum width plus the outer xsep for the east and west. The bumpy line is drawn starting from the lower left corner to the lower right corner with bumps being half-circles whose height is exactly the minimum height. The center of the shape is just above the south anchor, at a distance of the outer ysep.

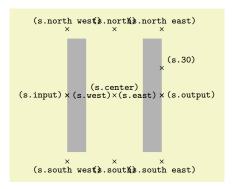


```
\begin{tikzpicture}
  \node[name=s,shape=inductor IEC,shape example,minimum width=7cm,minimum height=1cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
    north/above, south/below, east/left, west/right,
    north east/above, south east/below, south west/below, north west/above,
    input/left,output/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
  \end{tikzpicture}
```

Just as for a var resistor IEC, as many bumps as possible are added and the last bump is connected to the output anchor via a straight line.

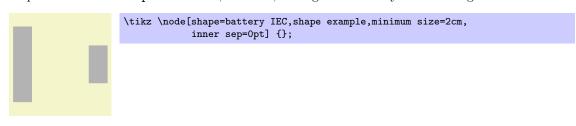
Shape capacitor IEC

This shape is based on a rectangle ee. However, instead of a rectangle as the background path, only the "left and right lines" that make up the rectangle are drawn.



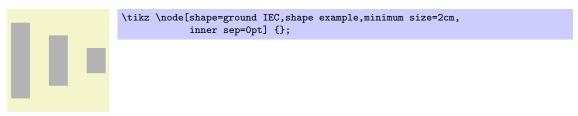
Shape battery IEC

This shape is similar to a capacitor IEC, however, the right line is only half the height of the left line.



Shape ground IEC

This shape is similar to a batter IEC, only three lines of different heights are drawn.



Shape make contact IEC

This shape consists of a line going from the lower left corner to the upper right corner. The size and anchors of this shape are computed in the same way as for an inductor IEC.

```
\begin{tikzpicture}
  \node[name=s,shape=make contact IEC,shape example,minimum width=3cm,minimum height=1cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
    north/above, south/below, east/left, west/right,
    north east/above, south east/below, south west/below, north west/above,
    input/left,output/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
  \end{tikzpicture}
```

Shape var make contact IEC

This shape works like make contact IEC, only a little circle is added to the path at the lower left corner. The radius of this circle is one twelfth of the width of the node.



Shape break contact IEC

This shape depicts a contact that can be broken. It works like make contact IEC.



31 Decoration Library

31.1 Overview and Common Options

The decoration libraries define a number of (more or less useful) decorations that can be applied to paths. The usage of decorations is not covered in the present section, please consult Sections 22, which explains how decorations are used in TikZ, and 80, which explains how new decorations can be defined.

The decorations are influenced by a number of parameters that can be set using the **decoration** option. These parameters are typically shared between different decorations. In the following, the general options are documented (they are defined directly in the **decoration** module), special-purpose keys are documented with the decoration that uses it.

Since you are encouraged to use these keys to make your own decorations configurable, it is indicated for each key where the value is stored (so that you can access it). Note that some values are stored in T_EX dimension registers while others are stored in macros.

$/pgf/decoration/amplitude=\langle dimension \rangle$

(no default, initially 2.5pt)

This key determines the "desired height" (or amplitude) of decorations for which this makes sense. For instance, the initial value of 2.5pt means that deforming decorations should deform a path by up to 2.5pt away from the original path.

This key sets the TFX-dimension \pgfdecorationsegmentamplitude.

/pgf/decoration/meta-amplitude=\langle dimension \rangle

(no default, initially 2.5pt)

This key determines the amplitude for a meta-decoration.

The key sets the TEX-macro (!) \pgfmetadecorationsegmentamplitude.

$/pgf/decoration/segment length=\langle dimension \rangle$

(no default, initially 10pt)

Many decorations are made up of small segments. This key determines the desired length of such segments.

This key sets the TFX-dimension \pgfdecorationsegmentlength.

$/pgf/decoration/meta-segment length=\langle dimension \rangle$

(no default, initially 1cm)

This determined the length of the meta-segments from which a meta-decoration is made up.

This key sets the T_EX-macro (!) \pgfmetadecorationsegmentlength.

/pgf/decoration/angle= $\langle degree \rangle$

(no default, initially 45)

The way some decorations look like depends on a configurable angle. For instance, a wave decoration consists of arcs and the opening angle of these arcs is given by the angle.

This key sets the TpX-macro \pgfdecorationsegmentangle.

$/pgf/decoration/aspect=\langle factor \rangle$

(no default, initially 0.5)

For some decorations there is a natural aspect ratio. For instance, for a brace decoration the aspect ratio determines where the brace point will be.

This key sets the TFX-macro \pgfdecorationsegmentaspect.

/pgf/decoration/start radius=\langle dimension \rangle

(no default, initially 2.5pt)

For some decorations there is a natural start radius (of some circle, presumably).

This key stores the value directly inside the key.

$/pgf/decoration/end radius=\langle dimension \rangle$

(no default, initially 2.5pt)

For some decorations there is a natural end radius (of some circle, presumably).

This key stores the value directly inside the key.

$/pgf/decoration/radius=\langle dimension \rangle$

(style, no default)

Sets the start and end radius simultaneously.

/pgf/decoration/path has corners= $\langle boolean \rangle$

(no default, initially false)

This is a hint to the decoration code as to whether the path has corners or not. If a path has a sharp corner, setting this option to true may result in better rendering of the decoration because the joins of input segments are approached "more carefully" than when this key is set to false. However, if the path is, say, a smooth circle, setting this key to true will usually look worse. Most decorations ignore this key, anyway. Internally, it sets the TeX-if \ifpgfdecoratepathhascorners.

31.2 Path Morphing Decorations

```
\usepgflibrary{\decorations.pathmorphing} \( \mathbb{M}\mathbb{E}\) and plain \( \mathbb{T}\mathbb{E}\) and pure pgf \usetikzlibrary{\decorations.pathmorphing} \( \mathbb{M}\mathbb{E}\) and plain \( \mathbb{T}\mathbb{E}\) when using \( \mathbb{T}ikZ \) \usetikzlibrary{\decorations.pathmorphing} \( \mathbb{C}\) \( \mathbb{T}\mathbb{E}\) when using \( \mathbb{T}ikZ \)
```

A path morphing decoration "morphs" or "deforms" the to-be-decorated path. This means that what used to be a straight line might afterwards be a snaking curve and have bumps. However, a line is still a line and path deforming decorations do not change the number of subpaths. For instance, if the path used to consist of two circles and an open arc, the path will, after the decoration process, still consist of two closed subpaths and one open subpath.

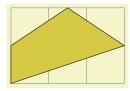
31.2.1 Decorations Producing Straight Line Paths

The following deformations use only straight lines in order to morph the paths.

Decoration lineto

This decoration replaces the path by straight lines. For each curve, the path simply goes directly from the start point to the end point. In the following example, the arc actually consist of two subcurves.

This decoration is actually always defined when the decoration module is loaded, but it is documented here for consistency.

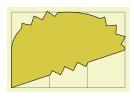


```
\begin{tikzpicture}[decoration=lineto]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
  \end{tikzpicture}
```

Decoration straight zigzag

This (meta-)decoration decorates the path by alternating between curveto and zigzag decorations. It always finishes with the curveto decoration. The following parameters influence the decoration:

- amplitude determines how much the zigzag line raises above and falls below a straight line to the target point.
- segment length determines the length of a complete "up-down" cycle.
- meta-segment length determines the length of the curveto and the zigzag decorations.

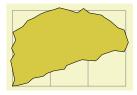


```
\begin{tikzpicture}[decoration={straight zigzag,meta-segment length=1.1cm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
  (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
  \end{tikzpicture}
```

Decoration random steps

This decoration consists of straight line segments. The line segments head towards the target, but each step is randomly shifted a little bit. The following parameters influence the decorations:

- segment length determines the basic length of each step.
- amplitude The end of each step is perturbed both in x- and in y-direction by two values drawn uniformly from the interval [-d, d], where d is the value of amplitude.

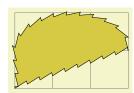


```
\begin{tikzpicture}
    [decoration={random steps,segment length=2mm}]
\draw [help lines] grid (3,2);
\draw [decorate,fill=examplefill]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration saw

This decoration looks like the blade of a saw. The following parameters influence the decoration:

- amplitude determines how much each spike raises above the straight line.
- segment length determines the length each spike.

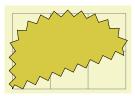


```
\begin{tikzpicture}[decoration=saw]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
  (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
  \end{tikzpicture}
```

Decoration zigzag

This decoration looks like a zigzag line. The following parameters influence the decoration:

- amplitude determines how much the zigzag line raises above and falls below a straight line to the target point.
- segment length determines the length of a complete "up-down" cycle.



```
\begin{tikzpicture}[decoration=zigzag]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
  \end{tikzpicture}
```

31.2.2 Decorations Producing Curved Line Paths

Decoration bent

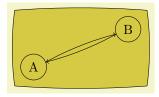
This decoration adds a slightly bent line from the start to the target. The amplitude of the bend is given amplitude (an amplitude of zero gives a straight line).

- amplitude determines the amplitude of the bend.
- aspect determines how tight the bend is. A good value is around 0.3.

Note that this decoration makes only little sense for curves. You should apply it only to straight lines.



```
\begin{tikzpicture}[decoration=bent]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) -- (1.5,2) -- (0,1);
  \end{tikzpicture}
```

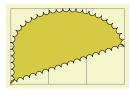


```
\begin{tikzpicture}[decoration={bent,aspect=.3}]
  \draw [decorate,fill=examplefill] (0,0) rectangle (3.5,2);
  \node[circle,draw] (A) at (.5,.5) {A};
  \node[circle,draw] (B) at (3,1.5) {B};
  \draw[->,decorate] (A) -- (B);
  \draw[->,decorate] (B) -- (A);
  \end{tikzpicture}
```

Decoration bumps

This decoration replaces the path by little half ellipses. The following parameters influence it.

- amplitude determines the height of the half ellipse.
- segment length determines the width of the half ellipse.

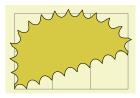


```
\begin{tikzpicture}[decoration=bumps]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
  (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

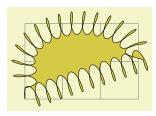
Decoration coil

This decoration replaces the path by a coiled line. To understand how this works, imagine a threedimensional spring. The spring's axis points along the path toward the target. Then, we "view" the spring from a certain angle. If we look "straight from the side" we will see a perfect sine curve, if we look "more from the front" we will see a coil. The following parameters influence the decoration:

- amplitude determines how much the coil rises above the path and falls below it. Thus, this is the radius of the coil.
- segment length determines the distance between two consecutive "curls." Thus, when the spring is see "from the side" this will be the wave length of the sine curve.
- aspect determines the "viewing direction." A value of 0 means "looking from the side" and a value of 0.5, which is the default, means "look more from the front."



```
\begin{tikzpicture}[decoration=coil]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
  (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```



```
\begin{tikzpicture}
    [decoration={coil,aspect=0.3,segment length=3mm,amplitude=3mm}]
    \draw [help lines] grid (3,2);
    \draw [decorate,fill=examplefill]
     (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
    \end{tikzpicture}
```

Decoration curveto

This decoration simply yields a line following the original path. This means that (ideally) it does not change the path and follows any curves in the path (hence the name). In reality, due to the internals of how decorations are implemented, this decoration actually replaces the path by numerous small straight lines.

This decoration is mostly useful in conjunction with meta-decorations. It is also actually defined in the decoration module and is always available.

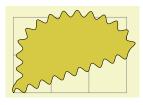


```
\begin{tikzpicture}[decoration=curveto]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration snake

This decoration replaces the path by a line that looks like a snake seen from above. More precisely, the snake is a sine wave with a "softened" start and ending. The following parameters influence the snake:

- amplitude determines the sine wave's amplitude.
- segment length determines the sine wave's wavelength.



```
\begin{tikzpicture}[decoration=snake]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

31.3 Path Replacing Decorations

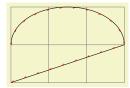
```
\label{thm:continuous_pathreplacing} $$ \mbox{MEX}$ and plain TeX and pure pgf $$ \usepgflibrary[decorations.pathreplacing] $$ \ConTeXt$ and pure pgf $$ \usetikzlibrary[decorations.pathreplacing] $$ \mbox{MEX}$ and plain TeX when using TikZ $$ \usetikzlibrary[decorations.pathreplacing] $$ \ConTeXt$ when using TikZ $$
```

This library defines decorations that replace the to-be-decorated path by another path. Unlike morphing decorations, the replaced path might be quite different, for instance a straight line might be replaced by a set of circles. Note that filling a path that has been replaced using one of the decorations in this library typically does not fill the original area but, rather, the smaller area of the newly-created path segments.

Decoration border

This decoration adds straight lines to the path that are at a specific angle to the line toward the target. The idea is to add these little lines to indicate the "border" of an area. The following parameters influence the decoration:

- segment length determines the distance between consecutive ticks.
- amplitude determines the length of the ticks.
- angle determines the angle between the ticks and the line of the path.

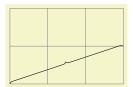


```
\begin{tikzpicture}[decoration=border]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate,draw,red}]
      (0,0) -- (3,1) arc (0:180:1.5 and 1);
  \end{tikzpicture}
```

Decoration brace

This decoration replaces a straight line path by a long brace. The left and right end of the brace will be exactly on the start and endpoint of the decoration. The decoration really only makes sense for paths that are a straight line.

- amplitude determines how much the brace rises above the path.
- aspect determines the fraction of the total length where the "middle part" of the brace will be.



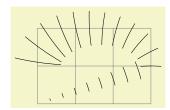
```
\begin{tikzpicture}[decoration=brace]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1);
  \end{tikzpicture}
```

Decoration expanding waves

This decoration adds arcs to the path that get bigger along the line towards the target. The following parameters influence the decoration:

• segment length determines the distance between consecutive arcs.

• angle determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.



```
\begin{tikzpicture}[decoration={expanding waves,angle=5}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
  \end{tikzpicture}
```

Decoration moveto

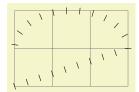
This decoration simply jumps to the end of the path using a move-to path operation. It is mainly useful as pre=moveto or post=moveto decorations.

This decoration is actually always defined when the decoration module is loaded, but it is documented here for consistency.

Decoration ticks

This decoration replaces the path by straight lines that are orthogonal to the path. The following parameters influence the decoration:

- segment length determines the distance between consecutive ticks.
- amplitude determines half the length of the ticks.

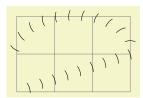


```
\begin{tikzpicture}[decoration=ticks]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
  \end{tikzpicture}
```

Decoration waves

This decoration replaces the path by arcs that have a constant size. The following parameters influence the decoration:

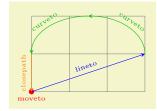
- segment length determines the distance between consecutive arcs.
- angle determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.
- radius determines the radius of each arc.



```
\begin{tikzpicture}[decoration={waves,radius=4mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration show path construction

This decoration allows "something different" to be done for each *type* of input segment (i.e., moveto, lineto, curveto or closepath). Typically, each segment will be replaced with another path, but this need not necessarily be the case.



```
\begin{tikzpicture}[>=stealth, every node/.style={midway, sloped, font=\tiny},
 decoration={show path construction,
   moveto code={
      \fill [red] (\tikzinputsegmentfirst) circle (2pt)
       node [fill=none, below] {moveto};},
    lineto code={
      \draw [blue,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
       node [above] {lineto};
   curveto code={
      \draw [green!75!black,->] (\tikzinputsegmentfirst) .. controls
        (\tikzinputsegmentsupporta) and (\tikzinputsegmentsupportb)
        ..(\tikzinputsegmentlast) node [above] {curveto};
    closepath code={
      \draw [orange,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
        node [above] {closepath};}
  \draw [help lines] grid (3,2);
  \path [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

The following keys can be used to specify the code to execute for each type of input segment.

/pgf/decoration/moveto code= $\langle code \rangle$

(no default, initially {})

Set the code to be executed for every move to input segment. It is important to remember that the transformations applied by the decoration automaton are turned off when $\langle code \rangle$ is executed.

/pgf/decoration/lineto code= $\langle code \rangle$

(no default, initially {})

Set the code to be executed for every line to input segment.

/pgf/decoration/curveto code= $\langle code \rangle$

(no default, initially {})

Set the code to be executed for every curveto input segment.

/pgf/decoration/closepath code= $\langle code \rangle$

(no default, initially {})

Set the code to be executed for every closepath input segment.

Within $\langle code \rangle$ the first and last points on the current input segment can be accessed using $\protect\prot$

In TikZ, you can use the following macros inside a TikZ coordinate.

\tikzinputsegmentfirst

The first point on the current input segment path.

\tikzinputsegmentlast

The last point on the current input segment path.

\tikzinputsegmentsupporta

The first support on the curveto input segment path.

\tikzinputsegmentsupportb

The second support on the curveto input segment path.

```
×----
```

```
\tikzset{
  show curve controls/.style={
    decoration={
      show path construction,
      curveto code={
       \draw [blue, dashed]
          (\tikzinputsegmentfirst)
                                      -- (\tikzinputsegmentsupporta)
          node [at end, cross out, draw, solid, red, inner sep=2pt]{};
        \draw [blue, dashed]
          (\tikzinputsegmentsupportb) -- (\tikzinputsegmentlast)
          node [at start, cross out, draw, solid, red, inner sep=2pt]{};
    },decorate
}
\tikzpicture
  \draw [postaction=show curve controls, thick]
    (0,2) .. controls (2.5,1.5) and (0.5,0.5) .. (3,0);
```

31.4 Marking Decorations

31.4.1 Overview

A marking on a path is any kind of graphic that is placed on a specific position on a path. Markings are useful in rather diverse situations: you can use them to, say, place little "footsteps" along a path as if someone where walking along the path; to place arrow tips on the middle of a path to indicate the "direction" in which something is flowing; or you can use them to place informative information at certain positions of a path.

For historical reasons there are three different libraries for placing marks on a path. They differ in what kind of markings can be added to a path. We start with the most general and most useful of these libraries.

31.5 Arbitrary Markings

```
\usepgflibrary{decorations.markings} % MEX and plain TeX and pure pgf \usepgflibrary[decorations.markings] % ConTeXt and pure pgf \usetikzlibrary{decorations.markings} % MEX and plain TeX when using TikZ \usetikzlibrary[decorations.markings] % ConTeXt when using TikZ
```

Markings are arbitrary "marks" that can be put on a path. Marks can be arrow tips or nodes or even whole pictures.

Decoration markings

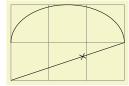
A marking can be thought of a "little picture" or more precisely of "some scope contents" that is placed "on" a path at a certain position. Suppose the marking should be a simple cross. We can produce this with the following code:

```
\draw (-2pt,-2pt) -- (2pt,2pt);
\draw (2pt,-2pt) -- (-2pt,2pt);
```

If we use this code as a marking at position 2cm on a path, then the following happens: PGF determines the position on the path that is 2cm along the path. Then is translates the coordinate system to this position and rotates it such that the positive x-axis is tangent to the path. Then a protective scope is created, inside which the above code is executed – resulting in a little cross on the path.

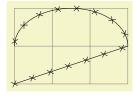
The markings decoration allows you to place one or more such markings on a path. The decoration destroys the input path (except in certain cases, detailed later), which means that it uses the path for determining positions on the path, but after the decoration is done this path is gone. You typically need to use a postaction to add markings.

Let us start with the above example in real code:



```
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=% actually add a mark
    at position 2cm
    with
    {
        \draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt);
    }
}
}
|
| draw [help lines] grid (3,2);
| draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
| end{tikzpicture}
```

We can also add the cross repeatedly:



The mark decoration option is used to specify a marking. It comes in two versions:

```
pgf/decoration/mark=at position pos with pos
```

(no default)

The options specifies that when a marking decoration is applied, there should be a marking at position $\langle pos \rangle$ on the path whose code is given by $\langle code \rangle$.

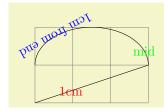
The $\langle pos \rangle$ can have four different forms:

- 1. It can be a non-negative dimension like Opt or 2cm or 5cm/2. In this case, it refers to the position along the path that is this far displaced from the start.
- 2. It can be a negative dimension like -1cm-2pt or -1sp. In this case, the position is taken from the end of the path. Thus, -1cm is the position that is -1cm displaced from the end of the path.
- 3. It can be a dimensionless non-negative number like 1/2 or 0.333+2*0.1. In this case, the $\langle pos \rangle$ is interpreted as a factor of the total path length. Thus, a $\langle pos \rangle$ or 0.5 refers to the middle of the path, 0.1 is near the start, and so on.
- 4. It can be a dimensionless negative number like -0.1. Then, again, the fraction of the path length counts "from the end."

The $\langle pos \rangle$ determines a position on the path. When the marking is applied, the (high level) coordinate system will have been transformed so that the origin lies at this position and the positive x-axis points along the path. For this coordinate system, the $\langle code \rangle$ is executed. It can contain all sorts of graphic drawing commands, including (even named) nodes.

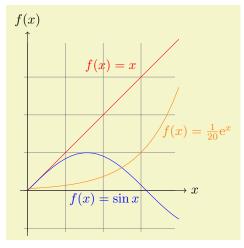
If the position lies past the end of the path (for instance if $\langle pos \rangle$ is set to 1.2), the marking will not be drawn.

It is possible to give the mark option several times, which causes several markings to be applied. In this case, however, it is necessary that the positions on the path are in increasing order. That is, it is not allowed (and will result in chaos) to have a marking that lies earlier on the path to follow a marking that is later on the path.



```
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=at position 1cm with \node[red]{1cm};,
    mark=at position .5 with \node[green]{mid};,
    mark=at position -1cm with {\node[blue,transform shape]{1cm from end};}}
    ]
    \draw [help lines] grid (3,2);
    \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
    \end{tikzpicture}
```

Here is an example that shows how markings can be used to place text on plots:



When the $\langle code \rangle$ is being executed, two special keys will have been set up, whose value may be of interest:

/pgf/decoration/mark info/sequence number

(no value)

This key can only be read. Its value (which can be obtained using the \pgfkeysvalueof command) is a "sequence number" of the mark. The first mark that is added to a path has number 1, the second number 2, and so on. This key is mainly useful in conjunction with repeated markings (see below).

/pgf/decoration/mark info/distance from start

(no value)

This key can only be read. Its value is the distance of the marking from the start of the path in points. For instance, if the path length is 100pt and the marking is in the middle of the path, the value of this key would be 50.0pt.

A second way to use the mark key is the following:

/pgf/decoration/mark=between positions $\langle start\ pos \rangle$ and $\langle end\ pos \rangle$ step $\langle stepping \rangle$ with $\langle code \rangle$ (no default)

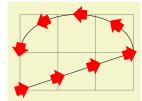
This works similarly to the at position version of this option, only multiple marks are placed, starting at $\langle start\ pos \rangle$ and then spaced apart by $\langle stepping \rangle$. The $\langle start\ pos \rangle$, the $\langle end\ pos \rangle$, and also the $\langle stepping \rangle$ may all be specified in the same way as for the at position version, that is, either using units or no units and also using positive or negative values.

Let us start with a simple example in which we place ten crosses along a path starting with the beginning of the path ($\langle start\ pos \rangle = 0$) and ending at the end ($\langle end\ pos \rangle = 1$).

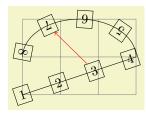


```
\begin{tikzpicture}[decoration={markings,
    mark=between positions 0 and 1 step 0.1
    with { \draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt); }} ]
    \draw [help lines] grid (3,2);
    \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
    \end{tikzpicture}
```

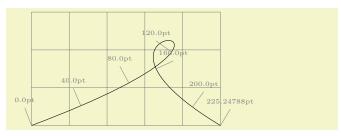
In the next example we place arrow shapes on the path instead of crosses. Note the use of the transform shape option to ensure that the nodes are actually rotated.



Using the key sequence number we can also "number" the nodes and even refer to them later on.



In the following example we use the distance info to place "length information" on a path:



/pgf/decoration/reset marks

(no value)

Since mark options accumulate, there needs to be a way to "reset" things, so that any mark options set in an enclosing scope do not interfere. This option does exactly this. Note that when the $\langle code \rangle$ of a marking is executed, the markings are automatically reset.

As mentioned earlier, the decoration usually destroys the path. However, this is no longer the case when the following key is set:

```
/pgf/decoration/mark connection node=\langle node \ name \rangle
```

(no default, initially empty)

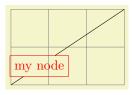
When this key is set to a non-empty $\langle node\ name \rangle$ while the decoration is being processed, the following happens: The marking code should, among possibly other things, define a node named $\langle node\ name \rangle$. Then, the output path of this decoration will contain a line-to to "one end" of this node, followed by a moveto to the "other end" of the node. More precisely, the first end is given by the position on the border of $\langle node\ name \rangle$ that lies in the direction "from which the path heads toward the node" while the other end lies on the border "where the path heads away from the node." Furthermore, this option causes the decoration to end with a line-to to the end instead of a move-to.

The net effect of all this is that when you decorate a straight line with one or more markings that contain just a node, the line will effectively connect these nodes.

Here are two examples that show how this works:



```
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .5 with
      {\node [draw,blue,transform shape] (my node) {my node};}}]
\draw [help lines] grid (3,2);
\draw decorate { (0,0) -- (3,2) };
\end{tikzpicture}
```



```
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .25 with
      {\node [draw,red] (my node) {my node};}}]
    \draw [help lines] grid (3,2);
    \draw decorate { (0,0) -- (3,2) };
    \end{tikzpicture}
```

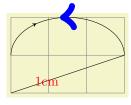
31.5.1 Arrow Tip Markings

Frequent markings that are hard to create correctly are arrow tips. For them, two special commands are available when the $\langle code \rangle$ of a mark option is executed. (They are only defined in this code):

```
\arrow[\langle options \rangle] \{\langle arrow\ end\ tip \rangle\}
```

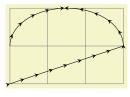
This command simply draws the $\langle arrow \ end \ tip \rangle$ at the origin, pointing right. This is exactly what you need when you want to draw an arrow tip as a marking.

The $\langle options \rangle$ can only be given when TikZ is used. In this case, they are executed in a scope that contains the arrow tip.



```
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=at position 1cm with {\node[red]{1cm};},
    mark=at position .75 with {\arrow[blue,line width=2mm]{>}},
    mark=at position -1cm with {\arrowreversed[black]{stealth};}}
    ]
    \draw [help lines] grid (3,2);
    \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
    \end{tikzpicture}
```

Here is a more useful example:



```
begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=between positions 0 and .75 step 4mm with {\arrow{stealth}},
    mark=between positions .75 and 1 step 4mm with {\arrowreversed{stealth}}}
]
    \draw [help lines] grid (3,2);
    \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

 $\arrowreversed[\langle options \rangle] \{\langle arrow\ end\ tip \rangle\}$

As above, only the arrow end tip is flipped and points in the other direction.

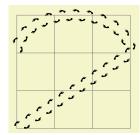
31.5.2 Footprint Markings

```
\usepgflibrary{\decorations.footprints} % MEX and plain TeX and pure pgf \usepgflibrary[\decorations.footprints] % ConTeXt and pure pgf \usetikzlibrary{\decorations.footprints} % MEX and plain TeX when using TikZ \usetikzlibrary[\decorations.footprints] % ConTeXt when using TikZ
```

The decorations of this library can be used to decorate a path with little footprints, as if someone had "walked" along the path.

Decoration footprints

The footprint decoration adds little footprints around the path. They start with the left foot.



```
\begin{tikzpicture}[decoration={footprints,foot length=5pt,stride length=10pt}]
\draw [help lines] grid (3,3);
\fill [decorate] (0,0) -- (3,2) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

You can influence the way this decoration looks using the following options:

/pgf/decoration/foot length

(initially 10pt)

The length or size of the footprint itself. A larger value makes the footprint larger, but does not change the stride length.

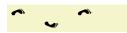


```
\begin{tikzpicture}[decoration={footprints,foot length=20pt}]
\fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/stride length

(initially 30pt)

The length of strides. This is the distance between the beginnings of left footprints along the path.



```
\begin{tikzpicture}[decoration={footprints,stride length=50pt}]
\fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot sep

(initially 4pt)

The separation in the middle between the footprints. The footprints are moved away from the path by half this amount.

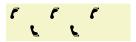


```
\begin{tikzpicture}[decoration={footprints,foot sep=10pt}]
\fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot angle

(initially 10)

Footprints are rotated by this much.

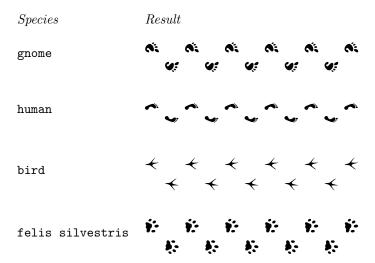


```
\begin{tikzpicture}[decoration={footprints,foot angle=60}]
\fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot of

(initially human)

The species whose footprints are shown. Possible values are:



31.5.3 Shape Background Markings

The third library for adding markings uses the background paths of certain shapes. This library is included mostly for historical reasons, using the markings library is usually preferable.

```
\usepgflibrary{decorations.shapes} % MEX and plain TeX and pure pgf \usepgflibrary[decorations.shapes] % ConTeXt and pure pgf \usetikzlibrary{decorations.shapes} % MEX and plain TeX when using TikZ \usetikzlibrary[decorations.shapes] % ConTeXt when using TikZ
```

This library defines decorations that use shapes or shape-like drawings to decorate a path. The following options are common options used by the decorations in this library:

The desired width of the shapes. For decorations that support varying shape sizes, this key sets both the start and end width (which can be overwritten using options like shape start width).

```
\label{eq:pgf_decoration_shape} \begin{tabular}{ll} \begin{tabul
```

Works like the previous key, only for the height.

```
/pgf/decoration/shape size = \langle dimension \rangle (no default)
```

Sets the desired width and height simultaneously.

For the exact places and macros where these keys store the values, please consult the beginning of the code of the library.

Decoration crosses

This decoration replaces the path by (diagonal) crosses. The following parameters influence the decoration:

- segment length determines the distance between (the centers of) consecutive crosses.
- shape height determines the height of each cross.
- shape width determines the width of each cross.

Decoration triangles

This decoration replaces the path by triangles that point along the path. The following parameters influence the decoration:

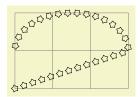
- segment length determines the distance between consecutive triangles.
- shape height determines the height of the triangle side that is orthogonal to the path.
- shape width determines the width of the triangle.

```
\begin{tikzpicture}[decoration=triangles]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=examplefill] (0,0) -- (3,1) arc (0:180:1.5 and 1);
  \end{tikzpicture}
```

Decoration shape backgrounds

This is a general decoration that replaces the to-be-decorated path by repeated copies of the background path of an arbitrary shape that has previously been defined using the \pgfdeclareshape command (that is, you can use any shape in the shape libraries).

Please note that the background path of the shapes is used, but no nodes are created. This means that you cannot have text inside the shapes of this path, you cannot name them, or refer to them. Finally, this decoration will not work with shapes that depend strongly on the size of the text box (like the arrow shapes). If any of these restrictions pose a problem, use the markings library instead.



```
\begin{tikzpicture}[decoration={shape backgrounds,shape=star,shape size=5pt}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
  \end{tikzpicture}
```

```
\tikzset{paint/.style={ draw=#1!50!black, fill=#1!50 },
         decorate with/.style=
          {decorate,decoration={shape backgrounds,shape=#1,shape size=2mm}}}
\begin{tikzpicture}
  \draw [decorate with=dart,
                                  paint=red]
                                                 (0,1.5) -- (3,1.5);
  \draw [decorate with=diamond,
                                  paint=green]
                                                 (0,1)
                                                        -- (3,1);
                                                 (0,0.5) -- (3,0.5);
 \draw [decorate with=rectangle, paint=blue]
  \draw [decorate with=circle,
                                  paint=yellow] (0,0)
\end{tikzpicture}
```

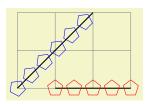
All shapes are positioned by the anchor that is specified via the anchor decoration option:

```
/pgf/decoration/anchor = \langle anchor \rangle
```

(no default, initially center)

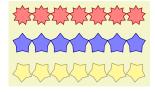
The anchor used to position the shape backgrounds.

A shape background path is added at the start point of the path and, if the distance between the shapes is appropriate, at the end point of the path.



Keys for customizing specific shapes can be specified (e.g., star points, cloud puffs, kite angles, and so on). The size of the shape is "enforced" using transformations. This means that the shape is typeset with an empty text box and some default size values, resulting in an initial shape. This shape is then rescaled using coordinate transformations so that it has the desired size (which may vary as we

travel along the to-be-decorated path). This means that settings involving angles and distances may not appear entirely accurate. More general options such as inner sep and minimum size will be ignored, but transformations can be applied to each segment as described below.



There are various keys to control the drawing of the shape decoration.

/pgf/decoration/shape=\langle shape name \rangle

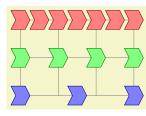
(no default, initially circle)

The shape whose background path is used.

```
/pgf/decoration/shape sep=\langle spacing \rangle
```

(no default, initially .25cm, between centers)

Set the spacing between the shapes on the decorations path. This can be just a distance on its own, but the additional keywords between centers, and between borders (which must be preceded by a comma), specify that the distance is between the center anchors of the shapes or between the edges of the boundaries of the shape borders.

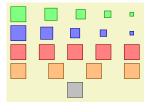


```
\begin{tikzpicture}[
    decoration={shape backgrounds,shape size=.5cm,shape=signal},
    signal from=west, signal to=east,
    paint/.style={decorate, draw=#1!50!black, fill=#1!50}]
    \draw [help lines] grid (3,2);
    \draw [paint=red, decoration={shape sep=.5cm}]
        (0,2) -- (3,2);
    \draw [paint=green, decoration={shape sep={1cm, between centers}}]
        (0,1) -- (3,1);
    \draw [paint=blue, decoration={shape sep={1cm, between borders}}]
        (0,0) -- (3,0);
    \end{tikzpicture}
```

/pgf/decoration/shape evenly spread= $\langle number \rangle$

(no default)

This key overrides the shape sep key and forces the decoration to fit $\langle number \rangle$ shapes evenly across the path. If $\langle number \rangle$ is less than 1, then no shapes will be used. If $\langle number \rangle$ equals 1, then one shape is put in the middle of the path. The additional keywords by centers (the default, if no keyword is specified) and by borders can be used (both preceded by a comma), to specify how the distance between shapes is determined. These keywords will only have a noticeable effect if the shapes sizes differ over time.

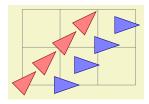


```
\tikzset{
 paint/.style={draw=#1!50!black, fill=#1!50},
  spreading/.style={
    decorate, decoration={shape backgrounds, shape=rectangle,
    shape start size=4mm, shape end size=1mm, shape evenly spread={#1}}}
\begin{tikzpicture}
  \fill [paint=green, spreading={5, by borders},
         decoration={shape scaled}]
                                                (0,2)
                                                         -- (3,2):
  \fill [paint=blue,spreading={5, by centers},
        decoration={shape scaled}]
                                                (0,1.5) -- (3,1.5);
                                                        -- (3,1);
                                                (0,1)
  \fill [paint=red,
                       spreading=5]
  \fill [paint=orange, spreading=4]
                                                        -- (3,.5);
                                                (0,.5)
  \fill [paint=gray,
                                                 (0,0)
                                                          -- (3,0);
                        spreading=1]
\end{tikzpicture}
```

/pgf/decoration/shape sloped= $\langle boolean \rangle$

(no default, initially true)

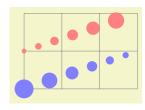
By default, shapes are rotated to the slope of the decorations path. If $\langle boolean \rangle$ is the value false, then this rotation is turned off. Internally this sets the TEX-if \ifpgfshapedecorationsloped accordingly.



It is possible to scale the width and height of the shapes along the length of the decorations path. The shapes are scaled between the starting size and the ending size. The following keys customize the way the decoration shapes are scaled:

/pgf/decoration/shape scaled= $\langle boolean \rangle$

(no default, initially false)



If this key is set to false (which is the default), then only the start width and height are used. Note that the keys shape width and shape height set the start and end height simultaneously.

```
/pgf/decoration/shape start width=\langle length \rangle
```

(no default, initially 2.5pt)

The starting width of the shape.

```
/pgf/decoration/shape start height=\langle length\rangle
```

(no default, initially 2.5pt)

The starting height of the shape.

```
/pgf/decoration/shape start size=\langle length \rangle
```

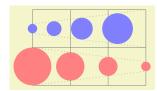
(style, no default)

Sets both the start height and start width simultaneously.

```
/pgf/decoration/shape end width=\langle length \rangle
```

(no default, initially 2.5pt)

The recommended ending width of the shape. Note that this is the width that a shape will take only if it is drawn exactly at the end of the path.



```
\tikzset{
 bigger/.style={decoration={shape start size=.25cm, shape end size=1cm}},
 smaller/.style={decoration={shape start size=1cm, shape end size=.25cm}},
 decoration={shape backgrounds,
              shape sep={.25cm, between borders},shape scaled}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \fill [decorate,bigger,
        decoration={shape sep={.25cm, between borders}}, blue!50]
    (0,1.5) -- (3,1.5);
 \fill [decorate, smaller,
        decoration={shape sep={1cm, between centers}},
    (0..5) -- (3..5):
 \draw [gray, dotted] (0,1.625) -- (3,2)
                                             (0,1.375) -- (3,1)
                              -- (3,.625) (0,0)
                                                    -- (3,.375);
                       (0,1)
\end{tikzpicture}
```

```
/pgf/decoration/shape end height=\langle length\rangle (no default)
```

The recommended ending height of the shape.

```
/pgf/decoration/shape end size=\langle length \rangle
```

Set both the end height and end width simultaneously.

31.6 Text Decorations

```
\usepgflibrary{\( \decorations.text \)} \% \text{MEX} \text{ and plain TeX} \text{ and pure pgf} \\ \usepgflibrary[\( \decorations.text \)] \% \text{ConTeXt} \text{ and pure pgf} \\ \usetikzlibrary{\( \decorations.text \)} \% \text{MEX} \text{ and plain TeX} \text{ when using TikZ} \\ \usetikzlibrary[\( \decorations.text \)] \% \text{ConTeXt} \text{ when using TikZ}
```

The decoration in this library decorates the path with some text. This can be used to draw text that follows a curve.

Decoration text along path

This decoration decorates the path with text. This drawing of the text is a "side effect" of the decoration. The to-be-decorated path is only used to determine where the characters should be put and it is thrown away after the decoration is done. This is why no line is shown in the following example.

```
Some long text alone
```

```
\catcode'\|12
\begin{tikzpicture}[decoration={text along path,
    text={Some long text along a ridiculously long curve that}}]
\draw [help lines] grid (3,2);
\draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

PGF "does its best" to typeset the text, however you should note the following points:

- Each character in the text is typeset in a separate \hbox. This means that if you want fancy things like kerning or ligatures you will have to manually annotate the characters in the decoration text within a group, for example, W{\kern-1ptA}TER.
- Each character is positioned using the center of its baseline. To move the text vertically (relative to the path), the additional transform key should be used.
- No attempt is made to ensure characters do not overlap when the angle between segments is considerably less than 180° (this is tricky to do in TEX without a huge processing overhead). In general this should not be too much of a problem, but, once again, kerning can be used in most cases to overcome any undesirable effects.
- It is only possible to typeset text in math mode under considerable restrictions. Math mode is entered and exited using any character of category code 3 (e.g., in plain TeX this is \$). Math subscripts and superscripts need to be contained within braces (e.g., {^y_i}) as do commands like \times or \cdot. However, even modestly complex mathematical typesetting is unlikely to be successful along a path (or even desirable).
- Some inaccuracies in positioning may be particularly apparent at input segment boundaries. This can (unfortunately) only be solved on a case-by-case basis by individually kerning the offending characters within a group.

The following keys are used by the text decoration:

```
/pgf/decoration/text = \langle text \rangle
```

(no default, initially empty)

(style, no default)

Sets the text to typeset along the curve. Consecutive spaces are ignored, so \ (or \space in LATEX) should be used to insert multiple spaces. It is possible to format the text using normal formatting commands, such as \it, \bf and \color, within customizable delimiters. Initially these delimiters are both | (however, care will be needed regarding the category codes of delimiters — see below).

```
green juicy apple.
```

By following the first delimiter with +, the formatting commands are added to any existing formatting.



Internally, the text is stored in the macro \pgfdecorationtext. Any characters that have not been typeset when the end of the path has been reached will be stored in \pgfdecorationrestoftext.

```
pgf/decoration/text format delimiters={\langle before \rangle}{\langle after \rangle} (no default, initially {|}{})
```

Set the characters that the text decoration will use to parse formatting commands. If $\langle after \rangle$ is empty, then $\langle before \rangle$ will be used for both delimiters. In general you should stick to characters whose category codes are 11 or 12. As + is used to indicate that the specified format commands are added to any existing ones, you should avoid using + as a delimiter.



```
/pgf/decoration/text color=\langle color \rangle
```

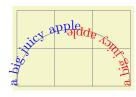
(no default, initially black)

The color of the text.

```
/pgf/decoration/reverse path=\langle boolean \rangle
```

(no default, initially false)

This key reverses the path. This is especially useful for typesetting text along different sides of curves.



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw [gray, ->]
  [postaction={decoration={text along path,
    text={a big juicy apple}, text color=red}, decorate}]
  [postaction={decoration={text along path,
    text={a big juicy apple}, text color=blue, reverse path}, decorate}]
  (3,0) .. controls (3,2) and (0,2) .. (0,0);
  \end{tikzpicture}
```

```
/pgf/decoration/text align={\langle alignment options \rangle}
```

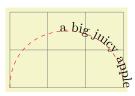
(no default)

This changes the key path to /pgf/decoration/text align and executes (alignment options).

```
/pgf/decoration/text align/align=\langle alignment \rangle
```

(no default, initially left)

Aligns the text according to \(alignment \), which should be one of left, right, or center.



/pgf/decoration/text align/left

(style, no value)

Aligns the text to the left end of the path.

/pgf/decoration/text align/right

(style, no value)

Aligns the text to the right end of the path.

```
/pgf/decoration/text align/center
```

(style, no value)

Aligns the text to the center of the path.

```
/pgf/decoration/text align/left indent=\langle length \rangle
```

(no default, initially Opt)

Specifies a distance which the automaton should move along before it starts typesetting the text.

```
/pgf/decoration/text align/right indent=\langle length\rangle
```

(no default, initially Opt)

Specifies a distance before the end of the path, where the automaton should stop typesetting the text.

```
/pgf/decoration/text align/fit to path=\langle boolean \rangle
```

(no default, initially false)

This key makes the decoration automaton try to fit the text to the length of the path. The automaton shifts forward by a small amount between each character in order to fit the text to the path. If, however, the length of the text is longer than the length of the path (i.e., the automaton would have to shift backwards between characters) this key will have no effect.

```
Juic
```

/pgf/decoration/text align/fit to path stretching spaces= $\langle boolean \rangle$ (no default, initially false)

This key works like the previous key except the automaton shifts forward only for space characters (including \space, but excluding \).



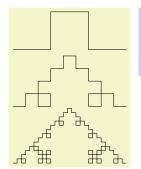
31.7 Fractal Decorations

```
\usepgflibrary{decorations.fractals} % \mbox{MEX} and plain \mbox{TeX} and pure pgf \usepgflibrary[decorations.fractals] % \mbox{ConTeX}t and pure pgf \usetikzlibrary{decorations.fractals} % \mbox{MEX} and plain \mbox{TeX} when using \mbox{Ti}kZ \usetikzlibrary[decorations.fractals] % \mbox{ConTeX}t when using \mbox{Ti}kZ
```

The decorations of this library can be used to create fractal lines. To use them, you typically have to apply the decoration repeatedly to an originally straight path.

Decoration Koch curve type 1

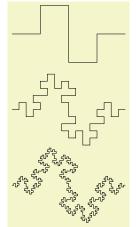
This decoration replaces a straight line by a "rectangular bump." By repeatedly applying this replacement, different levels of the Koch curve fractal can be created. Its Hausdorff dimension is $\log 5/\log 3$.



```
\begin{tikzpicture}[decoration=Koch curve type 1]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1.5) -- (3,-1.5) }};
  \draw decorate{ decorate{ (0,-3) -- (3,-3) }};
  \end{tikzpicture}
```

Decoration Koch curve type 2

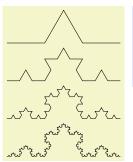
This decoration replaces a straight line by a "rectangular sine." Its Hausdorff dimension is 3/2.



```
\begin{tikzpicture}[decoration=Koch curve type 2]
\draw decorate{ (0,0) -- (3,0) };
\draw decorate{ decorate{ (0,-2) -- (3,-2) }};
\draw decorate{ decorate{ decorate{ (0,-4) -- (3,-4) }}};
\end{tikzpicture}
```

Decoration Koch snowflake

This decoration replaces a straight line by a "line with a spike." The Hausdorff dimension of Koch's snowflake's is $\log 4/\log 3$.



```
\begin{tikzpicture}[decoration=Koch snowflake]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1) -- (3,-1) }};
  \draw decorate{ decorate{ decorate{ (0,-2) -- (3,-2) }}};
  \draw decorate{ decorate{ decorate{ decorate{ (0,-3) -- (3,-3) }}};
  \end{tikzpicture}
```

Decoration Cantor set

This decoration replaces a straight line by a "line with a gap in the middle." The Hausdorff dimension of the Cantor set is $\log 2/\log 3$.

```
\begin{tikzpicture}[decoration=Cantor set,very thick]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-.5) -- (3,-.5) }};
  \draw decorate{ decorate{ decorate{ (0,-1) -- (3,-1) }}};
  \draw decorate{ decorate{ decorate{ decorate{ (0,-1.5) -- (3,-1.5) }}}};
  \end{tikzpicture}
```

32 Entity-Relationship Diagram Drawing Library

```
\usetikzlibrary{er} % MEX and plain TeX \usetikzlibrary[er] % ConTeXt
```

This packages provides styles for drawing entity-relationship diagrams.

This library is intended to help you in creating E/R-diagrams. It defines only few new styles, but using the style entity instead of saying rectangle, draw makes the code more expressive.

32.1 Entities

The package defines a simple style for drawing entities:

/tikz/entity (style, no value)

This style is to be used with nodes that represent entity types. It causes the node's shape to be set to a rectangle that is drawn and whose minimum size and width are set to sensible values.

Note that this style is called **entity** despite the fact that it is to be used for nodes representing entity types (the difference between an entity and an entity type is the same as the difference between an object and a class in object-oriented programming). If this bothers you, feel free to define a style **entity type** instead.

/tikz/every entity

(style, no value)

This style is evoked by the style entity. To change the appearance of entities, you can change this style.

32.2 Relationships

Relationships are drawn using styles that are very similar to the styles for entities.

/tikz/relationship

(style, no value)

This style works like entity, only it is to be used for relationships. Again, relationships are actually relationship types.

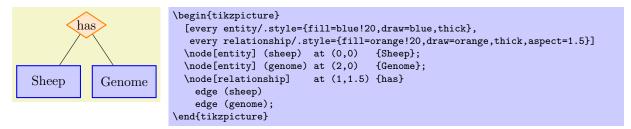
```
has

\[ \langle \text{login{tikzpicture}} \\ \node[entity] \text{ (sheep) at (0,0) {Sheep};} \\ \node[entity] \text{ (genome) at (2,0) {Genome};} \\ \node[relationship] \text{ at (1,1.5) {has}} \\ \text{ edge (sheep)} \\ \text{ edge (genome);} \\ \text{ end{tikzpicture}} \]
```

/tikz/every relationship

(style, no value)

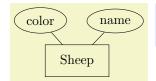
Works like every entity.



32.3 Attributes

/tikz/attribute (style, no value)

This style is used to indicate that a node is an attribute. To connect an attribute to its entity, you can use, for example, the child command or the pin option.



\begin{tikzpicture}[every pin edge/.style=draw]
\node[entity,pin={[attribute]60:name},pin={[attribute]120:color}] {Sheep};
\end{tikzpicture}

/tikz/key attribute

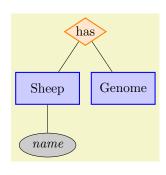
(style, no value)

This style is intended for key attributes. By default, the will cause the attribute to be typeset in italics. Typically, underlining is used instead, but that looks ugly and it is difficult to implement in T_FX.

/tikz/every attribute

(style, no value)

This style is used with every attribute, and therefore also for every key attribute.



```
begin{tikzpicture}
  [text depth=1pt,
    every attribute/.style={fill=black!20,draw=black},
    every entity/.style={fill=blue!20,draw=blue,thick},
    every relationship/.style={fill=orange!20,draw=orange,thick,aspect=1.5}]

\node[entity] (sheep) at (0,0) {Sheep}
    child {node [key attribute] {name}};
    \node[entity] (genome) at (2,0) {Genome};
    \node[relationship] at (1,1.5) {has}
    edge (sheep)
    edge (genome);
\end{tikzpicture}
```

33 Externalization Library

by Christian Feuersänger

```
\usetikzlibrary{external} % MEX and plain TeX \usetikzlibrary[external] % ConTeXt
```

This library provides a high-level automatic or semi-automatic export feature for TikZ pictures. Its purpose is to convert each picture to a separate PDF without changing the document as such.

It also externalizes \label information (and other aux file related stuff) using auxiliary files.

33.1 Overview

There are several reasons why external images for at least some pictures are of interest:

- 1. Larger picture require a considerable amount of time, which is necessary for every compilation. However, only few images will change from run to run. It can simply save time to export finished images and include them as final graphics.
- 2. It may be desirable to have final images for some graphics, for example to include them in third–party programs or to communicate them electronically.
- 3. It may be necessary to typeset a file in environments where PGF and TikZ are not available. In this case, external images are the only way to ensure compatibility.

The purpose of this library is to provide a way to export any TikZ-picture to separate PDF (or eps) images without changing the main document. It is actually a simple user interface to the $\ensuremath{\texttt{beginpgfgraphicnamed}}$... $\ensuremath{\texttt{endpgfgraphicnamed}}$ framework of PGF which is discussed in section 88.

33.2 Requirements

For most users, the library does not need special attention since requirements are met anyway. It collects all tokens between \begin{tikzpicture} and the next following \end{tikzpicture} and replaces them by the appropriate graphics or it takes steps to generate such an image.

It can't expand macros during this step, so the only requirement is that every picture's end is directly reachable from its beginning, without further macro expansion. Furthermore, the library assumes that all LaTeX pictures are ended with \end{tikzpicture}.

The library always searches for the *next* picture's end, \end{tikzpicture}. As a consequence, you can't use nested pictures directly. You *can* nest pictures, but you have to avoid that the nested picture's \end command is found before the outer \end command (for example using bracing constructs or by writing the nested picture into a separate macro call).

Consider using the \tikzexternaldisable method in case you'd like to skip selected pictures which do not meet the requirements.

33.3 A Word About ConTeXt And Plain TeX

Currently, the basic layer backend \beginpgfgraphicnamed ... \endpgfgraphicnamed relies on LATEX only, so externalization is currently only supported for LATEX.

33.4 Externalizing Graphics

After loading the library, a call to \tikzexternalize is necessary to activate the externalization.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize % activate!

\begin{document}
\begin{document}
\tikzpicture}
\node {root}
\child {node {left}}
\child {node {left}}
\child {node {child}}
\child {node {child}}
\thild {node {child}}
\t
```

The method works as follows: if the document is typeset normally, the library searches for replacement images for every picture. Filenames are generated automatically in the default configuration. In our case, the two file names will be main-figure0 and main-figure1. If they exist, those images are simply included and the pictures as such are not processed. If graphics files do not exist, steps are taken to generate the missing ones. Since (currently) only one output file can be set, each missing image needs to be generated by a separate run of IATEX in which the \jobname is set to the desired image file name. In the default configuration mode=convert with system call, these commands are issued automatically by using the \write18 method to call system commands. It is also possible to output every required file name or to generate a makefile; users will need to issue the required commands manually (or with make). The probably most comfortable way is to use the default configuration with

pdflatex -shell-escape main

which authorizes pdflatex to call itself recursively to generate the images. When it finishes, all images are generated and the document already includes them.

From this point on, successive runs of LATEX will use the final graphics files, the pictures won't be used anymore. Section 33.5 contains details about how to submit such a file to environments where PGF is not available.

$\verb|\tikzexternalize[\langle optional\ arguments \rangle]|$

This command activates the externalization. It installs commands to replace every TikZ-picture. It needs to be called before $\begin{document}$ because it may need to install its separate shipout routine.

The *(optional arguments)* can be any of the keys described below.

Note that the generation/modification of auxiliary files like .aux, .toc etc. is usually suppressed while a single image is externalized (details for \label support follow).

It is also possible to write $\texttt{tikzexternalize}\{\langle main\ job\ name \rangle\}$ if the argument is delimited by curly braces. This case is mainly for backwards compatibility and is no longer necessary. Since it might be useful in rare circumstances, it is documented in section 33.4.5.

A detailed description about the process of externalization is provided in section 33.4.5.

\tikzexternalrealjob

After the library is loaded, this macro will *always* contain the correct main job's name (in the example above, it is main). It is to be used instead of \jobname when the externalization is in effect.

\pgfactualjobname

Once \tikzexternalize has been called, \pgfactualjobname contains the name of the currently generated output file (which may be main or main-figure0 or main-figure1 in our example above).

\jobname

The value of \jobname is one of \tikzexternalrealjob or \pgfactualjobname, depending on the configuration. In short: if auxiliary file support (\label and \ref) is activated, \jobname=\tikzexternalrealjob (since that's the base file name of auxiliary files).

```
/tikz/external/system call=\{\langle template \rangle\}
```

(no default)

A template string used to generate system calls. Inside of $\{\langle template \rangle\}$, the macro \image can be used as placeholder for the image which is about to be generated while \texsource contains the main file name (in truth, it contains \input{\(main file name \)}, but that doesn't matter).

The default is

```
\tikzset{external/system call={pdflatex \tikzexternalcheckshellescape -halt-on-error
-interaction=batchmode -jobname "\image" "\texsource"}
```

where \tikzexternalcheckshellescape inserts the value of the configuration key shell escape if and only if the current document has been typeset with -shell-escape¹¹.

For eps output, you can (and need to) use

```
\tikzset{external/system call={latex \tikzexternalcheckshellescape -halt-on-error
   -interaction=batchmode -jobname "\image" "\texsource";
   dvips -o "\image".ps "\image".dvi}}
```

The argument $\{\langle template \rangle\}$ will be expanded using $\backslash edef$, so any control sequences will be expanded. During this evaluation, '\' will result in a normal backslash, '\'. Furthermore, double quotes '"', single quotes ''', semicolons and dashes '-' will be made to normal characters if any package uses them as macros. This ensures compatibility with the german package, for example.

```
/tikz/external/shell escape=\{\langle command-line \ arg \rangle\}
```

(no default, initially -shell-escape)

Contains the command line option for latex which enables the \write18 feature. For TeX-Live, this is -shell-escape. For MikTeX, you should use \tikzexternalize[shell escape=-enable-write18].

33.4.1 Support for Labels and References In External Files

The external library comes with extra support for \label and \ref (and other commands which usually store information in the .aux file) inside an external files.

There are, however, some points which need your attention when you try to use

- a) \ref to something in the main document inside an externalized graphics or
- b) \label in the externalized graphics which is referenced in the main document.

For point a), a \ref inside of an externalized graphics works *only* if you issue the required system call *manually* or by make. The initial configuration mode=convert with system call does *not* support \ref. But you can copy—paste the system call generated by mode=convert with system call and issue it manually. The reason is that \ref information is stored in the main .aux file – but this auxiliary file is not completely written when mode=convert with system call is invoked (there is a race condition). Note that \pageref is not supported (sorry). Thus: if you have \ref inside of external graphics, consider using mode=list and make or copy—paste the system call for the image(s) and issue it manually.

Point b) is realized automatically by the external library. In detail, a \label inside of an externalized graphics causes the external library to generate separate auxiliary files for every external image. These files are called $\langle imagename \rangle$.dpth. The extension .dpth indicates that the file also contains the image's depth (the baseline key of TikZ). Furthermore, anything which would have been written to an .aux file will be redirected to the .dpth file – but only things which occur inside of the externalized tikzpicture environment. When the main document loads the image, it will copy the .dpth file into the main .aux file. Then, successive compilations of the main document contain the external \label information. In other words, a \label in an external graphics needs the following work flow:

- 1. The external graphics needs to be generated together with its .dpth (usually automatically by TikZ).
- 2. The main document includes the external graphics and copies the .dpth content into its main .aux file.
- 3. The main document needs to be translated once again to re-read its .aux file¹².

¹¹Note that this is always true for the default configuration. This security consideration applies mainly for mode=list and make which will also work without shell escapes.

¹²Note that it is not possible to activate the content of an auxiliary file after \begin{document} in LATEX.

There is just one special case: if a \label/\ref combination is realized itsself by a tikzpicture which should be externalized, you need to proceed as for case a) since mode=convert with system call can't handle that stuff on its own. Thus, \label works automatically, just translate the main document often enough.

```
/tikz/external/aux in dpth=\{\langle boolean \rangle\}
```

(no default, initially true)

Allows to disable the feature. Disabling it will safe one \newwrite command, i.e. a write register.

Also see the disable dependency files feature.

33.4.2 Customizing the Generated File Names

The default filename for externalized graphics is ' $\langle real \ file \ name \rangle$ -figure_ $\langle number \rangle$ ' where $\langle number \rangle$ ranges from 0 to whatever is required. However, there are a couple of ways to change the generated filenames:

- Changing the overall file name using a prefix,
- Changing the file name for a single figure using \tikzsetnextfilename,
- Changing the file name for a restricted set of figures using figure name.

```
\tikz/external/prefix={\langle file\ name\ prefix\rangle}
```

(no default, initially empty)

A shortcut for $\mathsf{tikzsetexternalprefix}\{\langle file\ name\ prefix \rangle\}$, see below.

```
\tikzsetexternalprefix{\langle file name prefix\}
```

Assigns a common prefix used by all file names. For example,

```
\tikzsetexternalprefix{figures/}
```

will prepend figures/ to every external graphics file name.

Please note that \tikzsetexternalprefix is the *only* way to assign a prefix in case you want to prepare your document for environments where PGF is not installed (see section 33.5).

```
\tikzsetnextfilename{\langle file\ name \rangle}
```

Sets the file name for the next TikZ picture or \tikz short command. It will only be used for the next picture.

Pictures for which no explicit file name has been set (or the next file name is empty) will get automatically generated file names.

Please note that prefix will still be prepended to $\{\langle file\ name \rangle\}$.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}
\usetikzlibrary{external}
\tikzexternalize[prefix=figures/] % activate
\begin{document}
\tikzsetnextfilename{trees}
\begin{tikzpicture} % will be written to 'figures/trees.pdf'
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
\end{tikzpicture}
\tikzsetnextfilename{simple}
A simple image is \tikz \fill (0,0) circle(5pt);. % will be written to 'figures/simple.pdf'
\begin{tikzpicture} % will be written to 'figures/main-figure0.pdf'
   \draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
\end{document}
```

```
pdflatex -shell-escape main
```

```
/tikz/external/figure name=\{\langle name \rangle\}
```

(no default)

Same as $\tikzsetfigurename{\langle name \rangle}$.

$\tikzsetfigurename{\langle name \rangle}$

Changes the names of *all* following figures. It is possible to change figure name during the document either using $\tikzset{external/figure name} = {\langle name \rangle}$ or with this command. A unique counter will be used for each different $\{\langle name \rangle\}$, and each counter will start at 0.

The value of prefix will be applied after figure name has been evaluated.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}
\usetikzlibrary{external}
\tikzexternalize % activate
\begin{document}
\begin{tikzpicture} % will be written to 'main-figure0.pdf'
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
\end{tikzpicture}
  \tikzsetfigurename{subset_}
  A simple image is \tikz \fill (0,0) circle(5pt);. % will be written to 'subset_0.pdf'
  \begin{tikzpicture} % will be written to 'subset_1.pdf'
     \draw[help lines] (0,0) grid (5,5);
  \end{tikzpicture}
}% here, the old file name will be restored:
\begin{tikzpicture} % will be written to 'main-figure1.pdf'
   draw (0,0) -- (5,5);
\end{tikzpicture}
\end{document}
```

The scope of figure name ends with the next closing brace.

Remark: Use \tikzset{external/figure name/.add={prefix_}} to add a prefix_ and a _suffix_ to the actual value of figure name.

$\texttt{tikzappendtofigurename}\{\langle suffix\rangle\}$

Appends $\langle suffix \rangle$ to the actual value of figure name.

It is a shortcut for $\mathsf{tikzset}\{\mathsf{external/figure\ name/.add}=\{\}\{\langle \mathit{suffix}\rangle\}\}\$ (a shortcut which is also supported if $\mathsf{Ti}k\mathsf{Z}$ is not installed, see below).

33.4.3 Remaking Figures or Skipping Figures

$\tikzpicturedependsonfile{\langle file\ name \rangle}$

Adds a dependency for the *next* picture which is about to be externalized. If the command is invoked within a picture environment, it adds a dependency for the surrounding picture. Dependencies are written into $\langle target\ file \rangle$. dep in the format

```
\langle target \ file \rangle. \tikzexternalimgextension: \langle file \ name \rangle.
```

The effect is that if $\langle file\ name \rangle$ changes, the external graphics associated with the picture shall be remade.

This command uses the contents of \tikzexternalimgextension to check for graphics. If you encounter difficulties with image extensions, consider redefining this macro (after \tikzexternalize).

Limitations: this command is currently only supported for mode=list and make and the generated makefile.

```
\tikzexternalfiledependsonfile{\langle external graphics \rangle} {\langle file name \rangle}
```

A variant of $\tikzpicturedependsonfile$ which adds a dependency for an $\tikzpicturedependsonfile$ which adds a dependency for an $\tikzpicturedependsonfile$ which as it would have been generated by the external library, i.e. without file extension but including any prefixes.

/tikz/external/disable dependency files

(no value)

Allows to (irreversibly) disable the generation of file dependencies. Disabling it will safe one \newwrite command, i.e. a write register. Note that the write register is only allocated if the feature has been used at all. This key needs to be provided as argument to \tikzexternalize (or it needs to be set before calling \tikzexternalize).

Also see the aux in dpth key.

/tikz/external/force remake= $\{\langle boolean \rangle\}$

(default true)

A boolean which is used to customize the up-to-date checks of all following figures. Every up-to-date check will fail, resulting in automatic regeneration of every following figure.

```
\tikzset{external/force remake}
\begin{tikzpicture}
\draw (0,0) circle(5pt);
\end{tikzpicture}
```

You can also use force remake inside of a local TeX group to remake only selected pictures. The example

```
\tikz \draw (0,0) -- (1,1);
{
\tikzset{external/force remake}
\begin{tikzpicture}
    \draw (0,0) circle(5pt);
\end{tikzpicture}
}
\tikz \draw (0,0) -- (1,1);
```

will only apply force remake to the second figure.

Up-to-date checks are applied for mode=convert with system call and the makefile generated by mode=list and make.

```
/tikz/external/remake next=\{\langle boolean \rangle\}
```

(default true)

A variant of force remake which applies only to the next image.

```
/tikz/external/export next=\{\langle boolean \rangle\}
```

(default true)

A boolean which can be used to disable the export mechanism for single pictures.

```
\texttt{/tikz/external/export} = \{\langle boolean \rangle\}
```

(no default, initially true)

A boolean which can be used to disable the export mechanism for all pictures inside of the current TeX-scope.

```
\begin{document}
\begin{tikzpicture} % will be exported
...
\end{tikzpicture}

{
   \tikzset{external/export=false}
   \begin{tikzpicture} % won't be exported
...
\end{tikzpicture}
...
}

\begin{tikzpicture} % will be exported
...
\end{tikzpicture} % will be exported
...
\end{tikzpicture}
\end{document}
```

For LATEX, the feature lasts until the next $\ensuremath{\mbox{ }}$ (this holds for every call to \tikzset).

/tikz/external/up to date check= $\{\langle choice \rangle\}$

(no default, initially md5)

The external lib has to decide when some existing figure is up-to-date. In such a case, it can be used without remaking it. Outdated pictures will be remade.

The key up to date check allows to choose among a couple of heuristics which are supposed to catch the most important reasons to remake a figure.

The up to date check can be overrule by any of the force remake or remake next keys: if one of them is true, the figure is not up-to-date.

The choice simple is based on the existance of the file: the file is up-to-date if and only if it exists.

The choice md5 generates an MD5 checksum of the picture for which the up-to-date check is running. The MD5 is compared against the MD5 of the previous run, which, in turn, will be written into an extra file with the extension .md5. This file will be modified if and only if the MD5 comparison indicates a difference. The MD5 computation is based on the pdfTEX method \pdfmdfivesum. If it is unavailable for some reason, the choice diff will be used instead.

The choice diff is the same as MD5 – except that it compares the picture content as-is instead of a hash. The .md5 file will be used to compare an old version with the current one – but its content is some "normalized" version of the picture for internal use.

Attention: the content—based strategies md5 and diff operate on the picture content—and only on the picture content. Here, "picture content" only includes the top—level tokens; no expansion is applied and no included files are part of the strategies. If you change preamble styles, you have to rebuild the figures manually (for example by deleting the generated graphics files). If you have include files, consider using \tikzpicturedependsonfile and its variants. Since this key provides heuristics, you should always remake your figures before you finally publish your document.

The up to date check is applied for mode=convert with system call and mode=list and make.

\tikzexternaldisable

Allows to disable the complete externalization. While export next will still collect the contents of picture environments, this command uninstalls the hooks for the external library completely. Thus, nested picture environments or environments where \end{tikzpicture} is not directly reachable won't produce compilation failures – although it is not possible to externalize them automatically.

The externalization remains disabled until the end of the next TeX group (or environment) or until the next call to \tikzexternalenable.

\tikzexternalenable

Re-enables a previously running externalization after \tikzexternaldisable.

33.4.4 Customizing the Externalization

```
/tikz/external/figure list=\{\langle boolean \rangle\}
```

(no default, initially true)

A boolean which configures whether a figure list shall be generated. A figure list is an output file named $\{\langle jobname \rangle\}$.figlist which is filled with file names of each figure, one per line.

This file is not used by T_EX anymore, its purpose is to issue the required conversion commands pdflatex -jobname { $\langle picture\ file\ name \rangle$ } { $\langle main\ file \rangle$ } manually (or in a script). See section 33.4.5 for the details about the expected system call (or activate mode=convert with system call and inspect your log file).

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}
\usetikzlibrary{external}
\tikzexternalize[
  mode=graphics if exists,
  figure list=true,
  prefix=figures/]
\begin{document}
\tikzsetnextfilename{trees}
\begin{tikzpicture}
  \node {root}
   child {node {left}}
   child {node {right}
     child {node {child}}
     child {node {child}}
\end{tikzpicture}
\tikzsetnextfilename{simple}
A simple image is \tikz \fill (0,0) circle(5pt);.
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
\end{document}
```

pdflatex main

generates main.figlist containing

```
figures/trees
figures/simple
figures/main-figure0
```

$\texttt{/tikz/external/mode=}\{\langle choice \rangle\}$

(no default, initially convert with system call)

Configures what to do with TikZ pictures (unless we are currently externalizing one particular image, in that case, these modes are ignored).

The preconfigured mode convert with system call checks whether external graphics files are upto-date and includes them if that is the case. Any picture which is not up-to-date will be generated automatically using a system call. The system call can be configured using the system call template. The up-to-date check is applied according to the up to date check key. As soon as convert with system call is set, the figure list will be disabled – such a file is not required. In case you still need or want it, you can enable it after setting mode.

Please note that system calls may be disabled for security reasons. For pdflatex, they can be enabled using

```
pdflatex -shell-escape
```

while other T_FX variants may need other switches. The feature is sometimes called \write18.

The choice only graphics always tries to replace pictures with external graphics. It is an error if the graphics file does not exist.

The choice no graphics (or, equivalently, only pictures) typesets TikZ pictures without checking for external graphics.

A mixture is graphics if exists, it checks whether a suitable graphics file exists and includes it if that is the case. If it does not exist, the picture is typeset using T_FX.

Mode list only skips every TikZ picture; it only generates the file $\{\langle main\ file \rangle\}$. figlist containing file names for every picture, the contents of any picture environment is thrown away and a replacement text is shown. This implies figure list=true. See also the list and make mode which includes available graphics.

The mode list and make is similar to list only: it generates the same file $\{\langle main\ file \rangle\}$.figlist, but any images which exist already are included as graphics instead of ignoring them. Furthermore, this mode generates an additional file: $\{\langle main\ file \rangle\}$.makefile. This allows to use a work flow like

```
% step 1: generate main.makefile:
pdflatex main
% step 2: generate ALL graphics on 2 processors:
make -j 2 main.makefile
% step 3: include the graphics:
pdflatex main
```

This last make method is optional: list and make just assumes that images are generated somehow (not necessarily with the generated makefile). The generated makefile allows parallel externalization of graphics on multi-core systems and it supports any file dependencies configured with \tikzpicturedependsonfile. Furthermore, it respects the force remake and remake next keys.

```
/tikz/external/verbose IO=\{\langle boolean \rangle\}
```

(no default, initially true)

A boolean which configures whether I/O operations shall be listed in the logfile.

```
tikz/external/verbose optimize={\langle boolean \rangle}
```

(no default, initially true)

A boolean which configures whether optimization operations shall be listed in the logfile.

```
\texttt{/tikz/external/verbose=}\{\langle boolean \rangle\}
```

(no default, initially true)

Sets all verbosity flags to $\langle boolean \rangle$.

```
\texttt{/tikz/external/optimize=}\{\langle boolean \rangle\}
```

(no default, initially true)

Configures whether the conversion process shall be optimized. This affects only the case when \jobname differs from the main file name, i.e. when single pictures are converted.

In that case, the main file is compiled as usual - but everything except the selected picture is thrown away. If optimization is enabled, all other pictures won't be processed at all. Furthermore, expensive commands which do not contribute to the selected picture will be thrown away as well.

The default implementation discards \includegraphics commands which are *not* inside of the selected picture to reduce conversion time.

It is possible to add commands which shall be optimized away, see below.

```
\label{eq:command} $$ \text{tikz/external/optimize command away=} $$ (no default) $$
```

Installs commands to optimize $\langle \c command \rangle$ away. As is described above, optimization applies to the case when single pictures are converted: one usually doesn't need to process (probably expensive) commands which do not contribute to the selected picture.

The argument $\{\langle required\ argument\ count \rangle\}$ is either empty or a non-negative integer between 0 and 9. It denotes the number of arguments which should be consumed after $\langle \backslash command \rangle$. In any case, one argument in square brackets after the command will be recognized as well. To be more precise, the following cases for arguments of $\langle \backslash command \rangle$ are supported:

- 1. If $\{\langle required\ argument\ count \rangle\}$ is empty (the default), $\langle \langle command \rangle$ may take one optional argument in square brackets and one in curly braces (which is also optional).
- 2. If $\{\langle required\ argument\ count \rangle\}$ is not empty, $\{\langle \backslash command \rangle\}$ may take one optional argument in square brackets. Furthermore, it expects exactly $\{\langle required\ argument\ count \rangle\}$ following arguments.

Example:

```
\tikzset{external/optimize command away=\includegraphics}
```

```
\newcommand{\myExpensiveMacro}[1]{Very expensive!}
```

\tikzset{external/optimize command away=\myExpensiveMacro}

\newcommand{\myExpensiveMacroWithThreeArgs}[3]{Very expensive!}

\tikzset{external/optimize command away={\myExpensiveMacroWithThreeArgs}{3}}

```
% A command with optional argument:
\newcommand{\aFurtherExample}[3][]{Very expensive!}

% consume only two arguments: the first optional one will be processed
% anyway:
\tikzset{external/optimize command away={\myExpensiveMacroWithThreeArgs}{2}}
```

The argument $\langle \setminus command \rangle$ must be the name of a single macro. Any occurrence of this macro, together with its arguments, will be removed.

```
\begin{tikzpicture}
% this picture is currently converted!
\end{tikzpicture}

This here is outside of the converted picture and contains \myExpensiveMacro. It will be discarded.

This call: \myExpensiveMacro[argument=value]{Argument} as well.

And this here: \myExpensiveMacro{Argument} also.
```

The default is to optimize \includegraphics away.

This key is actually a style which sets the optimize/install and optimize/restore keys.

/tikz/external/optimize/install

(no value)

A command key which contains code to install optimizations. You can append code here (or clear the macro) if you need to modify the optimization.

/tikz/external/optimize/restore

(no value)

A command key which contains code to undo optimizations. You can append code here (or clear the macro) if you need to modify the optimization.

/tikz/external/only named= $\{\langle boolean \rangle\}$

(no default, initially false)

If enabled, only pictures for which file names have been set explicitly using \tikzsetnextfilename will be considered, no file names will be generated automatically.

/pgf/images/include external

 $(initially \pgfimage{#1})$

This command key constitutes the public interface to exchange the \includegraphics command used for the image inclusion. If can be overwritten using include external/.code= $\{\langle T_{EX} \ code \rangle\}$.

Its description can be found in the corresponding basic layer documentation on page 812.

Just one example here: you can use

```
\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{}\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{$\positive{}\positive{$\positive{$\positive{$\positive{$\positive{$\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive{}\positive
```

to manually change the viewport (bounding box) for included graphics.

Another example (of probably limited use) is

```
\pgfkeys{/pgf/images/include external/.code={\href{file:#1}{\pgfimage{#1}}}}
```

which will generate a clickable hyperlink around the image. Clicking on it opens the single exported file 13 .

If you want to limit the effects of this key to just one externalized figure, use

 $^{^{13}}$ This requires all external graphics files in the same base directory as the main .pdf file.

```
\tikzifexternalizing{\langle true\ code \rangle}{\langle false\ code \rangle}
```

This command can be used to check whether an image is currently written to its separate graphics file (if the "grab" procedure is running). If so, the $\{\langle true\ code \rangle\}$ will be executed. If not, that means if the main document is being typeset normally, the $\{\langle false\ code \rangle\}$ will be invoked.

This command must be used after \tikzexternalize.

```
\tikzifexternalizingnext{\langle true\ code \rangle} {\langle false\ code \rangle}
```

Like \tikzifexternalizing, but this variant also checks if the next following figure is the one which is about to be written to its separate graphics file.

33.4.5 Details About The Process

The standard run pdflatex \(\text{main document} \) causes the external library to check every occurrence of \(\text{begin{tikzpicture}} \) and every \(\text{tikz shortcommand}. \) If it finds a picture which shall be exported, it queries the respective file name and checks whether the file exists already. If so, it includes the external graphics. If not, it requires an externalization which can be done automatically (the default), semi-automatically (with mode=list and make) or manually (by issuing the requires system calls somehow).

The library can detect whether it runs in "conversion mode", i.e. if it should only process a single image. To do so, it checks whether the internal macro \tikzexternalrealjob exists. If so, its contents is assumed to be \(\text{main document} \) (without the suffix .tex). Usually, this macro is set by the conversion system call,

```
pdflatex -jobname "main-figure0" "\def\tikzexternalrealjob{main}\input{main}"
```

where main-figure0 is the picture we are currently externalizing and main.tex is the main document.

As soon as "conversion mode" has been detected, PGF changes the output routine. The complete file main.tex is processed as normal, but only the part of the desired picture will be written to the output file, in our case main-figure0.pdf. The rest of the document is silently thrown away. Of course, such a conversion process is quite expensive since we need to do it for every picture. Since everything except the current picture is thrown away, the library skips all other pictures. Furthermore, any \includegraphics commands which are outside of the converted TikZ-picture will be skipped as well. Thus, the conversion process should be much faster than typesetting the complete document, but it still requires its time. Eventually, the call \input{main} returns and the picture is ready. From this point on, the external graphics will be used.

There is another possibility to communicate $\langle main\ document \rangle$ to the subprocess performing the externalization: namely to write '\tikzexternalize{main}' into the document. In this case, the conversion system call will be

```
pdflatex -jobname "main-figure0" "main"
```

and the contents of \tikzexternalrealjob is set automatically. This case is detected by \tikzexternalize, and the system call is updated automatically (by patching its \texsource template argument). It is not necessary to change the system call manually.

The sequence in which system calls are performed and the decision whether they are issued automatically is governed by the mode key, consult its documentation for details.

33.5 Using External Graphics Without PGF Installed

Given that every picture has been exported correctly, one may want to compile a file without PGF and TikZ installed. TikZ comes with a minimal package which contains just enough commands to replace every tikzpicture environment and the \tikz short command with the appropriate external graphics. It can be found at

```
latex/pgf/utilities/tikzexternal.sty
```

and needs to be used instead of \usepackage{tikz}. So, we uncomment \usepackage{tikz} and our example from the beginning becomes

```
\documentclass{article}
% main document, called main.tex
%\usepackage{tikz}
\usepackage{graphicx}
\usepackage{tikzexternal}
%\usetikzlibrary{external}
\tikzexternalize
\begin{document}
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
\end{tikzpicture}
A simple image is \tikz \fill (0,0) circle(5pt);.
Furthermore, we might want to draw \tikz[baseline]\draw (0,-1) rectangle (1,1);
\end{document}
```

where the following files are necessary to compile the document:

```
tikzexternal.sty
main.tex
main-figure0.pdf
main-figure1.pdf
main-figure2.pdf
```

If there are any '.dpth' files, for example main-figure2.dpth, these files are also required. They contain information for the TikZ baseline option (or \labels inside external graphics).

Just copy the .sty file into the directory of your main.tex file and use it as part of your document.

Please keep in mind, that only tikzpicture environments and \tikz short images are available within the externalization framework. Additionally, calls to \tikzset and \pgfkeys won't lead to compilation errors because they are simply ignored. But since pgfkeys is not available, any option supplied to \tikzexternalize is ignored.

Attention: Since the simple replacement \usepackage{tikzexternal} doesn't support the key-value interface, you need to use \tikzsetexternalprefix instead of the prefix option and \tikzsetfigurename instead of the figure name option since \tikzset is not available in such a context.

Remark: Some of the features of this library are mainly useful to improve the speed of successive document compilations. In other words: you can't use all features in this context, keep it simple.

33.6 eps Graphics Export

It is also possible to use eps graphics instead of PDF files. There are different ways to produce them, for example to use pdflatex and call pdftops -eps $\{\langle pdf \ file \rangle\}\ \{\langle eps \ file \rangle\}\$ afterwards. You could add this command to the system call option.

Alternatively, you can use latex and dvips for image conversion as is explained for the system call option, see page 384. See the documentation for the basic level externalization in section 88 for restrictions of other drivers.

33.7 Bitmap Graphics Export

Occasionally, you may have an extremely large graphics which takes long times to render. It might be interesting to generate a bitmap (raster) image, which displays much faster (for example in a presentation). I have used this feature to speed-up the display of large shadings.

The external library can be customized to export bitmap images – with the help of external programs. Due to the dependence of external programs, you may need to adjust these commands manually. For example, on my computer, the ImageMagick Suite is installed which comes with the convert tool. Together with pdflatex, I can define the following style:

```
\tikzset{
   \ensuremath{\text{\%}} Defines a custom style which generates BOTH, .pdf and .png export
    % but prefers the .png on inclusion.
   % This style is not pre-defined, you may need to copy-paste and
    % adjust it.
   png export/.style={
        external/system call/.add=
            {}
            {; convert -density 300 -transparent white "\image.pdf" "\image.png"},
        /pgf/images/external info,
        /pgf/images/include external/.code={%
            \includegraphics
                 [width=\pgfexternalwidth,height=\pgfexternalheight]
                 {##1.png}%
        },
   }
```

The example above defines a new style called 'png export' which, when it is set with \tikzset{png export} somewhere in the document, modifies the configuration for both file generation and file input. The file generation is modified by appending the ImageMagick command to system call (separated by ';' as usual on Linux). This is, in principle, enough to generate a .png file. The include external command is overwritten such that it uses the .png file instead of the .pdf file (which exists as well in the configuration above). But since a .png file can have a much higher resolution than the desired image dimensions, we have to add width and height explicitly. Usually, the external library does not provide size information (it is unnecessary for .pdf or .eps since these formats have their bounding box information). To enable size information, the style uses the external info key, which, in turn, provides the \pgfexternalwidth and \pgfexternalwidth commands.

Now we can use \tikzset{png export} either document-wide or just for one particular image. The configuration remains in effect until the end of the current environment (or until the next closing curly brace '}').

```
/pgf/images/external info=\{\langle boolean \rangle\}
```

(no default, initially false)

If this key is activated, the size for any externalized image will be stored explicitly into the associated .dpth file.

When the file is included by \pgfincludeexternalgraphics (or automatically by the external library), the width is available as \pgfexternalwidth and the height as \pgfexternalheight.

33.8 Compatibility Issues

33.8.1 References In External Pictures

It is allowed if a picture contains references, for example \tikz \node {Reference to \ref{a:label}};.

There is just one issue: if the main job is currently compiling, its .aux file is not in its final state (even worse: it may not be readable at all). The picture externalization, however, needs the main .aux file to query any references.

Thus, you will need to invoke pdflatex -jobname $\langle image \rangle$ $\langle mainfile \rangle$ manually for any image which contains references.

This problem arises only for mode=convert with system call. In this case, the external library creates a special \jobname.auxlock file to check whether the main .aux file is currently usable.

33.8.2 Compatibility With Other Libraries or Packages

The external library has the following compatibility issues:

- 1. The external library comes with special support for \usetikzlibrary{fadings}: the fadings library may define local pictures which would be externalized (although they shouldn't). There is special handling to suppress this bug if \tikzexternalize is called after \usetikzlibrary{fadings} or if all fadings are defined before \tikzexternalize.
- 2. Problems have been reported when using \tikzexternalize (or the basic layer externalization) together with \usepackage{glossary}. This problem disappears if \tikzexternalize is called before \usepackage{glossary}.

3. Problems with \usepackage{pdfpages} and \usepackage{vmargin}: The external library replaces the current shipout routine of TEX during its externalization. This might raise problems with other packages which also manipulate the shipout routine (like the mentioned ones).

To fix those problems, use

```
\usetikzlibrary{external}

\tikzifexternalizing{%
% don't include package XYZ here
}{%
\usepackage{pdfpages}
\usepackage{vmargin}
...
}%
```

This uses the requested packages for the main document, but not for the single, exported graphics.

In general, the \tikzifexternalizing feature might be used to solve package conflicts and the \tikzexternaldisable and \tikzexternalenable features can be used to solve problems with single pictures.

33.8.3 Compatibility With Bounding Box Restrictions

Bounding box restrictions provide no problem when used with eps graphics. However, they pose problems for pdflatex, so you may need to use the latex / dvips combination if you use bounding box restrictions and externalization. Currently, the only possibility for bounding box restrictions and pdflatex is to use a combination of trim left / trim right / baseline: these keys do not really truncate the bounding box, they only store horizontal and vertical shifts (also see the trim lowlevel key in this context).

33.8.4 Interoperability With The Basic Layer Externalization

This library is fully compatible with \beginpgfgraphicnamed...\endpgfgraphicnamed environments. However, you will need to use the export next=false key to avoid conflicts:

```
\beginpgfgraphicnamed{picture4}
\tikzset{external/export next=false}
\begin{tikzpicture}
\draw (0,0) -- (4,4);
\end{tikzpicture}
\endpgfgraphicnamed
```

Please keep in mind that file prefixes do not apply to the basic layer.

34 Fading Library

```
\label{thm:configure} $$ \usepgflibrary[fadings] % MEX and plain TeX and pure pgf $$ \usetikzlibrary[fadings] % ConTeXt and pure pgf $$ \usetikzlibrary[fadings] % MEX and plain TeX when using TikZ $$ \usetikzlibrary[fadings] % ConTeXt when using TikZ $$
```

The package defines a number of fadings, see Section 21 for an introduction. The TikZ version defines special TikZ commands for creating fadings. These commands are explained in Section 21.

Fading name	Example (solid blue faded on checkerboard)
west	88
east	
north	
south	
circle with fuzzy edge 10 percent	
circle with fuzzy edge 15 percent	
circle with fuzzy edge 20 percent	
fuzzy ring 15 percent	

35 Fitting Library

```
\usetikzlibrary{fit} % MEX and plain TEX \usetikzlibrary[fit] % ConTEXt
```

The library defines (currently only two) options for fitting a node so that it contains a set of coordinates.

When you load this library, the following options become available:

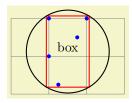
```
/tikz/fit=\langle coordinates or nodes\rangle
```

(no default)

This option must be given to a **node** path command. The $\langle coordinates \ or \ nodes \rangle$ should be a sequence of TikZ coordinates or node names, one directly after the other without commas (like with the plot coordinates path operation). Examples are (1,0) (2,2) or (a) (1,0) (b), where a and b are nodes.

For this sequence of coordinates, a minimal bounding box is computed that encompasses all the listed $\langle coordinates \ or \ nodes \rangle$. For coordinates in the list, the bounding box is guaranteed to contain this coordinate, for nodes it is guaranteed to contain the east, west, north and south anchors of the node. In principle (the details will be explained in a moment), things are now set up such that the text box of the node will be exactly this bounding box.

Here is an example: We fit several points in a rectangular node. By setting the inner sep to zero, we see exactly the text box of the node. Then we fit these points again in a circular node. Note how the circle encompasses exactly the same bounding box.



Every time the fit option is used, the following style is also applied to the node:

/tikz/every fit

(style, initially empty)

Set this style to change the appearance of a node that uses the fit option.

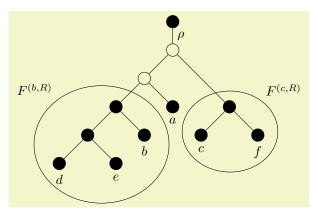
The exact effects of the fit option are the following:

- A minimal bounding box containing all coordinates is computed. Note that if a coordinate like (a) is used that contains a node name, this has the same effect as explicitly providing the (a.north) and (a.south) and (a.west) and (a.east). If you wish to refer only to the center of the a node, use (a.center) instead.
- 2. The text width option is set to the width of this bounding box.
- 3. The align=center option is set.
- 4. The anchor is set to center.
- 5. The at position of the node is set to the center of the computed bounding box.
- 6. After the node has been typeset, its height and depth are adjusted such that they add up to the height of the computed bounding box and such that the text of the node is vertically centered inside the box.

The above means that, generally speaking, if the node contains text like box in the above example, it will be centered inside the box. It will be difficult to put the text elsewhere, in particular, changing the anchor of the node will not have the desired effect. Instead, what you should do is to create a node with the fit option that does not contain any text, give it a name, and then use normal nodes to add text at the desired positions. Alternatively, consider using the label or pin options.

Suppose, for instance, that in the above example we want the word "box" to appear inside the box, but at its top. This can be achieved as follows:

Here is a real-life example that uses fitting:

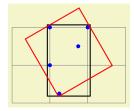


```
\begin{tikzpicture}
  [vertex/.style={minimum size=2pt,fill,draw,circle},
   open/.style={fill=none},
   sibling distance=1.5cm, level distance=.75cm,
   every fit/.style={ellipse,draw,inner sep=-2pt},
   leaf/.style={label={[name=#1]below:$#1$}},auto]
  \node [vertex] (root) {}
  child { node [vertex,open] {}
   child { node [vertex,open] {}
     child { node [vertex] (b's parent) {}
       child { node [vertex] {}
         child { node [vertex,leaf=d] {} }
         child { node [vertex,leaf=e] {} } }
       child { node [vertex,leaf=b] {} } }
     child { node [vertex,leaf=a] {} } }
   child { node [coordinate] {}
     child[missing]
     child { node [vertex] (f's parent) {}
       child { node [vertex,leaf=c] {} }
       child { node [vertex,leaf=f] {} } } }
   edge from parent node {$\rho$} };
  \label=above \ left: F^{(b,R)} \ \{\};
  \label=above \ right: F^{(c,R)} \]
\end{tikzpicture}
```

/tikz/rotate fit=\langle\rangle

(no default, initially 0)

This key fits $\langle coordinates\ or\ nodes \rangle$ inside a node that is rotated by $\langle angle \rangle$. As a side effect, it also sets the /tikz/rotate key.



```
\begin{tikzpicture}[inner sep=0pt,thick,
   dot/.style={fill=blue,circle,minimum size=3pt}]
   \draw[help lines] (0,0) grid (3,2);
   \node[dot] (a) at (1,1) {};
   \node[dot] (b) at (2,2) {};
   \node[dot] (c) at (1,2) {};
   \node[dot] (d) at (1.25,0.25) {};
   \node[dot] (e) at (1.75,1.5) {};
   \node[draw, fit=(a) (b) (c) (d) (e)] {};
   \node[draw=red, rotate fit=30, fit=(a) (b) (c) (d) (e)] {};
   \end{tikzpicture}
```

36 Fixed Point Arithmetic Library

```
\usepgflibrary{fixedpointarithmetic} % MEX and plain TeX and pure pgf \usepgflibrary[fixedpointarithmetic] % ConTeXt and pure pgf \usetikzlibrary{fixedpointarithmetic} % MEX and plain TeX when using TikZ \usetikzlibrary[fixedpointarithmetic] % ConTeXt when using TikZ
```

This library provides an interface to the LATEX package fp for fixed point arithmetic. In addition to loading this library you must ensure fp is loaded, otherwise errors will occur.

36.1 Overview

Whilst the mathematical engine that comes with PGF is reasonably fast and flexible when it comes to parsing, the accuracy tends to be fairly low, particularly for expressions involving many operations chained together. In addition the range of values that can be computed is very small: ± 16383.99999 . Conversely, the fp package has a reasonably high accuracy, and can perform computations over a wide range of values (approximately $\pm 9.999 \times 10^{17}$), but is comparatively slow and not very flexible, particularly regarding parsing.

This library enables the combination of the two: the flexible parser of the PGF mathematical engine with the evaluation accuracy of fp. There are, however, a number of important points to bear in mind:

- Whilst fp supports very large numbers, PGF and TikZ do not. It is possible to calculate the result of 2^20 or 1.2e10+3.4e10, but it is not possible to use these results in pictures directly without some "extra work".
- The PGF mathematical engine will still be used to evaluate lengths, such as 10pt or 3em, so it is not possible for an length to exceed the range of values supported by TEX-dimensions (±16383.99999pt), even though the resulting expression is within the range of fp. So, for example, one can calculate 3cm*10000, but not 3*10000cm.
- Not all of the functions listed in Section 71, have been mapped onto fp equivalents. Of those that have been, it is not guaranteed that functions will perform in the same way as they do in PGF. Reference should be made to the documentation for fp.
- In PGF, trigonometric functions such as sin and cos assume arguments are in degrees, and functions such as asin and acos return results in degrees. Although fp uses radians for such functions, PGF automatically converts arguments from degrees to radians, and converts results from radians to degrees, to ensure everything "works properly".
- The overall speed will actually be slower than using PGF mathematical engine. The calculating power of fp comes at the cost of an increased processing time.

36.2 Using Fixed Point Arithmetic in PGF and TikZ

The following key is provided to use fp in PGF and TikZ:

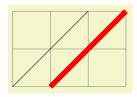
```
 \begin{tabular}{ll} \be
```

This key will set the key path to /pgf/fixed point, and execute $\langle options \rangle$. Then it will install the necessary commands so that the PGF parser will use fp to perform calculations. The best way to use this key is as an argument to a scope or picture. This means that fp does not always have to be used, and PGF can use its own mathematical engine at other times, which can lead to a significant reduction in the time for a document to compile.

Currently there are only a few keys key supported for $\langle options \rangle$:

```
/pgf/fixed point/scale results = \langle factor \rangle (no default)
```

As noted above, fp can process a far greater range of numbers than PGF and TikZ. In order to use results from fp in a {pgfpicture} or a {tikzpicture} they need to be scaled. When this key is used PGF will scale results of any evaluation by $\langle factor \rangle$. However, as it is not desirable for every part of every expression to be scaled, scaling will only take place if a special prefix * is used. If * is used at the beginning of an expression the evaluation of the expression will evaluated and then multiplied by $\langle factor \rangle$.



```
\begin{tikzpicture}[fixed point arithmetic={scale results=10^-6}]
\draw [help lines] grid (3,2);
\draw (0,0) -- (2,2);
\draw [red, line width=4pt] (*1.0e6,0) -- (*3.0e6,*2.0e6);
\end{tikzpicture}
```

A special case of scaling involves plots of data containing large numbers from files. It is possible to "pre-process" a file, typically using the application that generates the data, to either precede the relevant column with * or to perform the scaling as part of the calculation process. However, it may be desirable for the data in a plot to appear in a table as well, so, two files would be required, one pre-processed for plotting, and one not. This extra work may be undesirable so the following keys are provided:

```
/pgf/fixed point/scale file plot x=\langle factor \rangle (no default)
```

This key will scale the first column of data read from a file before it is plotted. It is independent of the scale results key.

```
/pgf/fixed point/scale file plot y=\langle factor \rangle (no default)
```

This key will scale the second column of data read from a file before it is plotted.

```
/pgf/fixed point/scale file plot z=\langle factor \rangle (no default)
```

This key will scale the third column of data read from a file before it is plotted.

37 Floating Point Unit Library

by Christian Feuersänger

```
\usepgflibrary{fpu} % MEX and plain TeX and pure pgf \usepgflibrary[fpu] % ConTeXt and pure pgf \usetikzlibrary{fpu} % MEX and plain TeX when using TikZ \usetikzlibrary[fpu] % ConTeXt when using TikZ
```

The floating point unit (fpu) allows the full data range of scientific computing for use in PGF. Its core is the PGF math routines for mantissa operations, leading to a reasonable trade—of between speed and accuracy. It does not require any third—party packages or external programs.

37.1 Overview

The fpu provides a replacement set of math commands which can be installed in isolated placed to achieve large data ranges at reasonable accuracy. It provides at least 14 the IEEE double precision data range, $-1 \cdot 10^{324}, \dots, 1 \cdot 10^{324}$. The absolute smallest number bigger than zero is $1 \cdot 10^{-324}$. The FPU's relative precision is at least $1 \cdot 10^{-4}$ although operations like addition have a relative precision of $1 \cdot 10^{-6}$.

Note that the library has not really been tested together with any drawing operations. It should be used to work with arbitrary input data which is then transformed somehow into PGF precision. This, in turn, can be processed by PGF.

37.2 Usage

```
/pgf/fpu={\langle boolean \rangle} (default true)
```

This key installs or uninstalls the FPU. The installation exchanges any routines of the standard math parser with those of the FPU: \pgfmathadd will be replaced with \pgfmathfloatadd and so on. Furthermore, any number will be parsed with \pgfmathfloatparsenumber.

```
1Y2.0e0] \pgfkeys{/pgf/fpu} \pgfmathparse{1+1}\pgfmathresult
```

The FPU uses a lowlevel number representation consisting of flags, mantissa and exponent¹⁵. To avoid unnecessary format conversions, \pgfmathresult will usually contain such a cryptic number. Depending on the context, the result may need to be converted into something which is suitable for PGF processing (like coordinates) or may need to be typeset. The FPU provides such methods as well.

Use fpu=false to deactivate the FPU. This will restore any change. Please note that this is not necessary if the FPU is used inside of a T_FX group – it will be deactivated afterwards anyway.

It does not hurt to call fpu=true or fpu=false multiple times.

Please note that if the fixed point arithmetics library of PGF will be activated after the FPU, the FPU will be deactivated automatically.

```
/pgf/fpu/output format=float|sci|fixed (no default, initially float)
```

This key allows to change the number format in which the FPU assigns \pgfmathresult.

The predefined choice float uses the low-level format used by the FPU. This is useful for further processing inside of any library.

```
1Y2.17765411e23 \pgfkeys{/pgf/fpu,/pgf/fpu/output format=float} \pgfmathparse{exp(50)*42}\pgfmathresult
```

The choice sci returns numbers in the format $\langle mantissa \rangle e \langle exponent \rangle$. It provides almost no computational overhead.

```
5.6154816e14 \pgfkeys{/pgf/fpu,/pgf/fpu/output format=sci} \pgfmathparse{4.22e-8^-2}\pgfmathresult
```

¹⁴To be more precise, the FPU's exponent is currently a 32 bit integer. That means it supports a significantly larger data range than an IEEE double precision number – but if a future T_EX version may provide lowlevel access to doubles, this may change.

 $^{^{15}}$ Users should always use high level routines to manipulate floating point numbers as the format may change in a future release.

The choice fixed returns normal fixed point numbers and provides the highest compatibility with the PGF engine. It is activated automatically in case the FPU scales results.

0.00000999985 \pgfkeys{/pgf/fpu,/pgf/fpu/output format=fixed} \pgfmathparse{sqrt(1e-12)}\pgfmathresult

/pgf/fpu/scale results= $\{\langle scale \rangle\}$

(no default)

A feature which allows semi-automatic result scaling. Setting this key has two effects: first, the output format for any computation will be set to fixed (assuming results will be processed by PGF's kernel). Second, any expression which starts with a star, *, will be multiplied with $\{\langle scale \rangle\}$.

```
\label{eq:continuous_pgf_fpu_scale} $$ \begin{array}{ll} \text{(no default)} \\ \text{pgf/fpu/scale file plot y={$\langle scale \rangle$}} \\ \text{pgf/fpu/scale file plot z={$\langle scale \rangle$}} \\ \end{array} $$ \begin{array}{ll} \text{(no default)} \\ \text{(no default)} \\ \end{array} $$
```

These keys will patch PGF's plot file command to automatically scale single coordinates by $\{\langle scale \rangle\}$. The initial setting does not scale plot file.

This command can be used to execute either $\langle true\text{-}code \rangle$ or $\langle false\text{-}code \rangle$, depending on whether the FPU has been activated or not.

37.3 Comparison to the fixed point arithmetics library

There are other ways to increase the data range and/or the precision of PGF's math parser. One of them is the fp package, preferable combined with PGF's fixed point arithmetic library. The differences between the FPU and fp are:

- The FPU supports at least the complete IEEE double precision number range, while fp covers only numbers of magnitude $\pm 1 \cdot 10^{17}$.
- The FPU has a uniform relative precision of about 4–5 correct digits. The fixed point library has an absolute precision which may perform good in many cases but will fail at the ends of the data range (as every fixed point routines does).
- The FPU has potential to be faster than fp as it has access to fast mantissa operations using PGF's math capabilities (which use TEX registers).

37.4 Command Reference and Programmer's Manual

37.4.1 Creating and Converting Floats

```
\verb|\pgfmathfloatparsenumber{} \{\langle x \rangle\}|
```

Reads a number of arbitrary magnitude and precision and stores its result into \pgfmathresult as floating point number $m \cdot 10^e$ with mantissa and exponent base 10.

The algorithm and the storage format is purely text-based. The number is stored as a triple of flags, a positive mantissa and an exponent, such as

```
1Y2.0e0] \pgfmathfloatparsenumber{2} \pgfmathresult
```

Please do not rely on the low-level representation here, use \pgfmathfloattomacro (and its variants) and \pgfmathfloatcreate if you want to work with these components.

The flags encoded in \pgfmathresult are represented as a digit where '0' stands for the number $\pm 0.10^0$, '1' stands for a positive sign, '2' means a negative sign, '3' stands for 'not a number', '4' means $+\infty$ and '5' stands for $-\infty$.

The mantissa is a normalized real number $m \in \mathbb{R}$, $1 \le m < 10$. It always contains a period and at least one digit after the period. The exponent is an integer.

Examples:

Flags: 0; Mantissa 0.0; Exponent 0.

```
\pgfmathfloatparsenumber{0}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 2.0; Exponent -1.

```
\pgfmathfloatparsenumber{0.2}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 4.2; Exponent 1.

```
\pgfmathfloatparsenumber{42}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 2.05; Exponent 3.

```
\pgfmathfloatparsenumber{20.5E+2}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 1.0; Exponent 6.

```
\pgfmathfloatparsenumber{1e6}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 5.21513; Exponent -11.

```
\pgfmathfloatparsenumber{5.21513e-11}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

The argument $\{\langle x \rangle\}$ may be given in fixed point format or the scientific 'e' (or 'E') notation. The scientific notation does not necessarily need to be normalized. The supported exponent range is (currently) only limited by the TEX-integer range (which uses 31 bit integer numbers).

```
pgf/fpu/handlers/empty number={\langle input\rangle}{\langle unreadable part\rangle} (no default)
```

This command key is invoked in case an empty string is parsed inside of \pgfmathfloatparsenumber. You can overwrite it to assign a replacement \pgfmathresult (in float!).

The initial setting is to invoke invalid number, see below.

```
/pgf/fpu/handlers/invalid number=\{\langle input\rangle\}\{\langle unreadable \ part\rangle\}  (no default)
```

This command key is invoked in case an invalid string is parsed inside of \pgfmathfloatparsenumber. You can overwrite it to assign a replacement \pgfmathresult (in float!).

The initial setting is to generate an error message.

```
pgf/fpu/handlers/wrong lowlevel format={\langle input\rangle}{\langle unreadable part\rangle} (no default)
```

This command key is invoked whenever \pgfmathfloattoregisters or its variants encounter something which is not a properly formatted lowlevel floating point number. As for invalid number, this key may assign a new \pgfmathresult (in floating point) which will be used instead of the offending $\{\langle input \rangle\}$.

The initial setting is to generate an error message.

$\protect\operatorname{\mathsf{pgfmathfloatqparsenumber}}\{\langle x\rangle\}$

The same as \pgfmathfloatparsenumber, but does not perform sanity checking.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Converts a number in floating point representation to a fixed point number. It is a counterpart to $\protect\$

Flags: 1; Mantissa 5.2; Exponent $-4 \rightarrow 0.00052$

```
\pgfmathfloatparsenumber{0.00052}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
$\to$
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

Flags: 1; Mantissa 1.23456; Exponent $6\rightarrow$ 1234560.00000000

```
\pgfmathfloatparsenumber{123.456e4}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
$\to$
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

$\protect\operatorname{\mathsf{pgfmathfloattoint}}\{\langle x \rangle\}$

Converts a number from low-level floating point representation to an integer (by truncating the fractional part).

```
123456 \pgfmathfloatparsenumber{123456} \pgfmathfloattoint{\pgfmathresult} \pgfmathresult
```

See also \pgfmathfloatint which returns the result as float.

$\protect\operatorname{\mathtt{f pgfmathfloattosci}}\{\langle float \rangle\}$

Converts a number from low-level floating point representation to scientific format, 1.234e4. The result will be assigned to the macro \pgfmathresult.

$\protect\pro$

Expands a number from low-level floating point representation to scientific format, 1.234e4.

Use \pgfmathfloatvalueof in contexts where only expandable macros are allowed.

```
\verb|\pgfmathfloatcreate|| \langle flags \rangle \} \{ \langle mantissa \rangle \} \{ \langle exponent \rangle \}
```

Defines \pgfmathresult as the floating point number encoded by $\{\langle flags \rangle\}$, $\{\langle mantissa \rangle\}$ and $\{\langle exponent \rangle\}$.

All arguments are characters and will be expanded using \edef.

Flags: 1; Mantissa 1.0; Exponent 327

```
\pgfmathfloatcreate{1}{1.0}{327}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
```

$\label{loss} $$ \operatorname{floating\ point\ number} {\langle flag \rangle} {\langle frue\text{-}code \rangle} {\langle false\text{-}code \rangle} $$$

Invokes $\{\langle true\text{-}code \rangle\}$ if the flag of $\{\langle floating\ point\ number \rangle\}$ equals $\{\langle flag \rangle\}$ and $\{\langle false\text{-}code \rangle\}$ otherwise.

The argument $\{\langle flag \rangle\}$ can be one of

- **0** to test for zero,
- 1 to test for positive numbers,
- + to test for positive numbers,
- 2 to test for negative numbers,
- to test for negative numbers,
- 3 for "not-a-number",
- 4 for $+\infty$,
- **5** for $-\infty$.

It's not zero!It's positive!It's not negative!It's positive!It's not negative!

```
\pgfmathfloatparsenumber{42}
\pgfmathfloatifflags{\pgfmathresult}{0}{It's zero!}{It's not zero!}
\pgfmathfloatifflags{\pgfmathresult}{1}{It's positive!}{It's not positive!}
\pgfmathfloatifflags{\pgfmathresult}{2}{It's negative!}{It's not negative!}

% or, equivalently
\pgfmathfloatifflags{\pgfmathresult}{+}{It's positive!}{It's not positive!}
\pgfmathfloatifflags{\pgfmathresult}{-}{It's negative!}{It's not negative!}
```

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \pr
```

Extracts the flags of a floating point number $\{\langle x \rangle\}$ to $\{\langle flagsmacro \rangle\}$, the mantissa to $\{\langle mantissamacro \rangle\}$ and the exponent to $\{\langle exponentmacro \rangle\}$.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

Takes a floating point number $\{\langle x \rangle\}$ as input and writes flags to count register $\{\langle flagscount \rangle\}$, mantissa to dimen register $\{\langle mantissadimen \rangle\}$ and exponent to count register $\{\langle exponent count \rangle\}$.

Please note that this method rounds the mantissa to T_FX-precision.

```
\verb|\pgfmathfloattoregisterstok| \{\langle x \rangle\} \{\langle flagscount \rangle\} \{\langle mantissatoks \rangle\} \{\langle exponent count \rangle\}
```

A variant of \pgfmathfloattoregisters which writes the mantissa into a token register. It maintains the full input precision.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

Extracts the flags of $\{\langle x \rangle\}$ into the count register $\{\langle flagscount \rangle\}$.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\be
```

Extracts the flags of $\{\langle x \rangle\}$ into the macro $\langle macro \rangle$.

```
\protect\operatorname{\mathtt{NPgfmathfloatgetmantissa}} \{\langle x \rangle\} \{\langle mantissadimen \rangle\}
```

Extracts the mantissa of $\{\langle x \rangle\}$ into the dimen register $\{\langle mantissadimen \rangle\}$.

```
\protect\operatorname{\mathtt{Npgfmathfloatgetmantissatok}} \
```

Extracts the mantissa of $\{\langle x \rangle\}$ into the token register $\{\langle mantissatoks \rangle\}$.

```
\protect\operatorname{\mathtt{ightheta}} \{\langle x \rangle\} \{\langle exponent count \rangle\}
```

Extracts the exponent of $\{\langle x \rangle\}$ into the count register $\{\langle exponent count \rangle\}$.

37.4.2 Symbolic Rounding Operations

Commands in this section constitute the basic level implementations of the rounding routines. They work symbolically, i.e. they operate on text, not on numbers and allow arbitrarily large numbers.

$\protect\$

Rounds a fixed point number to prescribed precision and writes the result to \pgfmathresult.

The desired precision can be configured with /pgf/number format/precision, see section 74. This section does also contain application examples.

Any trailing zeros after the period are discarded. The algorithm is purely text based and allows to deal with precisions beyond TeX's fixed point support.

As a side effect, the global boolean \ifpgfmathfloatroundhasperiod will be set to true if and only if the resulting mantissa has a period. Furthermore, \ifpgfmathfloatroundmayneedrenormalize will be set to true if and only if the rounding result's floating point representation would have a larger exponent than $\{\langle x \rangle\}$.

```
1 \pgfmathroundto{1} \pgfmathresult

4.69 \pgfmathroundto{4.685}
```

\pgfmathresult

```
20000 \pgfmathroundto{19999.9996} \pgfmathresult
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

A variant of \pgfmathroundto which always uses a fixed number of digits behind the period. It fills missing digits with zeros.

```
1.00 \pgfmathroundtozerofill{1} \pgfmathresult

4.69 \pgfmathroundto{4.685} \pgfmathresult

20000.00 \pgfmathroundtozerofill{19999.9996} \pgfmathresult
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Rounds a normalized floating point number to a prescribed precision and writes the result to \pgfmathresult.

The desired precision can be configured with /pgf/number format/precision, see section 74.

This method employs \pgfmathroundto to round the mantissa and applies renormalization if necessary.

As a side effect, the global boolean \ifpgfmathfloatroundhasperiod will be set to true if and only if the resulting mantissa has a period.

```
5.26el \pgfmathfloatparsenumber{52.5864}
\pgfmathfloatround{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult

lel \pgfmathfloatparsenumber{9.995}
\pgfmathfloatround{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult}
```

$\verb|\pgfmathfloatroundzerofill{} \langle x \rangle \}$

A variant of \pgfmathfloatround produces always the same number of digits after the period (it includes zeros if necessary).

```
5.26e1 \pgfmathfloatparsenumber{52.5864} \pgfmathfloatroundzerofill{\pgfmathresult} \pgfmathfloattosci{\pgfmathresult} \pgfmathresult

1.00e1 \pgfmathfloatparsenumber{9.995} \pgfmathfloatroundzerofill{\pgfmathresult} \pgfmathfloattosci{\pgfmathresult} \pgfmathresult} \pgfmathresult
```

37.4.3 Math Operations Commands

This section describes some of the replacement commands in more detail.

Please note that these commands can be used even if the fpu as such has not been activated – it is sufficient to load the library.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Methods of this form constitute the replacement operations where $\langle op \rangle$ can be any of the well–known math operations.

Thus, \pgfmathfloatadd is the counterpart for \pgfmathadd and so on. The semantics and number of arguments is the same, but all input and output arguments are expected to be floating point numbers.

$\protect\pro$

Renormalizes $\{\langle x \rangle\}$ to extended precision mantissa, meaning $100 \le m < 1000$ instead of $1 \le m < 10$.

The 'extended precision' means we have higher accuracy when we apply pgfmath operations to mantissas.

The input argument is expected to be a normalized floating point number; the output argument is a non-normalized floating point number (well, normalized to extended precision).

The operation is supposed to be very fast.

$\protect\operatorname{\mathtt{Npgfmathfloatsetextprecision}} \$

Sets the precision used inside of pgfmathfloattoextentedprecision to $\{\langle shift \rangle\}$.

< m < 10000

The different choices are

3

```
0 normalization to 0 \le m < 1 (disable extended precision)
1 normalization to 10 \le m < 100
2 normalization to 100 \le m < 1000 (default of \pgfmathfloattoextentedprecision)
```

\pgfmathfloatlessthan $\{\langle x \rangle\}\{\langle y \rangle\}$

normalization to 1000

Defines \pgfmathresult as 1.0 if $\langle x \rangle < \langle y \rangle$, but 0.0 otherwise. It also sets the global TeX-boolean \pgfmathfloatcomparison accordingly. The arguments $\{\langle x \rangle\}$ and $\{\langle y \rangle\}$ are expected to be numbers which have already been processed by \pgfmathfloatparsenumber. Arithmetic is carried out using TeX-registers for exponent- and mantissa comparison.

$\protect\pro$

Defines \pgfmathresult to be $\langle float \rangle \cdot \langle fixed \rangle$ where $\langle float \rangle$ is a floating point number and $\langle fixed \rangle$ is a fixed point number. The computation is performed in floating point arithmetics, that means we compute $m \cdot \langle fixed \rangle$ and renormalize the result where m is the mantissa of $\langle float \rangle$.

This operation renormalizes $\langle float \rangle$ with \pgfmathfloattoextentedprecision before the operation, that means it is intended for relatively small arguments of $\langle fixed \rangle$. The result is a floating point number.

$\verb|\pgfmathfloatifapproxequalrel|{\langle a \rangle} {\langle b \rangle} {\langle true\text{-}code \rangle} {\langle false\text{-}code \rangle}$

Computes the relative error between $\langle a \rangle$ and $\langle b \rangle$ (assuming $\langle b \rangle \neq 0$) and invokes $\langle true\text{-}code \rangle$ if the relative error is below /pgf/fpu/rel thresh and $\langle false\text{-}code \rangle$ if that is not the case.

The input arguments will be parsed with \pgfmathfloatparsenumber.

```
/pgf/fpu/rel thresh={\langle number \rangle}
```

(no default, initially 1e-4)

A threshold used by \pgfmathfloatifapproxequalrel to decide whether numbers are approximately equal.

$\protect\pro$

Defines \pgfmathresult to be $\langle x \rangle \cdot 10^{\langle num \rangle}$. The operation is an arithmetic shift base ten and modifies only the exponent of $\{\langle x \rangle\}$. The argument $\{\langle num \rangle\}$ is expected to be a (positive or negative) integer.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

Defines \pgfmathresult to be the absolute error between two floating point numbers x and y, |x - y| and returns the result as floating point number.

$\verb|\pgfmathfloatrelerror{|\langle x \rangle|}{\langle y \rangle}|$

Defines \pgfmathresult to be the relative error between two floating point numbers x and y, |x-y|/|y| and returns the result as floating point number.

$\protect\$

Returns the integer part of the floating point number $\{\langle x \rangle\}$, by truncating any digits after the period. This methods truncates the absolute value |x| to the next smaller integer and restores the original sign afterwards

The result is returned as floating point number as well.

See also \pgfmathfloattoint which returns the number in integer format.

$\protect\pro$

Defines \pgfmathresult to be the natural logarithm of $\langle x \rangle$, $\ln(\langle x \rangle)$. This method is logically the same as \pgfmathln, but it applies floating point arithmetics to read number $\langle x \rangle$ and employs the logarithm identity

$$\ln(m \cdot 10^e) = \ln(m) + e \cdot \ln(10)$$

to get the result. The factor $\ln(10)$ is a constant, so only $\ln(m)$ with $1 \le m < 10$ needs to be computed. This is done using standard pgf math operations.

Please note that $\langle x \rangle$ needs to be a number, expression parsing is not possible here.

If $\langle x \rangle$ is not a bounded positive real number (for example $\langle x \rangle \leq 0$), \pgfmathresult will be empty, no error message will be generated.

-15.7452	\pgfmathlog{1.452e-7} \pgfmathresult
20.28096	\pgfmathlog{6.426e+8} \pgfmathresult

37.4.4 Accessing the Original Math Routines for Programmers

As soon as the library is loaded, every private math routine will be copied to a new name. This allows library and package authors to access the TEX-register based math routines even if the FPU is activated. And, of course, it allows the FPU as such to perform its own mantissa computations.

The private implementations of PGF math commands, which are of the form $\protect\operatorname{pgfmath}(name)$ 0, will be available as $\protect\operatorname{pgfmath}(basic0(name)$ 0 as soon as the library is loaded.

38 Lindenmayer System Drawing Library

38.1 Overview

Lindenmayer systems (also commonly known as "L-systems"), were originally developed by Aristid Lindenmayer as a theory of algae growth patterns and then subsequently used to model branching patterns in plants and produce fractal patterns. Typically, an L-system consists of a set of symbols, each of which is associated with some graphical action (such as "turn left" or "move forward") and a set of rules ("production" or "rewrite" rules). Given a string of symbols, the rewrite rules are applied several times and the when resulting string is processed the action associated with each symbol is executed.

In PGF, L-systems can be used to create simple 2-dimensional fractal patterns...

```
\begin{tikzpicture}
\pgfdeclarelindenmayersystem{Koch curve}{
  \rule{F -> F-F++F-F}
}
\shadedraw [top color=white, bottom color=blue!50, draw=blue!50!black]
  [1-system={Koch curve, step=2pt, angle=60, axiom=F++F++F, order=3}]
  lindenmayer system -- cycle;
\end{tikzpicture}
```

... and "plant like" patterns...



```
\begin{tikzpicture}
\draw [green!50!black, rotate=90]
  [1-system={rule set={F -> FF-[-F+F]+[+F-F]}, axiom=F, order=4, step=2pt,
    randomize step percent=25, angle=30, randomize angle percent=5}]
  lindenmayer system;
\end{tikzpicture}
```

...but it is important to bear in mind that even moderately complex L-systems can exceed the available memory of TEX, and can be very slow. If possible, you are advised to increase the main memory and save stack to their maximum possible values for your particular TEX distribution. However, even by doing this you may find you still run out of memory quite quickly.

For an excellent introduction to L-systems (containing some "really cool" pictures – many of which are sadly not possible in PGF) see *The Algorithmic Beauty of Plants* by Przemyslaw Prusinkiewicz and Aristid Lindenmayer (which is freely available via the internet).

This PGF-library provides basic commands for defining and using simple L-systems. The TikZ-library provides, furthermore, a front end for using L-systems in TikZ.

38.1.1 Declaring L-systems

Before an L-system can be used, it must be declared using the following command:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command declares a Lindenmayer system called $\langle name \rangle$. The $\langle specification \rangle$ argument contains a description of the L-system's symbols and rules. Two commands \symbol and \rule are only defined when the $\langle specification \rangle$ argument is executed.

```
\symbol{\langle name \rangle} \{\langle code \rangle\}
```

This defines a symbol called $\langle name \rangle$ for a specific L-system, and associates it with $\langle code \rangle$.

A symbol should consist of a single alpha-numeric character (i.e., A-Z, a-z or 0-9). The symbols F, f, +, -, [and] are available by default so do not need to be defined for each L-system. However, if you are feeling adventurous, they can be redefined for specific L-systems if required. The L-system treats the default symbols as follows (the commands they execute are described below):

- F move forward a certain distance, drawing a line. Uses \pgflsystemdrawforward.
- f move forward a certain distance, without drawing a line. Uses \pgflsystemmoveforward.
- + turn left by some angle. Uses \pgflsystemturnleft.
- - turn right by some angle. Uses \pgflsystemturnright.
- [save the current state (i.e., the position and direction). Uses \pgflsystemsavestate.
-] restore the last saved state. Uses \pgflsystemrestorestate.

The symbols [and] act like a stack: [pushes the state of the L-system on to the stack, and] pops a state off the stack.

When $\langle code \rangle$ is executed, the transformation matrix is set up so that the origin is at the current position and the positive x-axis "points forward", so \pgfpathlineto{\pgfpoint{1cm}{0cm}} draws a line 1cm forward.

The following keys can alter the production of an L-system. However, they do not store values in themselves.

/pgf/lindenmayer system/step= $\langle length \rangle$

(no default, initially 5pt)

How far the L-system moves forward if required. This key sets the TEX dimension \pgflsystemstep.

/pgf/lindenmayer system/randomize step percent=\langle percentage \rangle

(no default, initially 0)

If the step is to be randomized, this key specifies by how much. The value is stored in the TEX macro \pgflsystemrandomizesteppercent.

/pgf/lindenmayer system/left angle= $\langle angle \rangle$

(no default, initially 90)

This key sets the angle through which the L-system turns when it turns left. The value is stored in the TeX macro \pgflsystemrleftangle.

/pgf/lindenmayer system/right angle= $\langle angle \rangle$

(no default, initially 90)

This key sets the angle through which the L-system turns when it turns right. The value is stored in the TeX macro \pgflsystemrrightangle.

/pgf/lindenmayer system/randomize angle percent= $\langle percentage \rangle$

(no default, initially 0)

If the angles are to be randomized, this key specifies by how much. The value is stored in the T_EX macro pgflsystemrandomizeanglepercent.

For speed and convenience, when the code for a symbol is executed, the following commands are available.

\pgflsystemcurrentstep

The current "step" of the L-system (i.e., how far the system will move forward if required). This is initially set to the value in the TEX-dimensions \pgflsystemstep, but the actual value may be changed if \pgflsystemrandomizestep is used (see below).

\pgflsystemcurrentleftangle

The angle the L-system will turn when it turns left. The value stored in this macro may be changed if \pgflsystemrandomizeleftangle is used.

\pgflsystemcurrentrightangle

The angle the L-system will turn when it turns right. The value stored in this macro may be changed if \pgflsystemrandomizerightangle is used.

The following commands may be useful if you wish to define your own symbols.

\pgflsystemrandomizestep

Randomizes the value in \pgflsystemcurrentstep according to the value of the randomize step percent key.

\pgflsystemrandomizeleftangle

Randomizes the value in \pgflsystemcurrentleftangle according to the value of the randomize angle percent key.

\pgflsystemrandomizerightangle

Randomizes the value in \pgflsystemcurrentrightangle according to the value of the randomize angle key.

\pgflsystemdrawforward

Move forward in the current direction, by \pgflsystemcurrentstep, drawing a line in the process. This macro calls \pgflsystemrandomizestep. Internally, PGF simply shifts the transformation matrix in the positive direction of the current (transformed) x-axis by \pgflsystemstep and then executes a line-to to the (newly transformed) origin.

\pgflsystemmoveforward

Move forward in the current direction, by \pgflsystemcurrentstep, without drawing a line. This macro calls \pgflsystemrandomizestep. PGF executes a transformation as above, but executes a move-to to the (newly transformed) origin.

\pgflsystemturnleft

Turn left by \pgflsystemcurrentleftangle. Internally, PGF simply rotates the transformation matrix. This macro calls \pgflsystemrandomizeleftangle.

\pgflsystemturnright

Turn right by \pgflsystemcurrentrightangle. Internally, PGF simply rotates the transformation matrix. This macro calls \pgflsystemrandomizerightangle.

\pgflsystemsavestate

Save the current position and orientation. Internally, PGF simply starts a new TFX-group.

\pgflsystemrestorestate

Restore the last saved position and orientation. Internally, PGF closes a TEX-group, restoring the transformation matrix of the outer scope, and a move-to command is executed to the (transformed) origin.

```
\mathbf{rule}\{\langle head \rangle -> \langle body \rangle\}
```

Declare a rule. $\langle head \rangle$ should consist of a single symbol, which need not have been declared using \symbol or exist as a default symbol (in fact, the more interesting L-systems depend on using symbols with no corresponding code, to control the "growth" of the system). $\langle body \rangle$ consists of a string of symbols, which again need not necessarily have any code associated with them.

As an example, the following shows an L-system that uses some of these commands. This example illustrates the point that some symbols, in this case A and B, do not have to have code associated with them. They simply control the growth of the system.

```
\pgfdeclarelindenmayersystem{Hilbert curve}{
   \symbol{X}{\pgflsystemdrawforward}
   \symbol{+}{\pgflsystemturnright} % Explicitly define + and - symbols.
   \symbol{-}{\pgflsystemturnleft}
   \rule{A -> +BX-AXA-XB+}
   \rule{B -> -AX+BXB+XA-}
}
\tikz\draw[lindenmayer system={Hilbert curve, axiom=A, order=4, angle=90}]
   lindenmayer system;
}
```

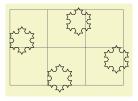
38.2 Using Lindenmayer Systems

38.2.1 Using L-Systems in PGF

The following command is used to run an L-system in PGF:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

Runs the L-system called $\langle name \rangle$ using the input string $\langle axiom \rangle$ for $\langle order \rangle$ iterations. In general, prior to calling this command, the transformation matrix should be set appropriately for shifting and rotating, and a move-to to the (transformed) origin should be executed. This origin will be where the L-system starts. In addition, the relevant keys should be set appropriately.



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \pgfset{lindenmayer system/.cd, angle=60, step=2pt}
  \foreach \x/\y in {0cm/1cm, 1.5cm/1.5cm, 2.5cm/0.5cm, 1cm/0cm}{
    \pgftransformshift{\pgfqpoint{\x}{\y}}
    \pgfpathmoveto{\pgfpointorigin}
    \pgflindenmayersystem{Koch curve}{F++F++F}{2}
    \pgfusepath{stroke}
  }
}
```

Note that it is perfectly feasible for an L-system to define special symbols which perform the move-to and use-path operations.

38.2.2 Using L-Systems in TikZ

In TikZ, an L-system is created using a path operation. However, TikZ is more flexible regarding the positioning of the L-system and also provides keys to create L-systems "on-line".

```
\path ... lindenmayer system [\langle keys \rangle] ...;
```

This will run an L-system according to the parameters specified in $\langle keys \rangle$ (which can also contain normal keys such as draw or thin). The syntax is flexible regarding the L-system parameters and the following all do the same thing:

```
\draw lindenmayer system [lindenmayer system={Hilbert curve, axiom=4, order=3}];
\draw [lindenmayer system={Hilbert curve, axiom=4, order=3}] lindenmayer system;
\tikzset{lindenmayer system={Hilbert curve, axiom=4, order=3}}
\draw lindenmayer system;
```

```
\path ... l-system [\langle keys \rangle] ...;
```

A more compact version of the lindenmayer system path command.

This library adds some additional keys for specifying L-systems. These keys only work in TikZ and all have the same path, namely, /pgf/lindenmayer system, but the following keys are provided for convenience, so that you do not have to keep repeating this path:

```
 \begin{array}{ll} \mbox{/pgf/lindenmayer system=} \{ \langle keys \rangle \} & \text{(style, no default)} \\ \mbox{alias /tikz/lindenmayer system} \end{array}
```

This key changes the key path to /pgf/lindenmayer systems and executes $\langle keys \rangle$.

```
 \begin{tabular}{ll} \be
```

A more compact version of the previous key.

```
/pgf/lindenmayer system/name={\langle name \rangle} (no default)
```

Sets the name for the L-system.

```
pgf/lindenmayer system/axiom={\langle string \rangle} (no default)
```

Sets the axiom (or input string) for the L-system.

```
/pgf/lindenmayer system/order={\langle integer \rangle} (no default)
```

Sets the number of iterations the L-system will perform.

```
/pgf/lindenmayer system/rule set=\{\langle list \rangle\} (no default)
```

This key allows an (anonymous) L-system to be declared "on-line". There is, however, a restriction that only the default symbols can be used for drawing (empty symbols can still be used to control the growth of the system). The rules in $\langle list \rangle$ should be separated by commas.

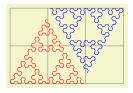


```
\tikz[rotate=65]\draw [green!60!black] 1-system
[1-system={rule set={F -> F[+F]F[-F]}, axiom=F, order=4, angle=25,step=3pt}];
```

$/pgf/lindenmayer system/anchor=\langle anchor \rangle$

(no default)

Be default, when this key is not used, the L-system will start from the last specified coordinate. By using this key, the L-system will be placed inside a special (rectangle) node which can be positioned using $\langle anchor \rangle$.



```
\begin{tikzpicture}[1-system={step=1.75pt, order=5, angle=60}]
  \pgfdeclarelindenmayersystem{Sierpinski triangle}{
    \symbol{X}{\pgflsystemdrawforward}
    \symbol{Y}{\pgflsystemdrawforward}
    \rule{X -> Y-X-Y}
    \rule{Y -> X+Y+X}
}
\draw [help lines] grid (3,2);
\draw [red] (0,0) 1-system
    [1-system={Sierpinski triangle, axiom=+++X, anchor=south west}];
\draw [blue] (3,2) 1-system
    [1-system={Sierpinski triangle, axiom=X, anchor=north east}];
\end{tikzpicture}
```

39 Matrix Library

```
\usetikzlibrary{matrix} % MEX and plain TeX \usetikzlibrary[matrix] % ConTeXt
```

This library package defines additional styles and options for creating matrices.

39.1 Matrices of Nodes

A matrix of nodes is a TikZ matrix in which each cell contains a node. In this case it is bothersome having to write $\node{$ at the beginning of each cell and $\}$; at the end of each cell. The following key simplifies typesetting such matrices.

```
/tikz/matrix of nodes (no value)
```

Conceptually, this key adds $\ne \{$ at the beginning and $\}$; at the end of each cell and sets the anchor of the node to base. Furthermore, it adds the option name option to each node, where the name is set to $\langle matrix\ name \rangle - \langle row\ number \rangle - \langle column\ number \rangle$. For example, if the matrix has the name my matrix, then the node in the upper left cell will get the name my matrix-1-1.

You may wish to add options to certain nodes in the matrix. This can be achieved in three ways.

1. You can modify, say, the row 2 column 3 style to pass special options to this particular cell.

2. At the beginning of a cell, you can use a special syntax. If a cell starts with a vertical bar, then everything between this bar and the next bar is passed on to the node command.

You can also use an option like | [red] (seven) | to give a different name to the node. Note that the & character also takes an optional argument, which is an extra column skip.

```
\begin{tikzpicture}
8
         1
              6
                       \matrix [matrix of nodes]
3
              7
                      {
         5
                        8 &[1cm] 1 &[3mm] |[red]| 6 \\
              2
4
         9
                                 5 &
                                           |[red]| 7 \\
                        3 &
                                  9 &
                                                   2 \\
                        4 &
                    \end{tikzpicture}
```

3. If your cell starts with a \path command or any command that expands to \path, which includes \draw, \node, \fill and others, the \node{ startup code and the }; code are suppressed. This means that for this particular cell you can provide totally different contents.

/tikz/matrix of math nodes

(no value)

This style is almost the same as the previous style, only \$ is added at the beginning and at the end of each node, so math mode will be switched on in all nodes.

```
/tikz/nodes in empty cells=\langle true \ or \ false \rangle
```

(default true)

When set to true, a node (with empty contents) is put in empty cells. Normally, empty cells are just, well, empty. The style can be used together with both a matrix of nodes and a matrix of math nodes.

```
\begin{tikzpicture}
     a_6
               \matrix [matrix of math nodes, nodes={circle, draw}]
                 a_8 &
                           & a_6 \\
                a_3 &
                           & a_7 \\
                a_4 & a_9 &
a_9
              }:
             \end{tikzpicture}
             \begin{tikzpicture}
               \matrix [matrix of math nodes, nodes={circle, draw}, nodes in empty cells]
               {
                a_8 &
                           & a_6 \\
                 a_3 &
                           & a_7 \\
                a_4 & a_9 &
              };
            \end{tikzpicture}
```

39.2 End-of-Lines and End-of-Row Characters in Matrices of Nodes

Special care must be taken about the usage of the \\ command inside a matrix of nodes. The reason is that this character is overloaded in TeX: On the one hand, it is used to denote the end of a line in normal text; on the other hand it is used to denote the end of a row in a matrix. Now, if a matrix contains node which in turn may have multiple lines, it is unclear which meaning of \\ should be used.

This problem arises only when you use the text width option of nodes. Suppose you write a line like

```
\matrix [text width=5cm,matrix of nodes]
{
  first row & upper line \\ lower line \\
  second row & hmm \\
};
```

This leaves TEX trying to riddle out how many rows this matrix should have. Do you want two rows with the upper right cell containing a two-line text. Or did you mean a three row matrix with the second row having only one cell?

Since T_FX is not clairvoyant, the following rules are used:

- 1. Inside a matrix, the \\ command, by default, signals the end of the row, not the end of a line in a cell.
- 2. However, there is an exception to this rule: If a cell starts with a TEX-group (this is, with {), then inside this first group the \\ command retains the meaning of "end of line" character. Note that this special rule works only for the first group in a cell and this group must be at the beginning.

The net effect of these rules is the following: Normally, \\ is an end-of-row indicator; if you want to use it as an end-of-line indicator in a cell, just put the whole cell in curly braces. The following example illustrates the difference:

```
row 1 upper line lower line row 2 hmm
```

```
\begin{tikzpicture}
  \matrix [matrix of nodes,nodes={text width=16mm,draw}]
  {
    row 1 & upper line \\ lower line \\
    row 2 & hmm \\
    };
\end{tikzpicture}
```

```
row 1 upper line lower line row 2 hmm
```

```
\begin{tikzpicture}
  \matrix [matrix of nodes,nodes={text width=16mm,draw}]
  {
    row 1 & {upper line \\ lower line} \\
    row 2 & hmm \\
    };
  \end{tikzpicture}
```

Note that this system is not fool-proof. If you write things like $a\&b\{c\d\}\$ in a matrix of nodes, an error will result (because the second cell did not start with a brace, so $\$ retained its normal meaning and, thus, the second cell contained the text $b\{c\}$, which is not balanced with respect to the number of braces).

39.3 Delimiters

Delimiters are parentheses or braces to the left and right of a formula or a matrix. The matrix library offers options for adding such delimiters to a matrix. However, delimiters can actually be added to any node that has the standard anchors north, south, north west and so on. In particular, you can add delimiters to any rectangle box. They are implemented by "measuring the height" of the node and then adding a delimiter of the correct size to the left or right using some after node magic.

```
/tikz/left delimiter=\langle delimiter \rangle
```

(no default)

This option can be given to a any node that has the standard anchors north, south and so on. The $\langle delimiter \rangle$ can be any delimiter that is acceptable to TeX's \left command.

```
\begin{pmatrix}
a_8 & a_1 & a_6 \\
a_3 & a_5 & a_7 \\
a_4 & a_9 & a_2
\end{pmatrix}
```

```
\begin{tikzpicture}
  \matrix [matrix of math nodes,left delimiter=(,right delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
};
\end{tikzpicture}
```

```
\left( \int_0^1 x \, dx \right)
```

```
\begin{tikzpicture}
 \node [fill=red!20,left delimiter=(,right delimiter=\}]
   {$\displaystyle\int_0^1 x\,dx$};
\end{tikzpicture}
```

/tikz/every delimiter

(style, initially empty)

This style is executed for every delimiter. You can use it to shift or color delimiters or do whatever.

/tikz/every left delimiter

(style, initially empty)

This style is additionally executed for every left delimiter.

```
\langle \begin{tikzpicture} \ a_8 & a_1 & a_6 \ a_3 & a_5 & a_7 \ a_4 & a_9 & a_2 \end{tikzpicture} \]
\text{ \langle levery left delimiter/.style=\{red,xshift=1ex\}, \ every right delimiter/.style=\{xshift=-1ex\}\]
\text{ \langle matrix [matrix of math nodes,left delimiter=(,right delimiter=\}]} \{ \ a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \\ \}; \ \end{\text{tikzpicture}}
```

/tikz/right delimiter=\langle delimiter \rangle

(no default)

Works as above.

/tikz/every right delimiter

(style, initially empty)

Works as above.

/tikz/above delimiter=\langle delimiter \rangle

(no default)

This option allows you to add a delimiter above the node. It is implemented by rotating a left delimiter.

```
\begin{bmatrix} a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \end{bmatrix}
```

/tikz/every above delimiter

(style, initially empty)

Works as above.

/tikz/below delimiter= $\langle delimiter \rangle$

(no default)

Works as above.

/tikz/every below delimiter

(style, initially empty)

Works as above.

40 Mindmap Drawing Library

```
\usetikzlibrary{mindmap} % MEX and plain TeX \usetikzlibrary[mindmap] % ConTeXt
```

This packages provides styles for drawing mindmap diagrams.

40.1 Overview

This library is intended to make the creation of mindmaps or concept maps easier. A *mindmap* is a graphical representation of a concept together with related concepts and annotations. Mindmaps are, essentially, trees, possibly with a few extra edges added, but they are usually drawn in a special way: The root concept is placed in the middle of the page and is drawn as a huge circle, ellipse, or cloud. The related concepts then "leave" this root concept via branch-like tendrils.

The mindmap library of TikZ produces mindmaps that look a bit different from the standard mindmaps: While the big root concept is still a circle, related concepts are also depicted as (smaller) circles. The related concepts are linked to the root concept via organic-looking connections. The overall effect is visually rather pleasing, but readers may not immediately think of a mindmap when they see a picture created with this library.

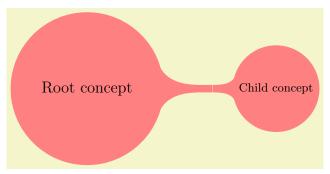
Although it is not strictly necessary, you will usually create mindmaps using TikZ's tree mechanism and some of the styles and macros of the package work best when used inside trees. However, it is still possible and sometimes necessary to treat parts of a mindmap as a graph with arbitrary edges and this is also possible.

40.2 The Mindmap Style

Every mindmap should be put in a scope or a picture where the mindmap style is used. This style installs some internal settings.

/tikz/mindmap (style, no value)

Use this style with all pictures or at least scopes that contain a mindmap. It installs a whole bunch of settings that are useful for drawing mindmaps.



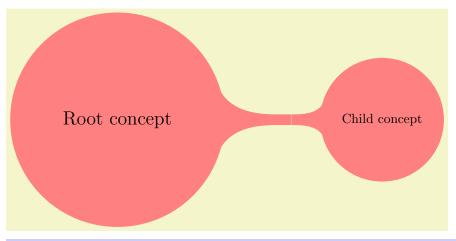
```
\tikz[mindmap,concept color=red!50]
\node [concept] {Root concept}
    child[grow=right] {node[concept] {Child concept}};
```

The sizes of concepts are predefined in such a way that a medium-size mindmap will fit on an A4 page (more or less).

/tikz/every mindmap

(style, no value)

This style is included by the mindmap style. Change this style to add special settings to your mindmaps.



\tikz[large mindmap,concept color=red!50]
\node [concept] {Root concept}
 child[grow=right] {node[concept] {Child concept}};

Remark: Note that mindmap redefines font sizes and sibling angle depending on the current concept level (i.e. inside of level 1 concept, level 2 concept etc.). Thus, if you need to redefine these variables, use

\tikzset{level 1 concept/.append style={font=\small}}

01

 $\tikzset{level 2 concept/.append style={sibling distance=90}} $$ after the mindmap style.$

/tikz/small mindmap

(style, no value)

This style includes the mindmap style, but additionally changes the default size of concepts, fonts and distances so that a medium-sized mindmap will fit on an A5 page (A5 pages are half as large as A4 pages). Mindmaps with small mindmap will also fit onto a standard frame of the beamer package.

/tikz/large mindmap

(style, no value)

This style includes the mindmap style, but additionally changes the default size of concepts, fonts and distances so that a medium-sized mindmap will fit on an A3 page (A3 pages are twice as large as A4 pages).

/tikz/huge mindmap

(style, no value)

This style causes concepts to be even bigger and it is best used with A2 paper and above.

40.3 Concepts Nodes

The basic entities of mindmaps are called concepts in TikZ. A concept is a node of style concept and it must be circular for some of the connection macros to work.

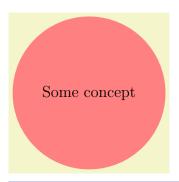
40.3.1 Isolated Concepts

The following styles influence how isolated concepts are rendered:

/tikz/concept (style, no value)

This style should be used with all nodes that are concepts, although some styles like extra concept install this style automatically.

Basically, this style makes the concept node circular and installs a uniform color called concept color, see below. Additionally, the style every concept is called.



\tikz[mindmap,concept color=red!50] \node [concept] {Some concept};

/tikz/every concept

(style, no value)

In order to change the appearance of concept nodes, you should change this style. Note, however, that the color of a concept should be uniform for some of the connection bar stuff to work, so you should not change the color or the draw/fill state of concepts using this option. It is mostly useful for changing the text color and font.

/tikz/concept color= $\langle color \rangle$

(no default)

This option tells TikZ which color should be used for filling and stroking concepts. The difference between this option and just setting every concept to the desired color is that this option allows TikZ to keep track of the colors used for concepts. This is important when you *change* the color between two connected concepts. In this case, TikZ can automatically create a shading that provides a smooth transition between the old and the new concept color; we will come back to this in the next section.

/tikz/extra concept

(style, no value)

This style is intended for concepts that are not part of the "mindmap tree," but stand beside it. Typically, they will have a subdued color or be smaller. In order to have these concepts appear in a uniform way and in order to indicate in the code that these concepts are additional, you can use this style.



/tikz/every extra concept

(style, no value)

Change this style to change the appearance of extra concepts.

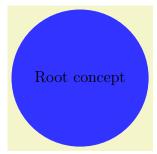
40.3.2 Concepts in Trees

As pointed out earlier, TikZ assumes that your mindmap is built using the child facilities of TikZ. There are numerous options that influence how concepts are rendered at the different levels of a tree.

/tikz/root concept

(style, no value)

This style is used for the roots of mindmap trees. By adding something to this, you can change how the root of a mindmap will be rendered.



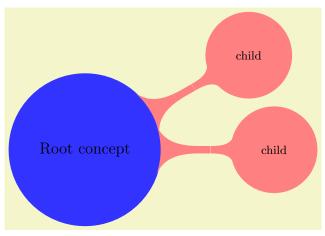
```
\tikz
  [root concept/.append style={concept color=blue!80,minimum size=3.5cm},
  mindmap]
  \node [concept] {Root concept};
```

Note that styles like large mindmap redefine these styles, so you should add something to this style only inside the picture.

/tikz/level 1 concept

(style, no value)

The mindmap style adds this style to the level 1 style. This means that the first level children of a mindmap tree will use this style.



```
\tikz
  [root concept/.append style={concept color=blue!80},
  level 1 concept/.append style={concept color=red!50},
  mindmap]
  \node [concept] {Root concept}
  child[grow=30] {node[concept] {child}}
  child[grow=0] {node[concept] {child}};
```

/tikz/level 2 concept

(style, no value)

Works like level 1 concept, only for second level children.

/tikz/level 3 concept

(style, no value)

Works like level 1 concept.

/tikz/level 4 concept

(style, no value)

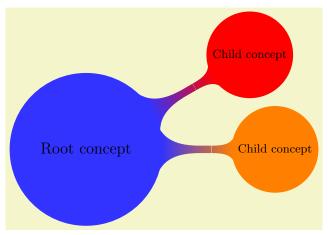
Works like level 1 concept. Note that there are no fifth and higher level styles, you need to modify level 5 directly in such cases.

/tikz/concept color= $\langle color \rangle$

(no default)

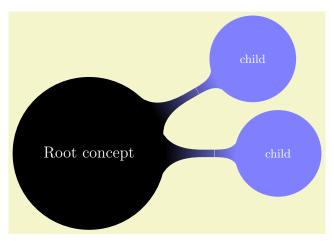
We saw already that this option is used to change the color of concepts. We now have a look at its effect when used on child nodes of a concept. Normally, this option simply changes the color of the children. However, when the option is given as an option to the child operation (and not to the node operation and also not as an option to all children via the level 1 style), TikZ will smoothly change the concept color from the parent's color to the color of the child concept.

Here is an example:



```
\tikz[mindmap,concept color=blue!80]
  \node [concept] {Root concept}
    child[concept color=red,grow=30] {node[concept] {Child concept}}
    child[concept color=orange,grow=0] {node[concept] {Child concept}};
```

In order to have a concept color which changes with the hierarchy level, a tiny bit of magic is needed:



```
\tikz[mindmap,text=white,
    root concept/.style={concept color=blue},
    level 1 concept/.append style=
        {every child/.style={concept color=blue!50}}]
    \node [concept] {Root concept}
    child[grow=30] {node[concept] {child}}
    child[grow=0] {node[concept] {child}};
```

40.4 Connecting Concepts

40.4.1 Simple Connections

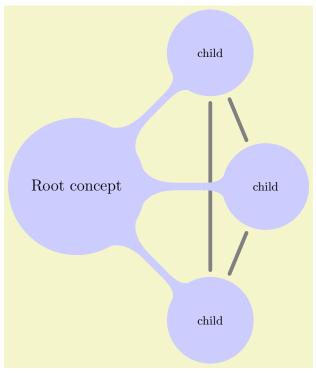
The easiest way to connect two concepts is to draw a line between them. In order to give such lines a consistent appearance, it is recommendable to use the following style when drawing such lines:

/tikz/concept connection

(style, no value)

This style can be used for lines between two concepts. Feel free to redefine this style.

A problem arises when you need to connect concepts after the main mindmap has been drawn. In this case you will want the connection lines to lie *behind* the main mindmap. However, you can draw the lines only after the coordinates of the concepts have been determined. In this case you should place the connecting lines on a background layer as in the following example:



40.4.2 The Circle Connection Bar Decoration

Instead of a simple line between two concepts, you can also add a bar between the two nodes that has slightly organic ends. These bars are also used by default as the edges from parents in the mindmap tree.

For the drawing of the bars a special decoration is used, which is defined in the mindmap library:

Decoration circle connection bar

This decoration can be used to connect two circles. The start of the to-be-decorated path should lie on the border of the first circle, the end should lie on the border of the second circle. The following two decoration keys should be initialized with the sizes of the circles:

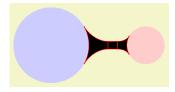
- start radius
- end radius

Furthermore, the following two decoration keys influence the decoration:

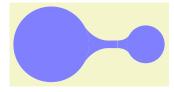
- amplitude
- angle

The decoration turns a straight line into a path that starts on the border of the first circle at the specified angle relative to the line connecting the centers of the circles. The path then changes into a rectangle whose thickness is given by the amplitude. Finally, the path ends with the same angles on the second circle.

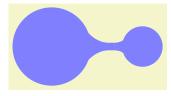
Here is an example that should make this clearer:



As can be seen, the decorated path consists of three parts and is not really useful for drawing. However, if you fill the decorated path only, and if you use the same color as for the circles, the result is better.



In the above example you may notice the small white line between the circles and the decorated path. This is due to rounding errors. Unfortunately, for larger distances, the errors can accumulate quite strongly, especially since TikZ and TeX are not very good at computing square roots. For this reason, it is a good idea to make the circles slightly larger to cover up such problems. When using nodes of shape circle, you can just add the draw option with a line width of one or two points (for very large distances you may need line width up to 4pt).



40.4.3 The Circle Connection Bar To-Path

The circle connection bar decoration is a bit complicated to use. Especially specifying the radii is quite bothersome (the amplitude and the angle can be set once and for all). For this reason, the mindmap library defines a special to-path that performs the necessary computations for you.

/tikz/circle connection bar

(style, no value)

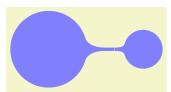
This style installs a rather involved to-path. Unlike normal to-paths, this path requires that the start and the target of the to-path are named nodes of shape circle – if this is not the case, this path will produce errors.

Assuming that the start and the target are circles, the to-path will first compute the radii of these circles (by measuring the distance from the center anchor to some anchor on the border) and will set the start circle keys accordingly. Next, the fill option is set to the concept color while draw=none is set. The decoration is set to circle connection bar. Finally, the following style is included:

/tikz/every circle connection bar

(style, no value)

Redefine this style to change the appearance of circle connection bar to-paths.



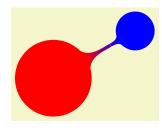
```
\begin{tikzpicture}[concept color=blue!50,blue!50,outer sep=0pt]
  \node (n1) at (0,0)   [circle,minimum size=2cm,fill,draw,thick] {};
  \node (n2) at (2.5,0)  [circle,minimum size=1cm,fill,draw,thick] {};
  \path (n1) to[circle connection bar] (n2);
  \end{tikzpicture}
```

Note that it is not a good idea to have more than one to operation together with the option circle connection bar in a single \path. Use the edge operation, instead, for creating multiple connections and this operation creates a new scope for each edge.

In a mindmap we sometimes want colors to change from one concept color to another. Then, the connection bar should, ideally, consist of a smooth transition between these two colors. Getting this right using shadings is a bit tricky if you try this "by hand," so the mindmap library provides a special option for facilitating this procedure.

/tikz/circle connection bar switch color=from (\(\first \color \\)) to (\(\second \color \\)) (no default)

This style works similarly to the circle connection bar. The only difference is that instead of filling the path with a single color a shading is used.



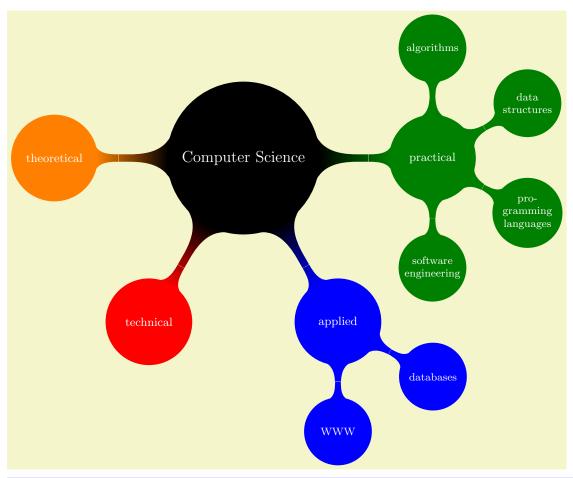
 $\label{lem:connection} $$ \phi (n1) to [circle connection bar switch color=from (red) to (blue)] (n2); $$ \end{tikzpicture}$

40.4.4 Tree Edges

Most of the time, concepts in a mindmap are connected automatically when the mindmap is built as a tree. The reason is that the mindmap installs a circle connection bar path as the edge from parent path. Also, the mindmap option takes care of things like setting the correct draw and outer sep settings and some other stuff.

In detail, the mindmap option sets the edge from parent path to a path that uses the to-path circle connection bar to connect the parent node and the child node. The concept color option (locally) changes this by using circle connection bar switch color instead with the from-color set to the old (parent's) concept color and the to-color set to the new (child's) concept color. This means that when you provide the concept color option to a child command, the color will change from the parent's concept color to the specified color.

Here is an example of a tree built in this way:



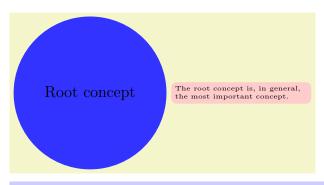
```
\begin{tikzpicture}
 \path[mindmap,concept color=black,text=white]
   node[concept] {Computer Science}
   [clockwise from=0]
   % note that 'sibling angle' can only be defined in
   % 'level 1 concept/.append style={}'
   child[concept color=green!50!black] {
     node[concept] {practical}
     [clockwise from=90]
     child { node[concept] {algorithms} }
     child { node[concept] {data structures} }
     child { node[concept] {pro\-gramming languages} }
     child { node[concept] {software engineer\-ing} }
   child[concept color=blue] {
     node[concept] {applied}
     [clockwise from=-30]
     child { node[concept] {databases} }
     child { node[concept] {WWW} }
   child[concept color=red] { node[concept] {technical} }
   child[concept color=orange] { node[concept] {theoretical} };
\end{tikzpicture}
```

40.5 Adding Annotations

An annotation is some text outside a mindmap that, unlike an extra concept, simply explains something in the mindmap. The following style is mainly intended to help readers of the code see that a node in an annotation node.

/tikz/annotation (style, no value)

This style indicates that a node is an annotation node. It includes the style every annotation, which allows you to change this style in a convenient fashion.



```
\begin{tikzpicture}
  [mindmap,concept color=blue!80,
  every annotation/.style={fill=red!20}]
  \node [concept] (root) {Root concept};

  \node [annotation,right] at (root.east)
  {The root concept is, in general, the most important concept.};
  \end{tikzpicture}
```

/tikz/every annotation

(style, no value)

This style is included by annotation.

41 Paper Folding Diagrams Library

\usetikzlibrary{folding} % MEX and plain TeX \usetikzlibrary[folding] % ConTeXt

This library defines commands for creating paper folding diagrams. Currently, it just contains a single command for creating a single diagram, but that one is really useful for creating calendars for your (real) desktop.

$\texttt{\tikzfoldingdodecahedron[}\langle options\rangle];$

This command draws a folding diagram for a dodecahedron. The syntax is intended to remind of the $\protect\operatorname{\mathtt{Npath}}$ command, but (currently) you must specify the $\langle options \rangle$ and nothing else may be specified between the command name and the closing semicolon.

The following keys may be used in the $\langle options \rangle$:

/tikz/folding line length= $\langle dimension \rangle$

(no default)

Sets the length of the base line for folding. For the dodecahedron this is the length of all the sides of the pentagons.

$$/\text{tikz/face } 1=\langle code \rangle$$
 (no default)

The $\langle code \rangle$ is executed for the first face of the dodecahedron. When it is executed, the coordinate system will have been shifted and rotated such that it lies at the middle of the first face of the dodecahedron.

```
/\text{tikz/face } 2 = \langle code \rangle (no default)
```

Same as face 1, but for the second face.

$$/\text{tikz/face } 3=\langle code \rangle$$
 (no default)

Same as face 1, but for the third face.

There are further similar options, ending with the following:

```
/\text{tikz/face } 12=\langle code \rangle (no default)
```

Same as face 1, but for the last face.

Here is a simple example:

```
\begin{tikzpicture}[transform shape]
                       \tikzfoldingdodecahedron
                       [folding line length=6mm,
11
                       face 1={
                                \node[red] {1};},
                       face 2={ \node
                                             {2}:}.
            ಉ
                       face 3={ \node
                                             {3};},
0
                       face 4={
                                \node
                       face 5={ \node
                                             {5}:}.
                       face 6={ \node
                                             {6};},
                       face 7=\{ \setminus node \}
                                             {7};},
                       face 8={ \node
                                            {8};},
                       face 9={ \node
                                            {9};},
                       face 10={\node
                                            {10}:}.
                       face 11={\node
                                             {11};}.
                       face 12={\node
                                             {12};}];
                    \end{tikzpicture}
```

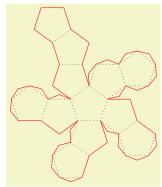
The appearance of the cut and folding lines can be influenced using the following styles:

/tikz/every cut (style, initially empty)

Executed for every line that should be cut using scissors.

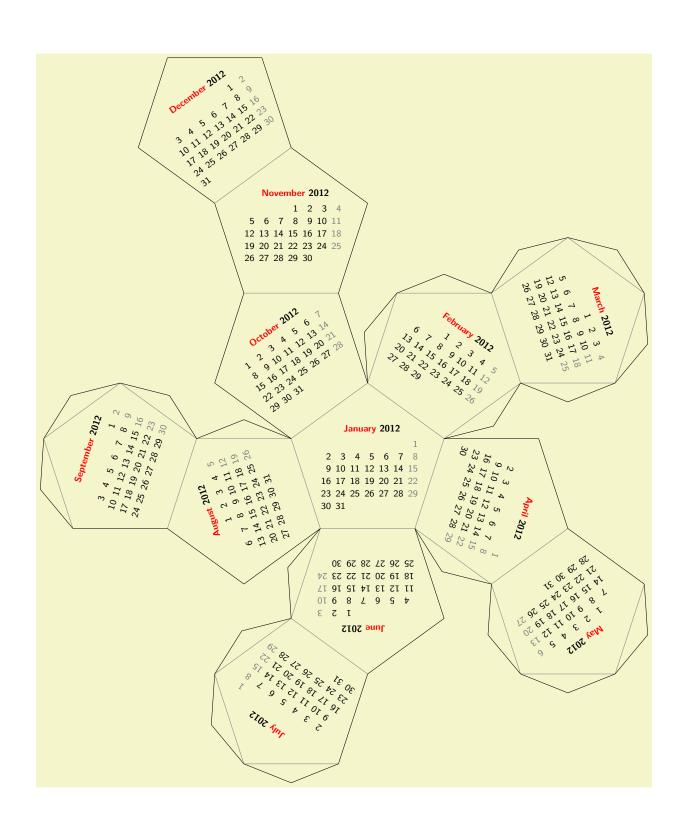
```
/tikz/every fold (style, initially help lines)
```

Executed for every line that should be folded.



\begin{tikzpicture}[every cut/.style=red,every fold/.style=dotted]
 \tikzfoldingdodecahedron[folding line length=6mm];
\end{tikzpicture}

Here is a big example that produces a diagram for a calendar:

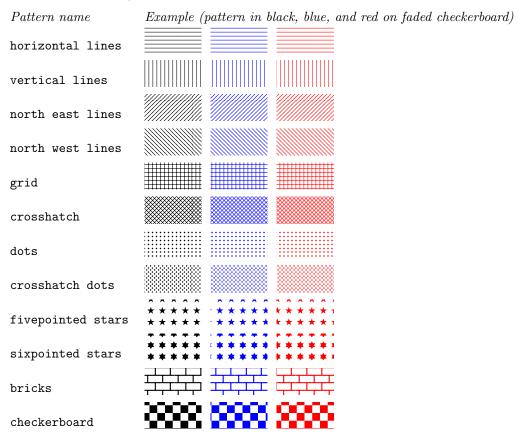


```
\sffamily\scriptsize
\begin{tikzpicture}
 [transform shape,
  every calendar/.style=
    at={(-8ex, 4ex)},
    week list,
    month label above centered,
    month text=\bfseries\textcolor{red}{\%mt} \%y0,
    if={(Sunday) [black!50]}
  }]
 \tikzfoldingdodecahedron
   folding line length=2.5cm,
   face 1={ \calendar [dates=\the\year-01-01 to \the\year-01-last];},
   face 2={ \calendar [dates=\the\year-02-01 to \the\year-02-last];},
   face 3={ \calendar [dates=\the\year-03-01 to \the\year-03-last];},
   face 4={ \calendar [dates=\the\year-04-01 to \the\year-04-last];},
   face 5=\{ \cdot \{ \cdot \}; \}, 
   face 6={ \calendar [dates=\the\year-06-01 to \the\year-06-last];},
   face 7={ \calendar [dates=\the\year-07-01 to \the\year-07-last];},
   face 9={ \calendar [dates=\the\year-09-01 to \the\year-09-last];},
   face 10={\calendar [dates=\the\year-10-01 to \the\year-10-last];},
   face 11={\calendar [dates=\the\year-11-01 to \the\year-11-last];},
   face 12={\calendar [dates=\the\year-12-01 to \the\year-12-last];}
 ];
\end{tikzpicture}
```

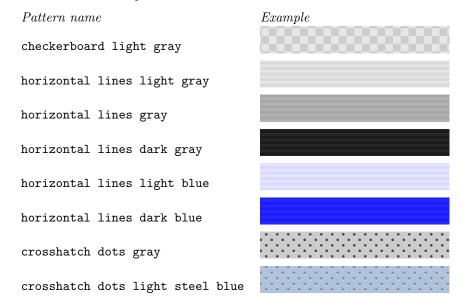
42 Pattern Library

The package defines patterns for filling areas.

42.1 Form-Only Patterns



42.2 Inherently Colored Patterns



43 Petri-Net Drawing Library

```
\usetikzlibrary{petri} % MEX and plain TEX \usetikzlibrary[petri] % ConTEXt
```

This packages provides shapes and styles for drawing Petri nets.

43.1 Places

The package defines a style for drawing places of Petri nets.

/tikz/place (style, no value)

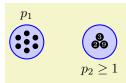
This style indicates that a node is a place of a Petri net. Usually, the text of the node should be empty since places do not contain any text. You should use the label option to add text outside the node like its name or its capacity. You should use the tokens options, explained in Section 43.3, to add tokens inside the place.

```
\begin{array}{c}
p_1 \\
\bullet \bullet \\
p_2 \ge 1
\end{array}
```

/tikz/every place

(style, no value)

This style is evoked by the style place. To change the appearance of places, you can change this style.



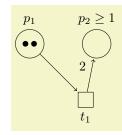
43.2 Transitions

Transitions are also nodes. They should be drawn using the following style:

/tikz/transition (style, no value)

This style indicates that a node is a transition. As for places, the text of a transition should be empty and the label option should be used for adding labels.

To connect a transition to places, you can use the edge command as in the following example:



/tikz/every transition

(style, no value)

This style is evoked by the style transition.

/tikz/pre (style, no value)

This style can be used with paths leading *from* a transition to a place to indicate that the place is in the pre-set of the transition. By default, this style is <-, shorten <=1pt, but feel free to redefine it.

/tikz/post (style, no value)

This style is also used with paths leading *from* a transition to a place, but this time the place is in the post-set of the transition. Again, feel free to redefine it.

/tikz/pre and post

(style, no value)

This style is to be used to indicate that a place is both in the pre- and post-set of a transition.

43.3 Tokens

Interestingly, the most complicated aspect of drawing Petri nets in TikZ turns out to be the placement of tokens.

Let us start with a single token. They are also nodes and there is a simple style for typesetting them.

/tikz/token (style, no value)

This style indicates that a node is a token. By default, this causes the node to be a small black circle. Unlike places and transitions, it *does* make sense to provide text for the token node. Such text will be typeset in a tiny font and in white on black (naturally, you can easily change this by setting the style every token).

/tikz/every token (style, no value)

Change this style to change the appearance of tokens.

In the above example, it is bothersome that we need an extra command for the token node. Worse, when we have *two* tokens on a node, it is difficult to place both nodes inside the node without overlap.

The Petri net library offers a solution to this problem: The children are tokens style.

/tikz/children are tokens

(style, no value)

The idea behind this style is to use trees mechanism for placing tokens. Every token lying on a place is treated as a child of the node. Normally this would have the effect that the tokens are placed below the place and they would be connected to the place by an edge. The children are tokens style, however, redefines the growth function of trees such that it places the children next to each other inside (or, rather, on top) of the place node. Additionally, the edge from the parent node is not drawn.

```
begin{tikzpicture}
    \node[place,label=above:$p_1$] {}

[children are tokens]
    child {node [token] {1}}
    child {node [token] {2}}
    child {node [token] {3}};

\end{tikzpicture}
```

In detail, what happens is the following: Tree growth functions tell TikZ where it should place the children of nodes. These functions get passed the number of children that a node has an the number of the child that should be placed. The special tree growth function for tokens has a special mapping for each possible number of children up to nine children. This mapping decides for each child where it should be placed on top of the place. For example, a single child is placed directly on top of the place. Two children are placed next to each other, separated by the token distance. Three children are placed in a triangle whose side lengths are token distance; and so on up to nine tokens. If you wish to place more than nice tokens on a place, you will have to write your own placement code.

```
p_2
```

```
\begin{tikzpicture}
  \node[place,label=above:$p_2$] {}
  [children are tokens]
  child {node [token] {1}}
  child {node [token,fill=red] {2}}
  child {node [token,fill=red] {2}}
  child {node [token,fill=red] {2}}
  child {node [token] {1}};
  \end{tikzpicture}
```

/tikz/token distance= $\langle distance \rangle$

(no default)

This specifies the distance between the centers of the tokens in the arrangements of the option children are tokens.



```
\begin{tikzpicture}
  \node[place,label=above:$p_3$] {}
  [children are tokens,token distance=1.1ex]
  child {node [token] {}}
  child {node [token,red] {}}
  child {node [token,blue] {}}
  child {node [token,blue] {}}
  child {node [token] {}};
  child {node [token] {}};
```

The children are tokens option gives you a lot of flexibility, but it is a bit cumbersome to use. For this reason there are some options that help in standard situations. They all use children are tokens internally, so any change to, say, the every token style will affect how these options depict tokens.

```
\texttt{/tikz/tokens} = \langle number \rangle
```

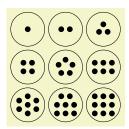
(no default)

This option is given to a place node, not to a token node. The effect of this option is to add $\langle number \rangle$ many child nodes to the place, each having the style token. Thus, the following two pieces of codes have the same effect:



```
\tikz
  \node[place] {}
  [children are tokens]
  child {node [token] {}}
  child {node [token] {}}
  child {node [token] {}};
  tikz
  \node[place,tokens=3] {};
```

It is legal to say tokens=0, no tokens are drawn in this case. This option does not handle ten or more tokens correctly. If you need this many tokens, you will have to program your own code.



/tikz/colored tokens= $\langle color \ list \rangle$

(no default)

This option, which must also be given when a place node is being created, gets a list of colors as parameter. It will then add as many tokens to the place as there are colors in this list, each filled correspondingly.



```
\tikz \node[place,colored tokens={black,black,red,blue}] {};
```

/tikz/structured tokens=⟨token texts⟩

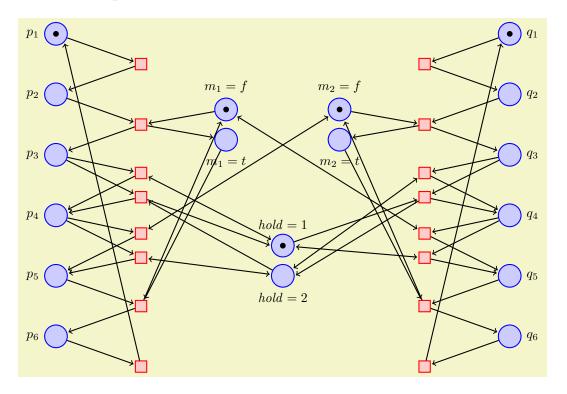
(no default)

This option, which must again be passed to a place, gets a list of texts for tokens. For each text, a new token will be added to the place.



If you use lots of structured tokens, consider redefining the every token style so that the tokens are larger.

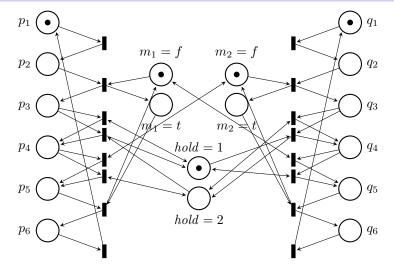
43.4 Examples



```
\begin{tikzpicture}[yscale=-1.6,xscale=1.5,thick,
 every transition/.style={draw=red,fill=red!20,minimum size=3mm},
 every place/.style={draw=blue,fill=blue!20,minimum size=6mm}]
 \foreach \i in \{1, \ldots, 6\} {
    \label=left: p_i \ (p\i) \ at \ (0,\i) \ \{\};
    \node[place,label=right:$q_i$] (qi) at (8,i) {};
 {m1/m_1/f/t/2.25/3,m2/m_2/f/t/2.25/5,h/mathit{hold}/1/2/4.5/4} {
    \node[place,label=above:{$\var = \vala$}] (\name\vala) at (\x,\height) {};
   \node[place,yshift=-8mm,label=below:{$\var = \valb$}] (\name\valb) at (\x,\height) {};
 \node[token] at (p1) {};
                            \node[token] at (q1) {};
 \node[token] at (m1f) {}; \node[token] at (m2f) {};
 \node[token] at (h1) {};
 \node[transition] at (1.5,1.5) {} edge [pre] (p1)
                                                   edge [post] (p2);
 \node[transition] at (1.5,2.5) {}
                                    edge [pre] (p2)
                                                     edge[pre]
                                                                 (m1f)
                                    edge [post](p3)
                                                     edge[post]
                                                                (m1t);
 \node[transition] at (1.5,3.3) {}
                                                   edge [post] (p4)
                                    edge [pre] (p3)
                                    edge [pre and post] (h1);
                                    edge [pre] (p3) edge [pre] (h2)
 \node[transition] at (1.5,3.7) {}
                                    edge [post] (p4) edge [post] (h1.west);
 \node[transition] at (1.5,4.3) {}
                                    edge [pre] (p4) edge [post] (p5)
                                    edge [pre and post] (m2f);
 \node[transition] at (1.5,4.7) {}
                                    edge [pre] (p4) edge [post] (p5)
                                    edge [pre and post] (h2);
 \node[transition] at (1.5,5.5) {}
                                    edge [pre] (p5) edge [pre] (m1t)
                                    edge [post] (p6) edge [post] (m1f);
 \node[transition] at (1.5,6.5) {}
                                    edge [pre] (p6) edge [post] (p1.south east);
 \node[transition] at (6.5,1.5) {}
                                    edge [pre] (q1)
                                                    edge [post] (q2);
 \node[transition] at (6.5,2.5) {}
                                    edge [pre] (q2)
                                                     edge [pre] (m2f)
                                    edge [post] (q3) edge [post] (m2t);
 \node[transition] at (6.5,3.3) {}
                                    edge [pre] (q3) edge [post] (q4)
                                    edge [pre and post] (h2);
 \node[transition] at (6.5,3.7) {}
                                    edge [pre] (q3) edge [pre] (h1)
                                    edge [post] (q4) edge [post] (h2.east);
 \node[transition] at (6.5,4.3) {}
                                    edge [pre] (q4) edge [post] (q5)
                                    edge [pre and post] (m1f);
 \node[transition] at (6.5,4.7) {}
                                    edge [pre] (q4) edge [post] (q5)
                                    edge [pre and post] (h1);
 \node[transition] at (6.5,5.5) {}
                                    edge [pre] (q5) edge [pre] (m2t)
                                    edge [post] (q6) edge [post] (m2f);
 \node[transition] at (6.5,6.5) {}
                                    edge [pre] (q6) edge [post] (q1.south west);
\end{tikzpicture}
```

Here is the same net once more, but with these styles changes:

```
\begin{tikzpicture}[yscale=-1.1,thin,>=stealth,
every transition/.style={fill,minimum width=1mm,minimum height=3.5mm},
every place/.style={draw,thick,minimum size=6mm}]
```



44 Plot Handler Library

```
\usepgflibrary{plothandlers} % MTEX and plain TEX and pure pgf
\usepgflibrary[plothandlers] % ConTEXt and pure pgf
\usetikzlibrary{plothandlers} % MTEX and plain TEX when using TikZ
\usetikzlibrary[plothandlers] % ConTEXt when using TikZ
```

This library packages defines additional plot handlers, see Section 89.3 for an introduction to plot handlers. The additional handlers are described in the following.

This library is loaded automatically by TikZ.

44.1 Curve Plot Handlers

\pgfplothandlercurveto

This handler will issue a \pgfpathcurveto command for each point of the plot, except possibly for the first. As for the line-to handler, what happens with the first point can be specified using \pgfsetmovetofirstplotpoint or \pgfsetlinetofirstplotpoint.

Obviously, the \pgfpathcurveto command needs, in addition to the points on the path, some control points. These are generated automatically using a somewhat "dumb" algorithm: Suppose you have three points x, y, and z on the curve such that y is between x and z:

```
\text{begin{tikzpicture}}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplothandlercurveto
\text{pgfplotstreamstart}
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

In order to determine the control points of the curve at the point y, the handler computes the vector z - x and scales it by the tension factor (see below). Let us call the resulting vector s. Then y + s and y - s will be the control points around y. The first control point at the beginning of the curve will be the beginning itself, once more; likewise the last control point is the end itself.

$\protect\pro$

Sets the factor used by the curve plot handlers to determine the distance of the control points from the points they control. The higher the curvature of the curve points, the higher this value should be. A value of 1 will cause four points at quarter positions of a circle to be connected using a circle. The default is 0.5.

```
\text{\text{begin{tikzpicture}}}
\text{\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\text{\pgfsetplottension{0.75}}
\text{\pgfplottandlercurveto}
\text{\pgfplotstreamstart}
\text{\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\text{\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\text{\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\text{\pgfplotstreamend}
\text{\pgfusepath{stroke}}
\end{tikzpicture}
```

\pgfplothandlerclosedcurve

This handler works like the curve-to plot handler, only it will add a new part to the current path that is a closed curve through the plot points.



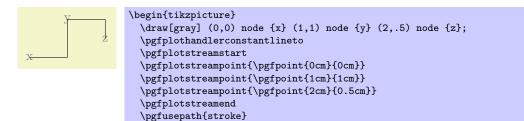
```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerclosedcurve
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
  \end{tikzpicture}
```

44.2 Constant Plot Handlers

There are several plot handlers which produce piecewise constant interpolations between successive points:

\pgfplothandlerconstantlineto

This handler works like the line-to plot handler, only it will produce a connected, piecewise constant path to connect the points.



\pgfplothandlerconstantlinetomarkright

A variant of \pgfplothandlerconstantlineto which places its mark on the right line ends.

\end{tikzpicture}

```
begin{tikzpicture}
    \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
    \pgfplothandlerconstantlinetomarkright
    \pgfplotstreamstart
    \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
    \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
    \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
    \pgfplotstreamend
    \pgfusepath{stroke}
    \end{tikzpicture}
```

\pgfplothandlerconstantlinetomarkmid

A variant of \pgfplothandlerconstantlineto which places its mark on the center of the line.

The plot handler always connects two data points by a horizontal line starting from the previous data points, followed by a vertical line in the middle between the two data points, followed by a horizontal line between the middle and the current data point. This results in a symmetric constant plot handler for constant mesh width.

\pgfplothandlerjumpmarkleft

This handler works like the line-to plot handler, only it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed on the left open pieces.

\pgfplothandlerjumpmarkright

This handler works like the line-to plot handler, only it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed on the right open pieces.

\pgfplothandlerjumpmarkmid

This handler works like the \pgfplothandlerconstantlinetomarkmid, but it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed in the center of the horizontal line segment..

See \pgfplothandlerconstantlinetomarkmid for details.

44.3 Comb Plot Handlers

There are three "comb" plot handlers. Their name stems from the fact that the plots they produce look like "combs" (more or less).

\pgfplothandlerxcomb

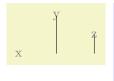
This handler converts each point in the plot stream into a line from the y-axis to the point's coordinate, resulting in a "horizontal comb."

```
begin{tikzpicture}
    \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
    \pgfplothandlerxcomb
    \pgfplotstreamstart
    \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
    \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
    \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
    \pgfplotstreamend
    \pgfusepath{stroke}
    \end{tikzpicture}
```

\pgfplothandlerycomb

This handler converts each point in the plot stream into a line from the x-axis to the point's coordinate, resulting in a "vertical comb."

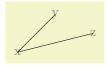
This handler is useful for creating "bar diagrams."



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerycomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
  \end{tikzpicture}
```

\pgfplothandlerpolarcomb

This handler converts each point in the plot stream into a line from the origin to the point's coordinate.



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerpolarcomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
  \end{tikzpicture}
```

PGF bar or comb plots usually draw something from zero to the current plot's coordinate. The "zero" offset can be changed using an input stream which returns the desired offset successively for each processed coordinate.

There are two such streams, which can be configured independently. The first one returns "zeros" for coordinate x, the second one returns "zeros" for coordinate y. They are used as follows.

```
\pgfplotxzerolevelstreamstart
\pgfplotxzerolevelstreamnext % assigns \pgf@x
\pgfplotxzerolevelstreamnext
\pgfplotxzerolevelstreamnext
\pgfplotxzerolevelstreamend
```

```
\pgfplotyzerolevelstreamstart
\pgfplotyzerolevelstreamnext % assigns \pgf@x
\pgfplotyzerolevelstreamend
```

Different zero level streams can be implemented by overwriting these macros.

$\protect\operatorname{\mathtt{f pgfplotxzerolevelstreamconstant}} \$

This zero level stream always returns $\{\langle dimension \rangle\}$ instead of x = 0pt.

It is used for xcomb and xbar.

$\verb|\pgfplotyzerolevelstreamconstant{} \langle dimension \rangle \}|$

This zero level stream always returns $\{\langle dimension \rangle\}$ instead of y = 0pt.

It is used for ycomb and ybar.

44.4 Bar Plot Handlers

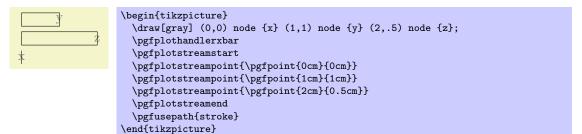
While comb plot handlers produce a line-to operation to generate combs, bar plot handlers employ rectangular shapes, allowing filled bars (or pattern bars).

\pgfplothandlerybar

This handler converts each point in the plot stream into a rectangle from the x-axis to the point's coordinate. The rectangle is placed centered at the x-axis.

\pgfplothandlerxbar

This handler converts each point in the plot stream into a rectangle from the y-axis to the point's coordinate. The rectangle is placed centered at the y-axis.



$/pgf/bar width={\langle dimension \rangle}$

(no default, initially 10pt)

alias /tikz/bar width

Sets the width of \pgfplothandlerxbar and \pgfplothandlerybar to $\{\langle dimension \rangle\}$. The argument $\{\langle dimension \rangle\}$ will be evaluated using the math parser.

```
 \begin{tabular}{ll} \be
```

Sets a shift used by $\protect\protec$

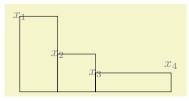
\pgfplotbarwidth

Expands to the value of /pgf/bar width.

\pgfplothandlerybarinterval

This handler is a variant of \pgfplothandlerybar which works with intervals instead of points.

Bars are drawn between successive input coordinates and the width is determined relatively to the interval length.



```
\begin{tikzpicture}
  \draw[gray] (0,2) node {$x_1$} (1,1) node {$x_2$} (2,.5) node {$x_3$} (4,0.7) node {$x_4$};
  \pgfplothandlerybarinterval
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreampoint{\pgfpoint{4cm}{0.7cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

In more detail, if (x_i, y_i) and (x_{i+1}, y_{i+1}) denote successive input coordinates, the bar will be placed above the interval $[x_i, x_{i+1}]$, centered at

$$x_i + \langle bar \ interval \ shift \rangle \cdot (x_{i+1} - x_i)$$

with width

$$\langle bar \ interval \ width \rangle \cdot (x_{i+1} - x_i).$$

Here, $\langle bar\ interval\ shift \rangle$ and $\langle bar\ interval\ width \rangle$ denote the current values of the associated options. If you have N+1 input points, you will get N bars (one for each interval). The y value of the last point will be ignored.

\pgfplothandlerxbarinterval

As \pgfplothandlerybarinterval, this handler provides bar plots with relative bar sizes and offsets, one bar for each y coordinate interval.

```
/pgf/bar interval shift=\{\langle factor \rangle\} (no default, initially 0.5) alias /tikz/bar interval shift
```

Sets the *relative* shift of \pgfplothandlerxbarinterval and \pgfplothandlerybarinterval to \(\lambda factor \rangle \). As \(\rangle pgf \rangle \) bar interval width, the argument is relative to the interval length of the input coordinates.

The argument $\{\langle scale \rangle\}$ will be evaluated using the math parser.

```
\label{eq:pgf/bar interval width=} $$ \begin{array}{c} \operatorname{pgf/bar interval width=} \{\langle scale \rangle\} \\ \operatorname{alias/tikz/bar interval width} \end{array} $$ (no default, initially 1)
```

\begin{tikzpicture}[bar interval width=0.5]

Sets the relative width of \pgfplothandlerxbarinterval and \pgfplothandlerybarinterval to $\{\langle scale \rangle\}$. The argument is relative to $(x_{i+1} - x_i)$ for y bar plots and relative to $(y_{i+1} - y_i)$ for x bar plots.

The argument $\{\langle scale \rangle\}$ will be evaluated using the math parser.

```
\draw[gray]
 (0,3) -- (0,-0.1)
  (1,3) -- (1,-0.1)
  (2,3) -- (2,-0.1)
  (4,3) -- (4,-0.1);
\pgfplothandlerybarinterval
\begin{scope}[bar interval shift=0.25,fill=blue]
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{2cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreampoint{\pgfpoint{4cm}{0.7cm}}
\pgfplotstreamend
\pgfusepath{fill}
\end{scope}
\begin{scope}[bar interval shift=0.75,fill=red]
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{3cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{0.2cm}}
```

\pgfplotstreampoint{\pgfpoint{2cm}{0.7cm}}
\pgfplotstreampoint{\pgfpoint{4cm}{0.2cm}}

\pgfplotstreamend
\pgfusepath{fill}
\end{scope}
\end{tikzpicture}

Please note that bars are always centered, so we have to use shifts 0.25 and 0.75 instead of 0 and 0.5.

44.5 Gapped Plot Handlers

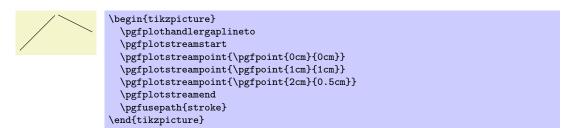
\pgfplothandlergaplineto

This handler will connect the points of the plots by straight line segments. However, at the start and the end of the lines there will be a small gap, given by the following key:

```
/pgf/gap around stream point=\langle dimension \rangle
```

(no default, initially 1.5pt)

The $\langle dimension \rangle$ by which the lines between consecutive stream points are shortened at the beginning and end.



\pgfplothandlergapcycle

Works like \pgfplothandlergaplineto, but the last point is connected to the first in the same fashion:



```
\begin{tikzpicture}
  \pgfplothandlergapcycle
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
  \end{tikzpicture}
```

44.6 Mark Plot Handler

$\protect\pro$

This command will execute the $\langle mark\ code \rangle$ for some points of the plot, but each time the coordinate transformation matrix will be set up such that the origin is at the position of the point to be plotted. This way, if the $\langle mark\ code \rangle$ draws a little circle around the origin, little circles will be drawn at some point of the plot.

By default, a mark is drawn at all points of the plot. However, two parameters r and p influence this. First, only every rth mark is drawn. Second, the first mark drawn is the pth. These parameters can be influenced using the commands below.

```
\begin\tikzpicture
\draw[gray] (0,0)
\pgfplothandlern
\pgfplotstreamst
\pgfplotstreampo
\pgfplotstreampo
\pgfplotstreampo
\pgfplotstreampo
```

```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
  \end{tikzpicture}
```

Typically, the $\langle code \rangle$ will be \pgfuseplotmark{ $\langle plot \ mark \ name \rangle$ }, where $\langle plot \ mark \ name \rangle$ is the name of a predefined plot mark.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\b$

Sets the r parameter to $\langle repeat \rangle$, that is, only every rth mark will be drawn.

$\protect\pro$

Sets the p parameter to $\langle phase \rangle$, that is, the first mark to be drawn is the pth, followed by the (p+r)th, then the (p+2r)th, and so on.

$\protect\pro$

This command works similar to the previous one. However, marks will only be placed at those indices in the given (index list). The syntax for the list is the same as for the \foreach statement. For

example, if you provide the list 1,3,...,25, a mark will be placed only at every second point. Similarly, 1,2,4,8,16,32 yields marks only at those points that are powers of two.

```
\text{begin{tikzpicture}}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\text{pgfplothandlermarklisted}
\{\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
\{1,3}
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

$\protect\operatorname{\mathtt{f plot}}$ mark $\{\langle plot \ mark \ name \rangle\}$

Draws the given $\langle plot \ mark \ name \rangle$ at the origin. The $\langle plot \ mark \ name \rangle$ must have been previously declared using \pgfdeclareplotmark .

$\protect\pro$

Declares a plot mark for later used with the \pgfuseplotmark command.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\b$

This command sets the TEX dimension \pgfplotmarksize to $\langle dimension \rangle$. This dimension is a "recommendation" for plot mark code at which size the plot mark should be drawn; plot mark code may choose to ignore this $\langle dimension \rangle$ altogether. For circles, $\langle dimension \rangle$ should be the radius, for other shapes it should be about half the width/height.

The predefined plot marks all take this dimension into account.

\end{tikzpicture}

\pgfplotmarksize

A TEX dimension that is a "recommendation" for the size of plot marks.

The following plot marks are predefined (the filling color has been set to yellow):

45 Plot Mark Library

```
\label{thm:configure} $$ \usepgflibrary[plotmarks] % MeX and plain TeX and pure pgf $$ \usepgflibrary[plotmarks] % ConTeXt and pure pgf $$ \usetikzlibrary[plotmarks] % MeX and plain TeX when using TikZ $$ \usetikzlibrary[plotmarks] % ConTeXt when using TikZ $$
```

This library defines a number of plot marks.

This library defines the following plot marks in addition to *, x, and + (the filling color has been set to a dark yellow):

```
\pgfuseplotmark{-}
\pgfuseplotmark{|}
\pgfuseplotmark{o}
\pgfuseplotmark{asterisk}
\pgfuseplotmark{star}
\pgfuseplotmark{10-pointed star}
\pgfuseplotmark{oplus}
\pgfuseplotmark{oplus*}
\pgfuseplotmark{otimes}
\pgfuseplotmark{otimes*}
\pgfuseplotmark{square}
\pgfuseplotmark{square*}
\pgfuseplotmark{triangle}
\pgfuseplotmark{triangle*}
\pgfuseplotmark{diamond}
\pgfuseplotmark{diamond*}
\pgfuseplotmark{halfdiamond*}
\pgfuseplotmark{halfsquare*}
\pgfuseplotmark{halfsquare right*}
\pgfuseplotmark{halfsquare left*}
\pgfuseplotmark{pentagon}
\pgfuseplotmark{pentagon*}
\pgfuseplotmark{Mercedes star}
\pgfuseplotmark{Mercedes star flipped}
                                       0 0
\pgfuseplotmark{halfcircle}
                                       \pgfuseplotmark{halfcircle*}
                                       A A A
\pgfuseplotmark{heart}
                                       p p p
\pgfuseplotmark{text}
```

Note that each of the provided marks can be rotated freely by means of mark options={rotate=90} or every mark/.append style={rotate=90}.

```
/pgf/mark color={\langle color \rangle}  (no default, initially empty)
```

Defines the additional fill color for the halfcircle, halfcircle*, halfdiamond* and halfsquare* markers. An empty value uses white (which is the initial configuration). The special value none disables filling of the respective parts.

Note that halfsquare will be filled with mark color, and the starred variant halfsquare* will be filled half with mark color and half with the actual fill color.

```
/pgf/text mark={\langle text \rangle} (no default, initially p)

Changes the text shown by mark=text.

With /pgf/text mark=m: m m

With /pgf/text mark=A: A A
```

There is no limitation about the number of characters or whatever. In fact, any T_EX material can be inserted as $\{\langle text \rangle\}$, including images.

/pgf/text mark as node= $\{\langle boolean \rangle\}$

(no default, initially false)

Configures how mark=text will be drawn: either as \node or as \pgftext.

The first choice is highly flexible and possibly slow, the second is very fast and usually enough.

/pgf/text mark style={\langle options for mark=text \rangle}

(no default)

Defines a set of options which control the appearance of mark=text.

If /pgf/text mark as node=false (the default), $\{\langle options \rangle\}$ is provided as argument to pgftext which provides only some basic keys like left, right, top, bottom, base and rotate.

If /pgf/text mark as node=true, $\{\langle options \rangle\}$ is provided as argument to \node. This means you can provide a very powerful set of options including anchor, scale, fill, draw, rounded corners etc.

46 Profiler Library

by Christian Feuersänger

```
\usepgflibrary{profiler} % MEX and plain TeX and pure pgf \usepgflibrary[profiler] % ConTeXt and pure pgf \usetikzlibrary{profiler} % MEX and plain TeX when using TikZ \usetikzlibrary[profiler] % ConTeXt when using TikZ
```

A library to simplify the optimization of runtime speed of T_EX programs.

It relies on the pdftex primitive \pdfelapsedtime¹⁶ to count (fractional) seconds and counts total time and self time for macro invocations.

46.1 Overview

The intended audience for this library are people writing TeX code which should be optimized. It is certainly not useful for the end-user.

The work flow for the optimization is simple: the preamble contains configuration commands like

```
\usepgflibrary{profiler}
\pgfprofilenewforenvironment{tikzpicture}
\pgfprofilenewforcommand{\pgfkeys}{1}
```

and then, the time between \begin{tikzpicture} and \end{tikzpicture} and the time required to call \pgfkeys will be collected.

At the end, a short usage summary like

```
pgflibraryprofiler(main job) {total time=1.07378sec; (100.0122%) self time=0.034sec; (3.1662%)} pgflibraryprofiler(<ENV>tikzpicture) {total time=1.03978sec; (96.84601%) self time=1.00415sec; (93.52722%)} pgflibraryprofiler(<CS>pgfkeys) {total time=0.03563sec; (3.31726%) self time=0.03563sec; (3.31726%)}
```

will be provided in the log file, furthermore, the same information is available in a text table called $\jobname.profiler.\langle datetime \rangle$.dat which is of the form:

profilerentry	totaltime[s]	totaltime[percent]	selftime[s]	selftime[percent]	
main job	1.07378	100.0122	0.034	3.1662	
<env>tikzpictu</env>	re 1.03978	96.84601	1.00415	93.52722	
<cs>pgfkeys</cs>	0.03563	3.31726	0.03563	3.31726	

Here, the totaltime means the time used for all invocations of the respective profiler entry (one row in the table). The selftime measures time which is not already counted for in another profiler entry which has been invoked within the current one. The example above is not very exciting: the main job consists only of several (quite complex) pictures and nothing else. Thus, its total time is large. However, the self time is very small because the tikzpictures are counted separately, and they have been invoked within the main job. The \pgfkeys control sequence has been invoked within the tikzpicture, that's why the selftime for the tikzpicture is a little bit smaller than its totaltime.

46.2 Requirements

The library works with pdftex and luatex. Furthermore, it requires a more or less recent version of pdftex which supports the \pdfelapsedtime directive.

46.3 Defining Profiler Entries

Unlike profilers for C/C++ or java, this library doesn't extract information about every TeX macro automatically, nor does it collect information for each of them. Instead, every profiler entry needs to be defined explicitly. Only defined profiler entries will be processed.

```
\profilenew{\langle name \rangle}
```

Defines a new profiler entry named $\langle name \rangle$.

This updates a set of internal registers used to track the profiler entry. The $\langle name \rangle$ can be arbitrary, it doesn't need to be related to any TeX macro.

¹⁶The primitive is emulated in luaT_FX.

The actual job of counting seconds is accomplished using $\profilestart{\langle name \rangle}$ followed eventually by the command $\profileend{\langle name \rangle}$.

It doesn't hurt if \pgfprofilenew is called multiple times with the same name.

```
\verb|\pgfprofilenewforcommand[|\langle profiler\ entry\ name\rangle] {\langle nacro\rangle} {\langle arguments\rangle}|
```

Defines a new profiler entry which will measure the time spent in $\langle \text{nacro} \rangle$. This calls \pgfprofilenew and replaces the current definition of $\langle \text{nacro} \rangle$ with a new one.

If $[\langle profiler\ entry\ name \rangle]$ has been provided, this defines the argument for $\protect\ pgfprofilenew$. It is allowed to use the same name for multiple commands; in this case, they are treated as if it where the same command. If the optional argument is not used, the profiler entry will be called '\pgfprofilecs\(macro\)' (\(macro\)\ without backslash) where \pgfprofilecs is predefined to be <CS>.

The replacement macro will collect all required arguments, start counting, invoke the original macro definition and stop counting.

The following macro types are supported within \pgfprofilenewforcommand:

- commands which take one (optional) argument in square brackets followed by one optional argument which has to be delimited by curly braces (use an empty argument for \(\arguments \) in this case),
- commands which take one (optional) argument in square brackets and exactly (arguments) arguments afterwards.

Take a look at \pgfprofilenewforcommandpattern in case you have more complicated commands. Note that the library can't detect if a command has been redefined somewhere.

A variant of \pgfprofilenewforcommand which can be used with arbitrary \(\argument \ patterns \)\. Example:

Note that \pgfprofilenewforcommand is a special case of \pgfprofilenewforcommandpattern:

```
\def\mymacro#1#2{ ... }
\pgfprofilenewforcommand\macro{2}
\pgfprofilenewforcommandpattern{\mymacro}{#1#2}{{#1}{#2}}
```

Thus, $\langle argument\ pattern \rangle$ is a copy-paste from the definition of your command. The $\langle invocation\ pattern \rangle$ is used by the profiler library to invoke the *original* command, so it is closely related to $\langle argument\ pattern \rangle$, but it needs extra curly braces around each argument.

The behavior of $\protect\operatorname{\sc home}$ gfprofilenewforcommandpattern is the same as discussed above: it defines a new profiler entry which will measure the time spent in $(\mbox{\sc home})$. The details about this definition has already been described. Note that up to one optional argument in square brackets is also checked automatically.

If you like to profile a command which doesn't match here for whatever reasons, you'll have to redefine it manually and insert \pgfprofilestart and \pgfprofileend in appropriate places.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

Enables verbose output for every invocation of $\langle profiler\ entry\ name \rangle$.

This is only available for profiler entries for commands (those created by \pgfprofilenewforcommand for example). It will also show all given arguments.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

A variant of $\protect\protec$

This expansion (with \edef) might yield unrecoverable errors for some commands. Handle with care.

$\protect{\protect\pr$

Defines a new profiler entry which measures time spent in the environment $\langle environment \ name \rangle$.

This calls \pgfprofilenew and handles the begin/end of the environment automatically.

The argument for $\protect\operatorname{profilerew}$ is $\protect\operatorname{profilerew}$, or, if this optional argument is not used, it is '\pgfprofileenv'(environment name)' where \pgfprofileenv is predefined as <ENV>. Again, it is permitted to use the same $\protect\operatorname{profilerentry}$ name \permitted multiple times to merge different commands into one output section.

$\protect\pro$

Starts (or resumes) timing of $\langle profiler\ entry\ name \rangle$. The argument must have been declared in the preamble using $\protect\prote$

Nested calls of \pgfprofilestart with the same argument will be ignored.

The invocation of this command doesn't change the environment: it doesn't introduce any TEX groups nor does it modify the token list.

$\protect\pro$

Stops (or interrupts) timing of $\langle profiler\ entry\ name \rangle$.

This command finishes a preceding call to \pgfprofilestart.

\pgfprofilepostprocess

For LATEX, this command is installed automatically in \end{document}. It stops all running timings, evaluates them and returns the result into the logfile. Furthermore, it generates a text table called \jobname.profiler. $\langle YYYY \rangle - \langle MM \rangle - \langle DD \rangle_{-} \langle HH \rangle h_{-} \langle MM \rangle m.dat$ with the same information.

Note that the profiler library predefines two profiler entries, namely main job which counts time from the beginning of the document until \pgfprofilepostprocess and preamble which counts time from the beginning of the document until \begin{document}.

$\protect{pgfprofilesetrel{profiler entry name}} (initially main job)$

Sets the profiler entry whose total time will be used to compute all other relative times. Thus, $\langle profiler\ entry\ name \rangle$ will use 100% of the total time per definition, all other relative times are relative to this one.

$\propto file if is running {\langle profiler\ entry\ name \rangle} {\langle true\ code \rangle} {\langle false\ code \rangle}$

Invokes $\{\langle true\ code \rangle\}$ if $\{\langle profiler\ entry\ name \rangle\}$ is currently running and $\{\langle false\ code \rangle\}$ otherwise.

47 Shadings Library

```
\usepgflibrary{shadings} % MEX and plain TeX and pure pgf \usepgflibrary[shadings] % ConTeXt and pure pgf \usetikzlibrary{shadings} % MEX and plain TeX when using TikZ \usetikzlibrary[shadings] % ConTeXt when using TikZ
```

The package defines a number of shadings in addition to the ball and axis shadings that are available by default.

In the following, the shadings defined in the library are listed in alphabetical order. The colors of some of these shadings can be configured using special options (like left color). These options implicitly select the shading.

The three shadings axis, ball, and radial are always defined, even when this library is not used.

Shading axis

In this always-defined shading the colors change gradually between three horizontal lines. The top line is at the top (uppermost) point of the path, the middle is in the middle, the bottom line is at the bottom of the path.

/tikz/top color= $\langle color \rangle$

(no default)

This option sets the color to be used at the top in an axis shading. When this option is given, several things happen:

- 1. The shade option is selected.
- 2. The shading=axis option is selected.
- 3. The middle color of the axis shading is set to the average of the given top color $\langle color \rangle$ and of whatever color is currently selected for the bottom.
- 4. The rotation angle of the shading is set to 0.



\tikz \draw[top color=red] (0,0) rectangle (2,1);

/tikz/bottom color= $\langle color \rangle$

(no default)

This option works like top color, only for the bottom color.

/tikz/middle color= $\langle color \rangle$

(no default)

This option specifies the color for the middle of an axis shading. It also sets the shade and shading=axis options, but it does not change the rotation angle.

Note: Since both top color and bottom color change the middle color, this option should be given *last* if all of these options need to be given:



\tikz \draw[top color=white,bottom color=black,middle color=red]
(0,0) rectangle (2,1);

/tikz/left color= $\langle color \rangle$

(no default)

This option does exactly the same as top color, except that the shading angle is set to 90°.

/tikz/right color= $\langle color \rangle$

(no default)

Works like left color.

Shading ball

This always-defined shading fills the path with a shading that "looks like a ball." The default "color" of the ball is blue (for no particular reason).

/tikz/ball color=(color)

(no default)

This option sets the color used for the ball shading. It sets the shade and shading=ball options. Note that the ball will never "completely" have the color $\langle color \rangle$. At its "highlight" spot a certain amount of white is mixed in, at the border a certain amount of black. Because of this, it also makes sense to say ball color=white or ball color=black



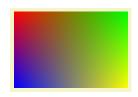




```
\begin{tikzpicture}
  \shade[ball color=white] (0,0) circle (2ex);
  \shade[ball color=red] (1,0) circle (2ex);
  \shade[ball color=black] (2,0) circle (2ex);
  \end{tikzpicture}
```

Shading bilinear interpolation

This shading fills a rectangle with colors that a bilinearly interpolated between the colors in the four corners of the rectangle. These four colors are called lower left, lower right, upper left, and upper right. By changing these color, you can change the way the shading looks. The library also defines four options, called the same way, that can be used to set these colors and select the shading implicitly.



/tikz/lower left= $\langle color \rangle$

(no default, initially white)

Sets the color to be used in a bilinear interpolation shading for the lower left corner. Also, this options selects this shading and sets the shade option.

/tikz/upper left= $\langle color \rangle$

(no default, initially white)

Works like lower left.

/tikz/upper right=⟨color⟩

(no default, initially white)

Works like lower left.

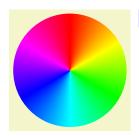
/tikz/lower right= $\langle color \rangle$

(no default, initially white)

Works like lower left.

Shading color wheel

This shading fills the path with a color wheel.



<page-header> \tikz \shade[shading=color wheel] (0,0) circle (1.5);

To produce a color ring, cut out a circle from the color wheel:



```
\tikz \shade[shading=color wheel] [even odd rule]
  (0,0) circle (1.5)
  (0,0) circle (1);
```

Shading color wheel black center

This shading looks like a color wheel, but the brightness drops to zero in the center.



\tikz \shade[shading=color wheel black center] (0,0) circle (1.5);

Shading color wheel white center

This shading looks like a color wheel, but the saturation drops to zero in the center.



\tikz \shade[shading=color wheel white center] (0,0) circle (1.5);

Shading Mandelbrot set

This shading is just for fun. It fills the path with a zoomable Mandelbrot set. Note that this is *not* a bitmap graphic. Rather, the Mandelbrot set is *computed by the PDF renderer* and can be zoomed arbitrarily (give it a try, if you have a fast computer).



\tikz \shade[shading=Mandelbrot set] (0,0) rectangle (2,2);

Shading radial

This always-defined shading fills the path with a gradual sweep from a certain color in the middle to another color at the border. If the path is a circle, the outer color will be reached exactly at the border. If the shading is not a circle, the outer color will continue a bit towards the corners. The default inner color is gray, the default outer color is white.

/tikz/inner color= $\langle color \rangle$

(no default)

This option sets the color used at the center of a radial shading. When this option is used, the shade and shading=radial options are set.



\tikz \draw[inner color=red] (0,0) rectangle (2,1);

/tikz/outer color=⟨color⟩

(no default)

This option sets the color used at the border and outside of a radial shading.



\tikz \draw[outer color=red,inner color=white]
 (0,0) rectangle (2,1);

48 Shadows Library

```
\label{thm:configure} $$ \usepgflibrary[shadows] % MeX and plain TeX and pure pgf $$ \usepgflibrary[shadows] % ConTeXt and pure pgf $$ \usetikzlibrary[shadows] % MeX and plain TeX when using TikZ $$ \usetikzlibrary[shadows] % ConTeXt when using TikZ $$
```

This library defines styles that help adding a (partly) transparent shadow to a path or node.

48.1 Overview

A *shadow* is usually a black or gray area that is drawn behind a path or a node, thereby adding visual depth to a picture. The **shadows** library defines options that make it easy to add shadows to paths. Internally, these options are based on using the **preaction** option to use a path twice: Once for drawing the shadow (slightly shifted) and once for actually using the path.

Note that you can only add shadows to paths, not to whole scopes.

In addition to the general shadow option, there exist special options like circular shadow. These can only (sensibly) be used with a special kind of path (for circular shadow, a circle) and, thus, they are not as general. The advantage is, however, that they are more visually pleasing since these shadows blend smoothly with the background. Note that these special shadows use fadings, which few printers will support.

48.2 The General Shadow Option

The shadows are internally created by using a single option called **general shadow**. The different options like **drop shadow** or **copy shadow** only differ in the commands that they preset.

You will not need to use this option directly under normal circumstances.

```
/tikz/general shadow=\langle shadow options\rangle
```

(default empty)

This option should be given to a **\path** or a **node**. It has the following effect: Before the path is used normally, it is used once with the $\langle shadow\ options \rangle$ in force. Furthermore, when the path is "preused" in this way, it is shifted and scaled a little bit.

In detail, the following happens: A preaction is used to paint the path in a special manner before it is actually painted. This "special" manner is as follows: The options in $\langle shadow\ options \rangle$ are used for painting this path. Typically, the $\langle shadow\ options \rangle$ will contain options like fill=black to create, say, a black shadow. Furthermore, after the $\langle shadow\ options \rangle$ have been set up, the following extra canvas transformations are applied to the path: It is scaled by shadow scale (with the origin of scaling at the path's center) and it is shifted by shadow xshift and shadow yshift.

Note that since scaling and shifting is done using canvas transformations, shadows are not taken into account when the picture's bounding box is computed.



```
\tikz [even odd rule]
\draw [general shadow={fill=red}] (0,0) circle (.5) (0.5,0) circle (.5);
```

/tikz/shadow scale= $\langle factor \rangle$

(no default, initially 1)

Shadows are scaled by $\langle factor \rangle$.



```
\tikz [even odd rule]
  \draw [general shadow={fill=red,shadow scale=1.25}]
  (0,0) circle (.5) (0.5,0) circle (.5);
```

/tikz/shadow xshift= $\langle dimension \rangle$

(no default, initially Opt)

Shadows are shifted horizontally by $\langle dimension \rangle$.



```
\tikz [even odd rule]
\draw [general shadow={fill=red,shadow xshift=-5pt}]
(0,0) circle (.5) (0.5,0) circle (.5);
```

```
/tikz/shadow yshift=\langle dimension\rangle
```

(no default, initially Opt)

Shadows are shifted vertically by $\langle dimension \rangle$.

48.3 Shadows for Arbitrary Paths and Shapes

48.3.1 Drop Shadows

/tikz/drop shadow=\langle shadow options\rangle

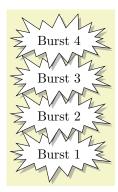
(default empty)

This option adds a drop shadow to a **\path** or a **node**. It uses the **general shadow** and passes the $\langle shadow\ options \rangle$ to it, plus, before them, the following extra options:

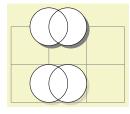
```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex, opacity=.5, fill=black!50, every shadow
```



\tikz [even odd rule]
 \filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);



\begin{tikzpicture}
 \foreach \i in {1,...,4}
 \node[starburst,drop shadow,fill=white,draw] at (0,\i) {Burst \i};
\end{tikzpicture}



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \filldraw [drop shadow={opacity=1},fill=white]
    (1,2) circle (.5) (1.5,2) circle (.5);

\filldraw [drop shadow={opacity=0.25},fill=white]
    (1,.5) circle (.5) (1.5,.5) circle (.5);
\end{tikzpicture}
```

/tikz/every shadow

(style, initially empty)

This style is executed in addition to any $\langle shadow\ options \rangle$ for each shadow. Use this style to reconfigure the way shadows are drawn.



\begin{tikzpicture}[every shadow/.style={opacity=.8,fill=blue!50!black}]
\filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);
\end{tikzpicture}

48.3.2 Copy Shadows

A copy shadow is not really a shadow. Rather, it looks like another copy of the path drawn behind the path and a little bit offset. This creates the visual impression of having multiple copies of the path/object present.

```
/tikz/copy shadow=\langle shadow\ options \rangle
```

(default empty)

This shadow installs the following default options:

```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex, every shadow
```

Furthermore, the options fill= $\langle fill\ color \rangle$ and draw= $\langle draw\ color \rangle$ are also set, where the $\langle fill\ color \rangle$ and $\langle draw\ color \rangle$ are the fill and draw colors used for the main path.

```
\begin{tikzpicture}
Hello World!
                   \node [copy shadow,fill=blue!20,draw=blue,thick] {Hello World!};
                   \node at (0,-1) [copy shadow={shadow xshift=1ex,shadow yshift=1ex},
Hello World!
                                    fill=blue!20,draw=blue,thick]
                     {Hello World!};
Hello World!
                   \node at (0,-2) [copy shadow={opacity=.5}, tape,
                                    fill=blue!20,draw=blue,thick]
                     {Hello World!};
Hello World!
                   \% We have to repeat the left color since shadings are not
                   % automatically applied to shadows
                   \node at (0,-3) [copy shadow={left color=blue!50},
                                    left color=blue!50,draw=blue,thick]
                     {Hello World!};
                 \end{tikzpicture}
```

/tikz/double copy shadow=(shadow options)

(default empty)

This shadow works like a copy shadow, only the shadow is added twice, the second time with the double xshift and yshift.

```
\begin{tikzpicture}
Hello World!
                   \node [double copy shadow,fill=blue!20,draw=blue,thick] {Hello World!};
                   \node at (0,-1) [double copy shadow={shadow xshift=1ex,shadow yshift=1ex},
Hello World!
                                    fill=blue!20,draw=blue,thick]
                     {Hello World!};
Hello World!
                   \node at (0,-2) [double copy shadow={opacity=.5},tape,
                                    fill=blue!20,draw=blue,thick]
                     {Hello World!};
Hello World!
                   \node at (0,-3) [double copy shadow={left color=blue!50},
                                    left color=blue!50,draw=blue,thick]
                     {Hello World!};
                 \end{tikzpicture}
```

48.4 Shadows for Special Paths and Nodes

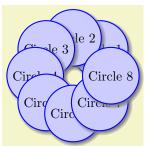
The shadows in this section should normally be added only to paths that have a special shape. They will look strange with other shapes.

```
/tikz/circular drop shadow=(shadow options)
```

(no default)

This shadow works like a drop shadow, only it adds a circular fading to the shadow. This means that the shadow will fade out at the border. The following options are preset for this shadow:

```
shadow scale=1.1, shadow xshift=.3ex, shadow yshift=-.3ex, fill=black, path fading={circle with fuzzy edge 15 percent}, every shadow,
```



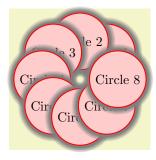
```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
   \node[circle,circular drop shadow,draw=blue,fill=blue!20,thick]
    at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

/tikz/circular glow=(shadow options)

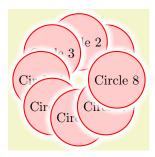
(no default)

This shadow works much like the circular shadow, only it is not shifted. This creates a visual effect of a "glow" behind the circle. The following options are preset for this shadow:

```
shadow scale=1.25, shadow xshift=0pt, shadow yshift=0pt, fill=black, path fading={circle with fuzzy edge 15 percent}, every shadow,
```



```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow,fill=red!20,draw=red,thick]
  at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

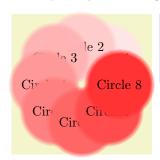


```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow={fill=white},fill=red!20,draw=red,thick]
  at (\i*45:1) {Circle \i};
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow={fill=green},fill=black,text=green!50!black]
  at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

An especially interesting effect can be achieved by only using the glow and not filling the path:



```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow={fill=red!\i0}]
  at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

49 Shape Library

49.1 Overview

In addition to the standard shapes rectangle, circle and coordinate, there exist a number of additional shapes defined in different shape libraries. Most of these shapes have been contributed by Mark Wibrow. In the present section, these shapes are described. Note that the library shapes is provided for compatibility only. Please include sublibraries like shapes.geometric or shapes.misc directly.

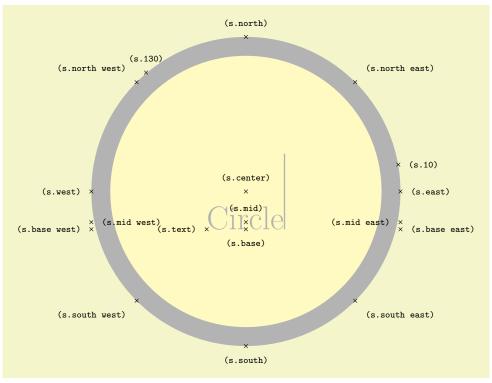
The appearance of shapes is influenced by numerous parameters like minimum height or inner xsep. These general parameters are documented in Section 16.2.2

49.2 Predefined Shapes

The three shapes rectangle, circle, and coordinate are always defined and no library needs to be loaded for them. While the coordinate shape defines only the center anchor, the other two shapes define a standard set of anchors.

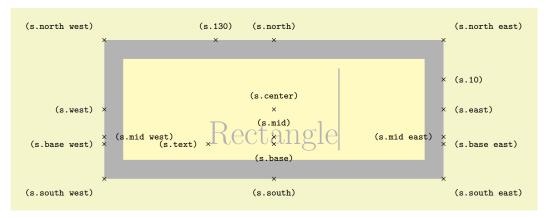
Shape circle

This shape draws a tightly fitting circle around the text. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



Shape rectangle

This shape, which is the standard, is a rectangle around the text. The inner and outer separations (see Section 16.2.2) influence the white space around the text. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\Huge
\begin{tikzpicture}
\node[name=s,shape=rectangle,shape example] {Rectangle\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
    west/left, center/above, east/right,
    mid west/right, mid/above, mid east/left,
    base west/left, base/below, base east/right,
    south west/below left, south/below, south east/below right,
    text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.3 Geometric Shapes

```
\usepgflibrary{shapes.geometric} % MEX and plain TeX and pure pgf \usepgflibrary[shapes.geometric] % ConTeXt and pure pgf \usetikzlibrary{shapes.geometric} % MEX and plain TeX when using TikZ \usetikzlibrary[shapes.geometric] % ConTeXt when using TikZ
```

This library defines different shapes that correspond to basic geometric objects like ellipses or polygons.

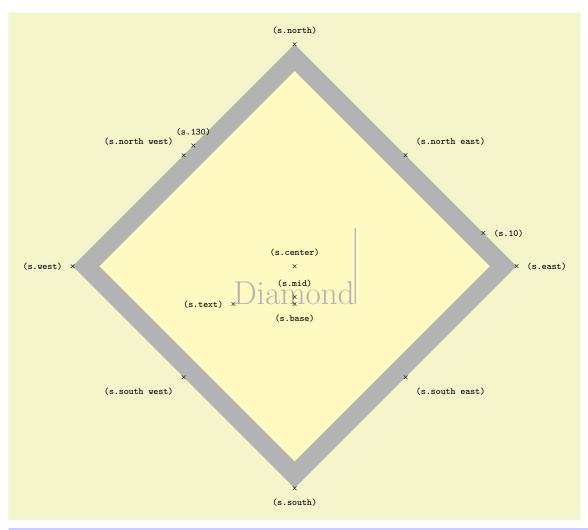
Shape diamond

This shape is a diamond tightly fitting the text box. The ratio between width and height is 1 by default, but can be changed by setting the shape aspect ratio using the following PGF key (to use this key in TikZ simply remove the /pgf/ path).

```
pgf/aspect = \langle value \rangle (no default, initially 1.0)
```

The aspect is a recommendation for the quotient of the width and the height of a shape. This key calls the macro \pgfsetshapeaspect.

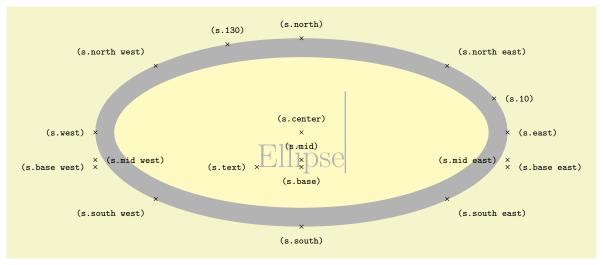
The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=diamond,shape example] {Diamond\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
      {north west/above left, north/above, north east/above right,
      west/left, center/above, east/right,
      mid/above,
      base/below,
      south west/below left, south/below, south east/below right,
      text/left, 10/right, 130/above}
      \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape ellipse

This shape is an ellipse tightly fitting the text box, if no inner separation is given. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\Huge
\begin{tikzpicture}
\node[name=s,shape=ellipse,shape example] {Ellipse\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
    west/left, center/above, east/right,
    mid west/right, mid/above, mid east/left,
    base west/left, base/below, base east/right,
    south west/below left, south/below, south east/below right,
    text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape trapezium

This shape is a trapezium, that is, a quadrilateral with a single pair of parallel lines (this can sometimes be known as a trapezoid). The trapezium shape supports the rotation of the shape border, as described in Section 16.2.2.

The lower internal angles at the lower corners of the trapezium can be specified independently, and the resulting extensions are in addition to the natural dimensions of the node contents (which includes any inner sep.

```
\text{\lambda} \text{
```

The PGF keys to set the lower internal angles of the trapezium are shown below. To use these keys in TikZ, simply remove the /pgf/ path.

```
/pgf/trapezium left angle=\langle \( \) (no default, initially 60)
   Sets the lower internal angle of the left side.
/pgf/trapezium right angle=\langle \( \) (no default, initially 60)
   Sets the lower internal angle of the right side.
/pgf/trapezium angle=\langle \( \) (style, no default)
```

This key stores no value itself, but sets the value of the previous two keys to $\langle angle \rangle$.

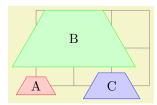
Regardless of the rotation of the shape border, the width and height of the trapezium are as follows:

```
height width
```

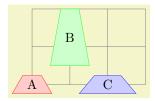
/pgf/trapezium stretches=\langle boolean \rangle

(default true)

This key controls whether PGF allows the width and the height of the trapezium to be enlarged independently, when considering any minimum size specification. This is initially false, ensuring that the shape "looks the same but bigger" when enlarged.



By setting \(\langle boolean \rangle \) to true, the trapezium can be stretched horizontally or vertically.

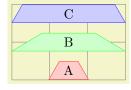


```
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\begin{tikzpicture}
\tikzset{trapezium stretches=true}
\draw [help lines] grid (3,2);
\node [my node=red] {A};
\node [my node=green, minimum height=1.5cm] at (1, 1.25) {B};
\node [my node=blue, minimum width=1.5cm] at (2, 0) {C};
\end{tikzpicture}
```

/pgf/trapezium stretches body= $\langle boolean \rangle$

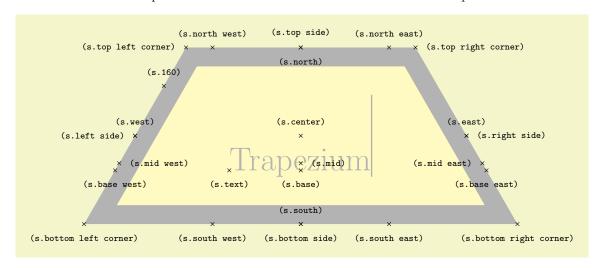
(default true)

This is similar to the trapezium stretches key except that when $\langle boolean \rangle$ is true, PGF enlarges only the body of the trapezium when applying minimum width.



```
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \node [my node=red] at (1.5,.25) {A};
  \node [my node=green, minimum width=3cm, trapezium stretches]
  at (1.5,1) {B};
  \node [my node=blue, minimum width=3cm, trapezium stretches body]
  at (1.5,1.75) {C};
\end{tikzpicture}
```

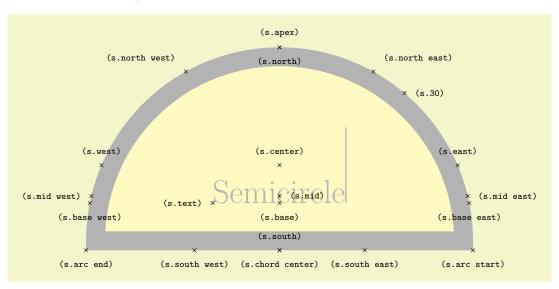
The anchors for the trapezium are shown below. The anchor 160 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=trapezium, shape example, inner sep=1cm]
   {Trapezium\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {bottom left corner/below, top right corner/right,
    top left corner/left,
                             bottom right corner/below,
    bottom side/below,
                             left side/left.
                             top side/above,
    right side/right,
    center/above, text/below,
                                    mid/right,
                                                      base/below.
    mid west/right, base west/below, mid east/left,
                                                      base east/below.
                 east/above,
    west/above,
                                    north/below,
                                                      south/above,
    north west/above, north east/above,
    south west/below, south east/below, 160/above}
 \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
   node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape semicircle

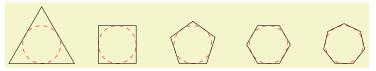
This shape is a semicircle, which tightly fits the node contents. This shape supports the rotation of the shape border, as described in Section 16.2.2. The anchors for the semicircle shape are shown below. Anchor 30 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=semicircle,shape border rotate=0,shape example, inner sep=1cm]
   {Semicircle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
   {apex/above,
                     arc start/below, arc end/below, chord center/below,
    center/above,
                     base/below,
                                    mid/right,
                                                      text/left,
    base west/below, base east/below, mid west/left, mid east/right,
    north/below,
                   south/above,
                                      east/above,
                                                      west/above,
    north west/above left, north east/above right,
                           south east/below, 30/right}
    south west/below.
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape regular polygon

This shape is a regular polygon, which, by default, is drawn so that a side (rather than a corner) is always at the bottom. This shape supports the rotation as described in Section 16.2.2, but the border of the polygon is *always* constructed using the incircle, whose radius is calculated to tightly fit the node contents (including any inner sep).



```
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[red, dashed] (\a*2,0) circle(0.5cm);
    \node[regular polygon, regular polygon sides=\a, draw,
    inner sep=0.3535cm] at (\a*2,0) {};
  }
\end{tikzpicture}
```

If the node is enlarged to any specified minimum size, this is interpreted as the diameter of the circumcircle, that is, the circle that passes through all the corners of the polygon border.

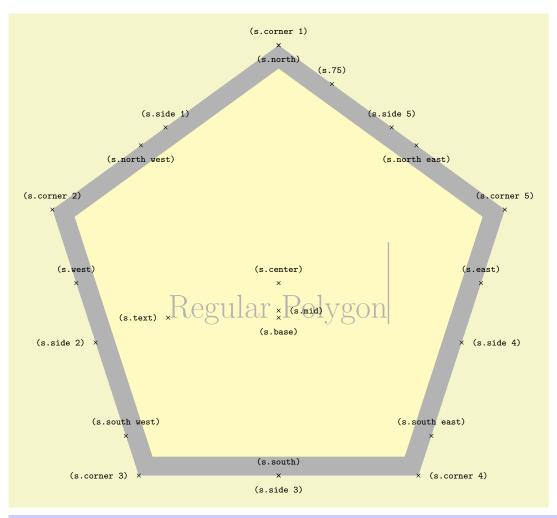


```
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[blue, dashed] (\a*2,0) circle(0.5cm);
    \node[regular polygon, regular polygon sides=\a, minimum size=1cm, draw] at (\a*2,0) {};
  }
\end{tikzpicture}
```

There is a PGF key to set the number of sides for the regular polygon. To use this key in TikZ, simply remove the /pgf/ path.

```
/pgf/regular polygon sides=\(integer\) (no default, initially 5)
```

The anchors for a regular polygon shape are shown below. The anchor 75 is an example of a border anchor.



Shape star

This shape is a star, which by default (minus any transformations) is drawn with the first point pointing upwards. This shape supports the rotation as described in Section 16.2.2, but the border of the star is always constructed using the incircle.

A star should be thought of as having a set of "inner points" and "outer points". The inner points of the border are based on the radius of the circle which tightly fits the node contents, and the outer points are based on the circumcircle, the circle that passes through every outer point. Any specified minimum size, width or height, is interpreted as the diameter of the circumcircle.



```
\begin{tikzpicture}
    \draw [help lines] (0,0) grid (2,2);
    \draw [blue, dashed] (1,1) circle(1cm);
    \draw [red, dashed] (1,1) circle(.5cm);
    \node [star, star point height=.5cm, minimum size=2cm, draw]
        at (1,1) {S};
\end{tikzpicture}
```

The PGF keys to set the number of star points, and the height of the star points, are shown below. To use these keys in TikZ, simply remove the /pgf/ path.

```
pgf/star\ points = \langle integer \rangle (no default, initially 5)
```

Sets the number of points for the star.

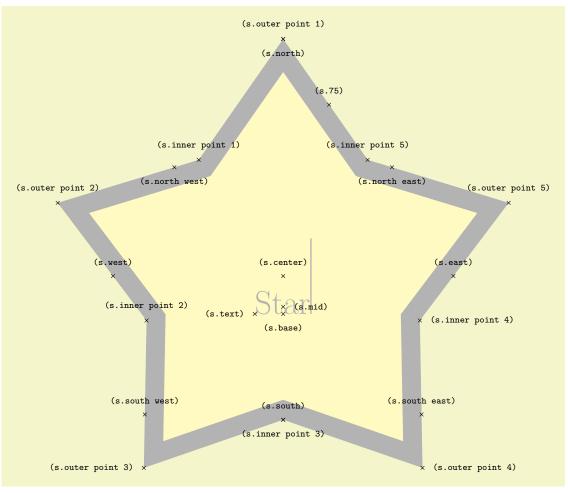
```
/pgf/star\ point\ height=\langle distance\rangle \qquad \qquad (no\ default,\ initially\ .5cm)
```

Sets the height of the star points. This is the distance between the inner point and outer point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the point height.

```
/pgf/star point ratio=\(\number\) (no default, initially 1.5)
```

Sets the ratio between the inner point and outer point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the ratio.

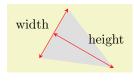
The inner and outer points form the principle anchors for the star, as shown below (anchor 75 is an example of a border anchor).



```
\Huge
\begin{tikzpicture}
\node[name=s, shape=star, star points=5, star point ratio=1.65, shape example, inner sep=1.5cm]
    {Star\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
        {inner point 1/above, inner point 2/above, inner point 3/below, inner point 4/right,
        inner point 5/above, outer point 1/above, outer point 2/above, outer point 3/left,
        outer point 4/right, outer point 5/above,
        center/above, text/left, mid/right, base/below, 75/above,
        west/above, east/above, north/below, south/above,
        north east/below, south east/above, north west/below, south west/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
        node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape isosceles triangle

This shape is an isosceles triangle, which supports the rotation of the shape border, as described in Section 16.2.2. The angle of rotation determines the direction in which the apex of the triangle points (provided no other transformations are applied). However, regardless of the rotation of the shape border, the width and height are always considered as follows:



```
\begin{tikzpicture}[>=stealth, every node/.style={text=black},
    shape border uses incircle, shape border rotate=-30]
\node [isosceles triangle, fill=gray!25, minimum width=1.5cm] (t) {};
\draw [red, <->] (t.left corner) -- (t.right corner)
    node [midway, above left] {width};
\draw [red, <->] (t.apex) -- (t.lower side)
    node [midway, above right] {height};
\end{tikzpicture}
```

There are PGF keys to customize this shape. To use these keys in TikZ, simply remove the /pgf/ path.

```
/pgf/isosceles triangle apex angle=\langle angle \rangle
```

(no default, initially 30)

Sets the angle of the apex of the isosceles triangle.

```
/pgf/isosceles triangle stretches=\langle boolean \rangle
```

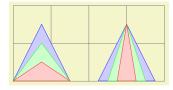
(default true)

By default $\langle boolean \rangle$ is false. This means, that when applying any minimum width or minimum height requirements, increasing the height will increase the width (and vice versa), in order to keep the apex angle the same.



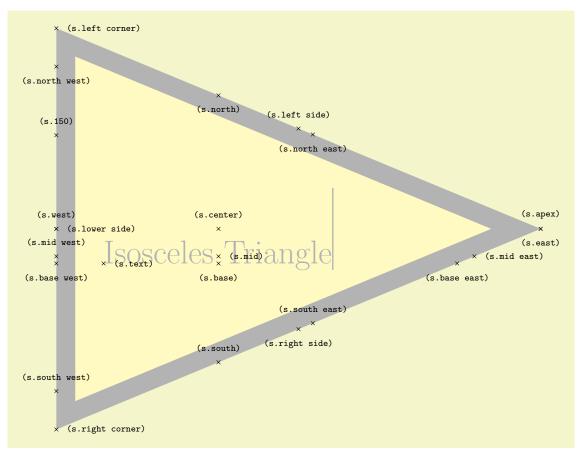
```
\begin{tikzpicture}[paint/.style={draw=#1!75, fill=#1!20}]
\tikzset{every node/.style={isosceles triangle, draw, inner sep=0pt,
    anchor=left corner, shape border rotate=90}}
\draw[help lines] grid(4,2);
\foreach \a/\c in {1.5/blue, 1/green, 0.5/red}{
    \node[paint=\c, minimum height=\a cm] at (0,0) {};
    \node[paint=\c, minimum width=\a cm] at (2,0) {};
}
\end{tikzpicture}
```

However, by setting (boolean) to true, minimum width and height can be applied independently.



```
\begin{tikzpicture}[paint/.style={draw=#1!75, fill=#1!20}]
  \tikzset{every node/.style={isosceles triangle, draw, inner sep=0pt,
      anchor=south, shape border rotate=90, isosceles triangle stretches}}
  \draw[help lines] grid(4,2);
  \foreach \a/\c in {1.5/blue, 1/green, 0.5/red}{
      \node[paint=\c, minimum height=\a cm, minimum width=1.5cm] at (0.75,0) {};
  \node[paint=\c, minimum width=\a cm, minimum height=1.5cm] at (3,0) {};
  }
  \end{tikzpicture}
```

The anchors for the isosceles triangle are shown below (anchor 150 is an example of a border anchor). Note that, somewhat confusingly, the anchor names such as left side and right corner are named as if the triangle is rotated to 90 degrees. Note also that the center anchor does not necessarily correspond to any kind of geometric center.



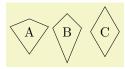
```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=isosceles triangle, shape example, inner xsep=1cm]
    {Isosceles Triangle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {apex/above,
                     left corner/right, right corner/right,
    left side/above, right side/below, lower side/right,
                                        150/above,
     center/above,
                     text/right,
     mid/right,
                     mid west/above,
                                         mid east/right,
    base/below,
                     base west/below, base east/below,
     west/above, east/below, north/below, south/above,
     north west/below, north east/below,
     south west/above, south east/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape kite

This shape is a kite, which supports the rotation of the shape border, as described in Section 16.2.2. There are PGF keys to specify the upper and lower vertex angles of the kite. To use these keys in TikZ, simply remove the /pgf/ path.

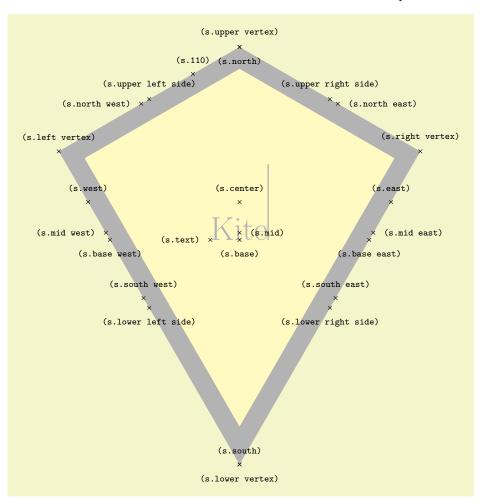
```
/pgf/kite upper vertex angle=⟨angle⟩
    Sets the upper internal angle of the kite.
/pgf/kite lower vertex angle=⟨angle⟩
    Sets the lower internal angle of the kite.
/pgf/kite vertex angles=⟨angle specification⟩
    (no default, initially 60)
    (no default)
```

This key sets the keys for both the upper and lower vertex angles (it stores no value itself). $\langle angle \ specification \rangle$ can be pair of angles in the form $\langle upper \ angle \rangle$ and $\langle lower \ angle \rangle$, or a single angle. In this latter case, both the upper and lower vertex angles will be the same.



```
\begin{tikzpicture}
  \tikzstyle{every node}=[kite, draw]
  \node[kite upper vertex angle=135, kite lower vertex angle=70] at (0,0) {A};
  \node[kite vertex angles=90 and 45] at (1,0) {B};
  \node[kite vertex angles=60] at (2,0) {C};
  \end{tikzpicture}
```

The anchors for the kite are shown below. Anchor 110 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
 \node[name=s, shape=kite, shape example, inner sep=1.5cm]
   {Kite\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {upper vertex/above, left vertex/above,
                                             lower vertex/below,
    right vertex/above, upper left side/above, upper right side/above,
    lower left side/below, lower right side/below,
    center/above, text/left,
                                    mid/right,
                                                        base/below,
    mid west/left, base west/below, mid east/right,
                                                       base east/below,
    west/above,
                   east/above,
                                     north/below,
                                                      south/above,
    north west/left, north east/right,
    south west/above, south east/above, 110/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
   node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape dart

This shape is a dart (which can also be known as an arrowhead or concave kite). This shape supports the rotation of the shape border, as described in Section 16.2.2. The angle of the border rotation determines the direction in which the dart points (unless other transformations have been applied).

There are PGF keys to set the angle for the 'tip' of the dart and the angle between the 'tails' of the dart. To use these keys in TikZ, simply remove the /pgf/ path.

```
tip angle

dart

tail angle
```

```
\begin{tikzpicture}
  \node[dart, draw, gray, shape border uses incircle, shape border rotate=45]
     (d) {dart};
  \draw [<->] (d.tip)++(202.5:.5cm) arc(202.5:247.5:.5cm);
  \node [left=.5cm] at (d.tip) {tip angle};
  \draw [<->] (d.tail center)++(157.5:.5cm) arc(157.5:292.5:.5cm);
  \node [right] at (d.tail center) {tail angle};
  \end{tikzpicture}
```

```
/pgf/dart tip angle=\langle angle \rangle
```

(no default, initially 45)

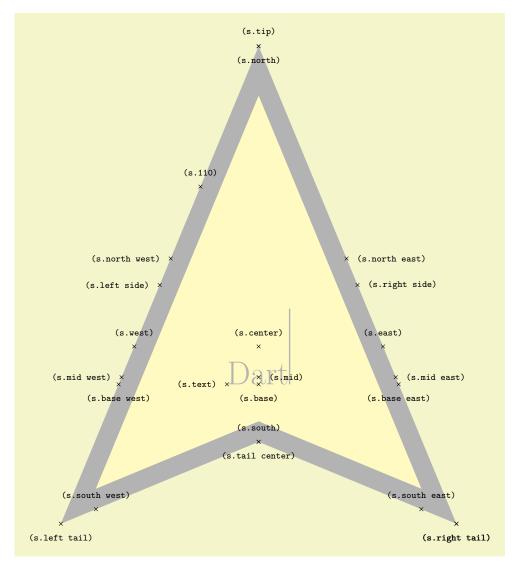
Sets the angle at the tip of the dart.

```
/pgf/dart tail angle=\langle angle \rangle
```

(no default, initially 135)

Sets the angle between the tails of the dart.

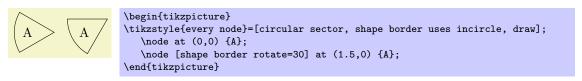
The anchors for the dart shape are shown below (note that the shape is rotated 90 degrees anti-clockwise). Anchor 110 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=dart, shape border rotate=90, shape example, inner sep=1.25cm]
   {Dart\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {tip/above,
                    tail center/below, right tail/below,
    left tail/below, right tail/below, left side/left, right side/right,
                     text/left,
                                       mid/right,
    center/above.
                                                          base/below.
    mid west/left,
                    base west/below, mid east/right, base east/below,
                                       north/below,
                                                         south/above,
    west/above.
                     east/above.
    north west/left, north east/right, south west/above, south east/above,
 \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
   node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape circular sector

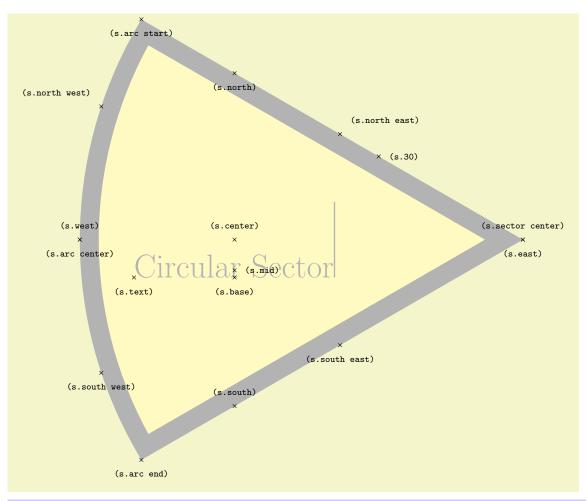
This shape is a circular sector (which can also be known as a wedge). This shape supports the rotation of the shape border, as described in Section 16.2.2. The angle of the border rotation determines the direction in which the 'apex' of the sector points (unless other transformations have been applied).



There is a PGF key to set the central angle of the sector, which is expected to be less than 180 degrees. To use this key in TikZ, simply remove the /pgf/ path.

```
/pgf/circular sector angle=\langle angle \rangle (no default, initially 60)
Sets the central angle of the sector.
```

The anchors for the circular sector shape are shown below. Anchor 30 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circular sector, style=shape example, inner sep=1cm]
   {Circular Sector\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
  {sector center/above, arc start/below, arc end/below, arc center/below,
                       base/below, mid/right, text/below,
   center/above,
   north/below,
                                        east/below,
                       south/above,
                                                       west/above,
   north west/above left, north east/above right,
   south west/below, south east/below, 30/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape cylinder

This shape is a 2-dimensional representation of a cylinder, which supports the rotation of the shape border as described in Section 16.2.2.

```
\lambda \begin{tikzpicture} \node[cylinder, draw, shape aspect=.5] {ABC}; \end{tikzpicture}
```

Regardless the rotation of the shape border, the height is always the distance between the curved ends, and the width is always the distance between the straight sides.

```
width
```

```
\begin{tikzpicture}[>=stealth]
\node [cylinder, gray!50, rotate=30, draw,
    minimum height=2cm, minimum width=1cm] (c) {Cylinder};
\draw[red, <->] (c.top) -- (c.bottom)
    node [at end, below, black] {height};
\draw[red, <->] (c.north) -- (c.south)
    node [at start, above, black] {width};
\end{tikzpicture}
```

Enlarging the shape to some minimum height will stretch only the body of the cylinder. By contrast, enlarging the shape to some minimum width will stretch the curved ends.

```
A B \[ \begin\{\tikzpicture\}\[ \shape \aspect=.5\] \\ \tikzset\{\tikzpicture\}\[ \style=\{cylinder\}, \shape \text{border rotate=90\}, \draw\}\ \\ \node \[ \liminimum \text{height=1.5cm}\] \quad \{A\}\; \\ \node \[ \liminimum \text{width=1.5cm}\] \at \((1.5,0)\) \{B\}\; \\ \end\{\tikzpicture\}\]
```

There are various keys to customize this shape (to use PGF keys in TikZ, simply remove the /pgf/ path).

```
/pgf/aspect=(value) (no default, initially 1.0)
```

The aspect is a recommendation for the quotient of the radii of the cylinder end. This may be ignored if the shape is enlarged to some minimum width.

```
/pgf/cylinder uses custom fill=\langle boolean \rangle
```

(default true)

This enables the use of a custom fill for the body and the end of the cylinder. The background path for the shape should not be filled (e.g., in TikZ, the fill option for the node must be implicity or explicitly set to none). Internally, this key sets the TEX-if \ightharpoonup internally, this key sets the TEX-if \ightharpoonup internally.

```
Cylinder \begin{tikzpicture}[aspect=0.5] \node [cylinder, cylinder uses custom fill, cylinder end fill=red!50, cylinder body fill=red!25] {Cylinder}; \end{tikzpicture}
```

```
/pgf/cylinder end fill=\langle color \rangle
```

(no default, initially white)

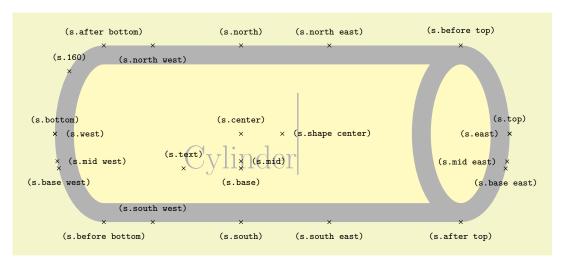
Sets the color for the end of the cylinder.

```
/pgf/cylinder body fill=\langle color \rangle
```

(no default, initially white)

Sets the color for the body of the cylinder.

The anchors of this shape are shown below (anchor 160 is an example of a border anchor). Note that the cylinder shape does not distinguish between outer xsep and outer ysep. Only the larger of the two values is used for the shape. Note also the difference between the center and shape center anchors: center is the center of the cylinder body and also the center of rotation. The shape center is the center of the shape which includes the 2-dimensional representation of the cylinder top.



```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=cylinder, shape example, aspect=.5, inner xsep=3cm,
       inner ysep=1cm] {Cylinder\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {before top/above, top/above,
                                           after top/below,
    before bottom/below, bottom/above,
                                           after bottom/above,
    mid/right,
                         mid west/right, mid east/left,
    base/below.
                         base west/below, base east/below,
                                           shape center/right,
    center/above.
                         text/above.
    west/right, east/left, north/above, south/below,
    north west/below, north east/above,
    south west/above, south east/below, 160/above}
 \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
   node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.4 Symbol Shapes

This library defines shapes that can be used for drawing symbols like a forbidden sign or a cloud.

Shape forbidden sign

This shape places the node inside a circle with a diagonal from the lower left to the upper right added. The circle is part of the background, the diagonal line part of the foreground path; thus, the diagonal line is on top of the text.



```
\begin{tikzpicture}
  \node [forbidden sign,line width=1ex,draw=red,fill=white] {Smoking};
  \end{tikzpicture}
```

The shape inherits all anchors from the circle shape.

Shape magnifying glass

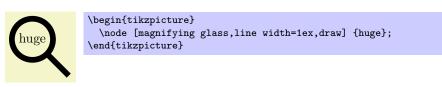
This shape places the node inside a circle with a handle attached to the node. The angle of the handle and its length can be adjusted using two keys:

```
/pgf/magnifying glass handle angle fill=\langle degree \rangle (default -45)

The angle of the handle.

/pgf/magnifying glass handle angle aspect=\langle factor \rangle (default 1.5)
```

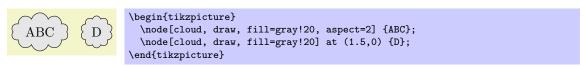
The length of the handle as a multiple of the circle radius.



The shape inherits all anchors from the circle shape.

Shape cloud

This shape is a cloud, drawn to tightly fit the node contents (strictly speaking, using an ellipse which tightly fits the node contents – including any inner sep).



A cloud should be thought of as having a number of "puffs", which are the individual arcs drawn around the border. There are PGF keys to specify how the cloud is drawn (to use these keys in TikZ, simply remove the /pgf/ path).

```
/pgf/cloud puffs=\langle integer \rangle
```

(no default, initially 10)

Sets the number of puffs for the cloud.

```
/pgf/cloud puff arc=\langle angle \rangle
```

(no default, initially 135)

Sets the length of the puff arc (in degrees). A shorter arc can produce better looking joins between puffs for larger line widths.

Like the diamond shape, the cloud shape also uses the aspect key to determine the ratio of the width and the height of the cloud. However, there may be circumstances where it may be undesirable to continually specify the aspect for the cloud. Therefore, the following key is implemented:

```
/pgf/cloud ignores aspect=\langle boolean \rangle
```

(default true)

Instruct PGF to ignore the aspect key. Internally, the TEX-if \ifpgfcloudignoresaspect is set appropriately. The initial value is false.

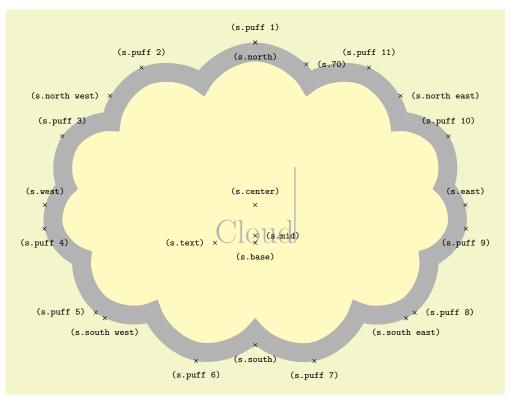
```
(rain) (snow)
```

Any minimum size requirements are applied to the "circum-ellipse", which is the ellipse which passes through all the midpoints of the puff arcs. These requirements are considered *after* any aspect specification is applied.



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw [blue, dashed] (1.5, 1) ellipse (1.5cm and 1cm);
  \node [cloud, cloud puffs=9, draw, minimum width=3cm, minimum height=2cm]
  at (1.5, 1) {};
  \end{tikzpicture}
```

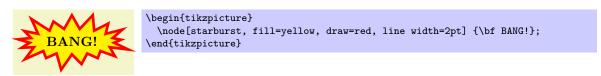
The anchors for the cloud shape are shown below for a cloud with eleven puffs. Anchor 70 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=cloud, style=shape example, cloud puffs=11, aspect=1.5,
       cloud puff arc=120,inner ysep=1cm] {Cloud\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
   {puff 1/above, puff 2/above, puff 3/above, puff 4/below,
   puff 5/left, puff 6/below, puff 7/below, puff 8/right,
    puff 9/below, puff 10/above, puff 11/above, 70/right,
    center/above, base/below,
                                 mid/right,
                                                text/left,
    north/below, south/below,
                                east/above,
                                                west/above,
                                 north east/right,
    north west/left,
    south west/below,
                                 south east/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape starburst

This shape is a randomly generated elliptical star, which supports the rotation of the shape border as described in Section 16.2.2.



Like the star shape, the starburst should be thought of as having a set of inner points and outer points. The inner points lie on the ellipse which tightly fits the node contents (including any inner sep).

Using a specified 'starburst point height' value, the outer points are generated randomly between this value and one quarter of this value. For a given starburst shape, the angle between each point is fixed, and is determined by the number of points specified for the starburst.

It is important to note that, whilst the maximum possible point height is used to calculate minimum width or height requirements, the outer points are randomly generated, so there is (unfortunately) no guarantee that any such requirements will be fully met.



```
\begin{tikzpicture}
  \draw[help lines] grid(3,2);
  \node[starburst, draw, minimum width=3cm, minimum height=2cm]
  at (1.5, 1) {\bf BOOM!};
\end{tikzpicture}
```

There are PGF keys to control the drawing of the starburst shape. To use these keys in TikZ, simply remove the /pgf/ path.

/pgf/starburst points=\langle integer\rangle

(no default, initially 17)

Sets the number of outer points for the starburst.

```
/pgf/starburst point height=\langle length \rangle
```

(no default, initially .5cm)

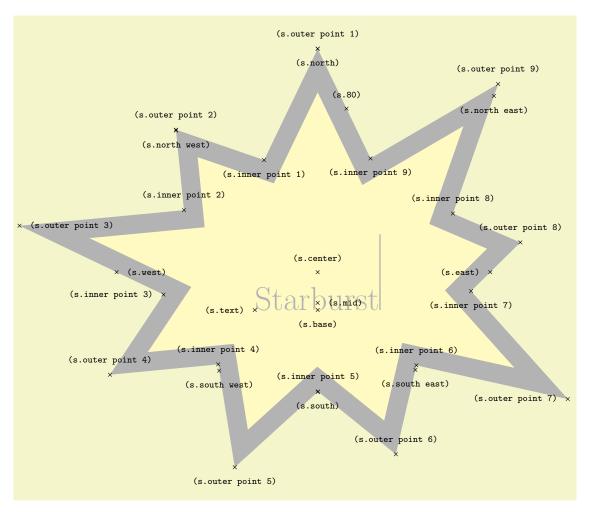
Sets the maximum distance between the inner point radius and the outer point radius.

$/pgf/random starburst=\langle integer \rangle$

(no default, initially 100)

Sets the seed for the random number generator for creating the starburst. The maximum value for $\langle integer \rangle$ is 16383. If $\langle integer \rangle = 0$, the random number generator will not be used, and the maximum point height will be used for all outer points. If $\langle integer \rangle$ is omitted, a seed will be randomly chosen.

The basic anchors for a nine point starburst shape are shown below. Anchor 80 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=starburst, starburst points=9, starburst point height=3.5cm,
       style=shape example,inner sep=1cm]
   {Starburst\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {outer point 1/above, outer point 2/above, outer point 3/right,
    outer point 4/above, outer point 5/below, outer point 6/above,
    outer point 7/left, outer point 8/above, outer point 9/above,
    inner point 1/below, inner point 2/above, inner point 3/left,
    inner point 4/above, inner point 5/above, inner point 6/above,
    inner point 7/below, inner point 8/above, inner point 9/below,
    center/above, text/left, mid/right, base/below, 80/above,
    north/below, south/below, east/left, west/right,
    north east/below, south west/below, south east/below, north west/below}
 \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
   node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape signal

This shape is a "signal" or sign shape, that is, a rectangle, with optionally pointed sides. A signal can point "to" somewhere, with outward points in that direction. It can also be "from" somewhere, with inward points from that direction. The resulting points extend the node contents (which include the inner sep).

```
To East

| Levery node/.style={signal, draw, text=white, signal to=nowhere}]
| Levery node/.style={signal, draw, text=white, signal, draw, text=white, signal, draw, text=white, signal, draw, text=white, signal,
```

There are PGF keys for drawing the signal shape (to use these keys in TikZ, simply remove the /pgf/ path):

```
/pgf/signal pointer angle=\langle angle \rangle
```

(no default, initially 90)

Sets the angle for the pointed sides of the shape. This angle is maintained when enforcing any minimum size requirements, so any adjustment to the width will affect the height, and vice versa.

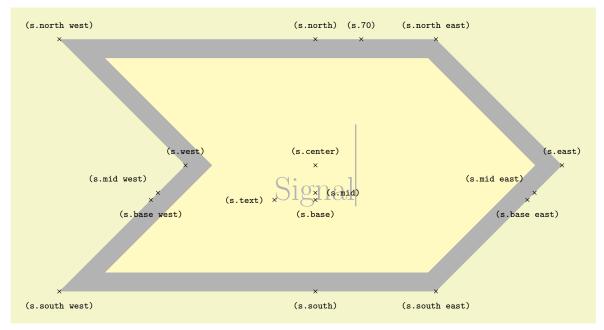
```
/pgf/signal from=\langle direction \rangle and \langle opposite direction \rangle (no default, initially nowhere)
```

Sets which sides take an inward pointer (i.e., that points towards the center of the shape). The possible values for $\langle direction \rangle$ and $\langle opposite\ direction \rangle$ are the compass point directions north, south, east and west (or above, below, right and left). An additional keyword nowhere can be used to reset the sides so they have no pointers. When used with signal from key, this only resets inward pointers; used with the signal to key, it only resets outward pointers.

```
/pgf/signal to=\langle direction \rangle and \langle opposite\ direction \rangle (no default, initially east)
Sets which sides take an outward pointer (i.e., that points away from the shape).
```

Note that PGF will ignore any instruction to use directions that are not opposites (so using the value east and north, will result in only north being assigned a pointer). This is also the case if non-opposite values are used in the signal to and signal from keys at the same time. So, for example, it is not possible for a signal to have an outward point to the left, and also have an inward point from below.

The anchors for the signal shape are shown below. Anchor 70 is an example of a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=signal, signal from=west, shape example, inner sep=2cm]
   {Signal\vrule width1pt height2cm};
  \foreach \anchor/\placement in
   {text/left, center/above,
                                   70/above,
    base/below, base east/below, base west/below,
    mid/right, mid east/above left, mid west/above left,
    north/above,
                      south/below.
    east/above,
                      west/above,
    north west/above, north east/above,
    south west/below, south east/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape tape

This shape is a rectangle with optional, "bendy" top and bottom sides, which tightly fits the node contents (including the inner sep).

```
\lambda \text{begin{tikzpicture}} \\ \node[tape, draw]{ABCD}; \\ \node[tape, draw, tape bend top=none] at (1.5, 0) {EFGH}; \\ \end{tikzpicture}
```

There are PGF keys to specify which sides bend and how high the bends are (to use these keys in TikZ, simply remove the /pgf/ path):

```
/pgf/tape bend top=\langle bend style\rangle (no default, initially in and out)
```

Specifies how the top side bends. The $\langle bend\ style \rangle$ is either in and out, out and in or none (i.e., a straight line). The bending sides are drawn in a clockwise direction, and using the bend style in and out will mean the side will first bend inwards and then bend outwards. The opposite holds true for out and in.

```
in and out

Tape

\text{begin{tikzpicture}[-stealth]}
\node[tape, draw, gray, minimum width=2cm](t){Tape};
\draw [blue]([yshift=5pt] t.north west) -- ([yshift=5pt]t.north east)
\text{node[midway, above, black]{in and out};}
\draw [blue]([yshift=-5pt]t.south east) -- ([yshift=-5pt]t.south west)
\text{node[sloped, allow upside down, midway, above, black]{in and out};}
\end{tikzpicture}
```

This might take a bit of getting used to, but just remember that when you want the bendy sides to be parallel, the sides take the same bend style. It is possible for the top and bottom sides to

take opposite bend styles, but the author of this shape cannot think of a single use for such a combination.



```
\begin{tikzpicture}
  \tikzstyle{every node}=[tape, draw]
  \node [tape bend top=out and in, tape bend bottom=out and in] {Parallel};
  \node at (2,0) [tape bend bottom=out and in] {Why?};
  \end{tikzpicture}
```

/pgf/tape bend bottom= $\langle bend \ style \rangle$

(no default, initially in and out)

Specifies how the bottom side bends.

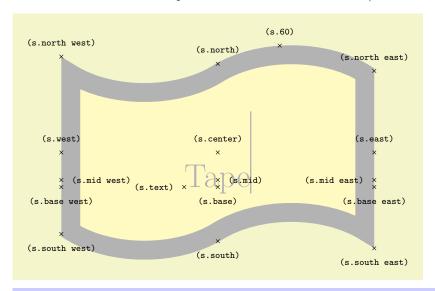
```
/pgf/tape bend height=\langle length \rangle
```

(no default, initially 5pt)

Sets the total height for a side with a bend.

```
tape bend height
```

The anchors for the tape shape are shown below. Anchor 60 is an example of a border anchor. Note that border anchors will snap to the center of convex curves (i.e. when bending in).



```
\Huge
\begin{tikzpicture}
\node[name=s, shape=tape, tape bend height=1cm, shape example, inner xsep=3cm]
    {Tape\vrule width1pt height2cm};
    \foreach \anchor/\placement in
     {text/left, center/above, 60/above,
        base/below, base east/below, base west/below,
        mid/right, mid east/left, mid west/right,
        north/above, south/below, east/above, west/above,
        north west/above, north east/above,
        south west/below, south east/below}
        \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
        node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.5 Arrow Shapes

```
\usepgflibrary{shapes.arrows} % MEX and plain TeX and pure pgf
\usepgflibrary[shapes.arrows] % ConTeXt and pure pgf
\usetikzlibrary{shapes.arrows} % MEX and plain TeX when using TikZ
```

\usetikzlibrary[shapes.arrows] % ConTeXt when using TikZ

This library defines arrow shapes. Note that an arrow shape is something quite different from a (normal) arrow tip: It is a shape that just "happens" to "look like" an arrow. In particular, you cannot use these shapes as arrow tips.

Shape single arrow

This shape is an arrow, which tightly fits the node contents (including any inner sep). This shape supports the rotation of the shape border, as described in Section 16.2.2. The angle of rotation determines in which direction the arrow points (provided no other rotational transformations are applied).

```
\begin{tikzpicture}[every node/.style={single arrow, draw}, rotate border/.style={shape border uses incircle, shape border rotate=#1}] \node {right}; \node at (2,0) [shape border rotate=90]{up}; \node at (1,1) [rotate border=37, inner sep=0pt]{$37^\circ$}; \end{tikzpicture}
```

Regardless of the rotation of the arrow border, the width is measured between the back ends of the arrow head, and the height is measured from the arrow tip to the end of the arrow tail.

```
height
height
width
width

\[
\text{begin{tikzpicture}[>=stealth,} \\
\text{rotate border/.style={shape border uses incircle, shape border rotate=#1}]} \\
\text{node[rotate border=-30, fill=gray!25, minimum height=3cm, single arrow,} \\
\text{single arrow head extend=.5cm, single arrow head indent=.25cm] (arrow) {};} \\
\text{draw[red, <->] (arrow.before tip) -- (arrow.after tip)} \\
\text{node [near end, left, black] {width};} \\
\text{draw[red, <->] (arrow.tip) -- (arrow.tail)} \\
\text{node [near end, below left, black] {height};} \\
\text{end{tikzpicture}}
\end{tikzpicture}
\]
```

There are PGF keys that can be used to customize this shape (to use these keys in TikZ, simply remove the /pgf/ path).

```
/pgf/single arrow tip angle=\langle angle \rangle
```

(no default, initially 90)

Sets the angle for the arrow tip. Enlarging the arrow to some minimum width may increase the height of the shape to maintain this angle.

```
/pgf/single arrow head extend=\langle length \rangle
```

(no default, initially .5cm)

This sets the distance between the tail of the arrow and the outer end of the arrow head. This may change if the shape is enlarged to some minimum width.

```
\begin{tikzpicture}
\node[single arrow, draw, single arrow head extend=.5cm, gray!50, rotate=60]
(a) {Arrow};
\draw[red, |<->|] (a.before tip) -- (a.before head)
node [midway, below, sloped, black] {head extend};
\end{tikzpicture}
```

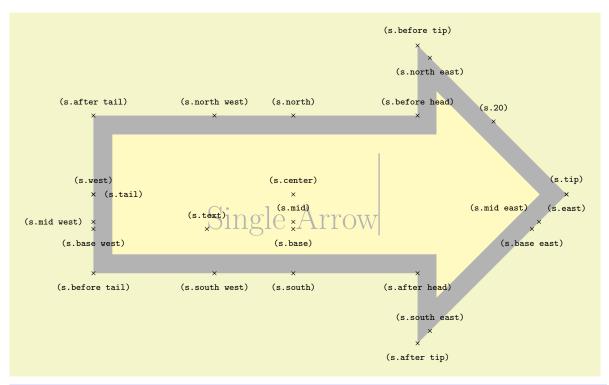
/pgf/single arrow head indent= $\langle length \rangle$

(no default, initially 0cm)

This moves the point where the arrow head joins the shaft of the arrow towards the arrow tip, by $\langle length \rangle$.



The anchors for this shape are shown below (anchor 20 is an example of a border anchor).



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=single arrow, shape example, single arrow head extend=1.5cm]
   {Single Arrow\vrule width1pt height2cm};
  \foreach \anchor/\placement in
                     center/above, 20/above,
   {text/above.
    mid west/left.
                     mid/above,
                                   mid east/above left,
     base west/below, base/below,
                                    base east/below,
     tip/above, before tip/above, after tip/below, before head/above,
     after head/below, after tail/above, before tail/below, tail/right,
     north/above, south/below, east/below, west/above,
     north west/above, north east/below, south west/below, south east/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape double arrow

This shape is a double arrow, which tightly fits the node contents (including any inner sep), and supports the rotation of the shape border, as described in Section 16.2.2.

```
\Left or Right \begin{tikzpicture}[every node/.style={double arrow, draw}] \node [double arrow, draw] {Left or Right}; \end{tikzpicture}
```

The double arrow behaves exactly like the single arrow, so you need to remember that the width is always the distance between the back ends of the arrow heads, and the height is always the tip-to-tip distance.

```
height

height

rotate border/.style={shape border uses incircle, shape border rotate=#1}]

node[rotate border=210, fill=gray!25, minimum height=3cm, double arrow,
double arrow head extend=.5cm, double arrow head indent=.25cm] (arrow) {};

draw[red, <->] (arrow.before tip 1) -- (arrow.after tip 1)

node [near start, right, black] {width};

draw[red, <->] (arrow.tip 1) -- (arrow.tip 2)

node [near end, above left, black] {height};

end{tikzpicture}
```

The PGF keys that can be used to customize the double arrow behave similarly to the keys for the single arrow (to use these keys in TikZ, simply remove the /pgf/ path).

/pgf/double arrow tip angle= $\langle angle \rangle$

(no default, initially 90)

Sets the angle for the arrow tip. Enlarging the arrow to some minimum width may increase the height of the shape to maintain this angle.

/pgf/double arrow head extend= $\langle length \rangle$

(no default, initially .5cm)

This sets the distance between the shaft of the arrow and the outer end of the arrow heads. This may change if the shape is enlarged to some minimum width.

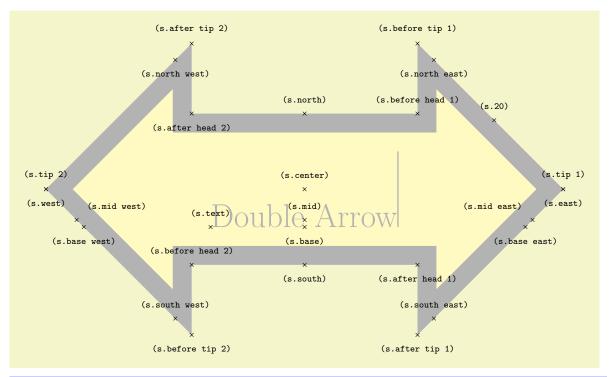
/pgf/double arrow head indent= $\langle length \rangle$

(no default, initially 0cm)

This moves the point where the arrow heads join the shaft of the arrow towards the arrow tips, by $\langle length \rangle$.



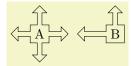
The anchors for this shape are shown below (anchor 20 is an example of a border anchor).



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=double arrow, double arrow head extend=1.5cm, shape example, inner xsep=2cm]
  {Double Arrow\vrule width1pt height2cm};
  \foreach \anchor/\placement in
  {text/above, center/above, 20/above,
    mid west/above right, mid/above, mid east/above left,
    base west/below, base/below, base east/below,
    before head 1/above, before tip 1/above, tip 1/above, after tip 1/below, after head 1/below,
    before head 2/above, before tip 2/below, tip 2/above, after tip 2/above, after head 2/below,
    north/above, south/below, east/below, west/below,
    north west/below, north east/below, south west/above, south east/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

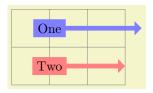
Shape arrow box

This shape is a rectangle with optional arrows which extend from the four sides.



```
\begin{tikzpicture}
  \node[arrow box, draw] {A};
  \node[arrow box, draw, arrow box arrows={north:.5cm, west:0.75cm}]
  at (2,0) {B};
\end{tikzpicture}
```

Any minimum size requirements are applied to the main rectangle *only*. This does not pose too many problems if you wish to accommodate the length of the arrows, as it is possible to specify the length of each arrow independently, from either the border of the rectangle (the default) or the center of the rectangle.



```
\begin{tikzpicture}
\tikzset{box/.style={arrow box, fill=#1}}
\draw [help lines] grid(3,2);
  \node[box=blue!50, arrow box arrows={east:2cm}] at (1,1.5){One};
  \node[box=red!50, arrow box arrows={east:2cm from center}] at (1,0.5){Two};
\end{tikzpicture}
```

There are various PGF keys for drawing this shape (to use these keys in TikZ, simply remove the pgf) path).

```
/pgf/arrow box tip angle=\langle angle \rangle
```

(no default, initially 90)

Sets the angle at the arrow tip for all four arrows.

```
/pgf/arrow box head extend=\langle length \rangle
```

(no default, initially .125cm)

Sets the distance the arrow head extends away from the shaft of the arrow. This applies to all arrows.

```
/pgf/arrow box head indent=\langle length \rangle
```

(no default, initially 0cm)

Moves the point where the arrow head joins the shaft of the arrow towards the arrow tip. This applies to all arrows.

```
/pgf/arrow box shaft width=\langle length \rangle
```

(no default, initially .125cm)

Sets the width of the shaft of all arrows.

```
/pgf/arrow box north arrow=\langle distance \rangle
```

(no default, initially .5cm)

Sets the distance the north arrow extends from the node. By default this is from the border of the shape, but by using the additional keyword from center, the distance will be measured from the center of the shape. If $\langle distance \rangle$ is Opt or a negative distance, the arrow will not be drawn.

```
/pgf/arrow box south arrow=\langle distance \rangle
```

(no default, initially .5cm)

Sets the distance the south arrow extends from the node.

```
/pgf/arrow box east arrow=\langle distance \rangle
```

(no default, initially .5cm)

Sets the distance the east arrow extends from the node.

```
/pgf/arrow box west arrow=\langle distance \rangle
```

(no default, initially .5cm)

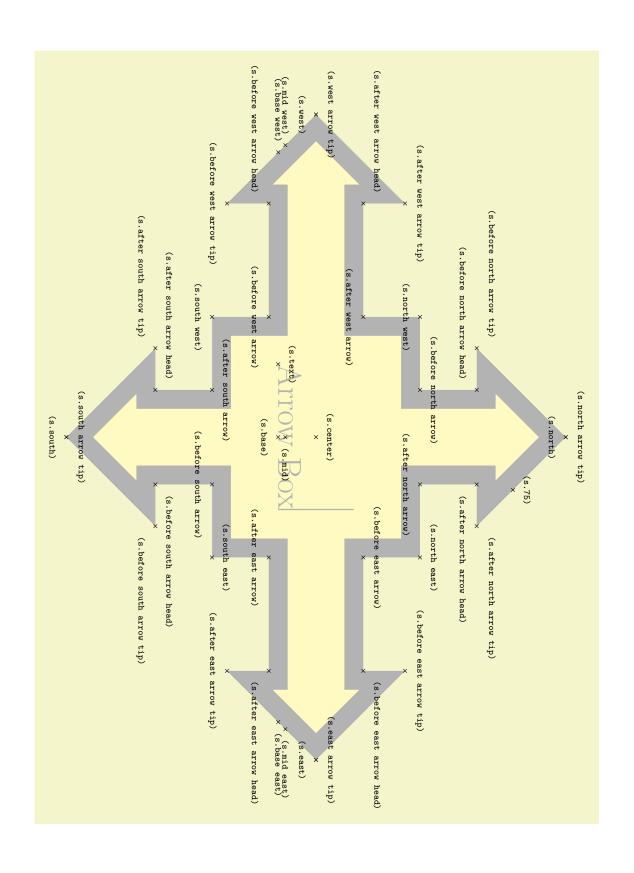
Sets the distance the west arrow extends from the node.

```
/pgf/arrow box arrows=\{\langle list \rangle\}
```

(no default)

Sets the distance that all arrows extend from the node. The specification in $\langle list \rangle$ consists of the four compass points north, south, east or west, separated by commas (so the list must be contained within braces). The distances can be specified after each side separated by a colon (e.g., north:1cm, or west:5cm from center). If an item specifies no distance, the most recently specified distance will be used (at the start of the list this is 0cm, so the first item in the list should specify a distance). Any sides not specified will not be drawn with an arrow.

The anchors for this shape are shown below (unfortunately, due to its size, this example must be rotated). Anchor 75 is an example of a border anchor. If a side is drawn without an arrow, the anchors for that arrow should be considered unavailable. They are (unavoidably) defined, but default to the center of the appropriate side.



```
\Huge
\begin{tikzpicture}
  \node[shape=arrow box, shape example, inner xsep=1cm, inner ysep=1.5cm, arrow box shaft width=2cm,
   arrow box arrows={north:3.5cm from border, south, east:5cm from border, west},
   arrow box head extend=0.75cm, rotate=-90](s) {Arrow Box\vrule width1pt height2cm};
  \foreach \anchor/\placement in
   {center/above, text/above, mid/right, base/below, 75/above,
   mid east/right, mid west/left, base east/right, base west/left,
   north/below, south/below, east/below, west/below,
   north east/above, south east/above, south west/below, north west/below,
   north arrow tip/above, south arrow tip/above, east arrow tip/above, west arrow tip/above,
   before north arrow/above, before north arrow head/below left, before north arrow tip/above left,
   after north arrow tip/above right, after north arrow head/below right, after north arrow/below,
   before south arrow/below, before south arrow head/above right, before south arrow tip/below right,
   after south arrow tip/below left, after south arrow head/above left, after south arrow/above,
   before east arrow/above, before east arrow head/above right, before east arrow tip/above,
   after east arrow tip/below, after east arrow head/below right, after east arrow/below,
   before west arrow/below, before west arrow head/below left, before west arrow tip/below,
   after west arrow tip/above, after west arrow head/above left, after west arrow/below}
      \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement, rotate=-90] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.6 Shapes with Multiple Text Parts

```
\usepgflibrary{shapes.multipart} % MEX and plain TeX and pure pgf \usepgflibrary[shapes.multipart] % ConTeXt and pure pgf \usetikzlibrary{shapes.multipart} % MEX and plain TeX when using TikZ \usetikzlibrary[shapes.multipart] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that are composed of multiple (text) parts.

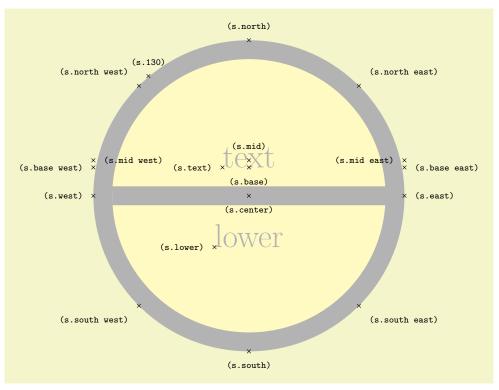
Shape circle split

This shape is a multi-part shape consisting of a circle with a line in the middle. The upper part is the main part (the text part), the lower part is the lower part.

```
\lambda begin{tikzpicture}
\node [circle split,draw,double,fill=red!20]

{
\text{ \q_1\text{ \nodepart{lower} \\ \$00\text{ \\ \}; \end{tikzpicture}}
```

The shape inherits all anchors from the circle shape and defines the lower anchor in addition. See also the following figure:



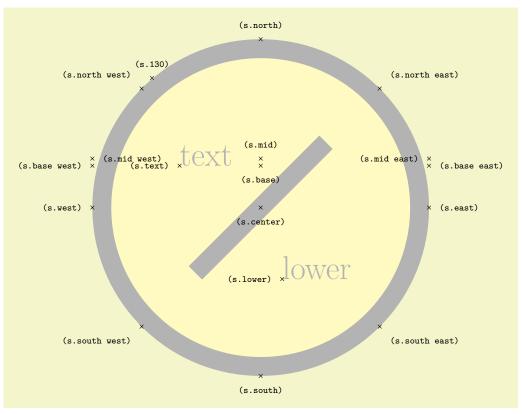
```
\Huge
\begin{tikzpicture}
\node[name=s,shape=circle split,shape example] {text\nodepart{lower}lower};
\foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
    west/left, center/below, east/right,
    mid west/right, mid/above, mid east/left,
    base west/left, base/below, base east/right,
    south west/below left, south/below, south east/below right,
    text/left, lower/left, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape circle solidus

This shape (due to Manuel Lacruz) is similar to the split circle, but the two text parts are arranged diagonally.



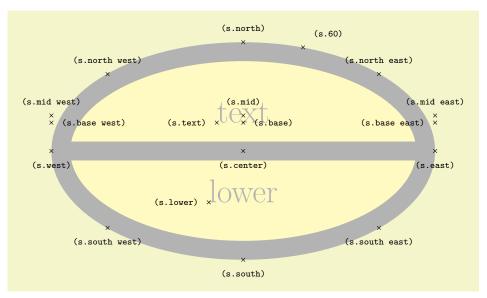
```
\begin{tikzpicture}
  \node [circle solidus,draw,double,fill=red!20]
  {
      $q_1$
      \nodepart{lower}
      $00$
   };
\end{tikzpicture}
```



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle solidus,shape example,inner xsep=1cm] {text\nodepart{lower}lower};
  \foreach \anchor/\placement in
      {north west/above left, north/above, north east/above right,
      west/left, center/below, east/right,
      mid west/right, mid/above, mid east/left,
      base west/left, base/below, base east/right,
      south west/below left, south/below, south east/below right,
      text/left, lower/left, 130/above}
      \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
            node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape ellipse split

This shape is a multi-part shape consisting of an ellipse with a line in the middle. The upper part is the main part (the text part), the lower part is the lower part. The anchors for this shape are shown below. Anchor 60 is a border anchor.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=ellipse split,shape example] {text\nodepart{lower}lower};
  \foreach \anchor/\placement in
    {center/below, text/left, lower/left, 60/above right,
    mid/above, mid east/above, mid west/above,
    base/right, base east/left, base west/right,
    north/above, south/below, east/below, west/below,
    north east/above, south east/below, south west/below, north west/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape rectangle split

This shape is a rectangle which can be split either horizontally or vertically into several parts.

```
\begin{tikzpicture}[my shape/.style={
        1 2 3 4
                   5
a
                            rectangle split, rectangle split parts=#1, draw, anchor=center}]
b
                            \node [mv shape=5] at (0.1)
                              {a\nodepart{two}b\nodepart{three}c\nodepart{four}d\nodepart{five}e};
\mathbf{c}
                     A
                             \node [my shape=5, rectangle split horizontal] at (2,2)
d
     1 | 2 | 3 |
                    В
              4
                              {1\nodepart{two}2\nodepart{three}3\nodepart{four}4\nodepart{five}5};
е
                             \node [my shape=3] at (3,0.5)
                    \mathbf{C}
                              {A\nodepart{two}B\nodepart{three}C};
                             \node [my shape=4, rectangle split horizontal] at (1.5,0.5)
                              {1\nodepart{two}2\nodepart{three}3\nodepart{four}4};
                          \end{tikzpicture}
```

The shape can be split into a maximum of twenty parts. However, to avoid allocating a lot of unnecessary boxes, PGF only allocates four boxes by default. To use the rectangle split shape with more than four boxes, the extra boxes must be allocated manually in advance (perhaps using \newbox or \let). The boxes take the form \pgfnodepart\(\lambda\) number\(\rangle\) box, where \(\lambda\) number\(\rangle\) is from the cardinal numbers one, two, three, ... and so on. \pgfnodepartonebox is special in that it is synonymous with \pgfnodeparttextbox. For compatibility with earlier versions of this shape, the boxes \pgfnodeparttwobox, \pgfnodepartthreebox and \pgfnodepartfourbox, can be referred to using the ordinal numbers: \pgfnodepartsecondbox, \pgfnodepartthirdbox and \pgfnodepartfourthbox. In order to facilitate the allocation of these extra boxes, the following key is provided:

```
/pgf/rectangle split allocate boxes=\langle number \rangle (no default)
```

This key checks if $\langle number \rangle$ boxes have been allocated, and if not, it allocates the required boxes using \newbox (some "magic" is performed to get around the fact that \newbox is declared \outer in plain TeX).

When split vertically, the rectangle split will meet any minimum width requirements, but any minimum height will be ignored. Conversely when split horizontally, minimum height requirements

will be met, but any minimum width will be ignored. In addition, inner sep is applied to every part that is used, so it cannot be specified independently for a particular part.

There are several PGF keys to specify how the shape is drawn. To use these keys in TikZ, simply remove the /pgf/ path:

```
/pgf/rectangle split parts=\langle number \rangle
```

(no default, initially 4)

Split the rectangle into $\langle number \rangle$ parts, which should be in the range 1 to 20. If more than four parts are needed, the boxes should be allocated in advance as described above.

```
Student
age:int
name:String
getAge():int
getName():String
```

```
\begin{tikzpicture} [every text node part/.style={align=center}]
  \node[rectangle split, rectangle split parts=3, draw, text width=2.75cm]
  {Student
    \nodepart{two}
    age:int \\
    name:String
    \nodepart{three}
    getAge():int \\
    getName():String};
\end{tikzpicture}
```

/pgf/rectangle split horizontal= $\langle boolean \rangle$

(default true)

This key determines whether the rectangle is split horizontally or vertically

```
/pgf/rectangle split ignore empty parts=\langle boolean \rangle
```

(default true)

When $\langle boolean \rangle$ is true, PGF will ignore any part that is empty except the text part. This effectively overrides the rectangle split parts key in that, if 3 parts (for example) are specified, but one is empty, only two will be shown.

```
text third third
```

```
\begin{tikzpicture}[every node/.style={draw, anchor=text, rectangle split,
    rectangle split parts=3}]
  \node {text \nodepart{second} \nodepart{third}third};
  \node [rectangle split ignore empty parts] at (2,0)
      {text \nodepart{second} \nodepart{third}third};
  \end{tikzpicture}
```

```
/pgf/rectangle split empty part width=\langle length \rangle
```

(no default, initially 1ex)

Sets the default width for a node part box if it is empty and empty parts are not ignored.

```
/pgf/rectangle split empty part height=\langle length \rangle
```

(no default, initially 1ex)

Sets the default height for a node part box if it is empty and empty parts are not ignored.

```
/pgf/rectangle split empty part depth=\langle length \rangle
```

(no default, initially Oex)

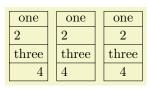
Sets the default depth for a node part box if it is empty and empty parts are not ignored.

```
/pgf/rectangle split part align=\{\langle list \rangle\}
```

(no default, initially center)

Sets the alignment of the boxes inside the node parts. Each item in $\langle list \rangle$ should be separated by commas (so if there is more than one item in $\langle list \rangle$, it must be surrounded by braces).

When the rectangle is split vertically, the entries in $\langle list \rangle$ must be one of left, right, or center. If $\langle list \rangle$ has less entries than node parts then the remaining boxes are aligned according to the last entry in the list. Note that this only aligns the boxes in each part and *does not* affect the alignment of the contents of the boxes.



When the rectangle is split horizontally, the entries in $\langle list \rangle$ must be one of top, bottom, center or base. Note that using the value base will only make sense if all the node part boxes are being aligned in this way. This is because the base value aligns the boxes in relation to each other, whereas the other values align the boxes in relation to the part of the shape they occupy.

```
/pgf/rectangle split draw splits=\langle boolean \rangle
```

(default true)

Sets whether the line or lines between node parts will be drawn. Internally, this sets the TeX-if \ifpgfrectanglesplitdrawsplits appropriately.

```
/pgf/rectangle split use custom fill=\langle boolean \rangle
```

(default true)

This enables the use of a custom fill for each of the node parts (including the area covered by the inner sep). The background path for the shape should not be filled (e.g., in TikZ, the fill option for the node must be implicitly or explicitly set to none). Internally, this key sets the TEX-if \ifpgfrectanglesplitusecustomfill appropriately.

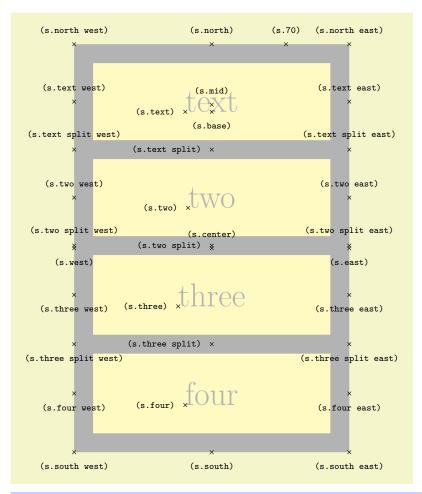
```
/pgf/rectangle split part fill=\{\langle list \rangle\}
```

(no default, initially white)

Sets the custom fill color for each node part shape. The items in $\langle list \rangle$ should be separated by commas (so if there is more than one item in $\langle list \rangle$, it must be surrounded by braces). If $\langle list \rangle$ has less entries than node parts, then the remaining node parts use the color from the last entry in the list. This key will automatically set /pgf/rectangle split use custom fill.

```
\begin{tikzpicture}
\tikzset{every node/.style={rectangle split, draw, minimum width=.5cm}}
\node[rectangle split part fill={red!50, green!50, blue!50, yellow!50}] {};
\node[rectangle split part fill={red!50, green!50, blue!50}] at (0.75,0) {};
\node[rectangle split part fill={red!50, green!50}] at (1.5,0) {};
\node[rectangle split part fill={red!50}] at (2.25,0) {};
\end{tikzpicture}
```

The anchors for the rectangle split shape split vertically into four, are shown below (anchor 70 is an example of a border angle). When a node part is missing, the anchors prefixed with the name of that node part should be considered unavailable. They are (unavoidably) defined, but default to other anchor positions.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rectangle split, rectangle split parts=4, shape example,
   inner ysep=0.75cm]
    {\nodepart{text}text\nodepart{two}two}
\nodepart{three}three\nodepart{four}four};
  \foreach \anchor/\placement in
   {text/left, text east/above, text west/above,
     two/left, two east/above, two west/above,
     three/left, three east/below, three west/below,
     four/left, four east/below, four west/below,
     text split/left, text split east/above, text split west/above,
     two split/left, two split east/above, two split west/above,
     three split/left, three split east/below, three split west/below,
     north/above, south/below, east/below, west/below,
     north west/above, north east/above, south west/below, south east/below,
     center/above, 70/above, mid/above, base/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.7 Callout Shapes

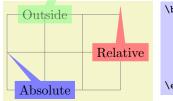
```
\usepgflibrary{shapes.callouts} % \mbox{MEX} and plain \mbox{TEX} and pure pgf \usepgflibrary[shapes.callouts] % \mbox{ConTEX} and plain \mbox{TEX} when using \mbox{Ti}k\mbox{Z} \usetikzlibrary[shapes.callouts] % \mbox{ConTEX} when using \mbox{Ti}k\mbox{Z}
```

Producing basic callouts can be done quite easily in PGF and TikZ by creating a node and then subsequently drawing a path from the border of the node to the required point. This library provides more fancy, 'balloon'-style callouts.

Callouts consist of a main shape and a pointer (which is part of the shape) which points to something in (or outside) the picture. The position on the border of the main shape to which the pointer is connected is determined automatically. However, the pointer is ignored when calculating the minimum size of the shape, and also when positioning anchors.

```
\text{hallo!} \text{begin{tikzpicture}[remember picture]} \text{node[ellipse callout, draw] (hallo) {Hallo!}; \text{end{tikzpicture}}
```

There are two kinds of pointer: the "relative" pointer and the "absolute" pointer. The relative pointer calculates the angle of a specified coordinate relative to the center of the main shape, locates the point on the border to which this angle corresponds, and then adds the coordinate to this point. This seemingly over-complex approach means than you do not have to guess the size of the main shape: the relative pointer will always be outside. The absolute pointer, on the other hand, is much simpler: it points to the specified coordinate absolutely (and can even point to named coordinates in different pictures).



The following keys are common to all callouts. Please remember that the callout relative pointer, and callout absolute pointer keys take a different format for their value depending on whether they are being used in PGF or TikZ.

```
/pgf/callout relative pointer=\langle coordinate\rangle (no default, initially \pgfpointpolar{315}{.5cm})
Sets the vector of the callout pointer 'relative' to the callout shape.
```

```
/pgf/callout absolute pointer=\langle coordinate \rangle (no default)
```

Sets the vector of the callout pointer absolutely within the picture.

```
/tikz/callout relative pointer=(coordinate) (no default, initially (315:.5cm))
```

The TikZ version of the callout relative pointer key. Here, $\langle coordinate \rangle$ can be specified using the TikZ format for coordinates.

```
/tikz/callout absolute pointer=(coordinate) (no default)
```

The TikZ version of the callout absolute pointer key. Here, $\langle coordinate \rangle$ can be specified using the TikZ format for coordinates.

It is also possible to shorten the pointer by some distance, using the following key:

```
/\texttt{pgf/callout pointer shorten=} \langle \textit{distance} \rangle \qquad \qquad \text{(no default, initially 0cm)}
```

Moves the callout pointer towards the center of the callout's main shape by $\langle distance \rangle$.

```
\tikzset{callout/.style={ellipse callout, callout pointer arc=30, callout absolute pointer={#1}}}
\draw (0,0) grid (3,2);
\node[callout={(3,1.5)}, fill=red!50] at (0,1.5) {A};
\node[callout={(3,.5)}, fill=green!50, callout pointer shorten=1cm]
at (0,.5) {B};
\end{tikzpicture}
```

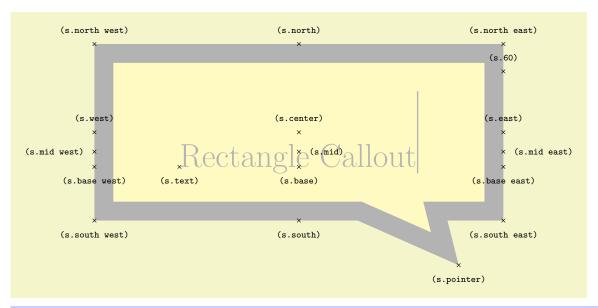
Shape rectangle callout

This shape is a callout whose main shape is a rectangle, which tightly fits the node contents (including any inner sep). It supports the keys described above and also the following key:

```
/pgf/callout pointer width=\langle length \rangle (no default, initially .25cm)
```

Sets the width of the pointer at the border of the rectangle.

The anchors for this shape are shown below (anchor 60 is an example of a border anchor). The pointer direction is ignored when placing anchors. Additionally, when using an absolute pointer, the pointer anchor should not to be used to used to position the shape as it is calculated whilst the shape is being drawn.



```
\begin{tikzpicture}
 \node[name=s,shape=rectangle callout, callout relative pointer={(1.25cm,-1cm)},
    callout pointer width=2cm, shape example, inner xsep=2cm, inner ysep=1cm]
  {Rectangle Callout\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {center/above, text/below,
                                    60/above,
    mid/right.
                  mid west/left. mid east/right.
    base/below,
                  base west/below, base east/below,
    north/above, south/below, east/above, west/above,
    north west/above, north east/above,
    south west/below, south east/below,
    pointer/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

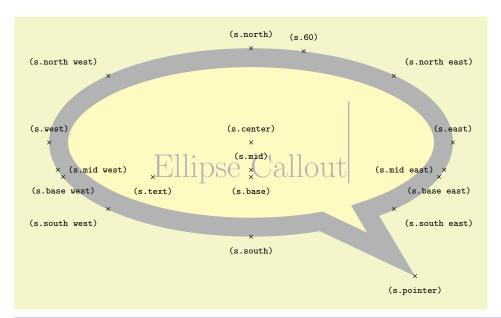
Shape ellipse callout

This shape is a callout whose main shape is an ellipse, which tightly fits the node contents (including any inner sep). It uses the absolute callout pointer, relative callout pointer and callout pointer shorten keys, and also the following key:

```
\label{eq:pgf/callout} \mbox{pointer arc=} \mbox{$\langle$angle$\rangle$} \qquad \qquad \mbox{(no default, initially 15)}
```

Sets the width of the pointer at the border of the ellipse according to an arc of length $\langle angle \rangle$.

The anchors for this shape are shown below (anchor 60 is an example of a border anchor). The pointer direction is ignored when placing anchors and the pointer anchor can only be used to position the shape when the relative anchor is specified.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=ellipse callout, callout relative pointer={(1.25cm,-1cm)},
   callout pointer width=2cm, shape example, inner xsep=1cm, inner ysep=.5cm]
   {Ellipse Callout\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {center/above, text/below,
                                   60/above.
    mid/above.
                  mid west/right, mid east/left,
                 base west/below, base east/below,
    base/below,
    north/above, south/below, east/above, west/above,
    north west/above left,
                            north east/above right,
    south west/below left,
                              south east/below right,
    pointer/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape cloud callout

This shape is a callout whose main shape is a cloud which fits the node contents. The pointer is segmented, consisting of a series of shrinking ellipses. This callout requires the symbol shape library (for the cloud shape). If this library is not loaded an error will result.

```
\begin{tikzpicture}
\node[cloud callout, cloud puffs=15, aspect=2.5, cloud puff arc=120,
shading=ball,text=white] {\bf Imagine...};
\end{tikzpicture}
```

The cloud callout supports the absolute callout pointer, relative callout pointer and callout pointer shorten keys, as described above. The main shape can be modified using the same keys as the cloud shape. The following keys are also supported:

```
/pgf/callout pointer start size=(value) (no default, initially .2 of callout)
```

Sets the size of the first segment in the pointer (i.e., the segment nearest the main cloud shape). There are three possible forms for $\langle value \rangle$:

- A single dimension (e.g., 5pt), in which case the first ellipse will have equal diameters of 5pt.
- Two dimensions (e.g., 10pt and 2.5pt), which sets the x and y diameters of the first ellipse.
- A decimal fraction (e.g., .2 of callout), in which case the x and y diameters of the first ellipse will be set as fractions of the width and height of the main shape. The keyword of callout cannot be omitted.

```
/pgf/callout pointer end size=(value) (no default, initially .1 of callout)
```

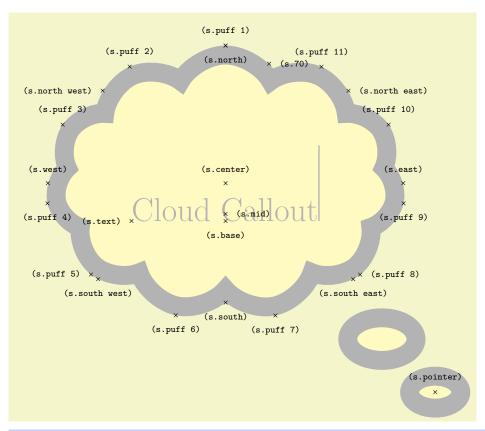
Sets the size of the last ellipse in the pointer.

```
/pgf/callout pointer segments=\langle number \rangle
```

(no default, initially 2)

Sets the number of segments in the pointer. Note that PGF will happily overlap segments if too many are specified.

The anchors for this shape are shown below (anchor 70 is an example of a border anchor). The pointer direction is ignored when placing anchors and the pointer anchor can only be used to position the shape when the relative anchor is specified. Note that the center of the last segment is drawn at the pointer anchor.



```
\Huge
\begin{tikzpicture}
 \node[name=s, shape=cloud callout, style=shape example, cloud puffs=11, aspect=1.5,
   cloud puff arc=120,inner xsep=.5cm, callout pointer start size=.25 of callout,
   callout pointer end size=.15 of callout, callout relative pointer={(315:4cm)},
   callout pointer segments=2] {Cloud Callout\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {puff 1/above, puff 2/above, puff 3/above, puff 4/below,
    puff 5/left, puff 6/below, puff 7/below, puff 8/right,
    puff 9/below, puff 10/above, puff 11/above, 70/right,
    center/above, base/below, mid/right, text/left,
    north/below, south/below, east/above, west/above,
    north west/left, north east/right,
    south west/below, south east/below,pointer/above}
 \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
   node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

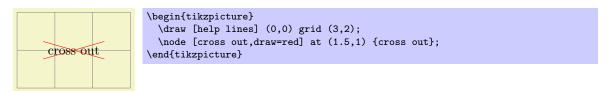
49.8 Miscellaneous Shapes

```
\usepgflibrary{shapes.misc} % MEX and plain TeX and pure pgf \usepgflibrary[shapes.misc] % ConTeXt and pure pgf \usetikzlibrary{shapes.misc} % MEX and plain TeX when using TikZ \usetikzlibrary[shapes.misc] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that do not fit into the previous categories.

Shape cross out

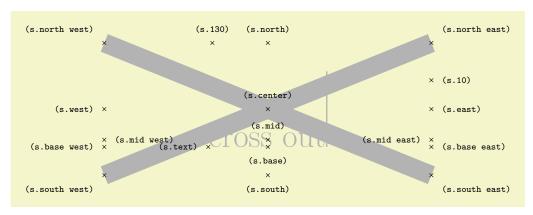
This shape "crosses out" the node. Its foreground path are simply two diagonal lines between the corners of the node's bounding box. Here is an example:



A useful application is inside text as in the following example:

```
Cross out! Cross \tikz[baseline] \node [cross out,draw,anchor=text] {me}; out!
```

This shape inherits all anchors from the rectangle shape, see also the following figure:



```
\Huge
\begin{tikzpicture}
\node[name=s,shape=cross out,shape example] {cross out\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
    west/left, center/above, east/right,
    mid west/right, mid/above, mid east/left,
    base west/left, base/below, base east/right,
    south west/below left, south/below, south east/below right,
    text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape strike out

This shape is identical to the **cross** out shape, only its foreground path consists of a single line from the lower left to the upper right.

```
Strike me out! Strike \tikz[baseline] \node [strike out,draw,anchor=text] {me}; out!
```

See the cross out shape for the anchors.

Shape rounded rectangle

This shape is a rectangle which can have optionally rounded sides.

```
Hallo \begin{tikzpicture} \node[rounded rectangle, draw, fill=red!20]{Hallo}; \end{tikzpicture}
```

There are keys to specify how the sides are rounded (to use these keys in TikZ, simply remove the /pgf/ path).

/pgf/rounded rectangle arc length= $\langle angle \rangle$

(no default, initially 180)

Sets the length of the arcs for the rounded ends. Recommended values for $\langle angle \rangle$ are between 90 and 180.

```
\lambda \begin{tikzpicture} \\ \matrix[row sep=5pt, every node/.style={draw, rounded rectangle}] { \\ \node[rounded rectangle arc length=180] {180}; \\ \\ \node[rounded rectangle arc length=120] {120}; \\ \\ \node[rounded rectangle arc length=90] {90}; \\}; \\ \end{tikzpicture}
```

/pgf/rounded rectangle west arc= $\langle arc \ type \rangle$

(no default, initially convex)

Sets the style of the rounding for the left side. The permitted values for $\langle arc\ type \rangle$ are concave, convex, or none.

```
Concave

\[
\text{Convex} \text{\text{None}} \text{\text{None}} \text{\text{begin{tikzpicture}}} \\
\text{\text{None}} \text{\t
```

/pgf/rounded rectangle left arc= $\langle arc \ type \rangle$

(style, no default)

Alternative key for specifying the west arc.

```
/pgf/rounded rectangle east arc=\langle arc \ type \rangle
```

(no default, initially convex)

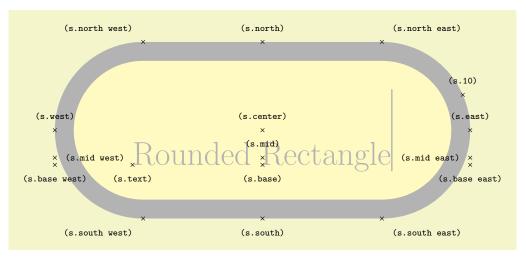
Sets the style of the rounding for the east side.

```
/pgf/rounded rectangle right arc=\langle arc type\rangle
```

(style, no default)

Alternative key for specifying the east arc.

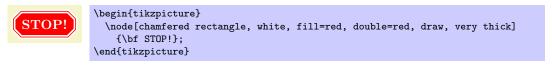
The anchors for this shape are shown below (anchor 10 is an example of a border angle). Note that if only one side is rounded, the center anchor will not be the precise center of the shape.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rounded rectangle, shape example, inner xsep=1.5cm, inner ysep=1cm]
  {Rounded Rectangle\vrule width 1pt height 2cm};
 \foreach \anchor/\placement in
   {center/above, text/below,
                                   10/above,
                  mid west/right, mid east/left,
    mid/above,
    base/below,
                  base west/below, base east/below,
    north/above, south/below, east/above, west/above,
    north west/above left, north east/above right,
    south west/below left,
                              south east/below right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape chamfered rectangle

This shape is a rectangle with optionally chamfered corners.



There are PGF keys to specify how this shape is drawn (to use these keys in TikZ simply remove the /pgf/ path).

```
/pgf/chamfered rectangle angle=\langle angle \rangle
```

(no default, initially 45)

Sets the angle from the vertical for the chamfer.

```
abc loss | loss
```

```
/pgf/chamfered rectangle xsep=\langle length \rangle
```

(no default, initially .666ex)

Sets the distance that the chamfer extends horizontally beyond the node contents (which includes the inner sep). If $\langle length \rangle$ is large, such that the top and bottom chamfered edges would cross, then $\langle length \rangle$ is ignored and the chamfered edges are drawn so that they meet in the middle.

```
def 456 \ \text{begin{tikzpicture}} \tikzset{every node/.style={chamfered rectangle, draw}} \node[chamfered rectangle xsep=2pt] {def}; \node[chamfered rectangle xsep=2cm] at (1.5,0) {456}; \end{tikzpicture}
```

```
/pgf/chamfered rectangle ysep=\langle length \rangle
```

(no default, initially .666ex)

Sets the distance that the chamfer extends vertically beyond the node contents. If $\langle length \rangle$ is large, such that the left and right chamfered edges would cross, then $\langle length \rangle$ is ignored and the chamfered edges are drawn so that they meet in the middle.

```
/pgf/chamfered rectangle sep=\langle length \rangle
```

(no default, initially .666ex)

Sets both the xsep and ysep simultaneously.

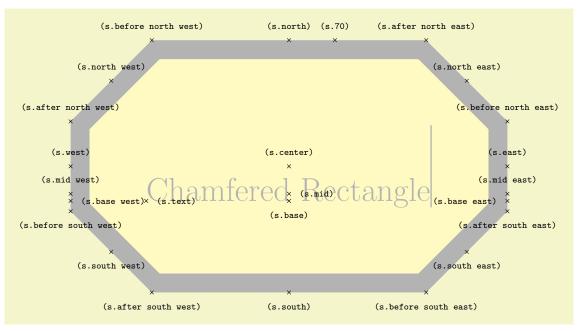
```
/pgf/chamfered rectangle corners=\langle list \rangle
```

(no default, initially chamfer all)

Specifies which corners are chamfered. The corners are identified by their "compass point" directions (i.e. north east, north west, south west, and south east), and must be separated by commas (so if there is more than one corner in the list, it must be surrounded by braces). Any corners not mentioned in $\langle list \rangle$ are automatically not chamfered. Two additional values chamfer all and chamfer none, are also permitted.

```
| The latest the latest tensor of the latest tensor
```

The anchors for this shape are shown below (anchor 60 is an example of a border angle.



```
\Huge
\begin{tikzpicture}
 \node[name=s,shape=chamfered rectangle, chamfered rectangle sep=1cm,
        shape example, inner ysep=1cm, inner xsep=.75cm]
    {Chamfered Rectangle\vrule width1pt height2cm};
 \foreach \anchor/\placement in
    {text/right, center/above,
                                   70/above,
    base/below, base east/left, base west/right,
    mid/right, mid east/above, mid west/above,
    north/above, south/below, east/above, west/above,
    before north east/above, north east/above, after north east/above,
    before north west/above, north west/above, after north west/above,
     before south west/below, south west/below, after south west/below,
    before south east/below, south east/below, after south east/below} \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

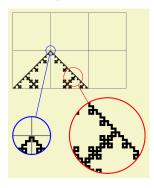
50 Spy Library: Magnifying Parts of Pictures

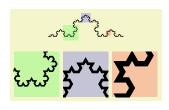
```
\usetikzlibrary{spy} % MEX and plain TeX \usetikzlibrary[spy] % ConTeXt
```

The package defines options for creating pictures in which some part of the picture is repeated in another area in a magnified way (as if you were looking through a spyglass, hence the name).

50.1 Magnifying a Part of a Picture

The idea behind the spy library is to make is easy to create high-density pictures in which some important parts are repeated somewhere, but magnified as if you were looking through a spyglass:





```
\begin{tikzpicture}[spy using overlays={size=12mm}]
  \draw [decoration=Koch snowflake]
    decorate { decorate{ decorate{ (0,0) -- (2,0) }}};

\spy [green,magnification=3] on (0.6,0.1) in node at (-0.3,-1);
  \spy [blue,magnification=5] on (1,0.5) in node at (1,-1);
  \spy [red,magnification=10] on (1.6,0.1) in node at (2.3,-1);
  \end{tikzpicture}
```

Note that this magnification uses what is called a *canvas transformation* in this manual: Everything is magnified, including line width and text.

In order for "spying" to work, the picture obviously has to be drawn several times: Once at its normal size and then again for each "magnifying glass." Several keys and commands work in concert to make this possible:

- You need to make TikZ aware of the fact that a picture (or just a scope) is to be magnified. This is done by adding the special key spy scope to a {scope} or {tikzpicture} (which is also just a scope). Some special keys like spy using outlines implicitly set the spy scope.
- Inside this scope you may then use the command \spy, which is only available inside such scopes (so there is no danger of you inadvertently using this command outside such a scope). This command has a special syntax and will (at some point) create two nodes: One node that shows the magnified picture (called the spy-in node) and another node showing which part of the original picture is magnified (called the spy-on node). The spy-in node is, indeed, a normal node, so it can have any shape or border that you like and you can apply all of TikZ's advanced features to it. The only difference compared to a normal node is that instead of some "text" it contains a magnified version of the picture, clipped to the size of the node.

The \spy command does not create the nodes immediately. Rather, the creation of these nodes is postponed till the end of the spy scope in which the \spy command is used. This is necessary since in order to repeat the whole scope inside the node containing the magnified version, the whole picture needs to be available when this node is created.

A basic question any library for "magnifying things" has to address is how you specify which part of the picture is to be magnified (the spy-on node) and where this magnified part is to be shown (the spy-in node). There are two possible ways:

1. You specify the size and position of the spy-on node. Then the size of the spy-in node is determined by the size of the spy-on node and the magnification factor – you can still decide where the spy-in node should be placed, but not its size.

2. Alternatively, you specify the size and position of the spy-in node. Then, similarly to the first case, the size of the spy-on node is determined implicitly and you can only decide where the spy-on node should be placed, but not its size.

The spy library uses the second method: You specify the size and position of the spy-in nodes, the sizes of the spy-on nodes are then computed automatically.

50.2 Spy Scopes

/tikz/spy scope=(options) (default empty)

This option may be used with a {scope} or any environment that creates such a scope internally (like {tikzpicture}). It has the following effects:

- It resets a number of graphic state parameters, including the color, line style, and others. This is necessary for technical reasons.
- It tells TikZ that the content of the scope should be saved internally in a special box.
- It defines the command \spy so that it can be used inside the scope.
- At the end of the scope, the nodes belonging to the \spy commands used inside the scope are created.
- The \(\langle options \rangle \) are saved in an internal style. Each time \(\spy\) is used, these \(\langle options \rangle \) will be used.
- Three keys are defined that provide useful shortcuts:

```
\texttt{/tikz/size=}\langle dimension \rangle (no default)
```

Inside a spy scope, this is a shortcut for minimum size.

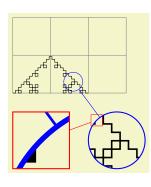
```
/\text{tikz/height} = \langle dimension \rangle (no default)
```

Inside a spy scope, this is a shortcut for minimum height.

```
/\text{tikz/width} = \langle dimension \rangle (no default)
```

Inside a spy scope, this is a shortcut for minimum width.

It is permissible to nest spy scopes. In this case, all \spy commands inside the inner spy scope only have an effect on material inside the scope, whereas \spy commands outside the inner spy scope but inside the outer spy scope allow you to "spy on the spy."



50.3 The Spy Command

 $\spy[\langle options \rangle]$ on $\langle coordinate \rangle$ in node $\langle node\ options \rangle$;

This command can only be used inside a spy scope. Let us start with the syntax:

- The \spy command is not a special case of \path. Rather, it has a small parser of its own.
- Following the optional $\langle options \rangle$, you must write on, followed by a coordinate. This coordinate will be the center of the area that is to be magnified.

• Following the $\langle coordinate \rangle$, you must write in node followed by some $\langle node\ options \rangle$. The syntax for these options is the same as for a normal node path command, such as [left] or (foo) [red] at (bar). However, $\langle node\ options \rangle$ are not followed by a curly brace. Rather, the $\langle node\ options \rangle$ must directly be followed by a semicolon.

The effect of this command is the following: The $\langle options \rangle$, $\langle coordinate \rangle$, and $\langle node\ options \rangle$ are stored internally till the end of the current spy scope. This means that, in particular, you can reference any node inside the spy scope, even if it is not yet defined when the \spy command is given. At the end of the current spy scope, two nodes are created, called the spy-in node and the spy-on node.

- The *spy-in node* is the node that contains a magnified part of the picture (the node *in* which we see on what we spy). This node is, indeed, a normal TikZ node, so you can use all standard options to style this node. In particular, you can specify a shape or a border color or a drop shadow or whatever. The only thing that is special about this node is that instead of containing some normal text, its "text" is the magnified picture.
 - To be precise, the picture of the spy scope is scaled by a certain factor, specified by the lens or magnification options discussed below, and is shifted in such a way that the $\langle coordinate \rangle$ lies at the center of the spy-on node.
- The spy-on node is a node that is centered on the $\langle coordinate \rangle$ and whose size reflects exactly the area shown inside the spy-in node (the node containing on what we spy).

Let us now go over what happens in detail when the two nodes are created:

- 1. A scope is started. Two sets of options are used with this scope: First, the options passed to the enclosing spy scope and then the *options* (which will, thus, overrule the options of the spy scope).
- 2. Then, the spy-on node is created. However, we will first discuss the spy-in node.
- 3. The spy-in node is created after the spy-on node (and, hence, will cover the spy-on node in case they overlap). When this node is created, the \(\lambda node \ options \rangle \) are used in addition to the effect caused by the \(\lambda options \rangle \) and the options of the \(\{ \text{spy scope} \} \). Additionally, the following style is used:

```
/tikz/every spy in node
```

(style, no value)

This style is used with every spy-in node.

The position of the node (the at option) is set to the $\langle coordinate \rangle$ by default, so that it will cover the to-be-magnified area. You can change this by providing the at option yourself:

```
\begin{tikzpicture}
    [spy using outlines={circle, magnification=3, size=1cm}]

\draw [decoration=Koch curve type 1]
    decorate{ decorate{ (0,0) -- (2,0) }};

\spy [red] on (1.6,0.3) in node;
    \spy [blue] on (1,1) in node at (1,-1);
\end{tikzpicture}
```

No "text" can be specified for the node. Rather, the "text" shown inside this node is the picture of the current spy scope, but canvas-transformed according to the following key:

```
\texttt{/tikz/lens} = \langle options \rangle (no default)
```

The $\langle options \rangle$ should contain transformation commands like scale or rotate. These transformations are applied to the picture when it is shown inside the spy-on node.

Since the most common transformation is undoubtedly a simple scaling, there is a special style for this:

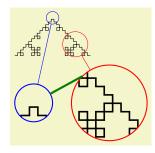
```
/\text{tikz/magnification} = \langle number \rangle (no default)
```

This has the same effect as saying lens={scale= $\langle number \rangle$ }.

Now, usually the size of a node is determined in such a way that it "fits" around the text of the node. For a spy-on node this is not a good approach since the "text" of this node would contain "the whole picture." Because of this, TikZ acts as if the "text" of the node has zero size. You

must then use keys like minimum size to cause the node to have a certain size. Note that the key size is an abbreviation for minimum size inside a spy scope.

You can name the spy-on node in the usual ways. Additionally, the node is (also) always named tikzspyinnode. Following the spy scope, you can use this node like any other node:



```
\begin{tikzpicture}
\begin{scope}
  [spy using outlines={circle, magnification=3, size=2cm, connect spies}]

\draw [decoration=Koch curve type 1]
  decorate{ decorate{ decorate{ (0,0) -- (2,0) }};

\spy [red] on (1.6,0.3) in node (a) [left] at (3.5,-1.25);

\spy [blue, size=1cm] on (1,1) in node (b) [right] at (0,-1.25);

\end{scope}
  \draw [ultra thick, green!50!black] (b) -- (a.north west);

\end{tikzpicture}
```

4. Once both nodes have been created, the current value of the following key is used to connect them:

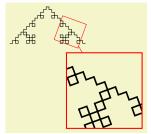
```
/tikz/spy connection path=\langle code \rangle
```

(no default, initially empty)

The $\langle code \rangle$ is executed after the spy-on and spy-in nodes have just been created. Inside this $\langle code \rangle$, the two nodes can be accessed as tikzspyinnode and tikzspyonnode. For example, the key connect spies sets this command to

```
\draw[thin] (tikzspyonnode) -- (tikzspyinnode);
```

Returning to the creation of the spy-in node: This node is centered on $\langle coordinate \rangle$ (more precisely, its anchor is set to center and the at option is set to $\langle coordinate \rangle$). Its size and shape are initially determined in the same way as the size and shape of the spy-on node (unless, of course, you explicitly provide a different shape for, say, the spy-on node locally, which is not really a good idea). Then, additionally, the *inverted* transformation done by the lens option is applied, resulting in a node whose size and shape exactly corresponds to the area in the picture that is shown in the spy-on node.



```
\begin{tikzpicture}
  [spy using outlines={lens={scale=3,rotate=20}, size=2cm, connect spies}]

\draw [decoration=Koch curve type 1]
  decorate{ decorate{ decorate{ (0,0) -- (2,0) }};

\spy [red] on (1.6,0.3) in node at (2.5,-1.25);
\end{tikzpicture}
```

Like for the spy-in node, a style can be used to format the spy-on node:

```
/tikz/every spy on node
```

(style, no value)

This style is used with every spy-on node.

The spy-on node is named tikzspyonnode (but, as always, this node is only available after the spy scope). If you have multiple spy-on nodes and you would like to access all of them, you need to use the name key inside the every spy on node style.

The inner sep and outer sep of both spy-in and spy-on nodes are set to Opt.

50.4 Predefined Spy Styles

There are some predefined styles that make using the spy library easier. The following two styles can be used instead of spy scope, they pass their $\langle options \rangle$ directly to spy scope. They additionally set up the graphic styles to be used for the spy-in nodes and the spy-on nodes in some special way.

```
/tikz/spy using outlines=(options)
```

(default empty)

This key creates a spy scope in which the spy-in node is drawn, but not filled, using a thick line; and the spy-on node is drawn, but not filled, using a very thin line.

```
\begin{tikzpicture}
  [spy using outlines={circle, magnification=3, size=1cm, connect spies}]

\draw [decoration=Koch curve type 1]
  decorate{ decorate{ decorate{ (0,0) -- (2,0) }};

\spy [red] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

/tikz/spy using overlays=(options)

(default empty)

This key creates a spy scope in which both the spy-in and spy-on nodes are filled, but with the fill opacity set to 20%.



```
\begin{tikzpicture}
  [spy using overlays={circle, magnification=3, size=1cm, connect spies}]

\draw [decoration=Koch curve type 1]
  decorate{ decorate{ decorate{ (0,0) -- (2,0) }};

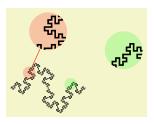
\spy [green] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

The following style is useful for connecting the spy-in and the spy-on nodes:

/tikz/connect spies

(no value)

Causes the spy-in and the spy-on nodes to be connected by a thin line.



```
\begin{tikzpicture}
  [spy using overlays={circle, magnification=3, size=1cm}]

\draw [decoration=Koch curve type 2]
  decorate{ decorate{ decorate{ (0,0) -- (2,0) }};

\spy [green] on (1.6,0.1) in node at (3,1);
  \spy [red,connect spies] on (0.5,0.4) in node at (1,1.5);
\end{tikzpicture}
```

50.5 Examples

Usually, the spy-in node and the spy-on node should have the same shape. However, you might also wish to use the circle shape for the spy-on node and the magnifying glass shape for the spy-in node:



```
\tikzset{spy using mag glass/.style={
    spy scope={
    every spy on node/.style={
        circle,
        fill, fill opacity=0.2, text opacity=1},
    every spy in node/.style={
        magnifying glass, circular drop shadow,
        fill=white, draw, ultra thick, cap=round},
    #1
    }}}
\begin{tikzpicture}[spy using mag glass={magnification=3, size=1cm}]
    \draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
    \spy [green!50!black] on (1.6,0.1) in node at (2.5,-0.5);
    \end{tikzpicture}
```

With the magnifying glass, you can also put it "on top" of the picture itself:



```
\begin{tikzpicture}
[spy scope={magnification=4, size=1cm},
  every spy in node/.style={
    magnifying glass, circular drop shadow,
    fill=white, draw, ultra thick, cap=round}]

\draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }};

\spy on (1.6,0.1) in node;
\end{tikzpicture}
```

51 SVG-Path Library

```
\usepgflibrary{svg.path} % MEX and plain TeX and pure pgf \usepgflibrary[svg.path] % ConTeXt and pure pgf \usetikzlibrary{svg.path} % MEX and plain TeX when using TikZ \usetikzlibrary[svg.path] % ConTeXt when using TikZ
```

This library defines a command that allows you to specify a path using the SVG-syntax.

$\protect\operatorname{pgfpathsvg}\{\langle path\rangle\}$

This command extends the current path by a $\langle path \rangle$ given in the SVG-path-data syntax. This syntax is described in detail in Section 8.3 of the SVG-specification, Version 1.1.

In principle, the complete syntax is supported and the library just provides a parser and a mapping to basic layer commands. For instance, M 0 10 is mapped to \pgfpathmoveto{\pgfpoint{0pt}{10pt}}. There are, however, a few things to be aware of:

- The computation underlying the arc commands A and a are not numerically stable, which may result in quite imprecise arcs. Bézier curves, both quadratic and cubic, are not affected, neither are arcs spanning degrees that are multiples of 90°.
- The dimensionless units of SVG are always interpreted as points (pt). This is a problem with paths like M 20000 0, which will raise an error message since TEX cannot handle dimensions larger than about 16000 points.
- All coordinate and canvas transformations apply to the path in the usual fashion.
- The \pgfpathsvg command can be freely intermixed with other path commands.



52 To Path Library

```
\usetikzlibrary{topaths} % MEX and plain TEX \usetikzlibrary[topaths] % ConTEXt
```

This library provides predefined to paths for use with the to path operation. After loading this package, you can say for instance to [loop] to add a loop to a node.

This library is loaded automatically by TikZ, so you do not need to load it yourself.

52.1 Straight Lines

The following style installs a to path that is simply a straight line from the start coordinate to the target coordinate.

/tikz/line to (no value)

Causes a straight line to be added to the path upon a to or an edge operation.

```
\tikz {\draw (0,0) to[line to] (1,0);}
```

52.2 Move-Tos

The following style installs a to path that simply "jumps" to the target coordinate.

/tikz/move to (no value)

Causes a move to be added to the path upon a to or an edge operation.

```
\tikz \draw (0,0) to[line to] (1,0) to[move to] (2,0) to[line to] (3,0);
```

52.3 Curves

The curve to style causes the to path to be set to a curve. The exact way this curve looks can be influenced via a number of options.

```
/tikz/curve to (no value)
```

Specifies that the to path should be a curve. This curve will leave the start coordinate at a certain angle, which can be specified using the out option. It reaches the target coordinate also at a certain angle, which is specified using the in option. The control points of the curve are at a certain distance that is computed in different ways, depending on which options are set.

All of the following options implicitly cause the curve to style to be installed.

```
/\text{tikz/out} = \langle angle \rangle (no default)
```

The angle at which the curve leaves the start coordinate. If the start coordinate is a node, the start coordinate is the point on the border of the node at the given $\langle angle \rangle$. The control point will, thus, lie at a certain distance in the direction $\langle angle \rangle$ from the start coordinate.

```
/\text{tikz/in} = \langle angle \rangle (no default)
```

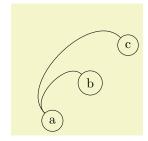
The angle at which the curve reaches the target coordinate.

```
/tikz/relative=\langle true or false \rangle (default true)
```

This option tells TikZ whether the in and out angles should be considered absolute or relative. Absolute means that an out angle of 30° means that the curve leaves the start coordinate at an angle of 30° relative to the paper (unless, of course, further transformations have been installed).

A relative angle is, by comparison, measured relative to a straight line from the start coordinate to the target coordinate. Thus, a relative angle of 30° means that the curve will bend to the left from the line going straight from the start to the target. For the target, the relative coordinate is measured in the same manner, namely relative to the line going from the start to the target. Thus, an angle of 150° means that the curve will reach target coming slightly from the left.

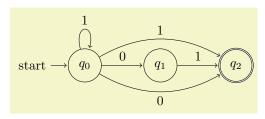
```
\begin{tikzpicture} [out=45,in=135,relative] \draw (0,0) to (1,0) to (2,1) to (2,2); \end{tikzpicture}
```



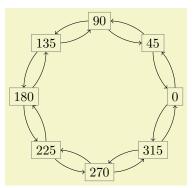
/tikz/bend left= $\langle angle \rangle$

(default last value)

This option sets $\operatorname{out}=\langle angle \rangle$, $\operatorname{in}=180-\langle angle \rangle$, relative. If no $\langle angle \rangle$ is given, the last given bend left or bend right angle is used.



```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid]
  \node[state,initial] (q_0)
 \node[state]
                          (q_1) [right=of q_0] {$q_1$};
 \label{local_problem} $$ \c [state, accepting](q_2) [right=of q_1] {$q_2$}; $$
 \path[->] (q_0) edge
                                      node [above]
                                                     {0} (q_1)
                   edge [loop above] node
                                                      {1} ()
                   edge [bend left] node [above]
                                                      {1} (q_2)
                   edge [bend right] node [below]
                                                      {0} (q_2)
            (q_1) edge
                                      node [above]
                                                      {1} (q_2);
\end{tikzpicture}
```



/tikz/bend right= $\langle angle \rangle$

(default last value)

Works like the bend left option, only the bend is to the other side.

/tikz/bend angle= $\langle angle \rangle$

(no default)

Sets the angle to be used by the bend left or bend right, but without actually selecting the curve to or the relative option. This is useful for globally specifying a bend angle for a whole picture.

/tikz/looseness= $\langle number \rangle$

(no default, initially 1)

This number specifies how "loose" the curve will be. In detail, the following happens: TikZ computes the distance between the start and the target coordinate (if the start and/or target coordinate are nodes, the distance is computed between the points on their border). This distance is then multiplied by a fixed factor and also by the factor $\langle number \rangle$. The resulting distance, let us call it d, is then used as the distance of the control points from the start and target coordinates.

The fixed factor has been chosen in such a way that if $\langle number \rangle$ is 1, if the in and out angles differ by 90°, then a quarter circle results:



```
\tikz \draw (0,0) to [out=0,in=-90] (1,1);
\tikz \draw (0,0) to [out=0,in=-90,looseness=0.5] (1,1);
```

/tikz/out looseness= $\langle number \rangle$

(no default)

Specifies the looseness factor for the out distance only.

/tikz/in looseness=⟨number⟩

(no default)

Specifies the looseness factor for the in distance only.

/tikz/min distance=⟨distance⟩

(no default)

If the computed distance for the start and target coordinates are below $\langle distance \rangle$, then $\langle distance \rangle$ is used instead.

/tikz/max distance= $\langle distance \rangle$

(no default)

If the computed distance for the start and target coordinates are above $\langle distance \rangle$, then $\langle distance \rangle$ is used instead.

/tikz/out min distance= $\langle distance \rangle$

(no default)

The minimum distance set only for the start coordinate.

/tikz/out max distance= $\langle distance \rangle$

(no default)

The maximum distance set only for the start coordinate.

/tikz/in min distance= $\langle distance \rangle$

(no default)

The minimum distance set only for the target coordinate.

/tikz/in max distance= $\langle distance \rangle$

(no default)

The maximum distance set only for the target coordinate.

/tikz/distance=\langle distance \rangle

(no default)

Set the minimum and maximum distance to the same value $\langle distance \rangle$. Note that this causes any computed distance d to be ignored and $\langle distance \rangle$ to be used instead.



/tikz/out distance=⟨distance⟩

(no default)

Sets the minimum and maximum out distance.

/tikz/in distance=⟨distance⟩

(no default)

Sets the minimum and maximum in distance.

/tikz/out control=(coordinate)

(no default)

This option causes the $\langle coordinate \rangle$ to be used as the start control point. All computations of d are ignored. You can use a coordinate like +(1,0) to specify a point relative to the start coordinate.

/tikz/in control= $\langle coordinate \rangle$

(no default)

This option causes the $\langle coordinate \rangle$ to be used as the target control point.

$\texttt{/tikz/controls=}\langle coordinate \rangle$ and $\langle coordinate \rangle$

(no default)

This option causes the $\langle coordinate \rangle$ s to be used as control points.



```
\tikz \draw (0,0) to [controls=+(90:1) and +(90:1)] (3,0);
```

52.4 Loops

/tikz/loop

(no value)

This key is similar to the curve to key, but differs in the following ways: First, the actual target coordinate is ignored and the start coordinate is used as the target coordinate. Thus, it is allowed not to provide any target coordinate, which can be useful with unnamed nodes. Second, the looseness is set to 8 and the min distance to 5mm. These settings result in rather nice loops when the opening angle (difference between in and out) is 30°.



```
\begin{tikzpicture}
 \node [circle,draw] {a} edge [in=30,out=60,loop] ();
\end{tikzpicture}
```

/tikz/loop above

(style, no value)

Sets the loop style and sets in and out angles such that loop is above the node. Furthermore, the above option is set, which causes a node label to be placed at the correct position.



```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [loop above] node {x} ();
\end{tikzpicture}
```

/tikz/loop below

(style, no value)

Works like the previous option.

/tikz/loop left

(style, no value)

Works like the previous option.

/tikz/loop right

(style, no value)

Works like the previous option.

/tikz/every loop

(style, initially ->,shorten >=1pt)

This style is installed at the beginning of every loop.



53 Through Library

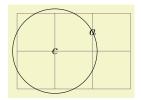
This library defines keys for creating shapes that go through given points.

```
/tikz/circle through=\langle coordinate\rangle
```

(no default)

When this key is given as an option to a node, the following happens:

- 1. The inner sep and the outer sep are set to zero.
- 2. The shape is set to circle.
- 3. The minimum size is set such that the circle around the center of the node (which is specified using at), goes through \(\langle coordinate \rangle \).



```
\begin{tikzpicture}
 \draw[help lines] (0,0) grid (3,2);
 \node (a) at (2,1.5) {\$a\};
 \node [draw] at (1,1) [circle through={(a)}] {\$c\};
 \end{tikzpicture}
```

54 Tree Library

```
\usetikzlibrary{trees} % MEX and plain TEX \usetikzlibrary[trees] % ConTEXt
```

This packages defines styles to be used when drawing trees.

54.1 Growth Functions

The package trees defines two new growth functions. They are installed using the following options:

```
/tikz/grow via three points=one child at (\langle x \rangle) and two children at (\langle y \rangle) and (\langle z \rangle) (no default)
```

This option installs a growth function that works as follows: If a parent node has just one child, this child is placed at $\langle x \rangle$. If the parent node has two children, these are placed at $\langle y \rangle$ and $\langle z \rangle$. If the parent node has more than two children, the children are placed at points that are linearly extrapolated from the three points $\langle x \rangle$, $\langle y \rangle$, and $\langle z \rangle$. In detail, the position is $x + \frac{n-1}{2}(y-x) + (c-1)(z-y)$, where n is the number of children and c is the number of the current child (starting with 1).

The net effect of all this is that if you have a certain "linear arrangement" in mind and use this option to specify the placement of a single child and of two children, then any number of children will be placed correctly.

Here are some arrangements based on this growth function. We start with a simple "above" arrangement:

```
begin{tikzpicture}[grow via three points={%
    one child at (0,1) and two children at (-.5,1) and (.5,1)}]
    \node at (0,0) {one} child;
    \node at (0,-1.5) {two} child child;
    \node at (0,-3) {three} child child child;
    \node at (0,-4.5) {four} child child child;
    \end{tikzpicture}

three

four
```

The next arrangement places children above, but "grows only to the right."

```
begin{tikzpicture}[grow via three points={%
    one child at (0,1) and two children at (0,1) and (1,1)}]
    \node at (0,0) {one} child;
    \node at (0,-1.5) {two} child child;
    \node at (0,-3) {three} child child child;
    \node at (0,-4.5) {four} child child child;
    \end{tikzpicture}

three

four
```

In the final arrangement, the children are placed along a line going down and right.

These examples should make it clear how you can create new styles to arrange your children along a line.

/tikz/grow cyclic (no value)

This style causes the children to be arranged "on a circle." For this, the children are placed at distance \tikzleveldistance from the parent node, but not on a straight line, but on points on a circle. Instead of a sibling distance, there is a sibling angle that denotes the angle between two given children.

```
/tikz/sibling angle=\langle angle \rangle (no default)
```

Sets the angle between siblings in the grow cyclic style.

Note that this function will rotate the coordinate system of the children to ensure that the grandchildren will grow in the right direction.

```
\begin{tikzpicture}
[grow cyclic,
  level 1/.style={level distance=8mm,sibling angle=60},
  level 2/.style={level distance=4mm,sibling angle=45},
  level 3/.style={level distance=2mm,sibling angle=30}]
  \coordinate [rotate=-90] % going down
  child foreach \x in {1,2,3}
    {child foreach \x in {1,2,3}}
    {child foreach \x in {1,2,3}};
  \end{tikzpicture}
```

/tikz/clockwise from= $\langle angle \rangle$

(no default)

This option also causes children to be arranged on a circle. However, the rule for placing children is simpler than with the grow cyclic style: The first child is placed at $\langle angle \rangle$ at a distance of \tikzleveldistance . The second child is placed at the same distance from the parent, but at angle $\langle angle \rangle - \tikzsiblingangle$. The third child is displaced by another \tikzsiblingangle in a clockwise fashion, and so on.

Note that this function will not rotate the coordinate system.

/tikz/counterclockwise from= $\langle angle \rangle$

(no default)

Works the same way as clockwise from, but sibling angles are added instead of subtracted.

54.2 Edges From Parent

The following styles can be used to modify how the edges from parents are drawn:

/tikz/edge from parent fork down

(style, no value)

This style will draw a line from the parent downwards (for half the level distance) and then on to the child using only horizontal and vertical lines.

```
root
left right
child child
```

```
\begin{tikzpicture}
  \node {root}
  [edge from parent fork down]
  child {node {left}}
  child {node {right}
    child[child anchor=north east] {node {child}}
    child {node {child}}
  };
  \end{tikzpicture}
```

/tikz/edge from parent fork right

(style, no value)

This style behaves similarly, only it will first draw its edge to the right.

```
\operatorname{root} - \begin{bmatrix} \operatorname{right} - \begin{bmatrix} \operatorname{child} \\ - \operatorname{child} \end{bmatrix} \\ \operatorname{left} \end{bmatrix}
```

```
\begin{tikzpicture}
  \node {root}
  [edge from parent fork right,grow=right]
  child {node {left}}
  child {node {right}}
    child {node {child}}
    child {node {child}}
  ;
};
\end{tikzpicture}
```

/tikz/edge from parent fork left

(style, no value)

Behaves similarly to the previous styles.

/tikz/edge from parent fork up

(style, no value)

Behaves similarly to the previous styles.

55 Turtle Graphics Library

This little library defines some keys to create simple turtle graphics in the tradition of the Logo programming language. These commands are mostly for fun, but they can also be used for more "serious" business.

```
\tikz[turtle/distance=2mm] \draw [turtle={home,forward,right,forward,left,forward,left,forward}];
```

Even though the turtle keys looks like an option, it uses the insert path option internally to produce a path.

The basic drawing model behind the turtle graphics is very simple: There is a (virtual) turtle that crawls around the page, thereby extending the path. The turtle always heads in a certain direction. When you move the turtle forward, you extend the path in that direction; turning the turtle just changes the direction, it does not cause anything to be drawn.

The turtle always moves relative to the last current point of the path and you can mix normal path commands with turtle commands. However, the direction of the turtle is managed independently of other path commands.

```
/\text{tikz/turtle} = \langle keys \rangle (no default)
```

This key executes the $\langle keys \rangle$ with the current key path set to /tikz/turtle.

```
\tikz[turtle/distance=2mm] \draw [turtle={home,fd,rt,fd,lt,fd,lt,fd}];
```

/tikz/turtle/home (no value)

Places the turtle at the origin and lets it head upward.

```
/\text{tikz/turtle/forward} = \langle distance \rangle (default see text)
```

Makes the turtle move forward by the given $\langle distance \rangle$. If no $\langle distance \rangle$ is specified, the current value of the following key is used:

```
/tikz/turtle/distance=\distance\ (no default, initially 1cm)
```

The default distance by which the turtle advances.

"Moving forward the turtle" actually means that, relative to the current last point on the path, a point at the given $\langle distance \rangle$ in the direction the turtle is currently heading is computed. Then, the operation to [turtle/how] is used to extend the path to this point.

```
/tikz/turtle/how (style, initially empty)
```

This style can set up the to path used by turtles. By setting this style you can change the to-path:

```
\tikz \draw [turtle={how/.style={bend left},home,forward,right,forward}];
```

/tikz/turtle/fd (no value)

An abbreviation for the forward key.

```
/\text{tikz/turtle/back} = \langle distance \rangle (default see text)
```

This has the same effect as a turtle/forward for the negated $\langle distance \rangle$ value.

```
/tikz/turtle/bk (no value)
```

An abbreviation for the back key.

```
/\text{tikz/turtle/left} = \langle angle \rangle (default 90)
```

Turns the turtle left by the given angle.

```
/tikz/turtle/lt (no value)
```

An abbreviation for the left key.

```
/\text{tikz/turtle/right} = \langle angle \rangle (default 90)
```

Turns the turtle right by the given angle.

```
/tikz/turtle/rt (no value)
```

An abbreviation for the right key.

Turtle graphics are especially nice in conjunction with the \foreach statement:



```
\tikz \filldraw [thick,blue,fill=blue!20]
  [turtle=home]
  \foreach \i in {1,...,5}
  {
    [turtle={forward,right=144}]
};
```

Part VI

Data Visualization

by Till Tantau

```
\tikz \datavisualization [scientific axes=clean]
  visualize as smooth line=Gaussian,
  \label{lem:continuous} Gaussian=\{pin in data=\{text=\{\$e^{-x^2}\}\$\}, when=x is 1\}\}
data [format=function] {
 var x : interval [-7:7] samples 51;
  func y = exp(-\value x*\value x);
  visualize as scatter,
  legend={south east outside},
  scatter={
    style={mark=*,mark size=1.4pt},
    label in legend={text={
        \sum_{i=1}^{10} x_i, where x_i \le U(-1,1)
data [format=function] {
  var i : interval [0:1] samples 20;
  func y = 0;
  func x = (rand + rand + rand + rand + rand +
            rand + rand + rand + rand + rand);
};
```

56 Introduction to Data Visualization

Data visualization is the process of converting data points, which typically consist of multiple numerical values, into a graphical representation. Examples include the well-known function plots, but pie charts, bar diagrams, box plots, or vector fields are also examples of data visualizations.

The data visualization subsystem of PGF takes a general, open approach to data visualization. Like everything else in PGF, there is a powerful, but not-so-easy-to-use basic layer in the data visualization system and a less flexible, but much simpler-to-use frontend layer. The present section gives an overview of the basic ideas behind the data visualization system.

56.1 Concept: Data Points

The most important input for a data visualization is always raw data. This data is typically present in different formats and the data visualization subsystem provides methods for reading such formats and also for defining new input formats. However, independently of the input format, we may ask what kind of data the data visualization subsystem should be able to process. For two-dimensional plots we need lists of pairs of real numbers. For a bar plot we usually need a list of numbers, possibly together with some colors and labels. For a surface plot we need a matrix of triples of real numbers. For a vector field we need even more complex data.

The data visualization subsystem makes no assumption concerning which kind of data is being processed. Instead, the whole "rendering pipeline" is centered around a concept called the *data point*. Conceptually, a data point is an arbitrarily complex record that represents one piece of data that should be visualized. Data points are *not* just coordinates in the plane or the numerical values that need to be visualized. Rather, they represent the basic units of the data that needs to be visualized.

Consider the following example: In an experiment we drive a car along a road and have different measurement instruments installed. We measure the position of the car, the time, the speed, the direction the car is heading, the acceleration, and perhaps some further values. A data point would consist of a record consisting of a timestamp together with the current position of the car (presumably two or three numbers), the speed vector (another two or three numbers), the acceleration (another two or three numbers), and perhaps the label text of the current experiment.

Data points should be "information rich." They might even contain more information than what will actually be visualized. It is the job of the rendering pipeline to pick out the information relevant to one particular data visualization – another visualization of the same data might pick different aspects of the data points, thereby hopefully allowing new insights into the data.

Technically, there is no special data structure for data points. Rather, when a special macro called \pgfdatapoint is called, the "totality" of all currently set keys with the /data point/ prefix in the current scope forms the data point. This is both a very general approach and quite fast since no extra data structures need to be created.

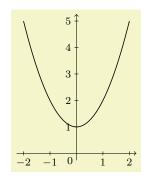
56.2 Concept: Visualization Pipeline

The visualization pipeline is a series of actions that are performed on the to-be-visualized data. The data is presented to the visualization pipeline in the form of a long stream of complex data points. The visualization pipeline makes several passes over this stream of data points. During the first pass(es), called the survey phase(s), information is gathered about the data points such as minimal and maximal values, which can be useful for automatic fitting of the data into a given area. In the main pass over the data, called the visualization phase, the data points are actually visualized, for instance in the form of lines or points.

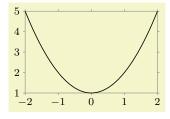
Like as for data points, the visualized pipeline makes no assumptions concerning what kind of visualization is desired. Indeed, one could even use it to produce a plain-text table. This flexibility is achieved by extensive use of objects and signals: When a data visualization starts, a number of signals (see Section 75.8 for an introduction to signals) are initialized. Then, numerous "visualization objects" are created that listen to these signals. These objects are all involved in processing the data points. For instance, the job of an interval mapper object is to map one attribute of a data point, such as a car's velocity, to another, such as the y-axis of a plot. For each data point the different signals are raised in a certain order and the different visualization objects now have a chance of preparing the data point for the actual visualization. Continuing the above example, there might be a second interval mapper that takes the computed y-position and applies a logarithm to it, because a log-plot was requested. Then another mapper, this time a polar mapper might be used to map everything to polar coordinates. Following this, a plot mark visualizer might actually draw something at the computed position.

The whole idea behind the rendering pipeline is that new kinds of data visualizations can be implemented, ideally, just by adding one or two new objects to the visualization pipeline. Furthermore, different kinds of plots can be combined in novel ways in this manner, which is usually very hard to do. For instance, the visualization pipeline makes it easy to create, say, polar-semilog-box-plots. At first sight, such new kinds of plots may seem frivolous, but data visualization is all about gaining insights into the data from as many different angles as possible.

Naturally, creating new classes and objects for the rendering pipeline is not trivial, so most users will just use the existing classes, which should, thus, be as flexible as possible. But even when one only intends to use existing classes, it is still tricky to setup the pipeline correctly since the ordering is obviously important and since things like axes and ticks need to be configured and taken care of. For this reason, the frontend libraries provide preconfigured rendering pipelines so that one can simply say that a data visualization should look like a line plot with school book axes or with scientific axes, which selects a certain visualization pipeline that is appropriate for this kind of plot:



```
\begin{tikzpicture}[scale=.7]
  \datavisualization [school book axes, visualize as smooth line]
  data [format=function] {
    var x : interval [-2:2];
    func y = \value x*\value x + 1;
  };
  \end{tikzpicture}
```



```
\begin{tikzpicture}[scale=.7]
  \datavisualization [scientific axes, visualize as smooth line]
  data [format=function] {
    var x : interval [-2:2];
    func y = \value x*\value x + 1;
  };
  \end{tikzpicture}
```

One must still configure such a plot (choose styles and themes and also specify which attributes of a data point should be used), but on the whole the plot is quite simple to specify.

57 Creating Data Visualizations

57.1 Overview

The present section explains how a data visualization is created in TikZ. For this, you need to include the datavisualization library and then use the command \datavisualization whose syntax is explained in the rest of the present section. This command is part of the following library:

```
\usetikzlibrary{datavisualization} % MEX and plain TEX \usetikzlibrary[datavisualization] % ConTEXt
```

This library must be loaded if you wish to use the \datavisualization command. It defines all styles needed to create basic data visualizations; additional, more specialized libraries need to be loaded for more advanced features.

In order to visualize, you basically need to do three things:

- 1. You need to select what kind of plot you would like to have (a "school book plot" or a "scientific 2d plot" or a "scientific spherical plot" etc.). This is done by passing an option to the \datavisualization command that selects this kind of plot.
- 2. You need to provide data points, which is done using the data command.
- 3. Additionally, you can add options that give you more fine-grained control over the way the visualization will look. You can configure the number of ticks and grid lines, where the labels are placed, the colors, or the fonts. Indeed, since the data visualization engine internally uses TikZ-styles, you can have extremely fine-grained control over how a plot will look like.

The syntax of the \datavisualization command is designed in such a way that if you only need to provide very few options to create plots that "look good by default".

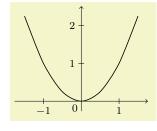
This section is structured as follows: First, the philosophy behind concepts like "data points," "axes," or "visualizers" is explained. Each of these concepts is further detailed in later section. Then, the syntax of the \datavisualization command is covered. The reference sections explain which predefined plot kinds are available.

57.2 Concept: Data Points and Data Formats

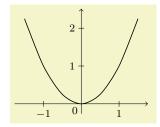
As explained in Section 56.1, data points are the basic entities that are processed by the data visualization engine. In order to specify data points, you use the data command, whose syntax is explained in more detail in Section 57.6. The data command allows you to either specify points "inline," directly inside your TeX-file; or you can specify the name of file that contains the data points.

Specifying data points. Data points can be formatted in different ways. For instance, in the so called *comma separated values* format, there is one line for each data point and the different attributes of a data point are separated by commas. Another common format is to specify data points using the so called *key-value* format, where on each line the different attributes of a data point are set using a comma-separated list of strings of the form attribute=value.

Here are two examples, where similar data is given in different formats:



```
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line]
  data {
      x, y
      -1.5, 2.25
      -1, 1
      -.5, .25
      0, 0
      .5, .25
      1, 1
      1.5, 2.25
    };
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line]
  data [format=function] {
    var x : interval [-1.5:1.5] samples 7;
    func y = \value x*\value x;
  };
\end{tikzpicture}
```

In the first example, no format needed to be specified explicitly since the default format is the one used for the data following the data keyword: A list of comma-separated values, where each line represents a data point.

Number accuracy. Data visualizations typically demand a much higher accuracy and range of values than TEX provides: TEX numbers are limited to 13 bits for the integer part and 16 bits for the fractional part. Because of this, the data visualization engine does not use PGF's standard representation of numbers and TEX dimensions and is does not use the standard parser when reading numbers in a data point. Instead, the fpu library, described in Section 37, is used to handle numbers.

This use of the fpu library has several effects that users of the data visualization system should be aware of:

- 1. You can use numbers like 1000000000000 or 0.0000000001 in a data points.
- 2. Since the fpu library does not support advanced parsing, you currently *cannot* write things like 3+2 in a data point number. This will result in an error
- 3. However, there is a loop-hole: If a "number" in a data point starts with a parenthesis, the value between the parentheses *is* parsed using the normal parser:
 - \bullet 100000 is allowed.
 - 2+3 yields an error.
 - (2+3) is allowed and evaluates to 5.
 - (100000) yields an error since 100000 is beyond the normal parser's precision.

The bottom line is that any normal calculations should be set inside round parentheses, while large numbers should not be surrounded by parentheses. Hopefully, in the future, these restrictions will be lifted

Section 58 gives an in-depth coverage of the available data formats and explains how new data formats can be defined.

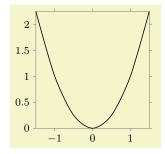
57.3 Concept: Axes, Ticks, and Grids

Most plots have two or three axes: A horizontal axis usually called the x-axis, a vertical axis called the y-axis, and possibly some axis pointing in a sloped direction called the z-axis. Axes are usually drawn as lines with ticks indicating interesting positions on the axes. The data visualization engine gives you detailed control over where these ticks are rendered and how many of them are used. Great care is taken to ensure that the position of ticks are chosen well by default.

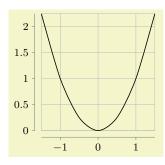
From the point of view of the data visualization engine, axes are a somewhat more general concept than "just" lines that point "along" some dimension: The data visualization engine uses axes to visualize any change of an attribute by varying the position of data points in the plane. For instance, in a polar plot, there is an "axis" for the angle and another "axis" for the distance if the point from the center. Clearly these axes vary the position of data points in the plane according to some attribute of the data points; but just as clearly they do not point in any "direction."

A great benefit of this approach is that the powerful methods for specifying and automatic inference of "good" positions for ticks or grid lines apply to all sorts of situations. For instance, you can use it to automatically put ticks and grid lines at well-chosen angles of a polar plot.

Typically, you will not need to specify axes explicitly. Rather, predefined styles take care of this for you:



```
\begin{tikzpicture}
  \datavisualization [
    scientific axes,
    x axis={length=3cm, ticks=few},
    visualize as smooth line
]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
\end{tikzpicture}
```

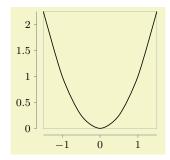


```
\detailstapicture}
\datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    all axes={grid},
    visualize as smooth line
]
    data [format=function] {
       var x : interval [-1.5:1.5] samples 7;
       func y = \value x*\value x;
    };
\end{tikzpicture}
```

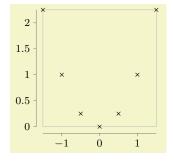
Section 59 explains in more detail how axes, ticks, and grid lines can be chosen and configured.

57.4 Concept: Visualizers

Data points and axes specify what is visualized and where. A visualizer specifies how they are visualized. One of the most common visualizers is a line visualizer which connects the positions of the data points in the plane using a line. Another common visualizer is the scatter plot visualizer where small marks are drawn at the positions of the data points. More advanced visualizers include, say, box plot visualizers or pie chart visualizers.



```
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    visualize as smooth line
]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
  \end{tikzpicture}
```



```
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    visualize as scatter
]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
  \end{tikzpicture}
```

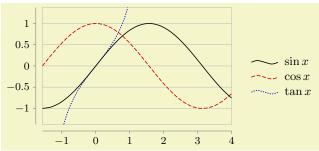
Section 60 provides more information on visualizers as well as reference lists.

57.5 Concept: Style Sheets and Legends

A single data visualizations may use more than one visualizer. For instance, if you wish to create a plot containing several lines, a separate visualizer is used for each line. In this case, two problems arise:

- 1. You may wish to make it easy for the reader to differentiate between the different visualizers. For instance, one line should be black, another should be red, and another blue. Alternatively, you might wish one line to be solid, another to be dashed, and a third to be dotted.
 - Specifying such styles is trickier than one might expect; experience shows that many plots use ill-chosen and inconsistent styling. For this reason, the data visualization introduces the notion of *style sheets* for visualizers and comes with some well-designed predefined style sheets.
- 2. You may wish to add information concerning what the different visualizers represent. This is typically done using a legend, but it is even better to add labels directly inside the visualization. Both approaches are supported.

An example where three functions are plotted and a legend is added is shown below. Two style sheets are used so that *both* the coloring and the dashing is varied.



```
\begin{tikzpicture}[baseline]
  \datavisualization [ scientific axes=clean,
                       y axis=grid,
                       visualize as smooth line/.list={sin,cos,tan},
                       style sheet=strong colors,
                       style sheet=vary dashing,
                       sin={label in legend={text=$\sin x$}},
                       cos={label in legend={text=$\cos x$}},
                       tan={label in legend={text=$\tan x$}},
                       data/format=function ]
 data [set=sin] {
   var x : interval [-0.5*pi:4];
    func y = sin(\forall x r);
  data [set=cos] {
    var x : interval [-0.5*pi:4];
    func y = cos(\forall x \ r);
 data [set=tan] {
    var x : interval [-0.3*pi:.3*pi];
    func y = tan(\forall x r);
 }:
\end{tikzpicture}
```

Section 61 details style sheets and legends.

57.6 Usage

Inside a TikZ picture you can use the \datavisualization command to create a data visualization. You can use this command several times in a picture to create pictures containing multiple data visualizations.

 $\datavisualization[\langle data\ visualization\ options \rangle] \langle data\ specification \rangle;$

This command is available only inside a {tikzpicture} environment.

The $\langle data\ visualization\ options \rangle$ are used to configure the data visualization, that is, how the data is to be depicted. The options are executed with the path prefix $/ \text{tikz/data}\ visualization$. This means that normal TikZ options like thin or red cannot be used here. Rather, a large number of options specific to data visualizations are available.

As a minimum, you should specify at least two options: First, you should use an option that selects an axis system that is appropriate for your plot. Typical possible keys are school book axes or scientific axes, detailed information on them can be found in Section 59.

Second, you use an option to select *how* the data should be visualized. This is done using a key like visualize as line which will, as the name suggests, visualize the data by connecting data points in the plane using a line. Similarly, visualize as smooth cycle will try to fit a smooth cycle through the data points. Detailed information on possible visualizers can be found in Section 60.

Following these options, the $\langle data\ specification \rangle$ is used to provide the actual to-be-visualized data. The syntax is somewhat similar to commands like \path: The $\langle data\ specification \rangle$ is a sequence of keywords followed by local options and parameters, terminated with a semicolon. (Indeed, like for the \path command, the $\langle data\ visualizers\ options \rangle$ need not be specified at the beginning, but additional option surrounded by square brackets may be given anywhere inside the $\langle data\ specification \rangle$.)

The different possible keywords inside the $\langle data \ specification \rangle$ are explained in the following.

```
\datavisualization ... data[\langle options \rangle] \{\langle inline \ data \rangle\} ...;
```

This command is used to specify data for the data visualization. It can be used several times inside a single visualization and each time the to-be-read data may have a different format, but the data will be visualized as if it have been specified inside a single data command.

The behaviour of the data command depends on whether the $\langle inline\ data \rangle$ is present. If it is not present, the $\langle options \rangle$ must be used to specify a source file from which the data is read; if the $\langle inline\ data \rangle$ is present no file will be used, instead the data should directly reside inside the TEX-file and be given between the curly braces surrounding the $\langle inline\ data \rangle$.

The $\langle options \rangle$ are executed with the prefix /pgf/data. The following options are always available:

```
/pgf/data/read from file=(filename) (no default, initially empty)
```

If you set the **source** attribute to a non-empty $\langle filename \rangle$, the data will be read from this file. In this case, no $\langle inline\ data \rangle$ may be present, not even empty curly braces should be provided.

```
\datavisualization ...
data [read from file=file1.csv]
data [read from file=file2.csv];
```

The other way round, if read from file is empty, the data must directly follow as (inline data).

```
\datavisualization ...
data {
    x, y
    1, 2
    2, 3
};
```

The second important key is format, which is used to specify the data format:

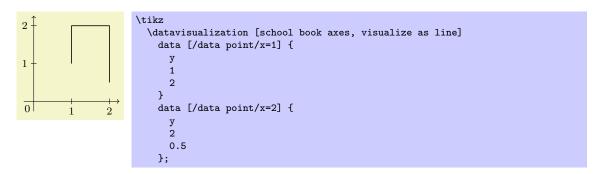
```
/pgf/data/format = \langle format \rangle (no default, initially table)
```

Use this key to locally set the format used for parsing the data, see Section 58 for a list of predefined formats.

The default format is the table-format, also known as "comma-separated values." The first line contains names of attributes separated by commas, all following lines constitute a data point where the attributes are given by the comma-separated values in that line.

Presetting attributes. Normally, the inline data or the external data contains for each data point the values of the different attributes. However, sometimes you may also wish to set an attribute to a fixed value for all data points of a data set. Suppose, for instance, that you have to source files experiment007.csv and experiment023.csv and you would like that for all data points of the first file the attribute /data point/experiment id is set to 7 while for the data points of the second file they are set to 23. In this case, you can specify the desired settings using an absolute path inside the $\langle options \rangle$. The effect will be local to the current data command:

```
\datavisualization...
data [/data point/experiment=7, read from file=experiment007.csv]
data [/data point/experiment=23, read from file=experiment023.csv];
```



Setting options for multiple data commands. You may wish to generally set the format once and for all. This can be done by using the following key:

```
/tikz/every data (style, no value)
```

This key is executed for every data command.

Another way of passing options to multiple data commands is to use the following facility: Whenever an option with the path /tikz/data visualization/data is used, the path will be remapped to /pgf/data. This means, in particular, that you can pass an option like data/format=table to the \datavisualization command to set the data format for all data commands of the data visualization.

Parsing inline data. When you specify data inline, T_EX needs to read the data "line-by-line," while T_EX normally largely ignores end-of-line characters. For this reason, the data visualization system temporarily changes the meaning of the end-of-line character. This is only possible if T_EX has not already processed the data in some other way (namely as the parameter to some macro).

The bottom line is that you cannot use inline data when the whole \datavisualization command is passed as a parameter to some macro that is not setup to handle "fragile" code. For instance, in a BEAMER frame you need to add the fragile option when a data visualization contains inline data.

The problem does not arise when an external data source is specified.

```
\datavisualization ... data\ point[\langle options \rangle] ...;
```

This command is used to specify data a single data point. The $\langle options \rangle$ are simply executed with the path /data point and then a data point is created. This means that inside the $\langle options \rangle$ you just specify the values of all attributes in key-value syntax.

```
2
data point [x=1, y=1] data point [x=1, y=2]
data point [x=2, y=2] data point [x=2, y=0.5];
```

```
/tikz/data visualization/data point=\langle options\rangle
```

(no default)

This key is the "key version" of the previous command. The difference is that this key can be used internally inside styles.

```
\datavisualization ... data group[\langle options \rangle] {\langle name \rangle}+={\langle data\ specifications \rangle} ...;
```

You can store a whole $\langle data\ specification \rangle$ in a data group. This allows you to reuse data in multiple places without having to write the data to an external file.

The syntax of this command comes in the following three variants:

```
• data group [\langle options \rangle] \{\langle name \rangle\} = \{\langle data \ specifications \rangle\}
```

- data group $[\langle options \rangle]$ $\{\langle name \rangle\}$ += $\{\langle data \ specifications \rangle\}$
- data group $[\langle options \rangle]$ $\{\langle name \rangle\}$

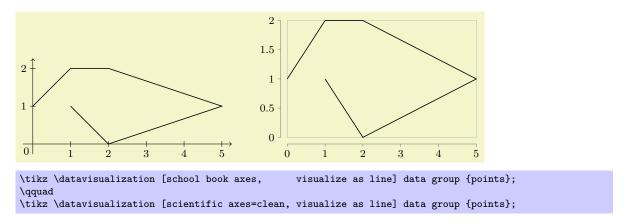
In the first case, a new data group called $\langle name \rangle$ is created (an existing data group of the same name will be erased) and the following $\langle data\ specifications \rangle$ is stored in this data group. The data group will not be fed to the rendering pipeline, but it is parsed at this point as if it were. The defined data group is defined globally, so you can used it in subsequent visualizations. The $\langle options \rangle$ are saved with the parsed $\langle data\ specifications \rangle$.

In the second case, an already existing data group is extended by adding the $\langle data \ specifications \rangle$ to it. In the third case (detected by noting that the $\langle name \rangle$ is neither followed by an equal sign nor a plus sign), the contents of the previously defined data group $\langle name \rangle$ is inserted. The $\langle options \rangle$ are also executed.

Let is now first create a data group. Note that nothing is drawn since the "dummy" data visualization is empty and used only for the definition of the data group.

```
\tikz \datavisualization data group {points} = {
    data {
        x, y
        0, 1
        1, 2
        2, 2
        5, 1
        2, 0
        1, 1
    }
};
```

We can now use this data in different plots:



```
\datavisualization ... scope[\langle options \rangle] \{\langle data \ specification \rangle\} ...;
```

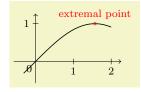
Scopes can be used to nest hierarchical data sets. The $\langle options \rangle$ will be executed with the path /pgf/data and will only apply to the data sets specified inside the $\langle data\ specification \rangle$, which may contain data or scope commands once more:

```
\datavisualization...
scope [/data point/experiment=7]
{
   data [read from file=experiment007-part1.csv]
   data [read from file=experiment007-part2.csv]
   data [read from file=experiment007-part3.csv]
}
scope [/data point/experiment=23, format=foo]
{
   data [read from file=experiment023-part1.foo]
   data [read from file=experiment023-part2.foo]
};
```

```
\datavisualization ... info[\langle options \rangle] \{\langle code \rangle\} ...;
```

This command will execute normal $TikZ \langle code \rangle$ at the end of a data visualization. The $\langle options \rangle$ are executed with the normal path /tikz/.

The only difference between this command and just giving the $\langle code \rangle$ directly following the data visualization is that inside the $\langle code \rangle$ following an info command you still have access to the coordinate system of the data visualization. In sharp contrast, TikZ code given after a data visualization can no longer access this coordinate system.



```
\begin{tikzpicture}[baseline]
  \datavisualization [ school book axes, visualize as line ]
  data [format=function] {
    var x : interval [-0.1*pi:2];
    func y = sin(\value x r);
  }
  info {
    \draw [red] (visualization cs: x={(.5*pi)}, y=1) circle [radius=1pt]
    node [above,font=\footnotesize] {extremal point};
  };
  \end{tikzpicture}
```

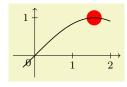
As can be seen, inside a data visualization a special coordinate system is available:

Coordinate system visualization

As for other coordinate systems, the syntax is (visualization cs: $\langle list \ of \ attribute-value \ pairs \rangle$). The effect is the following: For each pair $\langle attribute \rangle = \langle value \rangle$ in the $\langle list \rangle$ the key /data point/ $\langle attribute \rangle$ is set to $\langle value \rangle$. Then, it is computed where the resulting data point "would lie" on the canvas (however, no data point is passed to the visualizers).

```
\datavisualization ... info' [\langle options \rangle] {\langle code \rangle} ...;
```

This command works like info, only the $\langle code \rangle$ will be executed just before the visualization is done. This allows you to draw things behind the visualization.



```
\begin{tikzpicture}[baseline]
  \datavisualization [ school book axes, visualize as line ]
  data [format=function] {
    var x : interval [-0.1*pi:2];
    func y = sin(\value x r);
  }
  info' {
    \fill [red] (visualization cs: x={(.5*pi)}, y=1) circle [radius=2mm];
  };
  \end{tikzpicture}
```

Predefined node data visualization bounding box

This rectangle node stores a bounding box of the data visualization that is currently being constructed. This node can be useful inside info commands or when labels need to be added.

Predefined node data bounding box

This rectangle node is similar to data visualization bounding box, but it keeps track only of the actual data. The spaces needed for grid lines, ticks, axis labels, tick labels, and other all other information that is not part of the actual data is not part of this box.

57.7 Advanced: Executing User Code During a Data Visualization

The following keys can be passed to the \datavisualization command and allow you to execute some code at some special time during the data visualization process. For details of the process and on which signals are emitted when, see Section 62.

```
/tikz/data visualization/before survey=\langle code \rangle
```

(no default)

The $\langle code \rangle$ is passed to the **before survey** method of the data visualization object and then executed at the appropriate time (see Section 62 for details).

The following commands work likewise:

/tikz/data visualization/at start survey= $\langle code \rangle$	(no default)
/tikz/data visualization/at end survey= $\langle code \rangle$	(no default)
/tikz/data visualization/after survey= $\langle code \rangle$	(no default)
/tikz/data visualization/before visualization= $\langle code \rangle$	(no default)
/tikz/data visualization/at start visualization= $\langle code \rangle$	(no default)
/tikz/data visualization/at end visualization= $\langle code \rangle$	(no default)
/tikz/data visualization/after visualization= $\langle code \rangle$	(no default)

57.8 Advanced: Creating New Objects

You will need the following key only when you wish to create new rendering pipelines from scratch – instead of modifying an existing pipeline as you would normally do. In the following it is assumed that you are familiar with the concepts of Section 62.

$\label{eq:continuous} \verb|/tikz/data visualization/new object=|| options|| \\ (no default)$

This key servers two purposes:

- 1. This method makes it easy to create a new object as part of the rendering pipeline, using *options* to specify arguments rather that directly calling *pgfoonew*. Since you have the full power of the keys mechanism at your disposal, it is easy, for instance, to control whether or not parameters to the constructor are expanded or not.
- 2. The object is not created immediately, but only just before the visualization starts. This allows you to specify that an object must be created, but the parameter values of for its constructor may depend on keys that are not yet set. A typical application is the creating of an axis object: When you say scientific axes, the new object command is used internally to create two objects representing these axes. However, keys like x={length=5cm} can only later be used to specify the parameters that need to be passed to the constructor of the objects.

The following keys may be used inside the $\langle options \rangle$:

```
\label{eq:class_lambda} \mbox{\class=\class name)} \qquad \qquad \mbox{\constraints} \mbox{\co
```

The class of the to-be-created object.

```
/tikz/data visualization/when=\(\rho hase name\) (no default, initially before survey)
```

This key is used to specify when the object is to be created. As described above, the object is not created immediately, but at some time during the rendering process. You can specify any of the phases defined by the data visualization object, see Section 62 for details.

```
/\text{tikz/data visualization/store} = \langle key \ name \rangle (no default)
```

If the $\langle key \ name \rangle$ is not empty, once the object has been created, a handle to the object will be stored in $\langle key \ name \rangle$. If a handle is already stored in $\langle key \ name \rangle$, the object is not created twice.

```
/\text{tikz/data visualization/before creation} = \langle code \rangle (no default)
```

This code is executed right before the object is finally created. It can be used to compute values that are then passed to the constructor.

```
/\text{tikz/data visualization/after creation=}\langle code \rangle (no default)
```

This code is executed right after the object has just been created. A handle to the just-created object is available in \tikzdvobj.

```
/\text{tikz/data visualization/arg1} = \langle value \rangle (no default)
```

The value to be passed as the first parameter to the constructor. Similarly, the keys arg2 to arg8 specify further parameters passed. Naturally, only as many arguments are passed as parameters are set. Here is an example:

```
\tikzdatavisualizationset{
  new object={
    class = example class,
    arg1 = foo,
    arg2 = \bar
  }
}
```

causes the following object creation code to be executed later on:

```
\pgfoonew \tikzdvobj=new example class(foo,\bar)
```

Note that you key mechanisms like .expand once to pass the value of a macro instead of the macro itself:

Now, if \bar is set to This \emph{is} it. at the moment to object is created later on, the following object creation code is executed:

```
\pgfoonew \tikzdvobj=new example class(foo,This \emph{is} it)
```

```
/tikz/data visualization/arg1 from key=\langle key \rangle
```

(no default)

Works like the arg1, only the value that is passed to the constructor is the current value of the specified $\langle key \rangle$ at the moment when the object is created.

```
\tikzdatavisualizationset{
  new object={
    class = example class,
    arg1 from key = /tikz/some key
  }
}
\tikzset{some key/.initial=foobar}
```

causes the following to be executed:

```
\pgfoonew \tikzdvobj=new example class(foobar)
```

Naturally, the keys arg2 from key to arg8 from key are also provided.

```
/tikz/data visualization/arg1 handle from key=\langle key \rangle (no default)
```

Works like the arg1 from key, only the key must store an object and instead of the object a handle to the object is passed to the constructor.

58 Providing Data for a Data Visualization

58.1 Overview

The data visualization system needs a stream of data points as input. These data points can be directly generated by repeatedly calling the \pgfdatapoint command, but usually data is available in some special (text) format and one would like to visualize this data. The present section explains how data in some specific format can be fed to the data visualization system.

This section starts with an explanation of the main concepts. Then, the standard formats are listed in the reference section. It is also possible to define new formats, but this an advanced concept which requires an understanding of some of the internals of the parsing mechanism, explained in Section 58.5, and the usage of a rather low-level command, explained in Section 58.6.

58.2 Concepts

For the purposes of this section, let call a *data format* some standardized way of writing down a list of data points. A simple example of a data format is the CSV format (the acronym stands for *comma separated values*), where each line contains a data point, specified by values separated by commas. A different format is the *key-value format*, where data points are specified by lists of key-value pairs. A far more complex format is the PDB-format used by the protein database to describe molecules.

The data visualization system does not use any specific format. Instead, whenever data is read by the data visualization system, you must specify a format parser (or it is chosen automatically for you). It is the job of the parser to read (parse) the data lines and to turn them into data points, that is, to setup appropriate subkeys of /data point/.

To give a concrete example, suppose a file contains the following lines:

```
x, y, z
0, 0, 0
1, 1, 0
1, 1, 0.5
0, 1, 0.5
```

This file is in the CSV-format. This format can be read by the table parser (which is called thus, rather than "csv," since it can also read files in which the columns are separated by, say, a semicolon or a space). The table format will then read the data and for each line of the data, except for the headline of course, it will produce one data point. For instance, for the last data point the key /data point/x will be set to 0, the key /data point/y will be set to 1, and the key /data point/z will be set to 0.5.

All parsers are basically line-oriented. This means that, normally, each line in the input data should contain one data point. This rule may not always apply, for instance empty lines are typically ignored and sometimes a data point may span several lines, but deviating from this "one data point per line" rule makes parsers harder to program.

58.3 Reference: Build-In Formats

The following format is the default format, when no format=... is specified.

Format table

This format is used to parse data that is formatted in the following manner: Basically, each line consists of values that are separated by a separator like a comma or a space. The values are stored in different attributes, that is, subkeys of /data point like /data point/x. In order to decide which attribute is chosen for a give value, the headline is important. This is the first non-empty line of a table. It is formatted in the same way as normal data lines (value separated by the separator), but the meaning of the values is different: The first value in the headline is the name of the attribute where the first values in the following lines should go each time. Similarly, the second value in the headline is the name of the attribute for the second values in the following lines, and so on.

A simple example is the following:

```
angle, radius
0, 1
45, 2
90, 3
135, 4
```

The headline states that the values in the first column should be stored in the angle attribute (/data point/angle to be precise) and that the values in the second column should be stored in the radius attribute. There are four data points in this data set.

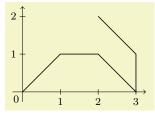
The format will tolerate too few or too many values in a line. If there are less values in a line than in the headline, the last attributes will simply be empty. If there are more values in a line than in the headline, the values are stored in attributes called /data point/attribute $\langle column\ number \rangle$, where the first value of a line gets $\langle column\ number \rangle$ equal to 1 and so on.

The table format can be configured using the following options:

/pgf/data/separator=\langle character \rangle

(no default, initially,)

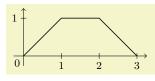
Use this key to change which character is used to separate values in the headline and in the data lines. To set the separator to a space, either set this key to an empty value or say separator=\space. Note that you must surround a comma by curly braces if you which to (re)set the separator character to a space.



$/pgf/data/headline = \langle headline \rangle$

(no default)

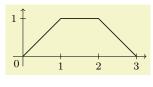
When this key is set to a non-empty value, the value of $\langle headline \rangle$ is used as the headline and the first line of the data is treated as a normal line rather than as a headline.



```
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as line]
  data [headline={x, y}] {
    0, 0
    1, 1
    2, 1
    3, 0
  };
\end{tikzpicture}
```

Format named

Basically, each line of the data must consist of a comma-separated sequence of attribute—values pairs like x=5, lo=500. This will cause the attribute /data point/x to be set to 5 and /data point/lo to be set to 500.

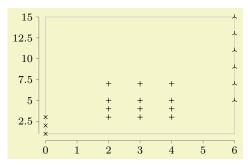


```
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as line]
  data [format=named] {
    x=0, y=0
    x=1, y=1
    x=2, y=1
    x=3, y=0
  };
\end{tikzpicture}
```

However, instead of just specifying a single value for an attribute as in x=5, you may also specify a whole set of values as in $x=\{1,2,3\}$. In this case, three data points will be created, one for each value

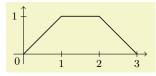
in the list. Indeed, the \foreach statement is used to iterate over the list of values, so you can write thighs like $x=\{1,\ldots,5\}$.

It is also permissible to specify lists of values for more than one attribute. In this case, a data point is created for each possible combination of values in the different lists:



Format TeX code

This format will simply execute each line of the data, each of which should contain some normal TeX code. Note that at the end of each line control returns to the format handler, so for instance the arguments of a command may not be spread over several lines. However, not each line needs to produce a data point.



```
\begin{tikzpicture}
\datavisualization [school book axes, visualize as line]
data [format=TeX code] {
   \pgfkeys{/data point/.cd,x=0, y=0} \pgfdatapoint
   \pgfkeys{/data point/.cd,x=1, y=1} \pgfdatapoint
   \pgfkeys{/data point/x=2} \pgfdatapoint
   \pgfkeyssetvalue{/data point/x}{3}
   \pgfkeyssetvalue{/data point/y}{0} \pgfdatapoint
   };
\end{tikzpicture}
```

58.4 Reference: Advanced Formats

\usetikzlibrary{datavisualization.formats.functions} % MEX and plain EX \usetikzlibrary[datavisualization.formats.functions] % ConEXt

This library defines the formats described in the following, which allow you to specify the data points indirectly, namely via a to-be-evaluated function.

Format function

This format allows you to specify a function that is then evaluated in order to create the desired data points. In other words, the data lines do not contain the data itself, but rather a functional description of the data.

The format used to specify the function works as follows: Each nonempty line of the data should contain at least one of either a *variable declaration* or a *function declaration*. A variable declaration signals that a certain attribute will range over a given interval. The function declarations will then, later, be evaluated for values inside this interval. The syntax for a variable declaration is one of the following:

```
1. var \langle variable \rangle : interval[\langle low \rangle: \langle high \rangle] samples \langle number \rangle;
2. var \langle variable \rangle : interval[\langle low \rangle: \langle high \rangle] step \langle step \rangle;
3. var \langle variable \rangle : {\langle values \rangle};
```

In the first case, if the optional samples part is missing, the number of samples is taken from the value stored in the following key:

```
/pgf/data/samples = \langle number \rangle (no default, initially 25)
```

Sets the number of samples to be used when no sample number is specified.

The meaning of declaring a variable declaration to range over an **interval** is that the attribute named $\langle variable \rangle$, that is, the key /data point/ $\langle variable \rangle$, will range over the interval [$\langle low \rangle, \langle high \rangle$]. If the number of samples is given (directly or indirectly), the interval is evenly divided into $\langle number \rangle$ many points and the attribute is set to each of these values. Similarly, when a $\langle step \rangle$ is specified, this stepping is used to increase $\langle low \rangle$ iteratively up to the largest value that is still less or equal to $\langle high \rangle$.

The meaning of declaring a variable using a list of $\langle values \rangle$ is that the variable will simply iterate over the values using \foreach.

You can specify more than one variable. In this case, each variable is varied independently of the other variables. For instance, if you declare an x-variable to range over the interval [0,1] in 25 steps and you also declare a y-variable to range over the same interval, you get a total of 625 value pairs.

The variable declarations specify which (input) variables will take which values. It is the job of the function declarations to specify how some additional attributes are to be computed. The syntax of a function declaration is as follows:

```
func \langle attribute \rangle = \langle expression \rangle;
```

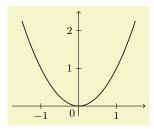
The meaning of such a declaration is the following: For each setting of the input variables (the variables specified using the var declaration), evaluate the $\langle expression \rangle$ using the standard mathematical parser of TikZ. The resulting value is then stored in /data point/ $\langle attribute \rangle$.

Inside $\langle expression \rangle$ you can reference data point attributes using the following command, which is only defined inside such an expression:

```
\vert {\vert variable}
```

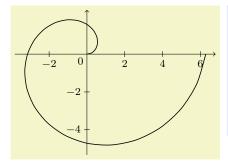
This expands to the current value of the key $/data\ point/\langle variable \rangle$.

There can be multiple function declarations in a single data specification. In this case, all of these functions will be evaluated for each setting of input variables.



```
\tikz
  \datavisualization [school book axes, visualize as smooth line]
  data [format=function] {
    var x : interval [-1.5:1.5];

    func y = \value x * \value x;
};
```



```
\tikz \datavisualization [
   school book axes,
   all axes={unit length=5mm, ticks={step=2}},
   visualize as smooth line]
data [format=function] {
   var t : interval [0:2*pi];

   func x = \value t * cos(\value t r);
   func y = \value t * sin(\value t r);
};
```

```
0.15

0.10

0.05

0.00

-0.05

-0.10

-0.15

0° 10° 20° 30° 40° 50°
```

58.5 Advanced: The Data Parsing Process

Whenever data is fed to the data visualization system, it will be handled by the \pgfdata command, declared in the datavisualization module. The command is both used to parse data stored in external sources (that is, in external files or which is produced on the fly by calling an external command) as well as data given inline. A data format does not need to know whether data comes from a file or is given inline, the \pgfdata command will take care of this.

Since TEX will always read files in a line-wise fashion, data is always fed to data format parsers in such a fashion. Thus, even it would make more sense for a format to ignore line-breaks, the parser must still handle data given line-by-line.

Let us now have a look at how \pgfdata works.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabula
```

This command is used to feed data to the visualization pipeline. This command can only be used when a data visualization object has been properly setup, see Section 57.

Basic options. The \pgfdata command may be followed by $\langle options \rangle$, which are executed with the path \rangle pgf/data/. Depending on these options, the $\langle options \rangle$ may either be followed by $\langle inline\ data \rangle$ or, alternatively, no $\langle inline\ data \rangle$ is present and the data is read from an external source.

The first important option is $\langle source \rangle$, which governs which of these two alternatives applies:

```
/pgf/data/read from file=\langle filename \rangle
```

(no default, initially empty)

If you set the read from file attribute to a non-empty $\langle filename \rangle$, the data will be read from this file. In this case, no $\langle inline\ data \rangle$ may be present, not even empty curly braces should be provided. If read from file is empty, the data must directly follow as $\langle inline\ data \rangle$.

```
% Data is read from two external files:
\pgfdata[format=table, read from file=file1.csv]
\pgfdata[format=table, read from file=file2.csv]
```

```
% Data is given inline:
\pgfdata[format=table]
{
    x, y
    1, 2
    2, 3
}
```

/pgf/data/inline (no value)

This is a shorthand file read from file={}. You can add this to make it clear(er) to the reader that data follows inline.

The second important key is format, which is used to specify the data format:

```
/pgf/data/format = \langle format \rangle
```

(no default, initially table)

Use this key to locally set the format used for parsing the data. The $\langle format \rangle$ must be a format that has been previously declared using the \pgfdeclaredataformat command. See the reference section for a list of the predefined formats.

In case all your data is in a certain format, you may wish to generally set the above key somewhere at the beginning of your file. Alternatively, you can use the following style to setup the format key and possibly further keys concerning the data format:

/pgf/every data

(style, no value)

This style is executed by \pgfdata before the $\langle options \rangle$ are parsed.

Note that the path of this key is just /pgf/, not /pgf/data/. Also note that TikZ internally sets the value of this key up in such a way that the keys /tikz/every data and also /tikz/data visualization/every data are executed. The bottom line of this is that when using TikZ, you should not set this key directly, set /tikz/every data instead.

Gathering of the data. Once the data format and the source have been decided upon, the data is "gathered." During this phase the data is not actually parsed in detail, but just gathered so that it can later be parsed during the visualization. There are two different ways in which the data is gathered:

- In case you have specified an external source, the data visualization object is told (by means of invoking the add data method) that it should (later) read data from the file specified by the source key using the format specified by the format key. The file is not read at this point, but only later during the actual visualization.
- Otherwise, namely when data is given inline, depending on which format is used, some catcodes
 get changed. This is necessary since TEX's special characters are often not-so-special in a certain
 format.

Independently of the format, the end-of-line character (carriage return) is made an active character. Finally, the $\langle inline\ data \rangle$ is then read as a normal argument and the data visualization object is told that later on it should parse this data using the given format parser. Note that in this case the data visualization object must store the whole data internally.

In both cases the "data visualization object" is the object stored in the /pgf/data visualization/obj key.

Parsing of the data. During the actual data visualization, all code that has been added to the data visualization object by means of the add data method is executed several times. It is the job of this code to call the \pgfdatapoint method for all data points present in the data.

When the \pgfdata method calls add data, the code that is passed to the data visualization object is just a call to internal macros of \pgfdata, which are able to parse the data stored in an external file or in the inlined data. Independently of where the data is stored, these macros always do the following:

- 1. The catcodes are setup according to what the data format requires.
- 2. Format-specific startup code gets called, which can initialize internal variables of the parsing process. (The catcode changes are not part of the startup code since in order to read inline data \pgfdata must be able to setup to temporarily setup the catcodes needed later on by the parsers, but since no reading is to be done, no startup code should be called at this point.)
- 3. For each line of the data a format-specific code handler, which depends on the data format, is called. This handler gets the current line as input and should call \pgfdatapoint once for each data point that is encoded by this line (a line might define multiple data points or none at all). Empty lines are handled by special format-specific code.
- 4. At the end, format-specific end code is executed.

For an example of how this works, see the description of the \pgfdeclaredataformat command.

Data sets. There are three options that allow you to create *data sets*. Such a data set is essentially a macro that stores a pre-parsed set of data that can be used multiple times in subsequent visualizations (or even in the same visualization).

$/pgf/data/new set=\langle name \rangle$ (no default)

Creates an empty data set called $\langle name \rangle$. If a data set of the same name already exists, it is overwritten and made empty. Data sets are global.

$/pgf/data/store in set=\langle name \rangle$ (no default)

When this key is set to any non-empty $\langle name \rangle$ and if this $\langle name \rangle$ has previously been used with the new set key, then the following happens: For the current \pgfdata command, all parsed data is not passed to the rendering pipeline. Instead, the parsed data is appended to the data set $\langle name \rangle$. This includes all options parsed to the \pgfdata command, which is why neither this key nor the previous key should be passed as options to a \pgfdata command.

$/pgf/data/use set=\langle name \rangle$ (no default)

This works similar to read from file. When this key is used with a $\protect\operatorname{pgfdata}$ command, no inline data may follow. Instead, the data stored in the data set $\langle name \rangle$ is used.

58.6 Advanced: Defining New Formats

In order to define a new data format you can use the following command, which is basic layer command defined in the module datavisualization:

```
\label{line code} $$ \operatorname{code}(\operatorname{code}) {\langle \operatorname{catcode} \ \operatorname{code} \rangle} {\langle \operatorname{catcode} \ \operatorname{code} \rangle} {\langle \operatorname{code} \ \rangle}
```

This command defines a new data format called $\langle format \ name \rangle$, which can subsequently be used in the $\protect\ pgfdata$ command. (The TikZ's data maps directly to $\protect\ pgfdata$, so the following applies to TikZ as well.)

As explained in the description of the $\protect\operatorname{pgfdata}$ command, when data is being parsed that is formatted according to $\langle format\ name \rangle$, the following happens:

- 1. The $\langle catcode\ code \rangle$ is executed. This code should just contain catcode changes. The $\langle catcode\ code \rangle$ will also be executed when inline data is read.
- 2. Next, the $\langle startup\ code \rangle$ is executed.
- 3. Next, for each non-empty line of the data, the line is passed to a macro whose argument list is given by \(\lambda line arguments \)\) and whose body is given by \(\lambda line code \)\). The idea is that you can use TEX's powerful pattern matching capabilities to parse the non-empty lines. See also the below example.
- 4. Empty lines are not processed by the $\langle line\ code \rangle$, but rather by the $\langle empty\ line\ code \rangle$. Typically, empty lines can simply be ignored and in this case you can let this parameter be empty.
- 5. At the end of the data, the $\langle end \ code \rangle$ is executed.

As an example, let us now define a simple data format for reading files formatted in the following manner: Each line should contain a coordinate pair as in (1.2,3.2), so two numbers separated by a comma and surrounded by parentheses. To make things more interesting, suppose that the hash mark symbol can be used to indicate comments. Here is an example of some data given in this format:

```
# This is some data formatted according to the "coordinates" format
(0,0)
(0.5,0.25)
(1,1)
(1.5,2.25)
(2,4)
```

A format parser for this format could be defined as follows:

```
\pgfdeclaredataformat{coordinates}
\% First comes the catcode argument. We turn the hash mark into a comment character.
{\catcode'\#=14\relax}
% Second comes the startup code. Since we do not need to setup things, we can leave
\% it empty. Note that we could also set it to something like \begingroup, provided we
% put an \endgroup in the end code
{}
% Now comes the arguments for non-empty lines. Well, these should be of the form
% (#1,#2), so we specify that:
{(#1,#2)}
% Now we must do something with a line of this form. We store the #1 argument in
  /data point/x and #2 in /data point/y. Then we call \pgfdatapoint to create a data point.
  \pgfkeyssetvalue{/data point/x}{#1}
  \pgfkeyssetvalue{/data point/y}{#2}
  \pgfdatapoint
  We ignore empty lines:
{}
% And we also have no end-of-line code.
```

This format could now be used as follows:

```
\begin{tikzpicture}
  \datavisualization[school book axes, visualize as smooth line]
  data [format=coordinates] {
    # This is some data formatted according
    # to the "coordinates" format
    (0,0)
    (0.5,0.25)
    (1,1)
    (1.5,2.25)
    (2,4)
  };
\end{tikzpicture}
```

59 Axes

59.1 Overview

When a data point is visualized, the most obvious way of creating a visual representation of its many attributes is to vary where the data point is shown. The data visualization system uses axes to turn data point attributes into positions on a page. The simplest – and most common – use of axes is to vary the horizontal position of data points according to one attribute and to vary the vertical position according to another attribute. In contrast, in a polar plot one attribute dictates the distance of the data point from the origin and another attribute describes the angle. From the data visualization engine's point of view, in both cases two axes are involved.

In addition to specifying how the value of a certain attribute is converted into a displacement on the page, an axis is also typically (but not always) visualized ("drawn") somewhere on the page. In this case, it is also customary to add a visual representation on this axis of which attribute values correspond to which positions on the page – something commonly known as *ticks*. Similar to ticks, *grid lines* also indicate positions where a certain attribute has a certain value, but instead of just indicating a single position on an axis, a grid line goes through all points that share an attribute value.

In the following, in Section 59.2 we first have a look at how axes can be defined and configured. As you will see, a lot of powerful configurations are available, but you will rarely define and configure an axis from scratch. Rather, it is more common to use a preconfigured axis instead. Section 59.3 introduces axis systems, which are predefined bundles of axes. You can define your own axis systems, but, again, in most cases it will suffice to just use one of the many preconfigured axis systems and use a few options to configure it so that it fits your need. Section 59.4 explains how ticks and grid lines can be configured. Again, several layers of options allow you to configure the way ticks look and where they are placed in great detail.

This section documents the standard axis systems that are always available. For polar axis systems, a special library needs to be loaded, which is documented in Section ??.

59.2 Basic Configuration of Axes

Inside the data visualization system, an *axis* is roughly a "systematic, named way of mapping an attribute to a position on a page". For instance, the classical "x-axis" is the "systematic way of mapping the value of the x attribute of data points to a horizontal position on the page". An axis is not its visual representation (such as the horizontal line with the ticks drawn to represent the x-axis), but a visual representation can be created once an axis has been defined.

The transformation of an attribute value (such as the value 1000000000 for the x attribute) to a specific displacement of the corresponding data point on the page involves two steps:

- 1. First, the range of possible values such as $[-5.6 \cdot 10^{12}, 7.8 \cdot 10^{12}]$ must be mapped to a "reasonable" interval such as [0cm, 5cm] or $[0^{\circ}, 180^{\circ}]$. TikZ's drawing routines will only be able to cope with values from such a "reasonable" interval.
- 2. Second, the values from the reasonable interval must be mapped to a transformation.

The first step is always the same for all axes, while the second requires different strategies. For this reason, the command new axis base is used to create a "basic" axis that has a "scaling mapper", whose job it is to map the range of values of a specific attribute to a reasonable interval, but such a basic axis does not define an actual transformation object. For this second step, additional objects such as a linear transformer need to be created separately.

59.2.1 Usage

To create an axis, the key new axis base is used first. Since this key does not create a transformation object, users typically do not use this key directly. Rather, it is used internally by other keys that create "real" axes. These keys are listed in Section 59.2.8.

/tikz/data visualization/new axis base= $\langle axis \ name \rangle$

(no default)

This key defines a new axis for the current data visualization called $\langle name \rangle$. This has two effects:

- 1. A so called *scaling mapper* is created that will monitor a certain attribute, rescale it, and map it to another attribute. (This will be explained in detail in a moment.)
- 2. The (axis name) is made available as a key that can be used to configure the axis:

```
/\text{tikz/data visualization}/\langle axis\ name \rangle = \langle options \rangle (no default)
```

This key becomes available once new axis base=metaaxis name has been called. It will execute the \(options \)\) with the path prefix \/ tikz/data visualization/axis options.

```
[new axis base=my axis,
my axis={attribute=some attribute}]
```

3. The $\langle axis\ name \rangle$ becomes part of the current set of axes. This set can be accessed through the following key:

```
/tikz/data visualization/all axes=(options) (no default)
```

This key passes the $\langle options \rangle$ to all axes inside the current scope, just as if you had written $\langle some\ axis\ name \rangle = \langle options \rangle$ for each $\langle some\ axis\ name \rangle$ in the current scope, including the just-created name $\langle axis\ name \rangle$.

There are many $\langle options \rangle$ that can be passed to a newly created axis. They are explained in the rest of this section.

Note the new axis base does *not* cause attributes to be mapped to positions on a page. Rather, special keys like new Cartesian axis first use new axis base to create an axis and then create an internal object that performs a linear mapping of the attribute to positions along a vectors.

59.2.2 The Axis Attribute

The first main job of an axis is to map the different values of some attribute to a reasonable interval. To achieve this, the following options are important (recall that these options are passed to the key whose name is the name of the axis):

```
/tikz/data visualization/axis options/attribute=\langle attribute \rangle (no default)
```

Specifies that the axis is used to transform the data points according the different values of the key /data point/(attribute). For instance, when we create a classical two-dimensional Cartesian coordinate system, then there are two axes called x axis and y axis that monitor the values of the attributes /data point/x and /data point/y, respectively:

```
[new axis base=x axis,
new axis base=y axis,
x axis={attribute=x},
y axis={attribute=y}]
```

In another example, we also create an x axis and a y axis. However, this time, we want to plot the values of the /data point/time attribute on the x-axis and, say, the value of the height attribute on the y-axis:

```
[new axis base=x axis,
new axis base=y axis,
x axis={attribute=time},
y axis={attribute=height}]
```

During the data visualization, the $\langle attribute \rangle$ will be "monitored" during the survey phase. This means that for each data point, the current value of $/data\ point/\langle attribute \rangle$ is examined and the minimum value of all of these values as well as the maximum value is recorded internally. Note that this works even when very large numbers like 100000000000 are involved.

Here is a real-life example. The scientific axes create two axes, called x axis and y axis, respectively.

59.2.3 The Axis Attribute Range Interval

Once an attribute has been specified for an axis, the data visualization engine will start monitoring this value. This means that before anything actual visualization is done, a "survey phase" is used to determine the range of values encountered for the attribute for all data points. This range of values results in what is called the *attribute range interval*. Its minimum is the smallest value encountered in the data and its maximum is the largest value.

Even though the attribute range interval is computed automatically and even though you typically do not need to worry about it, there are some situations where you may wish to set or enlarge the attribute range interval:

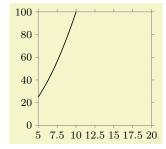
- You may wish to start the interval with 0, even though the range of values contains only positive values.
- You may wish to slightly enlarge the interval so that, say, the maximum is some "nice" value like 100 or 60.

The following keys can be used to influence the size of the attribute range interval:

```
/\text{tikz/data visualization/axis options/include value=} \langle \textit{list of value} \rangle (no default)
```

This key "fakes" data points for which the attribute's values are in the comma-separated $\langle list\ of\ values \rangle$. For instance, when you write include value=0, then the attribute range interval is guaranteed to contain 0 – even if the actual data points are all positive or all negative.

```
100
90
80
70
60
50
40
30
5 6 7 8 9 10
```



/tikz/data visualization/axis options/min value= $\langle value \rangle$

(no default)

This key allows you to simply set the minimum value, regardless of which values are present in the actual data. This key should be used with care: If there are data points for which the attribute's value is less than $\langle value \rangle$, they will still be depicted, but typically outside the normal visualization area. Usually, saying include value= $\langle value \rangle$ will achieve the same as saying min value= $\langle value \rangle$, but with less danger of creating ill-formed visualizations.

```
/tikz/data visualization/axis options/max value=\langle value \rangle
```

(no default)

Works like min value.

59.2.4 Scaling: The General Mechanism

The above key allows us specify which attribute should be "monitored". The next key is used to specify what should happen with the observed values.

```
/tikz/data visualization/axis options/scaling=\(scaling spec \)
```

(no default)

The $\langle scaling\ spec \rangle$ must have the following form:

```
\langle s_1 \rangle at \langle t_1 \rangle and \langle s_2 \rangle at \langle t_2 \rangle
```

This means that monitored values in the interval $[s_1, s_2]$ should be mapped to values the "reasonable" interval $[t_1, t_2]$, instead. For instance, we might write

```
[y axis = {scaling = 1900 at 0cm and 2000 at 5cm}]
```

in order to map dates between 1900 and 2000 to the dimension interval [0cm, 5cm].

```
\tikz \datavisualization
2,000
                               Scientific axes.
                                x axis={attribute=people, length=2.5cm, ticks=few},
                                y axis={attribute=year, scaling=1900 at 0cm and 2000 at 5cm},
1,980
                                visualize as scatter]
1,960
                                year, people
                                1900, 100
                                1910, 200
                                1950, 200
1,940
                                1960, 250
                                2000, 150
1,920
1,900
         150
               200
                     250
    100
```

So much for the basic idea. Let us now have a detailed look at what happens.

Number format and the min and max keywords. The source values s_1 and s_2 are typically just numbers like 3.14 or 1000000000. However, as described in Section 57.2, you can also specify expressions like (pi/2), provided that (currently) you put them in parentheses.

Instead of a number, you may alternatively also use the two key words min and max for s_1 and/or s_2 . In this case, min evaluates to the smallest value observed for the attribute in the data, symmetrically max evaluates to the largest values. For instance, in the above example with the year attribute ranging from 1900 to 2000, the keyword min would stand for 1900 and max for 2000. Similarly, for the people attribute min stands for 100 and max for 250. Note that min and max can only be used for s_1 and s_2 , not for t_1 and t_2 .

A typical use of the min and max keywords is to say

```
scaling = min at 0cm and max at 5cm
```

to map the complete range of values into an interval of length of 5cm.

The interval $[s_1, s_2]$ need not contain all values that the $\langle attribute \rangle$ may attain. It is permissible that values are less than s_1 or more than s_2 .

Linear transformation of the attribute. As indicated earlier, the main job of an axis is to map values from a "large" interval $[s_1, s_2]$ to a more reasonable interval $[t_1, t_2]$. Suppose that for the current data point the value of the key /data point/ $\langle attribute \rangle$ is the number v. In the simplest case, the following happens: A new value v' is computed so that $v' = t_1$ when $v = s_1$ and $v' = t_2$ when $v = s_2$ and v' is some value in between t_1 and t_2 then v is some value in between s_1 and s_2 . (Formally, in this basic case $v' = t_1 + (v - s_1) \frac{t_2 - t_1}{s_2 - s_1}$.)

Once v' has been computed, it is stored in the key /data point/ $\langle attribute \rangle$ /scaled. Thus, the "reasonable" value v' does not replace the value of the attribute, but it is placed in a different key. This means that both the original value and the more "scaled" values are available when the data point is visualized.

As an example, suppose you have written

```
[x axis = {attribute = x, scaling=1000 at 20 and 2000 at 30}]
```

Now suppose that /data point/x equals 1200 for a data point. Then the key /data point/x/scaled will be set to 22 when the data point is being visualized.

Nonlinear transformations of the attribute. By default, the transformation of $[s_1, s_2]$ to $[t_1, t_2]$ is the linear transformation described above. However, in some case you may be interested in a different

kind of transformation: For example, in a logarithmic plot, values of an attribute may range between, say, 1 and 1000 and we want an axis of length 3cm. So, we would write

```
[x axis = {attribute = x, scaling=1 at 0cm and 1000 at 3cm}]
```

Indeed, 1 will now be mapped to position 0cm and 1000 will be mapped to position 3cm. Now, the value 10 will be mapped to approximately 0.03cm because it is (almost) at one percent between 1 and 1000. However, in a logarithmic plot we actually want 10 to be mapped to the position 1cm rather than 0.03cm and we want 100 to be mapped to the position 2cm. Such a mapping a *nonlinear* mapping between the intervals.

In order to achieve such a nonlinear mapping, the function key can be used, whose syntax is described in a moment. The effect of this key is to specify a function $f: \mathbb{R} \to \mathbb{R}$ like, say, the logarithm function. When such a function is specified, the mapping of v to v' is computed as follows:

$$v' = t_1 + (f(s_2) - f(v)) \frac{t_2 - t_1}{f(s_2) - f(s_1)}.$$

The syntax of the function key is described next, but you typically will not call this key directly. Rather, you will use a key like logarithmic that installs appropriate code for the function key for you.

```
/\text{tikz/data visualization/axis options/function=}(code) (no default)
```

The $\langle code \rangle$ should specify a function f that is applied during the transformation of the interval $[s_1, s_2]$ to the interval $[t_1, t_2]$ in the following way: When the $\langle code \rangle$ is called, the macro \pgfvalue will have been set to an internal representation of the to-be-transformed value v. You can then call the commands of the math-micro-kernel of the data visualization system, see Section 62.4, to compute a new value. This new value must once more be stored in \pgfvalue.

The most common use of this key is to say

```
some axis={function=\pgfdvmathln{\pgfvalue}{\pgfvalue}}
```

This specifies that the function f is the logarithm function.

Another possibility might be to use the square-root function, instead:

Default scaling. When no scaling is specified, it may seem natural to use [0,1] both as the source and the target interval. However, this would not work when the logarithm function is used as transformations: In this case the logarithm of zero would be computed, leading to an error. Indeed, for a logarithmic axis it is far more natural to use [1,10] as the source interval and [0,1] as the target interval.

For these reasons, the default value for the scaling that is used when no value is specified explicitly can be set using a special key:

```
/\text{tikz/data visualization/axis options/scaling/default} = \langle text \rangle (no default)
```

The $\langle text \rangle$ is used as scaling whenever no other scaling is specified. This key is mainly used when a transformation function is set using function; normally, you will not use this key directly.

Most of the time, you will not use neither the scaling nor the function key directly, but rather you will use one of the following predefined styles documented in the following.

59.2.5 Scaling: Logarithmic Axes

/tikz/data visualization/axis options/logarithmic

(no value)

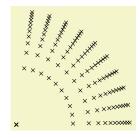
When this key is used with an axis, three things happen:

- 1. The transformation function of the axis is setup to the logarithm.
- 2. The strategy for automatically generating ticks and grid lines is set to the exponential strategy, see Section 59.4.13 for details.
- 3. The default scaling is setup sensibly.

All told, to turn an axis into a logarithmic axis, you just need to add this option to the axis.

Note that this will work with any axis, including, say, the degrees on a polar axis:

```
\tikz \datavisualization
  [new polar axes,
    angle axis={logarithmic, scaling=1 at 0 and 90 at 90},
    radius axis={scaling=0 at 0cm and 100 at 3cm},
    visualize as scatter]
data [format=named] {
    angle={1,10,...,90}, radius={1,10,...,100}
};
```



```
\tikz \datavisualization
  [new polar axes,
    angle axis={degrees},
    radius axis={logarithmic, scaling=1 at 0cm and 100 at 3cm},
    visualize as scatter]
data [format=named] {
    angle={1,10,...,90}, radius={1,10,...,100}
};
```

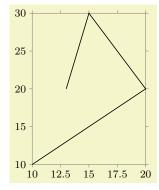
59.2.6 Scaling: Setting the Length or Unit Length

/tikz/data visualization/axis options/length= $\langle dimension \rangle$

(no default)

Sets scaling to min at 0cm and max at $\langle dimension \rangle$. The effect is that the range of all values of the axis's attribute will be mapped to an interval of exact length $\langle dimension \rangle$.

```
30
25
20
15
10
10 12.5 15 17.5 20
```

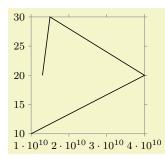


/tikz/data visualization/axis options/unit length= $\langle dimension \rangle$ per $\langle number \rangle$ units (no default)

Sets scaling to 0 at 0cm and 1 at $\langle dimension \rangle$. In other words, this key allows you to specify how long a single unit should be. This key is particularly useful when you wish to ensure that the same scaling is used across multiple axes or pictures.

```
30
25
20
15
10
20 30 40
```

The optional per $\langle number \rangle$ units allows you to apply more drastic scaling. Suppose that you want to plot a graph where one billion corresponds to one centimeter. Then the unit length would be need to be set to a hundredth of a nanometer – much too small for T_EX to handle as a dimension. In this case, you can write unit length=1cm per 1000000000 units:

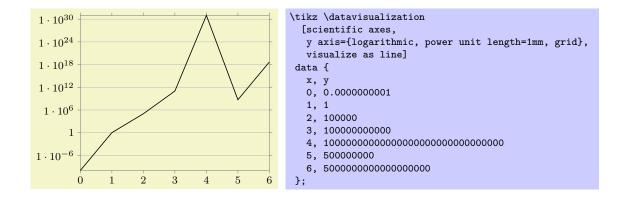


```
\tikz \datavisualization
[scientific axes,
   x axis={unit length=1mm per 1000000000 units, ticks=few},
   visualize as line]
data {
   x, y
   10000000000, 10
   40000000000, 20
   15000000000, 30
   13000000000, 20
};
```

(no default)

/tikz/data visualization/axis options/power unit length= $\langle dimension \rangle$

This key is used in conjunction with the logarithmic setting. It cases the scaling to be set to 1 at 0cm and 10 at $\langle dimension \rangle$. This causes a "power unit," that is, one power of ten in a logarithmic plot, to get a length of $\langle dimension \rangle$. Again, this key is useful for ensuring that the same scaling is used across multiple axes or pictures.



59.2.7 Axis Label

An axis can have a *label*, which is a textual representation of the attribute according to which the axis varies the position of the page. You can set the attribute using the following key:

```
/tikz/data visualization/axis options/label=\{[\langle options \rangle] \langle text \rangle\} (default axis's label in math mode)
```

This key sets the label of an axis to $\langle text \rangle$. This text will typically be placed inside a node and the $\langle options \rangle$ can be used to further configure the way this node is rendered. The $\langle options \rangle$ will be executed with the path prefix /tikz/data visualization/, so you need to say node style to configure the styling of a node, see Section 59.4.7.

```
25
                        \tikz \datavisualization [
                            scientific axes.
20
                            x axis = {label, length=2.5cm},
                            y axis = {label={[node style={fill=blue!20}]{$x^2$}}},
15
                            visualize as smooth line]
                         data [format=function] {
10
                           var x : interval [-3:5];
                           func y = \value x * \value x;
 5
 0
    -2
             2
         0
                  4
```

Note that using the label key does not actually cause a node to be created, because it is somewhat unclear where the label should be placed. Instead, the visualize label key is used (typically internally by an axis system) to show the label at some sensible position. This key is documented in Section 59.5.5.

59.2.8 Reference: Axis Types

As explained earlier, when you use **new axis base** to create a new axis, a powerful scaling and attribute mapping mechanism is installed, but no mapping of values to positions on the page is performed. For this, a transformation object must be installed. The following keys take care of this for you. Note, however, that even these keys do not cause a visual representation of the axis to be added to the visualization – this is the job of an axis system, see Section 59.3.

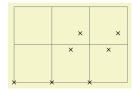
```
/tikz/data visualization/new Cartesian axis=\langle name \rangle (no default)
```

This key creates a new "Cartesian" axis, named $\langle name \rangle$. For such an axis, the (scaled) values of the axis's attribute are transformed into a displacement on the page along a straight line. The following key is used to configure in which "direction" the axis points:

Recall that an axis takes the values of an attribute and rescales them so that they fit into a "reasonable" interval $[t_1, t_2]$. Suppose that v' is the rescaled dimension in (TEX) points. Then when the data point is visualized, the coordinate system will be shifted by v' times the $\langle coordinate \rangle$.

As an example, suppose that you have said scaling=0 and 10pt and 50 and 20pt. Then when the underlying attribute has the value 25, it will be mapped to a v' of 15 (because 25 lies in the middle of 0 and 50 and 15pt lies in the middle of 10pt and 20pt). This, in turn, causes the data point to be displaced by 15 times the $\langle coordinate \rangle$.

The bottom line is that the $\langle coordinate \rangle$ should usually denote a point that is at distance 1pt from the origin and that points into the direction of the axis.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \datavisualization
    [new Cartesian axis=x axis, x axis={attribute=x},
    new Cartesian axis=y axis, y axis={attribute=y},
     x axis={unit vector=(0:1pt)},
    y axis={unit vector=(60:1pt)},
    visualize as scatter]
  data {
   х, у
   0,0
   1, 0
   2. 0
    1, 1
   2. 1
    1, 1.5
   2, 1.5
 }:
\end{tikzpicture}
```

59.3 Axis Systems

An axis system is, as the name suggests, a whole family of axes that act in concert. For example, in the "standard" axis system there is a horizontal axis called the x-axis that monitors the x attribute (by default, you can change this easily) and a vertical axis called the y-axis. Furthermore, a certain number of ticks are added and labels are placed at sensible positions.

59.3.1 Usage

Using an axis system is usually pretty easy: You just specify a key like scientific axes and the necessary axes get initialized with sensible default values. You can then start to modify these default values, if necessary.

First, you can (and should) set the attributes to which the difference axes refer. For instance, if the time attribute is plotted along the x-axis, you would write

```
x axis = {attribute = time}
```

Second, you may wish to modify the lengths of the axes. For this, you can use keys like length or further keys as described in the references later on.

Third, you may often wish to modify how many ticks and grid lines are shown. By default, no grid lines are shown, but you can say the following in order to cause grid lines to be shown:

```
all axes={grid}
```

Naturally, instead of all axes you can also specify a single axis, causing only grid lines to be shown for this axis. In order to change the number of ticks that are shown, you can say

```
all axes={ticks=few}
```

or also many instead of few or even none. Far more fine-grained control over the tick placement and rendering is possible, see Section 59.4 for details.

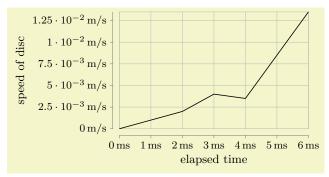
Fourth, consider adding units (like "cm" for centimeters or "m/s²" for acceleration) to your ticks:

```
x axis={ticks={tick unit=cm}}, y axis={ticks={tick unit=m/s^2}}
```

Finally, consider adding labels to your axes. For this, use the label option:

```
x axes={time $t$ (ms)}, y axis={distance $d$ (mm)}
```

Here is an example that employs most of the above features:



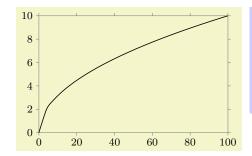
```
\tikz \datavisualization [
  scientific axes=clean.
  x axis={attribute=time, ticks={tick unit=ms},
    label={elapsed time}},
  y axis={attribute=v, ticks={tick unit=m/s},
    label={speed of disc}},
  all axes=grid,
  visualize as line]
data {
  time. v
  0,0
  1, 0.001
  2, 0.002
  3, 0.004
  4, 0.0035
  5, 0.0085
  6, 0.0135
};
```

59.3.2 Reference: Scientific Axis Systems

```
/tikz/data visualization/scientific axes=\langle options \rangle
```

(no default)

This key installs a two-dimensional coordinate system based on the attributes /data point/x and /data point/y.



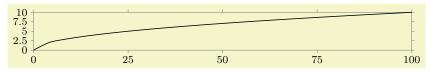
This axis system is usually a good choice to depict "arbitrary two dimensional data." Because the axes are automatically scaled, you do not need to worry about how large or small the values will be. The name scientific axes is intended to indicate that this axis system is often used in scientific publications.

You can use the $\langle options \rangle$ to fine tune the axis system. The $\langle options \rangle$ will be executed with the following path prefix:

/tikz/data visualization/scientific axes

All keys with this prefix can thus be passed as $\langle options \rangle$.

This axis system will always distort the relative magnitudes of the units on the two axis. If you wish the units on both axes to be equal, consider directly specifying the unit length "by hand":



The scientific axes have the following properties:

• The x-values are surveyed and the x-axis is then scaled and shifted so that it has the length specified by the following key.

 \t tikz/data visualization/scientific axes/width= \t dimension \t (no default, initially 5cm)

The minimum value is at the left end of the axis and at the canvas origin. The maximum value is at the right end of the axis.

• The y-values are surveyed and the y-axis is then scaled so that is has the length specified by the following key.

```
\ttikz/data visualization/scientific axes/height=\tdimension\t (no default)
```

By default, the height is the golden ratio times the width.

The minimum value is at the bottom of the axis and at the canvas origin. The maximum value is at the top of the axis.

 \bullet Lines (forming a frame) are depicted at the minimum and maximum values of the axes in 50% black.

The following keys are executed by default as options: outer ticks and standard labels.

You can use the following style to overrule the defaults:

```
/tikz/data visualization/every scientific axes
```

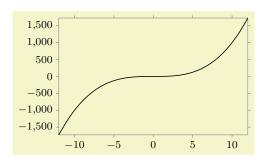
(style, no value)

The keys described in the following can be used to fine-tune the way the scientific axis system is rendered.

/tikz/data visualization/scientific axes/outer ticks

(no value)

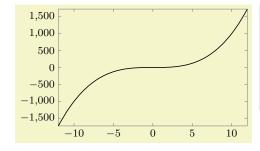
This causes the ticks to be drawn "on the outside" of the frame so that they interfere as little as possible with the data. It is the default.



/tikz/data visualization/scientific axes/inner ticks

(no value)

This axis system works like scientific axes, only the ticks are on the "inside" of the frame.



This axis system is also common in publications, but the ticks tend to interfere with marks if they are near to the border as can be seen in the following example:

```
0.8

0.6

0.4

0.2

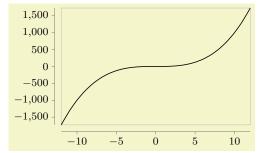
0 0.5 1 1.5 2
```

```
\begin{tikzpicture}
  \datavisualization [scientific axes={inner ticks, width=3.2cm},
                      style sheet=cross marks,
                      visualize as scatter/.list={a,b}]
    data [set=a] {
     х, у
     0, 0
     1, 1
     0.5, 0.5
     2, 1
   data [set=b] {
     х, у
     0.05, 0
     1.5, 1
     0.5, 0.75
     2, 0.5
   };
\end{tikzpicture}
```

/tikz/data visualization/scientific axes/clean

(no value)

The axes and the ticks are completely removed from the actual data, making this axis system especially useful for scatter plots, but also for most other scientific plots.



```
\tikz \datavisualization [
  scientific axes=clean,
  visualize as smooth line]
data [format=function] {
  var x : interval [-12:12];
  func y = \value x*\value x;
};
```

The distance of the axes from the actual plot is given by the padding of the axes.

For all scientific axis systems, different label placement strategies can be specified. They are discussed in the following.

/tikz/data visualization/scientific axes/standard labels

(no value)

As the name suggests, this is the standard placement strategy. The label of the x-axis is placed below the center of the x-axis, the label of the y-axis is rotated by 90° and placed left of the center of the y-axis.

```
\begin{array}{c}
0.15 \\
0.1 \\
5 \cdot 10^{-2} \\
0 \\
-5 \cdot 10^{-2} \\
-0.15 \\
-0.15
\end{array}

\begin{array}{c}
-10^{\circ} & -5^{\circ} & 0^{\circ} & 5^{\circ} & 10^{\circ} \\
\text{degree } d
\end{array}
```

```
\tikz \datavisualization
[scientific axes={clean, standard labels},
  visualize as smooth line,
  x axis={label=degree $d$,
    ticks={tick unit={}^\circ}},
  y axis={label=$\sin d$}]
data [format=function] {
  var x : interval [-10:10] samples 10;
  func y = sin(\value x);
};
```

/tikz/data visualization/scientific axes/upright labels

(no value)

Works like scientific axes standard labels, only the label of the y-axis is not rotated.

```
\begin{array}{c}
1.0000 \\
0.9975 \\
0.9950 \\
0.9900 \\
0.9875 \\
0.9850
\end{array}

\begin{array}{c}
0.9875 \\
0.9850
\end{array}

\begin{array}{c}
0^{\circ} \\
-10^{\circ} \\
-5^{\circ} \\
\end{array}

\begin{array}{c}
0^{\circ} \\
5^{\circ} \\
\end{array}

\begin{array}{c}
10^{\circ} \\
\end{array}

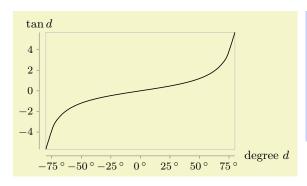
\begin{array}{c}
0^{\circ} \\
\end{array}

\begin{array}{c}
0^{\circ} \\
\end{array}
```

/tikz/data visualization/scientific axes/end labels

(no value)

Places the labels at the end of the x- and the y-axis, similar to the axis labels of a school book axis system.



```
\tikz \datavisualization [
   scientific axes={clean, end labels},
   visualize as smooth line,
   x axis={label=degree $d$,
      ticks={tick unit={}^\circ}},
   y axis={label=$\tan d$}]
data [format=function] {
   var x : interval [-80:80];
   func y = tan(\value x);
};
```

59.3.3 Reference: School Book Axis Systems

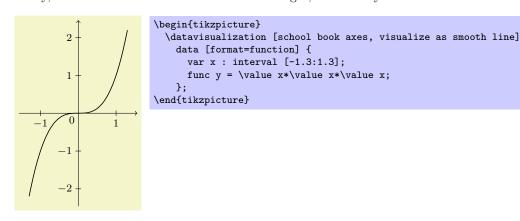
/tikz/data visualization/school book axes= $\langle options \rangle$

(no default)

This axis system is intended to "look like" the coordinate systems often used in school books: The axes are drawn in such a way that they intersect to origin. Furthermore, no automatic scaling is done to ensure that the lengths of units are the same in all directions.

This axis system must be used with care – it is nearly always necessary to specify the desired unit length by hand using the option unit length. If the magnitudes of the units on the two axes differ, different unit lengths typically need to be specified for the different axes.

Finally, if the data is "far removed" from the origin, this axis system will also "look bad."



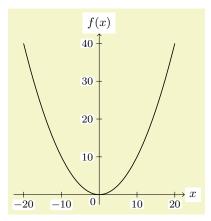
The stepping of the ticks is one unit by default. Using keys like ticks=some may help to give better steppings.

The $\langle options \rangle$ are executed with the key itself as path prefix. Thus, the following subkeys are permissible options:

/tikz/data visualization/school book axes/unit= $\langle value \rangle$

(no default)

Sets the scaling so that 1 cm corresponds to $\langle value \rangle$ units. At the same time, the stepping of the ticks will also be set to $\langle value \rangle$.

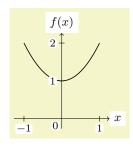


/tikz/data visualization/school book axes/standard labels

(no value)

This key makes the label of the x-axis appear at the right end of this axis and it makes the label of the y-axis appear at the top of the y-axis.

Currently, this is the only supported placement strategy for the school book axis system.



59.3.4 Advanced Reference: Underlying Cartesian Axis Systems

The axis systems described in the following are typically not used directly by the user. The systems setup directions for several axes in some sensible way, but they do not actually draw anything on these axes. For instance, the xy Cartesian creates two axes called x axis and y axis and makes the x-axis point right and the y-axis point up. In contrast, an axis system like scientific axes uses the axis system xy Cartesian internally and then proceeds to setup a lot of keys so that the axis lines are drawn, ticks and grid lines are drawn, and labels are placed at the correct positions.

/tikz/data visualization/xy Cartesian

(no value)

This axis system creates two axes called x axis and y axis that point right and up, respectively. By default, one unit is mapped to one cm.

```
\begin{tikzpicture}
  \datavisualization [xy Cartesian, visualize as smooth line]
  data [format=function] {
    var x : interval [-1.25:1.25];
    func y = \value x*\value x;
  };
  \end{tikzpicture}
```

```
/tikz/data visualization/xy axes=\langle options \rangle
```

(no default)

This key applies the $\langle options \rangle$ both to the x axis and the y axis.

/tikz/data visualization/xyz Cartesian cabinet

(no value)

This axis system works like xy Cartesian, only it additionally creates an axis called z axis that points left and down. For this axis, one unit corresponds to $\frac{1}{2} \sin 45^{\circ}$ cm. This is also known as a cabinet projection.

```
/tikz/data visualization/xyz axes=\langle options \rangle
```

(no default)

This key applies the $\langle options \rangle$ both to the x axis and the y axis.

/tikz/data visualization/uv Cartesian

(no value)

This axis system works like xy Cartesian, but it introduces two axes called u axis and v axis rather than the x axis and the y axis. The idea is that in addition to a "major" xy-coordinate system this is also a "smaller" or "minor" coordinate system in use for depicting, say, small vectors with respect to this second coordinate system.

```
/tikz/data visualization/uv axes=\langle options \rangle
```

(no default)

Applies the *(options)* to both the u axis and the y axis.

/tikz/data visualization/uvw Cartesian cabinet

(no value)

Like xyz Cartesian cabinet, but for the uvw-system.

```
/tikz/data visualization/uvw axes=\langle options \rangle
```

(no default)

Like xyz axes.

59.4 Ticks and Grids

59.4.1 Concepts

A *tick* is a small visual indication on an axis of the value of the axis's attribute at the position where the tick is shown. A tick may be accompanied additionally by a textual representation, but it need not. A *grid line* is similar to a tick, but it is not an indication on the axis, but rather a whole line that indicates all positions where the attribute has a certain value. Unlike ticks, grid lines (currently) are not accompanied by a textual representation.

Just as for axes, the data visualization system decouples the specification of which ticks are present *in principle* from where they are visualized. In the following, I describe how you specify which ticks and grid lines you would like to be drawn and how they should look like (their styling). The axis system of your choice will then visualize the ticks at a sensible position for the chosen system. For details on how to change where whole axis is shown along with its ticks, see Section 59.5.4.

Specifying which ticks you are interested in is done as follows: First, you use ticks key (or, for specifying which grid lines should be present, the grid key). This key takes several possible options, described in detail in the following, which have different effects:

1. Keys like step=10 or minor steps between steps cause a "semi-automatic" computation of possible steps. Here, you explicitly specify the stepping of steps, but the first stepping and their number are computed automatically according to the range of possible values for the attribute.

- 2. Keys like few, some, or many can be passed to ticks in order to have TikZ compute good tick positions automatically. This is usually what you want to happen, which is why most axis system will implicitly say ticks={some}.
- 3. Keys like at or also at provide "absolute control" over which ticks or grid lines are shown. For these keys, you can not only specify at what value a tick should be shown, but also its styling and also whether it is a major, minor, or subminor tick or grid line.

In the following, the main keys ticks and grids are documented first. Then the different kinds of ways of specifying where ticks or grid lines should be shown are explained.

59.4.2 The Main Options: Tick and Grid

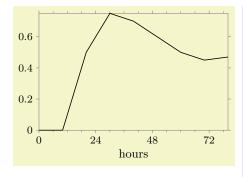
/tikz/data visualization/axis options/ticks=\langle options \rangle

(default some)

This key can be passed to an axis in order to configure which ticks are present for the axis. The possible $\langle options \rangle$ include, for instance, keys like step, which is used to specify a stepping for the ticks, but also keys like major or minor for specifying the positions of major and minor ticks in detail. The list of possible options is described in the rest of this section.

Note that the ticks option will only configure which ticks should be shown in principle. The actual rendering is done only when the visualize ticks key is used, documented in Section 59.5.4, which is typically done only internally by an axis system.

The $\langle options \rangle$ will be executed with the path prefix /tikz/data visualization/. When the ticks key is used multiple times for an axis, the $\langle options \rangle$ accumulate.



/tikz/data visualization/axis options/grid=(options)

(default at default ticks)

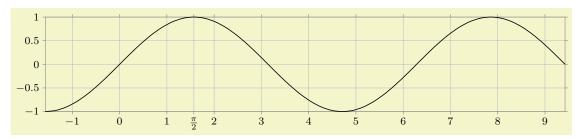
This key is similar to ticks, only it is used to configure where grid lines should be shown rather than ticks. In particular, the options that can be passed to the ticks key can also be passed to the grid key. Just like ticks, the $\langle options \rangle$ only specify which grid lines should be drawn in principle; it is the job of the visualize grid key to actually cause any grid lines to be shown.

If you do not specify any $\langle options \rangle$, the default text at default ticks is used. This option causes grid lines to be drawn at all positions where ticks are shown by default. Since this usually exactly what you would like to happen, most of the time you just need to all axes=grid to cause a grid to be shown.

/tikz/data visualization/axis options/ticks and grid=(options)

(no default)

This key passes the $\langle options \rangle$ to both the ticks key and also to the grid key. This is useful when you want to specify some special points explicitly where you wish a tick to be shown and also a grid line.



```
\tikz \datavisualization
[scientific axes,
  visualize as smooth line,
  all axes= {grid, unit length=1.25cm},
  y axis={ ticks=few },
  x axis={ ticks=many, ticks and grid={ major also at={(pi/2) as $\frac{\pii}{2}$}}]
  data [format=function] {
   var x : interval [-pi/2:3*pi] samples 50;
   func y = sin(\value x r);
};
```

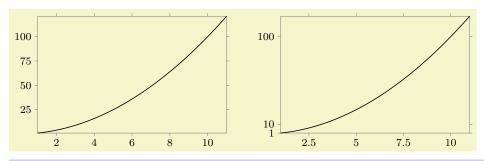
59.4.3 Semi-Automatic Computation of Tick and Grid Line Positions

Consider the following problem: The data visualization engine determines that in a plot the x-values vary between 17.4 and 34.5. In this case, we certainly do not want, say, ten ticks at exactly ten evenly spaced positions starting with 17.4 and ending with 34.5, because this would yield ticks at positions like 32.6. Ticks should be placed at "nice" positions like 20, 25, and 30.

Determining which positions are "nice" is somewhat difficult. In the above example, the positions 20, 25, and 30 are certainly nice, but only three ticks may be a bit few of them. Better might be the tick positions 17.5, 20, 22.5, through to 32.5. However, users might prefer even numbers over fractions like 2.5 as the stepping.

A tick placement strategy is a method of automatically deciding which positions are good for placing ticks. The data visualization engine comes with a number of predefined strategies, but you can also define new ones yourself. When the data visualization is requested to automatically determine "good" positions for the placement of ticks on an axis, it uses one of several possible basic strategies. These strategies differ dramatically in which tick positions they will choose: For a range of values between 5 and 1000, a linear steps strategy might place ticks at positions 100, 200, through to 1000, while an exponential steps strategy would prefer the tick positions 10, 100 and 1000. The exact number and values of the tick positions chosen by either strategy can be fine-tuned using additional options like step or about.

Here is an example of the different stepping chosen when one varies the tick placement strategy:



```
\begin{tikzpicture}
 \datavisualization [scientific axes, visualize as smooth line]
   data [format=function] {
     var x : interval [1:11];
     func y = \forall x \in x
\end{tikzpicture}
\qquad
\begin{tikzpicture}
 \datavisualization [scientific axes, visualize as smooth line,
   y axis={exponential steps},
   x axis={ticks={quarter about strategy}},
   data [format=function] {
     var x : interval [1:11];
     func y = \forall x \in x
   };
\end{tikzpicture}
```

Two strategies are always available: linear steps, which yields (semi)automatic ticks are evenly spaced positions, and exponential steps, which yields (semi)automatic steps at positions at exponentially increasing positions – which is exactly what is needed for logarithmic plots. These strategies are details in Section 59.4.13.

The following options are used to configure tick placement strategies like linear steps. Unlike the basic choice of a placement strategy, which is an axis option, the following should be passed to the option ticks or grid only. So, you would write things like x axis={ticks={step=2}}, but x axis={linear steps}.

```
/tikz/data visualization/step=\langle value \rangle
```

(no default, initially 1)

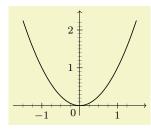
The value of this key is used to determine the spacing of the major ticks. The key is used by the linear steps and exponential steps strategies, see the explanations in Section 59.4.13 for details. Basically, all ticks are placed at all multiples of $\langle value \rangle$ that lie in the attribute range interval.

```
\tikz \datavisualization [
    school book axes, visualize as smooth line,
    y axis={ticks={step=1.25}},
]
    data [format=function] {
      var x : interval [0:3];
      func y = \value x*\value x/2;
    };
```

/tikz/data visualization/minor steps between steps= $\langle number \rangle$

(default 9)

Specifies that between any two major steps (whose positions are specified by the step key), there should be $\langle number \rangle$ many minor steps. Note that the default of 9 is exactly the right number so that each interval between two minor steps is exactly a tenth of the size of a major step. See also Section 59.4.13 for further details.



```
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line,
    x axis={ticks={minor steps between steps=3}},
    y axis={ticks={minor steps between steps}},
]
  data [format=function] {
    var x : interval [-1.5:1.5];
    func y = \value x*\value x;
    };
\end{tikzpicture}
```

/tikz/data visualization/phase= $\langle value \rangle$

(no default, initially 0)

See Section 59.4.13 for details on how the phase of steps influences the tick placement.

59.4.4 Automatic Computation of Tick and Grid Line Positions

The step option gives you "total control" over the stepping of ticks on an axis, but you often do not know the correct stepping in advance. In this case, you may prefer to have a good value for step being computed for you automatically.

Like the step key, these options are passed to the ticks option. So, for instance, you would write x axis={ticks={about=4}} to request about four ticks to be placed on the x-axis.

/tikz/data visualization/about= $\langle number \rangle$

(no default)

This key asks the data visualization to place about $\langle number \rangle$ many ticks on an axis. It is not guaranteed that $exactly \langle number \rangle$ many ticks will be used, rather the actual number will be the closest number of ticks to $\langle number \rangle$ so that their stepping is still "good". For instance, when you say about=10, it may happen that exactly 10, but perhaps even 13 ticks are actually selected, provided that these numbers of ticks lead to good stepping values like 5 or 2.5 rather than numbers like 3.4 or 7. The method that is used to determine which steppings a deemed to be "good" depends on the current tick placement strategy.

Linear steps. Let us start with linear steps: First, the difference between the maximum value v_{max} and the minimum value v_{min} on the axis is computed; let us call it r for "range." Then, r is divided by $\langle number \rangle$, yielding a target stepping s. If s is a number like 1 or 5 or 10, then this number could be used directly as the new value of step. However, s will typically something strange like 0.02345 or 345223.76, so s must be replaced by a better value like 0.02 in the first case and perhaps 250000 in the second case.

In order to determine which number is to be used, s is rewritten in the form $m \cdot 10^k$ with $1 \le m < 10$ and $k \in \mathbb{Z}$. For instance, 0.02345 would be rewritten as $2.345 \cdot 10^{-2}$ and 345223.76 as $3.4522376 \cdot 10^5$. The next step is to replace the still not-so-good number m like 2.345 or 3.452237 by a "good" value m'. For this, the current value of the about strategy is used:

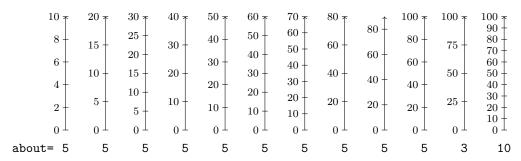
/tikz/data visualization/about strategy= $\langle list \rangle$

(no default)

The $\langle list \rangle$ is a comma-separated sequence of pairs $\langle threshold \rangle / \langle value \rangle$ like for instance 1.5/1.0 or 2.3/2.0. When a good value m' is sought for a given m, we iterate over the list and find the first pair $\langle threshold \rangle / \langle value \rangle$ where $\langle threshold \rangle$ exceeds m. Then m' is set to $\langle value \rangle$. For instance, if $\langle list \rangle$ is 1.5/1.0,2.3/2.0,4/2.5,7/5,11/10, which is the default, then for m=3.141 we would get m'=2.5 since 4>3.141, but $2.3\leq 3.141$. For m=6.3 we would get m'=5.

Once m' has been determined, the stepping is set to $s' = m' \cdot 10^k$.

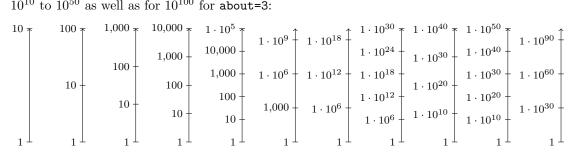
The net effect of all this is that for the default strategy the only valid stepping are the values 1, 2, 2.5 and 5 and every value obtainable by multiplying one of these values by a power of ten. The following example shows the effects of, first, setting about=5 (corresponding to the some option) and then having axes where the minimum value is always 0 and where the maximum value ranges from 10 to 100 and, second, setting about to the values from 3 (corresponding to the few option) and to 10 (corresponding to the many option) while having the minimum at 0 and the maximum at 100:



Exponential steps. For exponential steps the strategy for determining a good stepping value is similar to linear steps, but with the following differences:

- Naturally, since the stepping value refers to the exponent, the whole computation of a good stepping value needs to be done "in the exponent." Mathematically spoken, instead of considering the difference $r = v_{\text{max}} v_{\text{min}}$, we consider the difference $r = \log v_{\text{max}} \log v_{\text{min}}$. With this difference, we still compute $s = r/\langle number \rangle$ and let $s = m \cdot 10^k$ with $1 \le m < 10$.
- It makes no longer sense to use values like 2.5 for m' since this would yield a fractional exponent. Indeed, the only sensible values for m' seem to be 1, 3, 6, and 10. Because of this, the about strategy is ignored and one of these values or a multiple of one of them by a power of ten is used.

The following example shows the chosen steppings for a maximum varying from 10^1 to 10^5 and from 10^{10} to 10^{50} as well as for 10^{100} for about=3:



Alternative strategies.

In addition to the standard about strategy, there are some additional strategies that you might wish to use instead:

/tikz/data visualization/standard about strategy

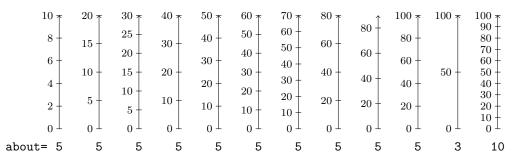
(no value)

Permissible values for m' are: 1, 2, 2.5, and 5. This strategy is the default strategy.

/tikz/data visualization/euro about strategy

(no value)

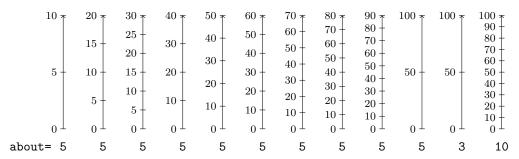
Permissible values for m' are: 1, 2, and 5. These are the same values as for the Euro coins, hence the name.



/tikz/data visualization/half about strategy

(no value)

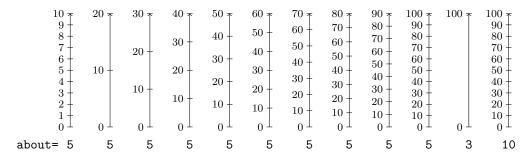
Permissible values for m': 1 and 5. Use this strategy if only powers of 10 or halves thereof seem logical.



/tikz/data visualization/decimal about strategy

(no value)

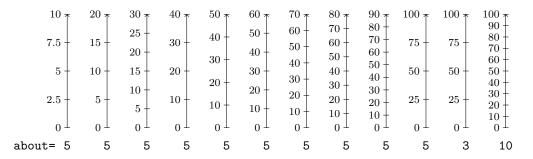
The only permissible value for m' is 1. This is an even more radical version of the previous strategy.



/tikz/data visualization/quarter about strategy

(no value)

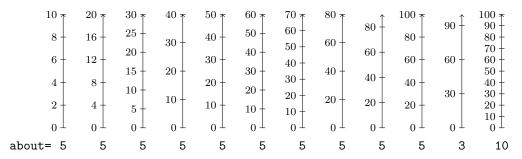
Permissible values for m' are: 1, 2.5, and 5.



/tikz/data visualization/int about strategy

(no value)

Permissible values for m' are: 1, 2, 3, 4, and 5.



/tikz/data visualization/many

(no value)

This is an abbreviation for about=10.

/tikz/data visualization/some

(no value)

This is an abbreviation for about=5.

/tikz/data visualization/few

(no value)

This is an abbreviation for about=3.

/tikz/data visualization/none

(no value)

Switches off the automatic step computation. Unless you use step= explicitly to set a stepping, no ticks will be (automatically) added.

59.4.5 Manual Specification of Tick and Grid Line Positions

The automatic computation of ticks and grid lines will usually do a good job, but not always. For instance, you might wish to have ticks exactly at, say, prime numbers or at Fibonacci numbers or you might wish to have an additional tick at π . In these cases you need more direct control over the specification of tick positions.

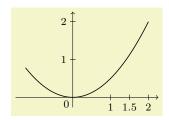
First, it is important to understand that the data visualization system differentiates between three kinds of ticks and grid lines: major, minor, and subminor. The major ticks are the most prominent ticks where, typically, a textual representation of the tick is shown; and the major grid lines are the thickest. The minor ticks are smaller, more numerous, and lie between major ticks. They are used, for instance, to indicate positions in the middle between major ticks or at all integer positions between major ticks. Finally, subminor ticks are even smaller than minor ticks and they lie between minor ticks.

Four keys are used to configure the different kinds:

/tikz/data visualization/major=\langle options\rangle

(no default)

The key can be passed as an option to the ticks key and also to the grid key, which in turn is passed as an option to an axis. The $\langle options \rangle$ passed to major specify at which positions major ticks/grid lines should be shown (using the at option and also at option) and also any special styling. The different possible options are described later in this section.

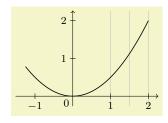


```
\tikz \datavisualization
[ school book axes, visualize as smooth line,
    x axis={ticks={major={at={1, 1.5, 2}}}}]
data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
};
```

/tikz/data visualization/minor=\langle options \rangle

(no default)

Like major, only for minor ticks/grid lines.



```
\tikz \datavisualization
  [ school book axes, visualize as smooth line,
    x axis={grid={minor={at={1, 1.5, 2}}}}]
data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
};
```

/tikz/data visualization/subminor=\langle options\rangle

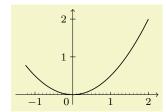
(no default)

Like major, only for subminor ticks/grid lines.

/tikz/data visualization/common=\langle options \rangle

(no default)

This key allows you to specify $\langle options \rangle$ that apply to major, minor and subminor alike. It does not make sense to use common to specify positions (since you typically do not want both a major and a minor tick at the same position), but it can be useful to configure, say, the size of all kinds of ticks:



```
\tikz \datavisualization
[ school book axes, visualize as smooth line,
    x axis={ticks={minor steps between steps, common={low=0}}} ]
data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
};
```

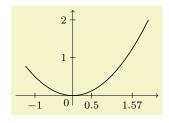
The following keys can now be passed to the major, minor, and subminor keys to specify where ticks or grid lines should be shown:

/tikz/data visualization/at=\langle list \rangle

(no default)

Basically, the $\langle list \rangle$ must be a list of values that is processed with the \foreach macro (thus, it can contain ellipses to specify ranges of value). Empty values are skipped.

The effect of passing at to a major, minor, or subminor key is that ticks or grid lines on the axis will be placed exactly at the values in $\langle list \rangle$. Here is an example:



```
\tikz \datavisualization
[ school book axes, visualize as smooth line,
    x axis={ticks={major={at={-1,0.5,(pi/2)}}}}]
data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
};
```

When this option is used, any previously specified tick positions are overwritten by the values in $\langle list \rangle$. Automatically computed ticks are also overwritten. Thus, this option gives you complete control over where ticks should be placed.

Normally, the individual values inside the $\langle list \rangle$ are just numbers that are specified in the same way as an attribute value. However, such a value may also contain the keyword **as**, which allows you so specify the styling of the tick in detail. Section 59.4.6 details how this works.

It is often a bit cumbersome that one has to write things like

```
some axis = {ticks = {major = {at = {...}}}}
```

A slight simplification is given by the following keys, which can be passed directly to ticks and grid:

```
/tikz/data visualization/major at=\langle list \rangle (no default)
```

A shorthand for major= $\{at=\{\langle list\rangle\}\}$.

/tikz/data visualization/minor at=
$$\langle list \rangle$$
 (no default)

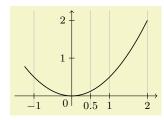
A shorthand for major= $\{at=\{\langle list\rangle\}\}$.

$$/\text{tikz/data visualization/subminor at=}\langle list \rangle$$
 (no default)

A shorthand for major= $\{at=\{\langle list\rangle\}\}$.

```
/tikz/data visualization/also at=\langle list \rangle (no default)
```

This key is similar to at, but it causes ticks or grid lines to be placed at the positions in the $\langle list \rangle$ in addition to the ticks that have already been specified either directly using at or indirectly using keys like step or some. The effect of multiple calls of this key accumulate. However, when at is used after an also at key, the at key completely resets the positions where ticks or grid lines are shown.



```
\tikz \datavisualization
[ school book axes, visualize as smooth line,
    x axis={grid, ticks and grid={major={also at={0.5}}}}]
data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
};
```

As for at, there are some shorthands available:

```
/tikz/data visualization/major also at=\langle list \rangle (no default)
```

A shorthand for major={also at={ $\langle list \rangle$ }}.

```
/tikz/data visualization/minor also at=\langle list \rangle (no default)
```

A shorthand for major={also at={ $\langle list \rangle$ }}.

```
/tikz/data visualization/subminor also at=\langle list \rangle (no default)
```

A shorthand for major={also at={ $\langle list \rangle$ }}.

59.4.6 Styling Ticks and Grid Lines: Introduction

When a tick, a tick label, or a grid line is visualized on the page, a whole regiment of styles influences the appearance. The reason for this large number of interdependent styles is the fact that we often wish to influence only a very certain part of how a tick is rendered while leaving the other aspects untouched: Sometimes we need to modify just the font of the tick label; sometimes we wish to change the length of the tick label and the tick label position at the same time; sometimes we wish to change the color of grid line, tick, and tick label; and sometimes we wish to generally change the thickness of all ticks.

Let us go over the different kinds of things that can be styled (grid lines, ticks, and tick labels) one by one and let us have a look at which styles are involved. We will start with the grid lines, since they turn out to be the most simple, but first let us have a look at the general style and styling mechanism that is used in many placed in the following:

59.4.7 Styling Ticks and Grid Lines: The Style and Node Style Keys

All keys of the data visualization system have the path prefix /tikz/data visualization. This is not only true for the main keys like scientific axes or visualize as line, but also for keys that govern how ticks are visualized. In particular, a style like every major grid has the path prefix /tikz/data visualization and all keys stored in this style are also executed with this path prefix.

Normally, this does not cause any trouble since most of the keys and even styles used in a data visualization are intended to configure what is shown in the visualization. However, at some point, we may also with to

specify options that no longer configure the visualization in general, but specify the appearance of a line or a node on the TikZ layer.

Two keys are used to "communicate" with the TikZ layer:

/tikz/data visualization/style= $\langle TikZ \ options \rangle$

(no default)

This key takes options whose path prefix is /tikz, not /tikz/data visualization. These options will be appended to a current list of such options (thus, multiple calls of this key accumulate). The resulting list of keys is not executed immediately, but it will be executed whenever the data visualization engine calls the TikZ layer to draw something (this placed will be indicated in the following).

```
100

90

80

70

60

50

40

30

5 6 7 8 9 10
```

```
\tikz \datavisualization
[scientific axes,
  all axes={ticks={style=blue}, length=3cm},
  y axis={grid, grid={minor steps between steps, major={style=red}}},
  visualize as line]
data [format=function] {
  var x : interval [5:10];
  func y = \value x * \value x;
};
```

/tikz/data visualization/styling

(no value)

Executing this key will cause all "accumulated" TikZ options from previous calls to the key /tikz/data visualization/style to be executed. Thus, you use style to set TikZ options, but you use styling to actually apply these options. Usually, you do not call this option directly since this application is only done deep inside the data visualization engine.

Similar to style (and styling) there also exist the node style (and node styling) key that takes TikZ options that apply to nodes only – in addition to the usual style.

/tikz/data visualization/node style=\langle TikZ options\rangle

(no default)

This key works like style, only it has an effect only on nodes that are created during a data visualization. This includes tick labels and axis labels:

```
100
90
80
70
60
50
40
30
```

```
\tikz \datavisualization
[scientific axes,
  all axes={ticks={node style=red}, length=3cm},
  visualize as line]
data [format=function] {
  var x : interval [5:10];
  func y = \value x * \value x;
};
```

Note that in the example the ticks themselves (the little thicker lines) are not red.

/tikz/data visualization/node styling

(no value)

Executing this key will cause all "accumulated" node stylings to be executed.

59.4.8 Styling Ticks and Grid Lines: Styling Grid Lines

When a grid line is visualized, see Section 59.5.3 for details on when this happens, the following styles are executed in the specified order.

- 1. grid layer.
- 2. every grid.
- 3. every major grid or every minor grid or every subminor grid, depending on the kind of grid line.

- 4. locally specified options for the individual grid line, see Section 59.4.11.
- 5. styling, see Section 59.4.7.

All of these keys have the path prefix /tikz/data visualization. However, the options stored in the first style (grid layer) and also in the last (styling) are executed with the path prefix /tikz (see Section 59.4.7).

Let us now have a look at these keys in detail:

/tikz/data visualization/grid layer

(style, initially on background layer)

This key is used to specified the *layer* on which grid lines should be drawn (layers are explained in Section 26). By default, all grid lines are placed on the **background** layer and thus behind the data visualization. This is a sensible strategy since it avoids obscuring the more important data with the far less important grid lines. However, you can change this style to "get the grid lines to the front":

```
100

90

80

70

60

50

40

30

5 6 7 8 9 10
```

```
\tikz \datavisualization
[scientific axes,
  all axes={
    length=3cm,
    grid,
    grid={minor steps between steps}
},
grid layer/.style=, % none, so on top of data (bad idea)
visualize as line]
data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
};
```

When this style is executed, the keys stored in the style will be executed with the prefix /tikz. Normally, you should only set this style to be empty or to on background layer.

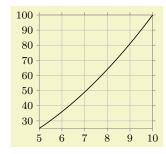
/tikz/data visualization/every grid

(style, no value)

This style provides overall configuration options for grid lines. By default, it is set to the following:

```
low=min, high=max
```

This causes grid lines to span all possible values when they are visualized, which is usually the desired behaviour (the low and high keys are explained in Section 59.5.4. You can append the style key to this style to configure the overall appearance of grid lines. In the following example we cause all grid lines to be dashed (which is not a good idea in general since it creates a distracting background pattern).



```
\tikz \datavisualization
  [scientific axes,
  all axes={length=3cm, grid},
  every grid/.append style={style=densely dashed},
  visualize as line]
data [format=function] {
  var x : interval [5:10];
  func y = \value x * \value x;
};
```

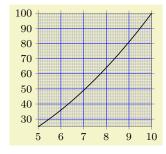
/tikz/data visualization/every major grid

(style, no value)

This style configures the appearance of major grid lines. It does so by calling the **style** key to setup appropriate TikZ options for visualizing major grid lines. The default definition of this style is:

```
style = {help lines, thin, black!25}
```

In the following example, we use thin major blue grid lines:



```
\tikz \datavisualization
[scientific axes,
all axes={
   length=3cm,
   grid,
   grid={minor steps between steps}
},
   every major grid/.style = {style={blue, thin}},
   visualize as line]
   data [format=function] {
    var x : interval [5:10];
   func y = \value x * \value x;
};
```

As can be seen, this is not exactly visually pleasing. The default settings for the grid lines should work in most situations; you may wish to increase the blackness level, however, when you experience trouble during printing or projecting graphics.

/tikz/data visualization/every minor grid

(style, no value)

Works like every major grid. The default is

```
style = {help lines, black!25}
```

/tikz/data visualization/every subminor grid

(style, no value)

Works like every major grid. The default is

```
style = {help lines, black!10}
```

59.4.9 Styling Ticks and Grid Lines: Styling Ticks and Tick Labels

Styling ticks and tick labels is somewhat similar to styling grid lines. Let us start with the tick *mark*, that is, the small line that represents the tick. When this mark is drawn, the following styles are applied:

- 1. every ticks.
- 2. every major ticks or every minor ticks or every subminor ticks, depending on the kind of ticks to be visualized.
- 3. locally specified options for the individual tick, see Section 59.4.11.
- 4. tick layer
- 5. every odd tick or every even tick, see Section 59.4.12.
- 6. draw
- 7. styling, see Section 59.4.7.

For the tick label node (the node containing the textual representation of the attribute's value at the tick position), the following styles are applied:

- 1. every ticks.
- 2. every major ticks or every minor ticks or every subminor ticks, depending on the kind of ticks to be visualized.
- 3. locally specified options for the individual tick, see Section 59.4.11.
- 4. tick node layer
- 5. every odd tick or every even tick, see Section 59.4.12.
- 6. styling, see Section 59.4.7.
- 7. node styling, see Section 59.4.7.

```
/tikz/data visualization/every ticks
```

(style, no value)

This style allows you to configure the appearance of ticks using the style and node style key. Here is (roughly) the default definition of this style:

```
node style={
  font=\footnotesize,
  inner sep=1pt,
  outer sep=.1666em,
  rounded corners=1.5pt
}
```

/tikz/data visualization/every major ticks

(style, no value)

The default is

```
style={line cap=round}, tick length=2pt
```

/tikz/data visualization/every minor ticks

(style, no value)

The default is

```
style={help lines,thin, line cap=round}, tick length=1.4pt
```

/tikz/data visualization/every subminor ticks

(style, no value)

The default is

```
style={help lines, line cap=round}, tick length=0.8pt
```

/tikz/data visualization/tick layer

(style, initially on background layer)

Like grid layer, this key specifies on which layer the ticks should be placed.

/tikz/data visualization/tick node layer

(style, initially empty)

Like tick layer, but now for the nodes. By default, tick nodes are placed on the main layer and thus on top of the data in case that the tick nodes are inside the data.

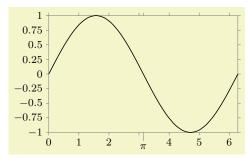
59.4.10 Styling Ticks and Grid Lines: Exceptional Ticks

You may sometimes wish to style a few ticks differently from the other ticks. For instance, in the axis system school book axes there should be a tick label at the 0 position only on one axis and then this label should be offset a bit. In many cases this is easy to achieve: When you add a tick "by hand" using the at or also at option, you can add any special options in square brackets.

However, in some situations the special tick position has been computed automatically for you, for instance by the step key or by saying tick=some. In this case, adding a tick mark with the desired options using also at would cause the tick mark with the correct options to be shown in addition to the tick mark with the wrong options. In cases like this one, the following option may be helpful:

```
/\text{tikz/data visualization/options at=}\langle value \rangle as [\langle options \rangle] (no default)
```

This key causes the $\langle options \rangle$ to be executed for any tick mark(s) at $\langle value \rangle$ in addition to any options given already for this position:



/tikz/data visualization/no tick text at= $\langle value \rangle$

(no default)

Shorthand for options at= $\langle value \rangle$ as [no tick text].

59.4.11 Styling Ticks and Grid Lines: Styling and Typesetting a Value

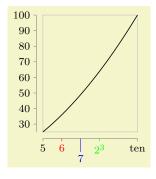
The at and also at key allow you to provide a comma-separated $\langle list \rangle$ of $\langle value \rangle$ s where ticks or grid lines should be placed. In the simplest case, the $\langle value \rangle$ is simply a number. However, the general syntax allows three different kinds of $\langle value \rangle$ s:

- 1. $\langle value \rangle$
- 2. $\langle value \rangle$ as $[\langle local\ options \rangle]$
- 3. $\langle value \rangle$ as $[\langle local\ options \rangle] \langle text \rangle$

In the first case, the $\langle value \rangle$ is just a number that is interpreted like any other attribute value.

In the second case, where the keyword as is present, followed by some option in square brackets, but nothing following the closing square bracket, when the tick or grid line at position $\langle value \rangle$ is shown, the $\langle local\ options \rangle$ are executed first. These can use the style key or the node style key to configure the appearance of this single tick or grid line. You can also use keys like low or high to influence how large the grid lines or the ticks are or keys like tick text at low to explicitly hide or show a tick label.

In the third case, which is only important for ticks and not for grid, the same happens as in the second case, but the text that is shown as tick label is $\langle text \rangle$ rather than the automatically generated tick label. This automatic generation of tick labels is explained in the following.



A value like "2" or "17" could just be used as $\langle text \rangle$ to be displayed in the node of a tick label. However, things are more difficult when the to-be-shown value is 0.0000000015, because we then would typically (but not always) prefer something like $1.5 \cdot 10^{-9}$ to be shown. Also, we might wish a unit to be added like 23m/s. Finally, we might wish a number like 3.141 to be replaced by π . For these reasons, the data visualization system does not simply put the to-be-shown value in a node as plain text. Instead, the number is passed to a typesetter whose job it is to typeset this number nicely using TeX's typesetting capabilities. The only exception is, as indicated above, the third syntax version of the at and also at keys, where $\langle text \rangle$ is placed in the tick label's node, regardless of what the typesetting would usually do.

The text produced by the automatic typesetting is computed as follows:

- 1. The current contents of the key tick prefix is put into the node.
- 2. This is followed by a call of the key tick typesetting which gets the $\langle value \rangle$ of the tick as its argument in scientific notation.
- 3. This is followed by the contents of the key tick suffix.

Let us have a look at these keys in detail:

/tikz/data visualization/tick prefix= $\langle text \rangle$

(no default, initially empty)

The $\langle text \rangle$ will be put in front of every typeset tick:

```
75
50
25
(6] (8] (10]
```

```
\tikz \datavisualization
[scientific axes, all axes={ticks=few, length=2.5cm},
   x axis={ticks={tick prefix=$\langle$, tick suffix=$]$}},
   visualize as line]
data [format=function] {
   var x : interval [5:10];
   func y = \value x * \value x;
};
```

```
/tikz/data visualization/tick suffix=\langle text \rangle
```

(no default, initially empty)

Works like tick prefix. This key is especially useful for adding units like "cm" or "m/s" to every tick label. For this reason, there is a (near) alias that is easier to memorize:

```
/tikz/data visualization/tick unit=\(roman \ math \ text\)
```

(no default)

A shorthand for tick suffix= $\{\$\setminus, \text{rm}(roman \ math \ text)\}$:

```
100 m/s<sup>2</sup>

90 m/s<sup>2</sup>

80 m/s<sup>2</sup>

70 m/s<sup>2</sup>

60 m/s<sup>2</sup>

50 m/s<sup>2</sup>

30 m/s<sup>2</sup>

5 s 6 s 7 s 8 s 9 s 10 s
```

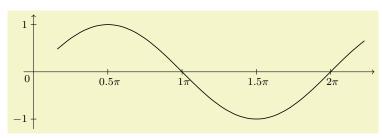
```
\tikz \datavisualization
[scientific axes, all axes={length=3cm},
   x axis={ticks={tick unit=s}},
   y axis={ticks={tick unit=m/s^2}},
   visualize as line]
data [format=function] {
   var x : interval [5:10];
   func y = \value x * \value x;
};
```

/tikz/data visualization/tick typesetting= $\langle value \rangle$

(no default)

The key gets called for each number that should be typeset. The argument $\langle value \rangle$ will be in scientific notation (like 1.0e1 for 10). By default, this key applies \pgfmathprintnumber to its argument. This command is a powerful number printer whose configuration is documented in Section 74.

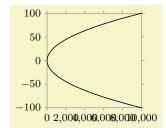
You are invited to code underlying this key so that a different type setting mechanism is used. Here is a (not quite finished) example that shows how, say, numbers could be printed in terms of multiples of π :



```
\def\mytypesetter#1{%
  \pgfmathparse{#1/pi}%
  \pgfmathprintnumber{\pgfmathresult}$\pi$%
}
\tikz \datavisualization
  [school book axes, all axes={unit length=1.25cm},
    x axis={ticks={step=(0.5*pi), tick typesetter/.code=\mytypesetter{##1}}},
    y axis={include value={-1,1}},
    visualize as smooth line]
data [format=function] {
    var x : interval [0.5:7];
    func y = sin(\value x r);
};
```

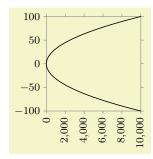
59.4.12 Stacked Ticks

Sometimes, the text of tick labels are so long or so numerous that the text of adjacent tick labels overlap (or have too little padding):



There are two ways to address this problem:

• One can rotate the labels on horizontal axes:



This is often a good solution, but may be hard to read. Also consider rotating labels only by 45° or 30°

• One can specify different shifts of the nodes for the different ticks, whereby the ticks text no longer overlap.

```
100

50

0

-50

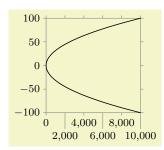
-100

0

4,000 8,000

2,000 6,000 10,000
```

However, specifying shifts "by hand" in the above way is not always an option, especially when the tick positions should be computed automatically. Instead, the stack option can be used, which is much easier to use and gives better results:



The stack option is actually just a style that gives you access to the general even/odd mechanism for ticks with labels. Whenever a tick mark is created where a tick label is also to be drawn, two special things happen:

1. For every odd tick mark, the every odd tick style is executed, for every even tick mark the every even tick. Here, "odd" and "even" are with respect to the order in which the ticks have been added to the list of at positions for each major, minor, or subminor tick list, not with respect to the order in which they will appear on the axis. Thus, when you write

```
ticks={major at=\{1,2,3,4\}, major at=\{0,-1,-2\}, minor at=\{9,8,7\}}
```

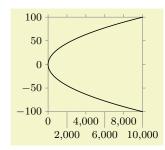
then for 1, 3, 0, and -2 as well as 9 and 7 the key every odd tick will be executed, while every even tick will be executed for positions 2, 4, -1, and also 8.

2. When a tick node label is shown at the low position of the tick mark, the dimension stored in the key tick text low even padding is added to the low value. Provided that this padding is not zero (which is the default), the length of the even tick marks will be increased and the tick label node will be placed at a greater distance from the axis.

Similar keys exist for padding ticks with labels at high positions and also at even positions.

/tikz/data visualization/tick text low even padding=\langle dimension \rangle (no default, initially Opt)

When a tick label is shown at the low position of an even tick, the $\langle distance \rangle$ is added to the low value, see also Section 59.5.4.



Note that $\langle dimension \rangle$ should usually be non-positive.

The following keys work similarly:

```
/tikz/data visualization/tick text low odd padding=(dimension) (no default, initially Opt)
```

/tikz/data visualization/tick text high even padding=\(\langle dimension\rangle\) (no default, initially Opt)

/tikz/data visualization/tick text high odd padding=\langle dimension \rangle (no default, initially Opt)

```
/\text{tikz/data visualization/tick text odd padding=} \langle dimension \rangle (no default)
```

A shorthand for setting tick text odd low padding and tick text odd high padding at the same time

```
/\text{tikz/data visualization/tick text even padding} = \langle dimension \rangle (no default)
```

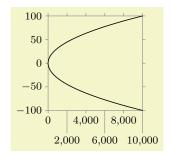
A shorthand for setting tick text even low padding and tick text even high padding at the same time.

```
/tikz/data visualization/tick text padding=\langle dimension \rangle (no default)
```

Sets all text paddings to $\langle dimension \rangle$.

```
/tikz/data visualization/stack=\langle dimension \rangle
```

Shorthand for tick text even padding= $\langle dimension \rangle$.

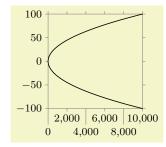


(default 1em)

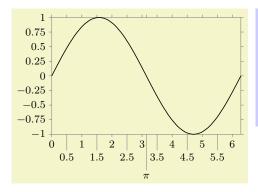
```
/tikz/data visualization/stack'=\langle dimension\rangle
```

(no default)

Shorthand for tick text odd padding= $\langle dimension \rangle$. The difference to stack is that the set of value that are "lowered" is exactly exchanged with the set of value "lowered" by stack.



Note that the above keys have an effect on all tick labels of an axis, also on special ticks that you may have added using the also at key. When using the stack key, you should specify a tick text padding explicitly for such keys:



```
\tikz \datavisualization
[scientific axes,
   x axis={ticks={stack, many, major also at=
          {(pi) as [{tick text padding=2.5em}] $\pi$}},
   visualize as smooth line]
   data [format=function] {
      var x : interval[0:(2*pi)];
      func y = sin(\value x r);
   };
```

59.4.13 Reference: Basic Strategies

/tikz/data visualization/axis options/linear steps

(no value)

This strategy places ticks at positions that are evenly spaced by the current value of step.

In detail, the following happens: Let a be the minimum value of the data values along the axis and let b be the maximum. Let the current stepping be s (the stepping is set using the \mathtt{step} option, see below) and let the current phasing be p (set using the \mathtt{phase}) option. Then ticks are placed all positions $i \cdot s + p$ that lie in the interval [a, b], where i ranges over all integers.

The tick positions computed in the way described above are mayor step positions. In addition to these, if the key minor steps between steps is set to some number n, then n many minor ticks are introduced between each two mayor ticks (and also before and after the last mayor tick, provided the values still lie in the interval [a, b]). Note that is n is 1, then one minor tick will be added in the middle between any two mayor ticks. Use a value of 9 (not 10) to partition the interval between two mayor ticks into ten equally sized minor intervals.

```
\begin{tikzpicture}
  \datavisualization
  [scientific axes={inner ticks, width=3cm},
    x axis={ticks={step=3, minor steps between steps=2}},
    y axis={ticks={step=.36}},
    visualize as scatter]
  data {
     x, y
     17, 30
     34, 32
  };
\end{tikzpicture}
```

/tikz/data visualization/axis options/exponential steps

(no value)

This strategy produces ticks at positions that are appropriate for logarithmic plots. It is automatically selected when you use the logarithmic option with an axis.

In detail, the following happens: As for linear steps let numbers a, b, s, and p be given. Then, mayor ticks are placed at all positions $10^{i \cdot s + p}$ that lie in the interval [a, b] for $i \in \mathbb{Z}$.

The minor steps are added in the same way as for linear steps. In particular, they interpolate *linearly* between mayor steps.

```
\begin{tikzpicture}
1 \cdot 10^{6}
                              \datavisualization
1 \cdot 10^{5}
                                [scientific axes,
                                 x axis={logarithmic, length=2cm, ticks={step=1.5}},
10,000
                                 y axis={logarithmic, ticks={step=1, minor steps between steps=9}},
                                 visualize as scatter]
1,000
                                data {
                                   х, у
  100
                                   1, 10
                                  1000, 1000000
    10
                                };
            31.62
                   1.000
                            \end{tikzpicture}
```

59.4.14 Advanced: Defining New Placement Strategies

```
/\text{tikz/data visualization/axis options/tick placement strategy} = \langle macro \rangle (no default)
```

This key can be used to install a so-called *tick placement strategy*. Whenever visualize ticks is used to request some ticks to be visualized, it is checked whether some automatic ticks should be created. This is the case when the following key is set:

```
/tikz/data visualization/compute step=\langle code \rangle (no default)
```

The $\langle code \rangle$ should compute a suitable value for the stepping to be used by the $\langle macro \rangle$ in the tick placement strategy.

For instance, the step key sets compute step to $\def\tikz@lib@dv@step{#1}$. Thus, when you say step=5, then the desired stepping of 5 is communicated to the $\langle macro \rangle$ via the macro $\tikz@lib@dv@step$.

Provided compute step is set to some nonempty value, upon visualization of ticks the $\langle macro \rangle$ is executed. Typically, $\langle macro \rangle$ will first call the $\langle code \rangle$ stored in the key compute step. Then, it should implement some strategy then uses the value of the computed or desired stepping to create appropriate at commands. To be precise, it should set the keys major, minor, and/or subminor with some appropriate at values.

Inside the call of $\langle macro \rangle$, the macro \tikzdvaxis will have been set to the name of the axis for which default ticks need to be computed. This allows you to access the minimum and the maximum value stored in the scaling mapper of that axis.

```
15
12.5
10
7.5
5
2.5
0 × 2 3 5 7 11 13
```

```
\def\silly{
  \tikzdatavisualizationset{major={at={
          2,3,5,7,11,13}}}
}
\begin{tikzpicture}
  \datavisualization [
     scientific axes, visualize as scatter,
     x axis={tick placement strategy=\silly}
     ]
     data {
          x, y
          0, 0
          15, 15
     };
\end{tikzpicture}
```

59.5 Advanced: Creating New Axis Systems

The data visualization library comes with a number of predefined axis systems, like scientific axes=clean, but it is also possible and to define new axis systems. Doing so involves the following steps:

- 1. Creating a number of axes.
- 2. Configuring attributes of these axes like their length or default scaling.
- 3. Creating visual representations of the axes.
- 4. Creating visual representations of the ticks and grid lines.

The first step uses new ... axis keys to create new axes, the last steps use visualize ... keys to create the visual representations of the axes.

Note that the axis system has no control over the actual attribute value ranges and neither over which ticks need to be drawn. The axis system can only provide good defaults and then specify *how* the ticks or labels should be drawn and *where* on the page – but not at which values.

In the following, as a running example let us develop an axis system our system that does the following: For the x-axis is looks like a normal scientific axis system, but there are actually two y-axes: One at the left and one at the right, each using a different attribute, but both coexisting in the same picture.

59.5.1 Creating the Axes

A new axis system is created as a style key with the prefix /tikz/data visualization. Thus, we would write:

```
\tikzset{
  data visualization/our system/.style={
    ...
  }
}
```

In our system we need three axis: The x-axis, the left axis and the right axis. Since all of these axes are Cartesian axes, we write the following:

```
\tikzset{
  data visualization/our system/.style={
    new Cartesian axis=x axis,
    new Cartesian axis=left axis,
    new Cartesian axis=right axis,
    x axis={attribute=x},
    left axis={unit vector={(0cm,1pt)}},
    right axis={unit vector={(0cm,1pt)}},
}
```

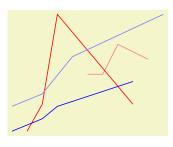
As can be seen, we also configure things so that the x-axis will use the x attribute by default (users can later change this by saying x axis={attribute= $\langle some\ other\ attribute \rangle$ }), but we do not configure the attributes of the left axis nor the right axis. We also make the left and right axis point upward (the x axis needs no configuration here since a Cartesian axis points right by default). The reason is the left would not be a particularly good attribute name and this way we ensure that users have to pick names themselves (hopefully good ones).

The next step is to define a standard scaling for the axes. Here, we can use the same as for scientific axes, so we would add the following keys to the definition of our system:

```
x axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/width}},
left axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}},
right axis={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}}
```

We now already have enough to try our system, although we will not yet see any axes or ticks, but we will see the correct scaling of the attributes. Let us first define a data group:

```
\tikz \datavisualization data group {people and money} = {
    data [set=people 1] {
      time, people
      1900, 1000000000
      1920, 1500000000
      1930, 2000000000
      1980, 3000000000
    data [set=people 2] {
      time, people
      1900. 2000000000
      1920, 2500000000
      1940, 4000000000
      2000, 5700000000
    data [set=money 1] {
      time, money
      1910, 1.1
      1920, 2
      1930, 5
      1980, 2
    data [set=money 2] {
      time, money
      1950, 3
      1960, 3
      1970, 4
      1990, 3.5
  };
```



```
\tikz \datavisualization [
   our system,
   x axis={attribute=time, length=4cm},
   left axis ={attribute=money},
   right axis={attribute=people},
   visualize as line/.list={people 1, people 2, money 1, money 2},
   people 1={style={visualizer color=blue}},
   people 2={style={visualizer color=blue!50}},
   money 1={style={visualizer color=red}},
   money 2={style={visualizer color=red!50}}]
   data group {people and money};
```

59.5.2 Visualizing the Axes

We must now show the axes themselves. For this we can use the visualize axis key:

```
/tikz/data visualization/axis options/visualize axis=\langle options\rangle (no default)
```

This key is passed to an axis as an option. It causes a visual representation of the axis to be created during the data visualization. The $\langle options \rangle$ are used to determine where the axis should be drawn and how long it should be. We can specify, for instance, that an axis should be drawn at the minimum value of another axis or where another axis has the value 0.

The goto, high, and low Keys. In our example, the left axis should be shown at the left hand side. This is the position where the x axis has its minimum value. To specify this, we would use the following code:

```
left axis={ visualize axis={ x axis={ goto=min } }
```

As can be seen, we can pass another axis as an $\langle option \rangle$ to visualize axis, where we pass the following key to the axis in turn:

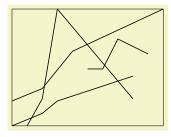
```
/tikz/data visualization/axis options/goto=\langle value \rangle (no default)
```

The key can be passed to an axis. It will set the attribute monitored by the axis to the given $\langle value \rangle$, which is usually some number. However, $\langle value \rangle$ may also be one of the following, which causes a special behaviour:

• min: The attribute is set to the minimal value that the attribute has attained along this axis.

- max: Like min.
- padded min: This will also set the \(\alpha ttribute \rangle \) monitored by the axis to the same value as min. Additionally, however, the subkey \(\alpha \text{data point} / \alpha ttribute \rangle / \text{offset} \) is set to the current padding for the minimum, see the description of padding min later on. The effect of this is that the actual point "meant" by the attribute is offset by this padding along the attribute's axis.
- padded max: Like padded min.

The right axis would be visualized the same way, only at goto=max. The x-axis actually need to be visualized twice: Once at the bottom and once at the top. Thus, we need to call visualize axis twice for this axis:



```
\tikzset{
 data visualization/our system/.append style={
   left axis= {visualize axis={x axis=
                                          {goto=min}}},
   right axis={visualize axis={x axis=
                                          {goto=max}}},
               {visualize axis={left axis={goto=min}},
    x axis=
                visualize axis={left axis={goto=max}}},
}
\tikz \datavisualization [
   our system,
   x axis={attribute=time, length=4cm},
   left axis ={attribute=money},
   right axis={attribute=people}
    visualize as line/.list={people 1, people 2, money 1, money 2}]
 data group {people and money};
```

There is another key that is similar to goto, but has a slightly different semantics:

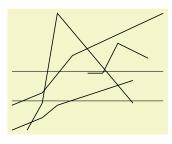
```
/tikz/data visualization/axis options/goto pos=\langle fraction \rangle
```

(no default)

The key works like goto, only the $\langle fraction \rangle$ is not interpreted as a value but as a fraction of the way between the minimum and the maximum value for this axis.

Suppose that for an axis the attribute range interval is [500, 1000] and the reasonable interval is [1,3]. Then for a $\langle fraction \rangle$ of 0, the mapping process would choose value 1 from the reasonable interval, for a $\langle fraction \rangle$ of 1 the position 3 from the reasonable interval, and for a $\langle fraction \rangle$ or 0.25 the position 1.5 since it is one quarter at the distance from 1 to 3.

Note that neither the attribute range interval nor the transformation function for the attribute are important for the goto pos option – the $\langle fraction \rangle$ is computed with respect to the reasonable interval. Also note that the values of the actual attribute corresponding to the fractional positions in the reasonable interval are not computed.



By default, when an axis is visualized, it spans the set of all possible values for the monitored attribute, that is, from min to max. However, there are actually two keys that allow you to adjust this:

```
/tikz/data visualization/low=\langle value \rangle
```

(no default)

This is the attribute value where the axis visualization starts. The same special values as for goto are permissible (like min or padded min, but also 0 or 1).

```
/tikz/data visualization/high=\langle value \rangle
```

(no default)

Like low, only for where the axis ends.

By default, low=min and high=max are set for an axis visualization. Another sensible setting is low=padded min and high=padded max. The following key provides a shorthand for this:

```
/tikz/data visualization/padded
```

(no value)

Shorthand for low-padded min, high-padded max.

As an example, consider the scientific axes=clean. Here, each axis is actually drawn three times: Once at the minimum, once at the maximum and then once more at the padded minimum.

The axis line. When an axis is drawn, TikZ does not simply draw a straight line from the low position to the high position. In reality, the data visualization system uses the two commands pgfpathdvmoveto and pgfpathdvlineto internally. These will replace the straight line by a curve in certain situations. For instance, in a polar coordinate system, if an axis should be drawn along an angle axis for a fixed radius, an arc will be used instead of a straight line.

Styling the axis. As can be seen, we now get the axis we want (but without the ticks, visualizing them will be explained later). The axis is, however, simply a black line. We can *style* the axis in a manner similar to styling ticks and grid lines, see Section 59.4.7. In detail, the following styles get executed:

- 1. axis layer
- 2. every axis
- 3. styling

Additionally, even before every axis is executed, low-min and high-max are executed.

/tikz/data visualization/axis layer

(style, initially on background layer)

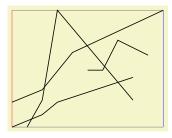
The layer on which the axis is drawn. See the description of grid layer on page 59.4.8 for details.

/tikz/data visualization/every axis

(style, no value)

Put styling of the axis here. It is usually a good idea to set this style to style={black!50}.

Recall that the styling key is set using the style key, see Section 59.4.7.

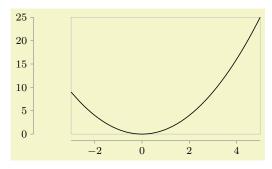


```
\tikzset{
 data visualization/our system/.append style={
   every axis/.style={style=black!50}, % make this the default
   left axis= {visualize axis={x axis=
                                         {goto=min}, style=red!75}},
   right axis={visualize axis={x axis=
                                          {goto=max}, style=blue!75}},
               {visualize axis={left axis={goto=min}},
                visualize axis={left axis={goto=max}}},
}
\tikz \datavisualization [
   our system,
   x axis={attribute=time, length=4cm},
   left axis ={attribute=money},
   right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
 data group {people and money};
```

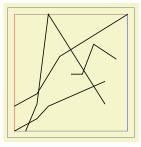
Padding the Axis. When an axis is visualized, it is often a good idea to make it "a little bit longer" or to "remove it a bit from the border," because the visualization of an axis should not interfere with the actual data. For this reason, a *padding* can be specified for axes:

```
/tikz/data visualization/axis options/padding min=\langle dimension \rangle (no default)
```

This is the dimension that is used whenever goto=padded min is used. The $\langle dimension \rangle$ is then put into the offset subkey of the attribute monitored by the axis. When a data point is transformed by a linear transformer and when this subkey is nonzero, this offset is added. (For an angle axis of a polar transformer, the $\langle dimension \rangle$ is interpreted as an additional angle rather than as an additional distance). Note that $\langle dimension \rangle$ should typically be negative since "adding the $\langle dimension \rangle$ " will then make the axis longer (because it starts at a smaller value). The standard axis systems set the padding to some default and take its value into account:



Using padded and using the padded key, we can visualize our axis "a little removed from the actual data":



```
\tikzset{
 data visualization/our system/.append style={
   all axes= {padding=.5em},
   left axis= {visualize axis={x axis=
                                          {goto=padded min}, padded}},
   right axis={visualize axis={x axis=
                                         {goto=padded max}, padded}},
               {visualize axis={left axis={goto=padded min}, padded},
    x axis=
                visualize axis={left axis={goto=padded max}, padded}},
\tikz \datavisualization [
   our system,
   x axis={attribute=time, length=3cm},
   left axis ={attribute=money};
   right axis={attribute=people},
   visualize as line/.list={people 1, people 2, money 1, money 2}]
 data group {people and money};
```

```
/tikz/data visualization/axis options/padding max=\langle dimension \rangle
```

(no default)

Works like padding min, but $\langle dimension \rangle$ should typically be positive.

```
/tikz/data visualization/axis options/padding=\langle dimension \rangle
```

(no default)

Sets both padding min to the negated value of $\langle dimension \rangle$ and padding max to $\langle dimension \rangle$.

59.5.3 Visualizing Grid Lines

As explained earlier, the grid key is used to specify at which positions grid lines should be drawn in principle. However, this key does not actually cause any grid lines to be drawn. Instead, the visualize grid key is used by the axis system to specify how grid lines are drawn.

```
/tikz/data visualization/axis options/visualize grid=\langle options \rangle
```

(no default)

This key is passed to an axis. It causes grid lines to be drawn at the positions specified by the grid key for this axis. The $\langle options \rangle$ govern where and how the grid lines will be drawn.

The direction axis. At first sight, one might expect that the grid lines for an axis should simply be drawn perpendicular to the axis between the minimum and maximum value of the axis. However, things are somewhat more difficult in reality:

- 1. A grid line is supposed to indicate all positions where a certain attribute attains a fixed value. But, then, a grid line does not really need to be a grid *line*. Consider for instance a three dimensional axis system. A "grid line" for the x-coordinate 3 would actually be a "grid plane."
- 2. For a polar coordinate system and a fixed radius, this set of positions at a certain radius is not a straight line, but an arc. For more complicated coordinate systems such as the one arising from three-dimensional spherical projections, a grid line may well be a fairly involved curve.

The visualize grid command addresses these complications as follows:

1. A grid line is always a line, not a plane or a volume. This means that in the example of a three dimensional axis system and the x-attribute being 3, one would have to choose whether the grid line should go "along" the y-axis or "along" the z-axis for this position. One can, however, call

the visualize grid command twice, once for each direction, to cause grid lines to be shown for both directions.

2. A grid line is created by moving to a start position and then doing a line to to the target position. However, the "moveto" and "line to" are done by calling special commands of the data visualization system. These special commands allow coordinate system to "notice" that the line is along an axis and will allow them to replace the straight line by an appropriate curve. The polar axes systems employ this strategy, for instance.

By the above discussion, in order to create a grid line for attribute a having value v, we need to specify an axis "along" which the line should be drawn. When there are only two axes, this is usually "the other axis". This "other axis" is specified using the following key:

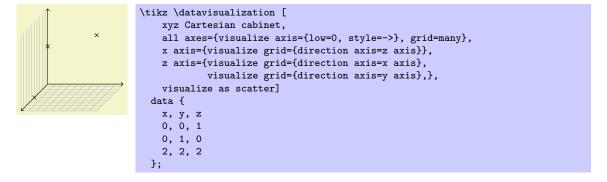
```
/tikz/data visualization/direction axis=\(axis name\)
```

(no default)

You must pass this key as an $\langle option \rangle$ each time you use visualize axis. When the grid line is drawn, the attribute a is set to v and the axis $\langle axis\ name \rangle$'s attribute is set once to the current value of low and once to high. Then a line is drawn between these two positions using \pgfpathdvlineto.

The low and high keys are the same as the ones used in the visualize axis key.

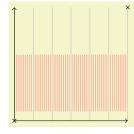
```
\tikz \datavisualization [
    xyz Cartesian cabinet,
    all axes={visualize axis={low=0, style=->}},
    x axis={visualize grid={direction axis=y axis}, grid=many},
    visualize as scatter]
data {
    x, y, z
    0, 0, 1
    0, 1, 0
    2, 2, 2
};
```



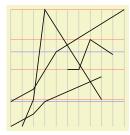
Styling the grid lines. When a grid line is draw, styles are applied as described in Section 59.4.8.

The major, minor, and subminor grid lines. The grid option allows you to specify for each kind of grid line (major, minor, or subminor) a set of different values for which these grid lines should be drawn. Correspondingly, it is also possible to configure for each kind of grid line how it should be drawn. For this, the major, minor, subminor, and also the common keys can be used inside the $\langle options \rangle$ of visualize grid. While as option to grid these keys are used to specify at values, as options of visualize grid they are used to configure the different kinds of grid lines.

Most of the time, no special configuration is necessary since all styling is best done by configuring keys like every major grid. You need to use a key like major only if you wish to configure for instance the low or high values of a major grid line differently from those of minor grid lines – are rather unlikely setting – or when the styling should deviate from the usual settings.



Returning to the example of our system with the two axis systems, it is straight-forward to configure the grid lines of the x-axis: The direction axis is either of the other two axis (they point in the same direction and they have the same range). For the other two axes, we visualize one grid independently of the other, using different colors.



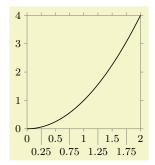
59.5.4 Visualizing the Ticks and Tick Labels

```
/tikz/data visualization/axis options/visualize ticks=(options)
```

(no default)

Visualizing a tick involves (possibly) drawing a tick mark and adding (possibly) the tick node. The process is similar to visualize grid: Users use the ticks key to configure how many ticks they would like for an axis and at which positions. The axis system uses the visualize ticks key to specify where these ticks should actually be shown.

Unlike grid lines, which are typically only visualized once for each combination of an axis and a direction axis, tick marks might be visualized at different places for the same axis. Consider for instance the scientific axes:

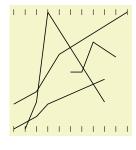


Have a look at the ticks on the y-axis: There are ticks at values 0, 1, 2, 3, and 4. These are visualized both at the left side (where the tick nodes are also shown) and additionally also at the right side, but only as small marks. Similarly, the ticks on the x-axis appear at the bottom, but also (in much simpler versions) at the top. Both for the x-axis and for the y-axis the visualize ticks key was called twice.

The tick marks. Drawing a tick mark is quite similar to visualizing a grid line; indeed a tick mark can be thought of as a "mini grid line": Just like a grid line it "points a long an axis." However, a tick will

always be a short straight line – even when the coordinate system is actually twisted (experimentation has shown that ticks that follow the curvature of the coordinate system like grid lines are hard to recognize). For this reason, the low and high keys have a different meaning from the one used with the visualize grid key. In detail to configure the size and position of a tick mark for the value v of attribute a, proceed as follows:

- The visualize ticks key will have setup attribute a to be equal to v.
- You should now use the goto or goto pos key together with all *other* axes to configure at which position with respect to these other options the tick mark should be shown. For instance, suppose we want tick marks in our system for the x-axis at the bottom and at the top. This corresponds to once setting the left axis to its minimal value and once to its maximal value:



• In the above example, we may wish to shorten the ticks a bit at the bottom and at the top. For this, we use the low and high key:

```
/tikz/data visualization/low=\(dimension\) (no default)
```

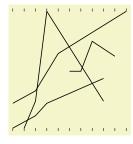
When used with the visualize ticks option, the low key contains a dimension that specifies the extend of the tick going "toward the minimum" of the direction axis. More precisely, when a tick mark is visualized, a unit tangent vector at the current data point in the direction of the direction axis is computed and this vector is multiplied by $\langle dimension \rangle$ to compute the start position of the tick line. The end position is given by this vector times the high value.

Note that the $\langle dimension \rangle$ should usually be negative for the low key and positive for the high key.

For tick marks where a tick label node is shown, the $\langle dimension \rangle$ is increased by the current values of keys like tick text even low padding, see Section 59.4.12 for details.

```
/tikz/data visualization/tick length=\langle dimension \rangle (no default)
Shorthand for low=-\langle dimension \rangle, high=\langle dimension \rangle.
```

What we want to happen is that in the upper visualization of the ticks the low value is Opt, while in the lower one the high value is Opt:



```
\tikzset{
   data visualization/our system/.append style={
      x axis={
      visualize ticks={direction axis=left axis,high=0pt,left axis={goto=min}},
      visualize ticks={direction axis=left axis,low=0pt,left axis={goto=max}},
   }
}

\tikz \datavisualization [
   our system,
   x axis={attribute=time, length=3cm, ticks=many},
   left axis ={attribute=money},
   right axis={attribute=people},
   visualize as line/.list={people 1, people 2, money 1, money 2}]
   data group {people and money};
```

In order to style the tick mark, use the styling mechanism that is detailed in Section 59.4.9.

The tick label node. At certain tick positions, we may wish to add a node indicating the value of the attribute at the given position. The visualize ticks command has no influence over which text should be shown at a node – the text is specified and typeset as explained in Section 59.4.11.

Each time visualize ticks, for each tick position up to two tick label nodes will be created: One at the low position and one at the high position. The following keys are used to configure which of these cases happen:

```
/tikz/data visualization/tick text at low=(true or false) (default true)
```

Pass this option to visualize ticks when you want tick label nodes to be placed at the low position of each tick mark.

```
1,920 1,960 2,000

1,900 | 1,940 | 1,980 |

1,900 | 1,940 | 1,980 |

1,920 1,960 2,000
```

```
/tikz/data visualization/tick text at high=\langle true\ or\ false \rangle
```

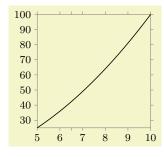
(default true)

Like tick text at low.

```
/tikz/data visualization/no tick text
```

(no value)

Shorthand for tick text at low=false, tick text at high=false.



When a tick label node is to be placed at the low or the high position, the next step is to determine the exact position and the correct anchor of the node. This is done as follows:

• In order to compute an appropriate anchor, the tick mark is considered: This is a short line pointing in a certain direction. For a tick label node at the low position, the anchor attribute is setup in such a way that the node label will be below the low position when the tick mark direction points up, it will be to the right when the direction points left, above when it points down, and so on also for diagonal directions. Similarly, for the high position, when the direction points up, the node will be placed above the tick mark and so on.

This computation is done automatically.

- The tick label node is styled. The styles that are applied are described in Section 59.4.9.
- A tick label node for the low position is usually anchored at this low position, but an additional padding will be added as described in Section 59.4.12.

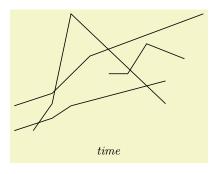
59.5.5 Visualizing the Axis Labels

The label option can be used with an axis to specify a text should be shown next to the axis to indicates which attribute this axis refers to. Like ticks or grid, the label option does not actually draw the label, this is the job of the visualize label key, which is configured by the axis system.

```
/\text{tikz/data visualization/axis options/visualize label=}\langle options \rangle (no default)
```

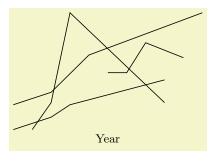
The $\langle options \rangle$ should be used to configure a "good place" for the axis label. Usually, you will use the goto or the goto pos key.

For the example of our system, we would like the label of the x axis to be placed below at the middle of the axis, so we use goto pos=.5 to determine this position. Concerning the other axes, we want it to be placed at the minimum position of the left axis with a lot of padding.



```
\tikzdatavisualizationset{
  our system/.append style={
    x axis={visualize label={
        x axis={goto pos=.5},
        left axis={padding=1.5em, goto=padded min}}}
}
}
\tikz \datavisualization [
  our system,
    x axis={attribute=time, ticks=some, label},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={
        people 1, people 2, money 1, money 2}]
    data group {people and money};
```

In the above example, the padding of 1.5em was rather arbitrary and "suboptimal". It would be outright wrong if the labels on the x axis were larger or if they were missing. It would be better if the vertical position of the x axis label were always "below" all other options. For such cases a slightly strange approach is useful: You position the node using node style={at=...} where at is now the normal TikZ option that is used to specify the position of a node. Inside the ..., you specify that the horizontal position should be the bottom of up-to-now-constructed data visualization and the vertical position should be at the "origin," which is, however, the position computed by the goto keys for the axes:



```
\tikzdatavisualizationset{
  our system/.append style={
    x axis={visualize label={
        x axis={goto pos=.5},
        node style={
            at={(0,0 |- data visualization bounding box.south)},
            below
} } } } }
\tikz \datavisualization [
    our system,
    x axis={attribute=time, ticks=some, label=Year},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={
        people 1, people 2, money 1, money 2}]
    data group {people and money};
```

Two additional keys are useful for positioning axis labels:

/tikz/data visualization/axis option/anchor at min

(no value)

When passed to an axis, this key sets the anchor so that a node positioned at either the min or the padded min value of the axis will be placed "nicely" with respect to the axis. For instance, if the axis points upwards from the min value to the max value, the anchor would be set to north since this gives a label below the axis's start. Similarly, if the axis points right, the anchor would be set to east, and so on.

```
/tikz/data visualization/axis option/anchor at max
```

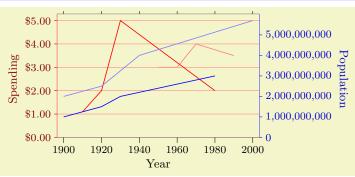
(no value)

Like anchor at min.

59.5.6 The Complete Axis System

Here is the code for the complete axis system developed above and an example of how it is used:

```
\tikzdatavisualizationset{ our system/.style={
  % The axes
  new Cartesian axis=x axis,
                                 new Cartesian axis=left axis,
                                                                       new Cartesian axis=right axis,
  % The directions of the axes
  all axes={padding=.5em},
                                 left axis={unit vector={(0cm,1pt)}}, right axis={unit vector={(0cm,1pt)}},
  % The default attributes, other attributes must be configured
  x axis={attribute=x},
  % The lengths of the axes
           ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/width}},
  left axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}},
  right axis={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}},
  % The styling of the axes
  every axis/.style={style=black!50}, % make this the default
  % Visualizing the axes themselves
  left axis= {visualize axis={x axis=
                                        {goto=padded min}, style=red!75, padded}},
                                       {goto=padded max}, style=blue!75,padded}},
 right axis={visualize axis={x axis=
  x axis=
             {visualize axis={left axis={goto=padded min}, padded},
              visualize axis={left axis={goto=padded max}, padded}},
  % Visualizing the grid, when requested
             {visualize grid={direction axis=left axis}},
  left axis= {visualize grid={direction axis=x axis, common={style=red!50}}},
  right axis={visualize grid={direction axis=x axis, common={style=blue!50}}},
  % Visualizing the ticks, when requested
  left axis={visualize ticks={style={red!50!black}, direction axis=x axis,
                              x axis={goto=padded min}, high=Opt, tick text at low}},
 right axis={visualize ticks={style={blue!80!black}, direction axis=x axis,
                              x axis={goto=padded max}, low=Opt, tick text at high}},
  x axis={visualize ticks={direction axis=left axis, left axis={goto=padded min}, high=0pt,
                           tick text at low}.
          visualize ticks={direction axis=left axis, left axis={goto=padded max}, low=0pt}},
  % By default, there are ticks on all axes
  all axes={ticks},
  % Visualizing the axis labels, when requested
 x axis={visualize label={x axis={goto pos=.5}, node style={
        at={(0,0 |- data visualization bounding box.south)}, below}}},
  left axis={visualize label={left axis={goto pos=.5}, node style={
        at={(0,0 -| data visualization bounding box.west)}, rotate=90, anchor=south, red!50!black}}},
  right axis={visualize label={right axis={goto pos=.5}, node style={
        at={(0,0 -| data visualization bounding box.east)}, rotate=-90, anchor=south, blue!80!black}}},
}}
```



```
\tikz \datavisualization [
 our system.
 x axis={attribute=time, label=Year,
   ticks={tick text padding=2pt, style={/pgf/number format/set thousands separator=}}},
 left axis={attribute=money, label=Spending,
   padding min=0, include value=0, grid,
   ticks={tick prefix=\$, style={/pgf/number format/fixed,
        /pgf/number format/fixed zerofill, /pgf/number format/precision=2}}},
 right axis={attribute=people,
   label=Population,
   padding min=0, include value=0,
   ticks={style=/pgf/number format/fixed}},
 visualize as line/.list={
   people 1, people 2, money 1, money 2},
 people 1={style={visualizer color=blue}},
 people 2={style={visualizer color=blue!50}},
 money 1={style={visualizer color=red}};
 money 2={style={visualizer color=red!50}} ]
data group {people and money};
```

59.5.7 Using the New Axis System Key

The axis system our system that we developed in the course of the previous section is not yet "configurable." The only configuration that was possible was to "misuse" the width and height keys of the scientific axes.

In order to make our system configurable so that we can say our system= $\langle options \rangle$, where $\langle options \rangle$ are executed with the path prefix

```
/tikz/data visualization/our system
```

we can use the following key:

The new axis system key takes four parameters. The first one, $\langle system \ name \rangle$, is the name of the to-be-created axis system, our system in our case. The new axis system will create the following new key:

```
/\text{tikz/data visualization}/\langle axis system name \rangle = \langle options \rangle (no default)
```

When the key $\langle axis\ system\ name \rangle$ is used, the following keys will be executed in the following order:

- 1. The \(\lambda axis \ setup\rangle\) with the path prefix \(/\text{tikz/data visualization/}\).
- 2. The $\langle default\ options \rangle$ with the same path prefix.
- 3. The following style:

```
/tikz/data visualization/every (axis system name) (style, no value)
```

Even though this style has the path prefix /tikz/data visualization itself, the keys stored in this style will be executed with the path prefix /tikz/data visualization/ $\langle axis system name \rangle$.

- 4. The $\langle options \rangle$ with the path prefix /tikz/data visualization/ $\langle axis\ system\ name \rangle$.
- 5. The \(\lambda application \ options \rangle \) with the path prefix \(/\text{tikz/data visualization/}\)

Let us now have a look at what all of this means. First, the $\langle axis\ setup \rangle$ will contain all options that setup the axis system in all ways that need not be configured. For instance, the $\langle axis\ setup \rangle$ for the scientific axes will create an x axis and also a y axis (because these are always present), but will not setup the label visualization (because this can be configured in different ways). For our system, which cannot be configured at all, we would place all of our configuration in the $\langle axis\ setup \rangle$.

The $\langle default\ options \rangle$ can be used to pick default values that would usually be passed to the $\langle options \rangle$ of the newly created axis system. For instance, for scientific axis, the $\langle default\ options \rangle$ are set to outer ticks, standard labels, because these are the defaults.

Finally, the $\langle application\ options \rangle$ can be used to actually apply the configuration that has been chosen by the $\langle options \rangle$. The idea is that $\langle default\ options \rangle$, $\langle options \rangle$, and also every $\langle axis\ system\ name \rangle$ all have a chance of changing, re-changing and re-setting all sorts of styles and keys. Then, with the last change "winning," the resulting setting of a style can be executed, which may then cause a label visualization to be installed.

60 Visualizers

60.1 Overview

In a data visualization a long stream of data points is *visualized* using *visualizers*. Recall that it is the job of the axis systems as described in Section 59 to determine *where* data points are visualized. It is the job of the visualizers to determine *how* they are visualized.

The most basic and common visualizer is the *line visualizer*. It simply connects subsequent data points by straight lines to indicate either that the points on these lines interpolate between the real data points or the straight lines are used to indicate the order in which the data points appear. A different, more "conservative" visualizer is the *scatter visualizer* or *mark visualizer*, which just places a small mark at each data point. Such a visualizer does not imply any interpolation or ordering between the data points.

Visualizers may, however, also be more complicated. For instance, a visualizer used for a box plot could visualize a data point as a box with a median value, standard deviation, outliers, and other information; a rectangle visualizer might visualize data points as larger areas; a projection visualizer might visualize the projection of data points onto different axes; and so.

Creating a new visualizer is not quite trivial since a new PGF class needs to be implemented. Fortunately, using visualizers is much simpler: For each kind of visualizer there is a key that allows you to create such a visualizer. You can then use further keys to configure the visualizer and to connect it to the data.

In a data visualization multiple visualizers may exist at the same time. This happens in different situations:

A data visualization may contain several independent data sets that are to be visualized. There might
be a line plot, for which a line visualizer is used, and also a scatter plot, for which a scatter visualizer
would be used.

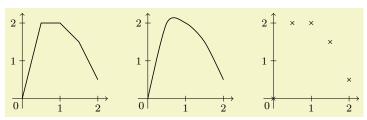
In this case, for each data point only one visualizer will do anything. To achieve this, each data point has an attribute called visualizer which tells the visualizer objects whether they should "react" to the data point or not.

A single data point might be visualized several times. For instance, a scatter visualizer might draw a
mark at the data point's position on the page and a projection visualizer might draw, additionally, a
mark at the projected position.

60.2 Usage

60.2.1 Using a Single Visualizer

The simplest scenario for using visualizers are data visualizations in which there is only a single data set that is visualized in one style. In this case, all that needs to be done in order to choose a visualizer is use one of the options starting with visualize as ... together with the \datavisualization command:



```
% Define a data set:
\tikz \datavisualization data group {example} = {
data {
  х, у
  0, 0
 0.5, 2
  1, 2
  1.5, 1.5
  2, 0.5
}};
\tikz \datavisualization [school book axes, visualize as line]
                                                                        data group {example};
\qquad
\tikz \datavisualization [school book axes, visualize as smooth line] data group {example};
\tikz \datavisualization [school book axes, visualize as scatter]
                                                                        data group {example};
```

Methods for styling visualizers are discussed in Section 60.2.3.

60.2.2 Using Multiple Visualizers

A data visualization may contain multiple data groups and for each data set we might wish to use a different visualizer. In this case, we need some way of telling the data visualization engine to which visualizer should be used with the different data points.

To solve this problem, you can *name* a visualizer. The visualizer's name can then both be used to configure the visualizer and also to indicate that data points "belong" to the visualizer.

Naming a visualizer is quite simple: The visualize as ... keys actually take a single parameter, which is the name of the visualizer. For instance, the following code creates three visualizers, named sin, cos, and tan:

```
visualize as line=sin,
visualize as line=cos,
visualize as scatter=tan
```

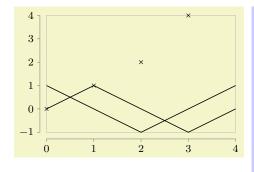
(When you just say visualize as line without providing a name, the name line is chosen as a default, for visualize as scatter the name scatter is the default and so.)

In order to indicate which data points should be visualized by which of these visualizers, the following key is important:

```
/data point/set (no value)
```

A visualizer will only act on a data point when its name matches the value of this key. Initially, this key is set to the last visualizer created, so if there is only one, there is no need to set or worry about this key.

Since the set key has the path prefix /data point, it can be set like any other attribute of a data key:



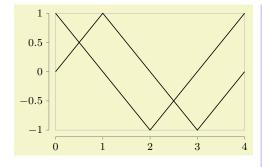
```
\tikz \datavisualization
 [scientific axes=clean,
 visualize as line=sin.
 visualize as line=cos.
 visualize as scatter=tan]
data {
 x, y, set
 0, 0, sin
 1, 1, sin
 2, 0, sin
 3, -1, \sin
 4, 0, sin
 0, 1, cos
 1. 0. cos
 0, 0, tan
 1, 1, tan
 2. 2. tan
 3, 4, tan
 2, -1, cos
 3, 0, cos
 4, 1, cos
```

As can be seen, the data points with the same set attribute do not need to be consecutive.

The above method of specifying the visualizer works nicely, but in most cases it would be more natural to keep the **set** attribute out of the table. This is easy to achieve by using multiple **data** and using the following key:

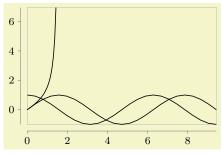
```
/pgf/data/set=\langle name \rangle (no default)
```

Shorthand for /data point/set= $\langle name \rangle$.



```
\tikz \datavisualization
 [scientific axes=clean.
  visualize as line=sin,
  visualize as line=cos]
data [set=sin] {
  х, у
  0, 0
  1, 1
  2, 0
  3. -1
  4, 0
}
data [set=cos] {
  х, у
  0, 1
  1, 0
  2, -1
  3, 0
  4,
     1
```

When you need to visualize several similar things in a single plot (like ten lines that all get visualized by visualize as line), it is somewhat cumbersome having to write this ten times. In this case you can shorten your code by making use of the .list key handler: When you add it to a key, the "value" passed to the key is parsed as a list of values. The key is then executed once for each of these values:



```
\tikz \datavisualization
[scientific axes=clean,
  visualize as line/.list={sin, cos, tan}]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
}
data [set=tan, format=function] {
  var x : interval[0:pi/2.2];
  func y = tan(\value x r);
};
```

60.2.3 Styling a Visualizer

In order to style a visualizer that has been created using for instance visualize as $line=\langle visualizer\ name \rangle$, you can use the following key:

```
/\text{tikz/data visualization}/\langle visualizer name \rangle = \langle options \rangle (no default)
```

For each visualizer, a key of the same name is created with the path prefix /tikz/data visualization. This key takes the $\langle options \rangle$ and executes them with the path prefix

```
/tikz/data visualization/visualizer options/
```

These options are then used to configure the appearance of the current visualizer. (This is quite similar to the way options are passed to an axis in order to configure the axis.) Possible options include style, but also label in legend and label in data. The latter two options are discussed in Section 61.3, the first option below.

```
\tikz \datavisualization
[scientific axes=clean,
  visualize as smooth line/.list={sin, cos},
  sin={style=red},
  cos={style=blue}]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
};
```

/tikz/data visualization/visualizer options/style=\langle options \rangle

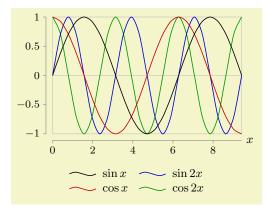
(no default)

The $\langle options \rangle$ given to this key should be normal TikZ options. They will be executed when the visualizer is used.

```
\begin{bmatrix} 1 \\ 0.5 \\ 0 \\ -0.5 \\ -1 \end{bmatrix}
\begin{bmatrix} 0 \\ 2 \\ 4 \\ 6 \\ 8 \end{bmatrix}
```

```
\tikz \datavisualization
[scientific axes=clean,
  visualize as smooth line=sin,
  sin={style={red, densely dotted}},
  visualize as smooth line=cos,
  cos={style={mark=x}},
]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
};
```

When you have multiple visualizers in a single data visualization, you can use the **style** option with each visualizer to configure their different appearances as in the above example. However, it is usually much better (and easier) to use a style sheet, see Section 61.



```
\tikz \datavisualization
 [scientific axes={clean, end labels},
  x axis={label=$x$}, y axis={grid={major also at=0}},
  visualize as smooth line/.list={sin,cos,sin 2,cos 2},
  legend={below, rows=2},
  sin={label in legend={text=$\sin x$}},
  cos={label in legend={text=$\cos x$}},
  sin 2={label in legend={text=$\sin 2x$}},
  cos 2={label in legend={text=$\setminus cos 2x$}},
  style sheet=strong colors]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\forall x r);
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\forall x r);
data [set=sin 2, format=function] {
  var x : interval[0:3*pi];
  func y = \sin(2*\langle value x r);
data [set=cos 2, format=function] {
  var x : interval[0:3*pi]:
  func y = cos(2*\langle value x r);
```

/tikz/data visualization/visualizer options/ignore style sheets

(no value)

This option, which should be passed to a visualizer after its creation before another visualizer is created, causes style sheets *not* to apply to the visualizer (but the style option will still have an effect). This allows you to create visualizers that are used for special purposes and that do not "take part" in the

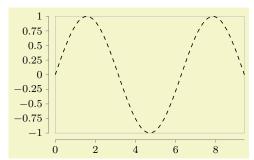
usual styling. For instance, a visualizer might be used internally to depict a regression line, even though the regression line itself should not participate in the usual styling by, say, dashing or different coloring.

In addition to the options passed to a visualizer via style, the following also gets executed when a visualizer is used:

/tikz/data visualization/every visualizer

(style, no value)

This style is used with every visualizer. Note that it should contain normal TikZ keys.



```
\tikz \datavisualization
[scientific axes=clean,
  every visualizer/.style={dashed},
  visualize as smooth line]
data [format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
};
```

60.3 Reference: Basic Visualizers

60.3.1 Visualizing Data Points Using Lines

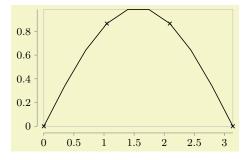
```
/tikz/data visualizers/visualize as line=(visualizer name)
```

(default line)

Creates a new visualizer named $\langle visualizer\ name \rangle$. Basically, this visualizer connects all data points for which the /data point/set attribute equals $\langle visualizer\ name \rangle$ by a line that is styled by the visualizer's style.

In more detail, the following happens:

- 1. A new object is created (of class plot handler visualizer) that is configured to collect the canvas positions of all data points whose set attribute equals (visualizer name).
- 2. During the end of the data visualization, PGF's plotting mechanism (see Section 89) is used to plot the stream of recorded data points.
 - This means that, in principle, all of the plot handlers available in TikZ could be used for the visualization (such as the smooth handler). However, some plot handlers such as, say, the xcomb are unsuitable as plot handlers since they do not support the advanced axis handling done by the data visualization engine. Because of this (and also for other reasons), you cannot set the plot handler directly, but must use one of the options like straight line, smooth line and others, documented in a moment.
- 3. Additionally, plot marks can be drawn at the collected data points. Here, all of the options available to TikZ for drawing plot marks are available. To configure them, all options offered by TikZ for configuring marks are available such as mark repeat:



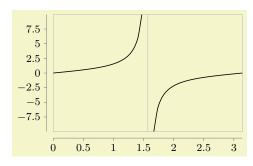
```
\tikz \datavisualization
[scientific axes=clean,
  visualize as line=my data,
  my data={style={mark=x, mark repeat=3}}]
data [format=function] {
  var x : interval [0:pi] samples 10;
  func y = sin(\value x r);
};
```

The line visualizer also provides a method of dealing with gaps in a line. Take for instance the function $f(x) = \tan x$. When this function is plotted over the interval $[0, \pi]$, then the function will go to $\pm \infty$ at $\pi/2$. When we plot this, we might plot the function in the interval $[0, \frac{\pi}{2} - \epsilon]$ and then continue in the interval $[\frac{\pi}{2} + \epsilon, \pi]$. However, we do not want the point at coordinate $(\frac{\pi}{2} - \epsilon, \tan(\frac{\pi}{2} - \epsilon))$ to be connected to the coordinate $(\frac{\pi}{2} + \epsilon, \tan(\frac{\pi}{2} + \epsilon))$ by a line. Rather, there should be a "gap" or a "jump" between these coordinates. To achieve this, the following key can be used:

/data point/outlier= $\langle value \rangle$

(default true, initially empty)

When this key is set to anything non-empty value, a visualizer will consider this data point to be an "outlier." For a line visualizer this means that the point is not shown and that the current line ends at the previous data point and a new line starts at the next data point.



```
\tikz \datavisualization
[scientific axes=clean, x axis={grid={major at=(pi/2)}},
  visualize as smooth line]
data [format=function] {
  var x : interval[0:pi/2-0.1];
  func y = tan(\value x r);
}
data point [outlier]
data [format=function] {
  var x : interval[pi/2+0.1:pi];
  func y = tan(\value x r);
};
```

/tikz/data visualizers/visualize as smooth line=\(visualizer name\)

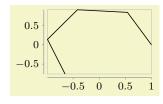
(default line)

A shorthand visualize as line= $\langle visualizer\ name \rangle$ followed $\langle visualizer\ name \rangle$ =smooth line.

/tikz/data visualization/visualizer options/straight line

(no value)

Causes the data points to be connected by straight lines.

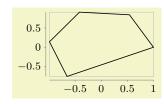


```
\tikz [scale=.55] \datavisualization
[scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data, my data={straight line}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/straight cycle

(no value)

Causes the data points to be connected by a polygon.



```
\tikz [scale=.55] \datavisualization
  [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data, my data={straight cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/polygon

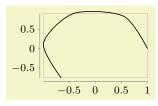
(no value)

This is an alias for straight cycle.

/tikz/data visualization/visualizer options/smooth line

(no value)

Causes the data points to be connected by a line that is smoothed at the joins:



```
\tikz [scale=.55] \datavisualization
[scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data, my data={smooth line}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/smooth cycle

(no value)

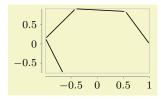
Causes the data points to be connected by a circular line that is smoothed at the joins:

```
\tikz [scale=.55] \datavisualization
[scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data, my data={smooth cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/gap line

(no value)

This key causes the data points to be connected by lines that "do not quite touch" the data points. This is implemented by using the \pgfplothandlergaplineto, see Section 44.5.

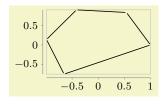


```
\tikz [scale=.55] \datavisualization
[scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data, my data={gap line}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/gap cycle

(no value)

Like gapped line, only with a cycle:

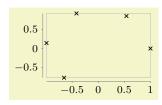


```
\tikz [scale=.55] \datavisualization
[scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data, my data={gap cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/no lines

(no value)

Suppresses the line. This option only makes sense when the mark option is used.



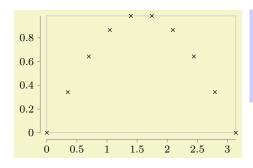
```
\tikz [scale=.55] \datavisualization
  [scientific axes=clean, all axes={ticks=few},
   visualize as smooth line=my data, my data={no lines, style={mark=x}}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

60.3.2 Visualizing Data Points Using Marks

/tikz/data visualizers/visualize as scatter= $\langle visualizer\ name \rangle$

(default scatter)

A shorthand visualize as line= $\langle visualizer\ name \rangle$ followed $\langle visualizer\ name \rangle$ =no lines and setting the style of the visualizer so that is will use mark=x (plus some size adjustments) to draw marks at the data points.



```
\tikz \datavisualization
[scientific axes=clean,
  visualize as scatter]
data [format=function] {
  var x : interval [0:pi] samples 10;
  func y = sin(\value x r);
};
```

60.4 Advanced: Creating New Visualizers

Creating a new visualizer is a two-stage process that does, unfortunately, require in-depth knowledge of the data visualization backend:

- 1. First, you need to create a new class using \pgfooclass whose instances react to the signal visualize datapoint signal. This requires detailed knowledge of the data visualization engine, see Section 62.
- 2. Second, you should provide keys on the TikZ level for creating the necessary objects. These keys invoke the key new visualizer internally.

/tikz/data visualization/new visualizer= $\{\langle name \rangle\}$ $\{\langle options \rangle\}$ $\{\langle legend\ entry\ options \rangle\}$ (no default) This key configures a new visualizer named $\langle name \rangle$. This entails the following actions:

- The key /tikz/data visualization/\(\langle name \rangle\) is created. As described earlier, this key can be used to pass for instance style options to the visualizer.
- The style key /tikz/data visualization/visualizers/ $\langle name \rangle$ /styling is created and made empty. This is the key in which the style key will store the options passed to the visualizer.
- The style key /tikz/data visualization/visualizers/ $\langle name \rangle$ /label in legend options is set to $\langle legend\ entry\ options \rangle$. These options are used to configure how the visualizer should be rendered in a legend, see Section 61.9.9 for details.
- The key /data point/set/ $\langle name \rangle$ is set to a number that is increased for each visualizer in the current data visualization. This number is important for style sheets, see Section 61.
- The key /data point/ $\langle name \rangle$ /execute at begin is set to code that creates a {scope} that executes the following styles as options:
 - 1. The *(options)* passed to the new visualizer key.
 - 2. The every visualizer style.
 - 3. The styling from the currently active style sheets, see Section 61.
 - 4. The styling stored in the styling key mentioned above.
- The key /data point/ $\langle name \rangle$ /execute at end is set to code that will finish all paths that may have been created by the visualizer and closes the scope.

All of the above mean the following in practice:

- Inside a new visualize as ... key, you pass the name of the to-be-created to new visualizer as the first parameter and any special default styling setup of the visualizer as the second parameter.
- The new visualize as ... key should also create a visualizer object using new object.
- When this object finally is about to create the actual visualization, it should surround the code by invoking the code stored in the execute at begin and the execute at end keys of the visualizer.

Everything else is usually taken care of by the new visualizer key automatically.

As an example, let us create a simple visualizer that creates a circle whose radius is dictated by the radius attribute. To keep things simple in this example, this attribute cannot be configured.

First, we need the visualizer class. For this example I have boiled it down to a minimum:

```
\pgfooclass{circle visualizer}
  % Stores the name of the visualizer. This is needed for filtering and configuration
  \attribute name:
  \mbox{\ensuremath{\it\%}} The constructor. Just setup the attribute.
  \method circle visualizer(#1) { \pgfooset{name}{#1} }
  % Connect to visualize signal.
  \method default connects() {
     \pgfoothis.get handle(\me)
     \pgfkeysvalueof{/pgf/data visualization/obj}.connect(\me,visualize,visualize datapoint signal)
  % This method is invoked for each data point. It checks whether the data point belongs to the correct
  % visualizer and, if so, calls the macro \dovisualization to do the actual visualization.
  \method visualize() {
    \pgfdvfilterpassedtrue
     \pgfdvnamedvisualizerfilter
    \ifpgfdvfilterpassed
      \dovisualization
    \fi
  }
}
```

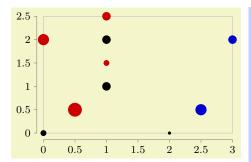
The \dovisualization method must now do the correct visualization.

```
\def\dovisualization{
  \pgfkeysvalueof{/data point/\pgfoovalueof{name}/execute at begin}
  \pgfpathcircle{\pgfpointdvdatapoint}{\pgfkeysvalueof{/data point/radius}}
  % \pgfusepath is done by |execute at end|
  \pgfkeysvalueof{/data point/\pgfoovalueof{name}/execute at end}
}
```

Finally, we create a visualize as key:

```
\tikzdatavisualizationset{
 visualize as circle/.style={
   new object={
     when=after survey,
     store=/tikz/data visualization/visualizers/#1,
     class=circle visualizer,
     arg1=#1
   },
   new visualizer={#1}{%
      color=visualizer color,
                                     % a color setup by the style sheet
      every path/.style={fill,draw}, % fill and draw the circle by default,
   \{\}, % let's ignore legends in this example
   /data point/set=#1
 },
  visualize as circle/.default=circle
```

Now, let's see how this works:



```
\tikz \datavisualization [
   scientific axes=clean,
   visualize as circle/.list={a, b, c},
   style sheet=strong colors]

data [set=a] {
    x, y, radius
    0, 0, 2pt
    1, 1, 3pt
    1, 2, 3pt
    2, 0, 1pt
}

data [set=b] {
    x, y, radius
    0.5, 0.5, 5pt
    1, 1.5, 2pt
    1, 2.5, 3pt
    0, 2, 4pt
}

data [set=c] {
    x, y, radius
    3, 2, 3pt
    2.5, 0.5, 4pt
};
```

61 Style Sheets and Legends

61.1 Overview

In many data visualizations, different sets of data need to be visualized in a single visualization. For instance, in a plot there might be a line for the sine of x and another line for the cosine of x; in another visualization there might be a set of points representing data from a first experiment and another set of points representing data from a second experiment; and so on. In order to indicate to which data set a data point belongs, one might plot the curve of the sine in, say, black, and the curve of the cosine in red; we might plot the data from the fist experiment using stars and the data from the second experiment using circles; and so on. Finally, at some place in the visualization – either inside the data or in a legend next to it – the meaning of the colors or symbols need to be explained.

Just as you would like TikZ to map the data points automatically onto the axes, you will also typically wish TikZ to choose for instance the coloring of the lines automatically for you. This is done using *style sheets*. There are at least two good reasons why you should prefer style sheets over configuring the styling of each visualizer "by hand" using the **style** key:

- 1. It is far more convenient to just say style sheet=strong colors than having to individually picking the different colors.
- 2. The style sheets were chosen and constructed rather carefully.

For instance, the **strong colors** style sheet does not pick colors like pure green or pure yellow, which have very low contrast with respect to a white background and which often lead to unintelligible graphics. Instead, opposing primary colors with maximum contrast on a white background were picked that are visually quite pleasing.

Similarly, the different dashing style sheets are constructed in such a way that there are only few and small gaps in the dashing so that no data points get lost because the dashes are spaced too far apart. Also dashing patterns were chosen that have a maximum optical difference.

As a final example, style sheets for plot marks are constructed in such a way that even when two plot marks lie directly on top of each other, they are still easily distinguishable.

The bottom line is that whenever possible, you should use one of the predefined style sheets rather than picking colors or dashings at random.

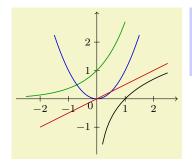
61.2 Concepts: Style Sheets

A style sheet is a predefined list of styles such as a list of colors, a list of dashing pattern, a list of plot marks, or a combinations thereof. A style sheet can be attached to a data point attribute. Then, the value of this attribute is used with data points to choose which style in the list should be chosen to visualize the data point.

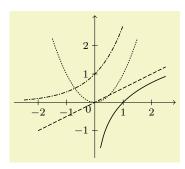
In most cases, there is just one attribute to which style sheets get attached: the /data point/visualizer attribute. The effect of attaching a style sheet to this attribute is that each visualizer is styled differently.

For the following examples, let us first define a simple data set:

```
\tikz \datavisualization data group {function classes} = {
  data [set=log, format=function] {
    var x : interval [0.2:2.5];
    func y = ln(\value x);
  }
  data [set=lin, format=function] {
    var x : interval [-2:2.5];
    func y = 0.5*\value x;
  }
  data [set=squared, format=function] {
    var x : interval [-1.5:1.5];
    func y = \value x*\value x;
  }
  data [set=exp, format=function] {
    var x : interval [-2.5:1];
    func y = exp(\value x);
  }
};
```



```
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=strong colors]
data group {function classes};
```



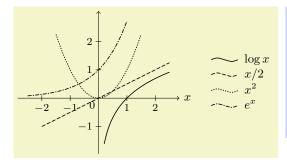
```
\tikz \datavisualization [
   school book axes, all axes={unit length=7.5mm},
   visualize as smooth line/.list={log, lin, squared, exp},
   style sheet=vary dashing]
data group {function classes};
```

61.3 Concepts: Legends

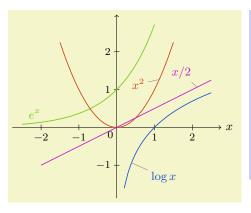
A legend is a box that is next to a data visualization (or inside it at some otherwise empty position) that contains a textual explanation of the different colors or styles used in a data visualization.

Just as it is difficult to get colors and dashing patterns right "by hand," it is also difficult to get a legend right. For instance, when a small line is shown in the legend that represents the actual line in the data visualization, if the line is too short and the dashing is too large, it may be impossible to discern which dashing is actually meant. Similarly, when plot marks are shown on such a short line, using a simple straight line may make it hard to read the plot marks correctly.

The data visualization engine makes some effort to make it easy to create high-quality legends. Additionally, it also offers ways of easily adding labels for visualizers directly inside the data visualization, which is even better than adding a legend, in general.



```
\tikz \datavisualization [
   school book axes, all axes={unit length=7.5mm},
   x axis={label=$x$},
   visualize as smooth line/.list={log, lin, squared, exp},
   log=   {label in legend={text=$\log x$}},
   lin=   {label in legend={text=$x/2$}},
   squared={label in legend={text=$x^2$}},
   exp=   {label in legend={text=$x^2$}},
   style sheet=vary dashing]
data group {function classes};
```



61.4 Usage: Style Sheets

61.4.1 Picking a Style Sheet

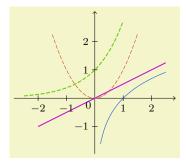
To use a style sheet, you need to *attach* it to an attribute. You can attach multiple style sheets to an attribute and in this case all of these style sheets can influence the appearance of the data points.

Most of the time, you will attach a style sheet to the **set** attribute. This has the effect that each different data set inside the same visualization is rendered in a different way. Since this use of style sheets is the most common, there is a special, easy-to-remember option for this:

/tikz/data visualization/style sheet= $\langle style \ sheet \rangle$

(no default)

Adds the $\langle style \ sheet \rangle$ to the list of style sheets attached to the set attribute.



```
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=vary thickness and dashing,
  style sheet=vary hue]
data group {function classes};
```

While the style sheet key will attach a style sheet only to the set attribute, the following key handler can be used to attach a style sheet to an arbitrary attribute:

Key handler $\langle key \rangle / .style sheet = \langle style sheet \rangle$

Inside a data visualization you can use this key handler together with an attribute, that is, with a key having the path prefix /data point. For instance, in order to attach the \(\style \) strong colors to the attribute set, you could write

/data point/set/.style sheet=strong colors

Indeed, the style sheet key is just a shorthand for the above.

The effect of attaching a style sheet is the following:

- A new object is created that will monitor the attribute.
- Each time a special *styling key* is emitted by the data visualization engine, this object will inspect the current value of the attribute to which it is attached.
- Depending on this value, one of the styles stored in the style sheet is chosen (how this works, exactly, will be explained in a moment).
- The chosen style is then locally applied.

In reality, things are a bit more complicated: If the attribute of the data point happens to have a subkey named in the same way as the value, then the value of is this subkey is used instead of the value itself. This allows you to "rename" a value.

In a sense, a style sheet behaves much like a visualizer (see Section 60): In accordance with the value of a certain attribute, the appearance of data points change. However, there are a few differences: First, the styling of a data point needs to be triggered explicitly and this triggering is not necessarily done for each data point individually, but only for a whole visualizer. Second, styles can be computed even when no data point is present. This is useful for instance in a legend since, here, a visual representation of a visualizer needs to be created independently of the actual data points.

61.4.2 Creating a New Style Sheet

Creating a style sheet works as follows: For each possible value that an attribute can attain we must specify a style. This is done by creating a style key for each such possible value with a special path prefix and setting this style key to the desired value. The special path prefix is /pgf/data visualization/style sheets followed by the name of the style sheet.

As an example, suppose we wish to create a style sheet test that makes styled data points red when the attribute has value foo and green when the attribute has value bar and dashed, blue when the attribute is foobar. We could then write

```
/pgf/data visualization/style sheets/test/foo/.style={red},
/pgf/data visualization/style sheets/test/bar/.style={green},
/pgf/data visualization/style sheets/test/foobar/.style={dashed, blue},
```

We could then attach this style sheet to the attribute code as follows:

```
/data point/code/.style sheet=test
```

Then, when /data point/code=foobar holds when the styling signal is raised, the stying dashed, blue will get executed.

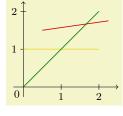
A natural question arises concerning the situation that the value of the attribute is not defined as a subkey of the style sheet. In this case, a special key gets executed:

```
\verb|/pgf/data visualization/style sheets/| & sheet| & | default style = | & value| & (style, no default) \\
```

This key gets during styling whenever /pgf/data visualization/style sheet/ $\langle style \ sheet \rangle / \langle value \rangle$ is not defined.

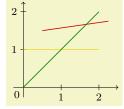
Let us put all of this together in a real-life example. Suppose we wish to create a style sheet that makes the first data set green, the second yellow and the third one red. Further data sets should be, say, black. The attribute that we intend to style is the set attribute. For the moment, we assume that the data sets will be named 1, 2, 3, and so on (instead of, say, experiment 1 or sin or something more readable – we will get rid of this restriction in a minute).

We would now write:



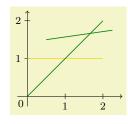
```
/pgf/data visualization/style sheets/traffic light/.cd,
  % All these styles have the above prefix.
  1/.style={green!50!black},
  2/.style={yellow!90!black},
  3/.style={red!80!black},
  default style/.style={black}
\tikz \datavisualization [
  school book axes,
 visualize as line=1,
  visualize as line=2,
 visualize as line=3,
  style sheet=traffic light]
data point [x=0, y=0, set=1]
data point [x=2, y=2, set=1]
data point [x=0, y=1, set=2]
data point [x=2, y=1, set=2]
data point [x=0.5, y=1.5, set=3]
data point [x=2.25, y=1.75, set=3];
```

In the above example, we have to name the visualizers 1, 2, 3 and so one since the value of the set attribute is used both assign data points to visualizers and also pick a style sheet. However, it would be much nicer if we could name any way we want. To achieve this, we use the special rule for style sheets that says that if there is a subkey of an attribute whose name is the same name as the value, then the value of this key is used instead. This slightly intimidating definition is much easier to understand when we have a look at an example:



```
% Definition of traffic light keys as above
\begin{tikzpicture}
  \datavisualization data group {lines} = {
    data point [x=0, y=0,
                                set=normal]
    data point [x=2, y=2,
                                set=normal]
    data point [x=0, y=1,
                                set=heated]
    data point [x=2, y=1,
                                set=heated]
    data point [x=0.5, y=1.5,
                                set=critical]
    data point [x=2.25, y=1.75, set=critical]
  \datavisualization [
    school book axes,
    visualize as line=normal,
    visualize as line=heated,
    visualize as line=critical,
    /data point/set/normal/.initial=1,
    /data point/set/heated/.initial=2,
    /data point/set/critical/.initial=3,
    style sheet=traffic light]
  data group {lines};
\end{tikzpicture}
```

Now, it is a bit bothersome that we have to set all these /data point/set/... keys by hand. It turns out that this is not necessary: Each time a visualizer is created, a subkey of /data point/set with the name of the visualizer is created automatically and a number is stored that is increased for each new visualizer in a data visualization. This means that the three lines starting with /data point are inserted automatically for you, so they can be left out. However, you would need them for instance when you would like several different data sets to use the same styling:



```
% Definition of traffic light keys as above
\tikz \datavisualization [
   school book axes,
   visualize as line=normal,
   visualize as line=heated,
   visualize as line=critical,
   /data point/set/critical/.initial=1, % same styling as first set
   style sheet=traffic light]
data group {lines};
```

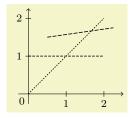
We can a command that slightly simplifies the definition of style sheets:

$\protect\operatorname{\colored} {\colored} {\colored}$

This command executes the $\langle keys \rangle$ with the path prefix /pgf/data visualization/style sheets/ $\langle name \rangle$. The above definition of the traffic light style sheet could be rewritten as follows:

```
\pgfdvdeclarestylesheet{traffic light}{
  1/.style={green!50!black},
  2/.style={yellow!90!black},
  3/.style={red!80!black},
  default style/.style={black}
}
```

As a final example, let us create a style sheet that changes the dashing pattern according to the value of the attribute. We do not need to define an large number of styles in this case, but can use the default style key to "calculate" the correct dashing.



```
\pgfdvdeclarestylesheet{my dashings}{
  default style/.style={dash pattern={on #1pt off 1pt}}
}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=my dashings]
data group {lines};
```

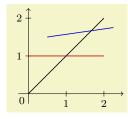
61.4.3 Creating a New Color Style Sheet

Creating a style sheet that varies colors according to an attribute works the same way as creating a normal style sheet: Subkeys lies 1, 2, and so on use the style attribute to setup a color. However, instead of using the color attribute to set the color, you should use the visualizer color key to set the color:

```
/tikz/visualizer color=\langle color \rangle
```

(no default)

This key is used to set the color visualizer color to $\langle color \rangle$. This color is used by visualizers to color the data they visualize, rather than the current "standard color." The reason for not using the normal current color is simply that it makes many internals of the data visualization engine a bit simpler.

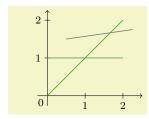


```
\pgfdvdeclarestylesheet{my colors}
{
  default style/.style={visualizer color=black},
  1/.style={visualizer color=black},
  2/.style={visualizer color=red!80!black},
  3/.style={visualizer color=blue!80!black},
}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=my colors]
data group {lines};
```

There is an additional command that makes it easy to define a style sheet based on a *color series*. Color series are a concept from the xcolor package: The idea is that we start with a certain color for the first data set and then add a certain "color offset" for each next data point. Please consult the documentation of the xcolor package for details.

```
\tikzdvdeclarestylesheetcolorseries{\langle name \rangle} {\langle color\ model \rangle} {\langle initial\ color \rangle} {\langle step \rangle}
```

This command creates a new style sheet using $\protect\protec$



```
\tikzdvdeclarestylesheetcolorseries{greens}{hsb}{0.3,1.3,0.8}{0,-.4,-.1}
\tikz \datavisualization [
    school book axes,
    visualize as line=normal,
    visualize as line=heated,
    visualize as line=critical,
    style sheet=greens]
data group {lines};
```

61.5 Reference: Style Sheets for Lines

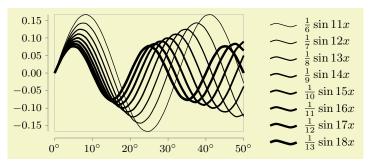
The following style sheets can be applied to visualizations that use the visualize as line and related keys. For the examples, the following style and data set are used:

```
\tikzdatavisualizationset {
  example visualization/.style={
    scientific axes=clean,
    y axis={ticks={style={
           /pgf/number format/fixed,
           /pgf/number format/fixed zerofill,
           /pgf/number format/precision=2}}},
    x axis={ticks={tick suffix=${}^\circ$}},
    1={label in legend={text=$\frac{1}{6}\sin 11x$}},
    2=\{label in legend=\{text=\$frac\{1\}\{7\}\sin 12x$\}\},
    3={label in legend={text=$\frac{1}{8}\sin 13x$}},
    4={label in legend={text=$\frac{1}{9}\sin 14x$}},
    5={label in legend={text=$\frac{1}{10}\sin 15x$}},
    6={label in legend={text=\frac{1}{11} \sin 16x}},
    7={label in legend={text=$\frac{1}{12}\sin 17x$}},
     8= \{label in legend= \{text=\$ frac \{1\} \{13\} \} in 18x \} \} 
}
```

```
\tikz \datavisualization data group {sin functions} = {
  data [format=function] {
    var set : {1,...,8};
    var x : interval [0:50];
    func y = sin(\value x * (\value{set}+10))/(\value{set}+5);
  }
};
```

Style sheet vary thickness

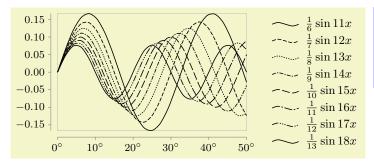
This style varies the thickness of lines. It should be used only when there are only two or three lines, and even then it is not particularly pleasing visually.



```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
    example visualization,
    style sheet=vary thickness]
data group {sin functions};
```

Style sheet vary dashing

This style varies the dashing of lines. Although it is not particularly pleasing visually and although visualizations using this style sheet tend to look "excited" (but not necessarily "exciting"), this style sheet is often the best choice when the visualization is to be printed in black and white.

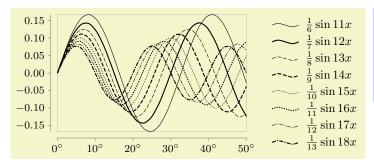


\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=vary dashing]
data group {sin functions};

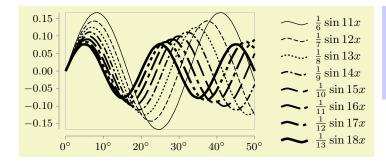
As can be seen, there are only seven distinct dashing patterns. The eighth and further lines will use a solid line once more. You will then have to specify the dashing "by hand" using the style option together with the visualizer.

Style sheet vary dashing and thickness

This style alternates between varying the thickness and the dashing of lines. The difference to just using both the vary thickness and vary dashing is that too thick lines are avoided. Instead, this style creates clearly distinguishable line styles for many lines (up to 14) with a minimum of visual clutter. This style is the most useful for visualizations when many different lines (ten or more) should be printed in black and white.



For comparison, here is the must-less-than-satisfactory result of combining the two independent style sheets:



\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=vary thickness,
 style sheet=vary dashing]
data group {sin functions};

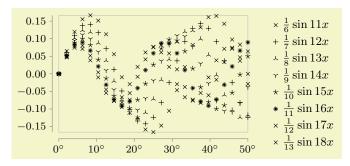
61.6 Reference: Style Sheets for Scatter Plots

The following style sheets can be used both for scatter plots and also with lines. In the latter case, the marks are added to the lines.

Style sheet cross marks

This style uses different crosses to distinguish between the data points of different data sets. The crosses were chosen in such a way that when two different cross marks lie at the same coordinate, their overall shape allows one to still uniquely determine which marks are on top of each other.

This style supports only up to six different data sets.



\tikz \datavisualization [
 visualize as scatter/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=cross marks]
data group {sin functions};

```
0.15
                                                                                                   \frac{1}{6}\sin 11x
  0.10
                                                                                                   \frac{1}{7}\sin 12x
  0.05
                                                                                                   \frac{1}{8}\sin 13x
  0.00
                                                                                                  \frac{1}{9}\sin 14x
                                                                                                   \frac{1}{10}\sin 15x
-0.05
                                                                                                   \frac{1}{11}\sin 16x
-0.10
                                                                                                   \frac{1}{12}\sin 17x
-0.15
                                                                                                   \frac{1}{13}\sin 18x
                        10°
                                                               40
```

\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=cross marks]
data group {sin functions};

61.7 Reference: Color Style Sheets

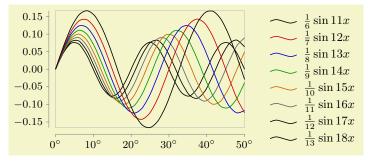
Color style sheets are very useful for creating visually pleasing data visualizations that contain multiple data sets. However, there are two things to keep in mind:

- At some point, every data visualization is printed or photo copied in black and white by someone. In this case, data sets can often no longer be distinguished.
- A few people are color blind. They will not be able to distinguish between red and green lines (and some people are not even able to distinguish colors at all).

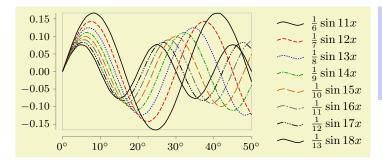
For these reasons, if there is any chance that the data visualization will be printed in black and white at some point, consider combining color style sheets with style sheets like vary dashing to make data sets distinguishable in all situations.

Style sheet strong colors

This style sheets uses pure primary colors that can very easily be distinguished. Although not as visually pleasing as the vary hue style sheet, the visualizations are easier to read when this style sheet is used. Up to six different data sets are supported.



\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=strong colors]
data group {sin functions};

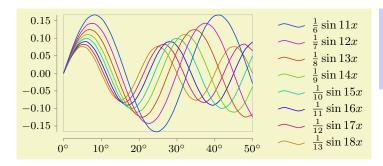


\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=strong colors,
 style sheet=vary dashing]
data group {sin functions};

Unlike strong colors, the following style sheets support, in principle, an unlimited number of data set. In practice, as always, more than four or five data sets lead to nearly indistinguishable data sets.

Style sheet vary hue

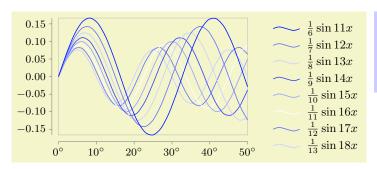
This style uses a different hue for each data set.



\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=vary hue]
data group {sin functions};

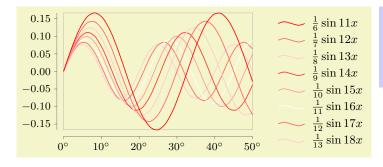
Style sheet shades of blue

As the name suggests, different shades of blue are used for different data sets.



\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=shades of blue]
data group {sin functions};

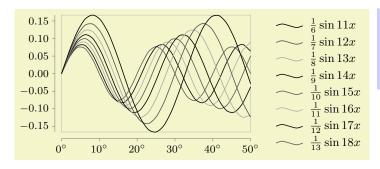
Style sheet shades of red



\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=shades of red]
data group {sin functions};

Style sheet gray scale

For once, this style sheet can also be used when the visualization is printed in black and white.



\tikz \datavisualization [
 visualize as smooth line/.list=
 {1,2,3,4,5,6,7,8},
 example visualization,
 style sheet=gray scale]
data group {sin functions};

61.8 Usage: Labeling Data Sets Inside the Visualization

In a visualization that contains multiple data sets, it is often necessary to clearly point out which line or mark type corresponds to which data set. This can be done in the main text via a sentence like "the normal data (black) lies clearly below the critical values (red)," but it often a good idea to indicate data sets ideally directly inside the data visualization or directly next to it in a so-called legend.

The data visualization engine has direct support both for indicating data sets directly inside the visualization and also for indicating them in a legend.

The "best" way of indicating where a data set lies or which color is used for it is to put a label directly inside the data visualization. The reason this is the "best" way is that people do not have to match the legend entries against the data, let alone having to look up the meaning of line styles somewhere in the text. However, adding a label directly inside the visualization is also the most tricky way of indicating data sets since it is hard to compute good positions for the labels automatically and since there needs to be some empty space where the label can be put.

61.8.1 Placing a Label Next to a Data Set

The following key is used to create a label inside the data visualization for a data set:

/tikz/data visualization/visualizer options/label in data=(options) (no default)

This key is passed to a visualizer that has previously been created using keys starting visualize as It will create a label inside the data visualization "next" to the visualizer (the details are explained in a moment). You can use this key multiple times with a visualizer to create multiple labels at different points with different texts.

The $\langle options \rangle$ determine which text is shown and where it is shown. They are executed with the following path prefix:

/tikz/data visualization/visualizer label options

In order to configure which text is shown and where, use the following keys inside the $\langle options \rangle$:

/tikz/data visualization/visualizer label options/text= $\langle text \rangle$ (no default)

This is the text that will be displayed next to the data. It will be to the "left" of the data, see the description below.

```
/tikz/data visualization/visualizer label options/text'=\langle text \rangle (no default)
```

Like text, only the text will be to the "right" of the data.

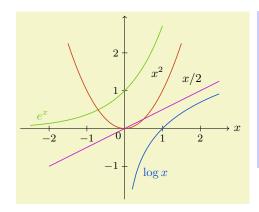
The following keys are used to configure where the label will be shown. They use different strategies to specify one data point where the label will be anchored. The coordinate of this data point will be stored in (label visualizer coordinate). Independently of the strategy, once the data point has been chosen, the coordinate of the next data point is stored in (label visualizer coordinate). Then, a (conceptual) line is created from the first coordinate to the second and a node is placed at the beginning of this line to its "left" or, for the text' option, on its "right." More precisely, an automatic anchor is computed for a node placed implicitly on this line using the auto option or, for the text' option, using auto, swap.

The node placed at the position computed in this way will have the $\langle text \rangle$ set by the text or text' option and its styling is determined by the current node style.

Let us now have a look at the different ways of determining the data point at which the label in anchored:

```
/tikz/data visualization/visualizer label options/when=\langle attribute \rangle is\langle number \rangle (no default)
```

This key causes the value of the $\langle attribute \rangle$ to be monitored in the stream of data points. The chosen is data point is the first data point where the $\langle attribute \rangle$ is at least $\langle number \rangle$ (if this never happens, the last data point is used).

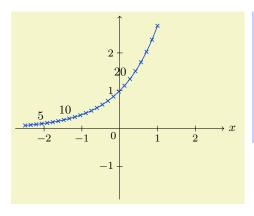


```
\tikz \datavisualization [
  school book axes.
  x axis={label=$x$}
  visualize as smooth line/.list={log, lin, squared, exp},
          {label in data={text'=$\log x$, when=y is -1,
  log=
                          text colored}},
          {label in data={text=$x/2$,
                                           when=x is 2},
  lin=
  squared={label in data={text=$x^2$,
                                           when=x is 1.1},
  exp=
          {label in data={text=$e^x$,
                                           when=x is -2,
                          text colored}},
  style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/visualizer label options/index= $\langle number \rangle$

(no default)

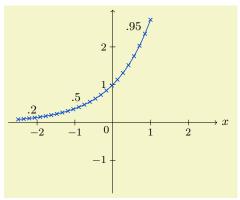
This key chooses the $\langle number \rangle$ th data point belonging to the visualizer's data set.



```
\tikz \datavisualization [
   school book axes,
   x axis={label=$x$},
   visualize as smooth line/.list={exp},
   exp=   {label in data={text=$5$, index=5},
        label in data={text=$10$, index=10},
        label in data={text=$20$, index=20},
        style={mark=x}},
   style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/visualizer label options/pos= $\langle fraction \rangle$ (no default)

This key chooses the first data point belonging to the data set whose index is at least $\langle fraction \rangle$ times the number of all data points in the data set.



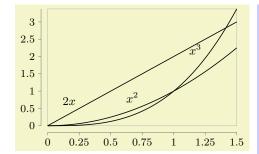
```
\tikz \datavisualization [
   school book axes,
   x axis={label=$x$},
   visualize as smooth line=exp,
   exp= {label in data={text=$.2$, pos=0.2},
        label in data={text=$.5$, pos=0.5},
        label in data={text=$.95$, pos=0.95},
        style={mark=x}},
   style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/visualizer label options/auto

(no value)

This key is executed automatically by default. It works like the pos option, where the $\langle fraction \rangle$ is set to $(\langle data\ set's\ index \rangle - 1/2)/\langle number\ of\ data\ sets \rangle$. For instance, when there are 10 data sets, the fraction for the first one will be 5%, the fraction for the second will be 15%, for the third it will be 25%, ending with 95% for the last one.

The net effect of all this is that when there are several lines, labels will be placed at different positions along the lines with hopefully only little overlap.



```
\tikz \datavisualization [
    scientific axes=clean,
    visualize as smooth line/.list={linear, squared, cubed},
    linear ={label in data={text=$2x$}},
    squared={label in data={text=$x^2$}},
    cubed ={label in data={text=$x^3$}}

data [set=linear, format=function] {
    var x : interval [0:1.5];
    func y = 2*\value x;
}

data [set=squared, format=function] {
    var x : interval [0:1.5];
    func y = \value x * \value x;
}

data [set=cubed, format=function] {
    var x : interval [0:1.5];
    func y = \value x * \value x * \value x;
};

func y = \value x * \value x * \value x;
};
```

As can be seen in the example, the result is not always satisfactory. In this case, the pin in data option might be preferable, see below.

The following keys allow you to style labels.

```
/tikz/data visualization/visualizer label options/node style=(options) (no default)
```

Just passes the options to /tikz/data visualization/node style.

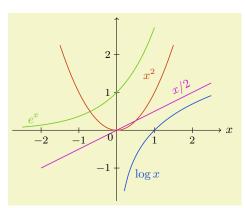
```
/tikz/data visualization/visualizer label options/text colored (no value)
```

Causes the node style to set the text color to visualizer color. The effect of this is that the label's text will have the same color as the data set to which it is attached.

```
/tikz/data visualization/every data set label
```

(style, no value)

This style is executed with every label that represents a data set. Inside this style, use node style to change the appearance of nodes. This style has a default definition, usually you should just append things to this style.



```
\tikz \datavisualization [
  school book axes.
  x axis={label=$x$}
 visualize as smooth line/.list={log, lin, squared, exp},
  every data set label/.append style={text colored},
  log=
          {label in data={text'=\frac{x}{y}, when=y is -1}},
          {label in data={text=$x/2$,
  lin=
                    node style=sloped,
                                           when=x is 2}}.
  squared={label in data={text=$x^2$,
                                           when=x is 1.1},
          {label in data={text=$e^x$.
  exp=
                    node style=sloped,
                                           when=x is -2},
  style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/every label in data

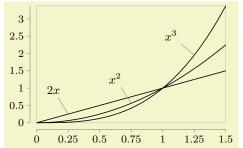
(style, no value)

Like every data set label, this key is also executed with labels. However, this key is executed after the style sheets have been executed, giving you a chance to overrule their styling.

61.8.2 Connecting a Label to a Data Set via a Pin

```
/tikz/data visualization/visualizer options/pin in data=\langle options \rangle (no default)
```

This key is a variant of the label in data key and takes the same options, plus two additional ones. The difference to label in data is that the label node is shown a bit removed from the data set, but connected to it via a small line (this is like the difference between the label and pin options).



```
\tikz \datavisualization [
    scientific axes=clean,
    visualize as smooth line/.list={linear, squared, cubed},
    linear ={pin in data={text=$2x$}},
    squared={pin in data={text=$x^2$}},
    cubed ={pin in data={text=$x^3$}}]

data [set=linear, format=function] {
    var x : interval [0:1.5];
    func y = \value x;
}

data [set=squared, format=function] {
    var x : interval [0:1.5];
    func y = \value x * \value x;
}

data [set=cubed, format=function] {
    var x : interval [0:1.5];
    func y = \value x * \value x * \value x;
};

func y = \value x * \value x * \value x;
};
```

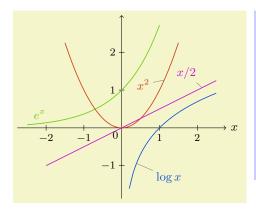
The following keys can be used additionally:

```
/tikz/data visualization/visualizer label options/pin angle=\langle angle \rangle (no default)
```

The position of the label of a pin in data is mainly computed in the same way as for a label in data. However, once the position has been computed, the label is shifted as follows:

- When an $\langle angle \rangle$ is specified using the present key, the shift is by the current value of pin length in the direction of $\langle angle \rangle$.
- When $\langle angle \rangle$ is empty (which is the default), then the shift is also by the current value of pin length, but now in the direction that is orthogonal and to the left of the line between the coordinate of the data point and the coordinate of the next data point. When text' is used, the direction is to the right instead of the left.

/tikz/data visualization/visualizer label options/pin length= $\langle dimension \rangle$ (no default) See the description of pin angle.



61.9 Usage: Labeling Data Sets Inside a Legend

The "classical" way of indicating the style used for the different data sets inside a visualization is a *legend*. It is a description next to or even inside the visualization that contains one line for each data set and displays an iconographic version of the data set next to some text labeling the data set. Note, however, that even though legend are quite common, also consider using a label in data or a pin in data instead.

Creating a high-quality legend is by no means simple. A legend should not distract the reader, so aggressive borders should definitively be avoided. A legend should make it easy to match the actual styling of a data set (like, say, using a red, dashed line) to the "iconographic" representation of this styling. An example of what can go wrong here is using short lines to represent lines dashed in different way where the lines are so short that the differences in the dashing cannot be discerned. Another example is showing straight lines with plot marks on them where the plot marks are obscured by the horizontal line itself, while the plot marks are clearly visible in the actual visualization since no horizontal lines occur.

The data visualization engine comes with a large set of options for creating and placing high-quality legends next or inside data visualizations.

61.9.1 Creating Legends and Legend Entries

A data visualization can be accompanied by one or more legends. In order to create a legend, the following key can be used (although, in practice, you will usually use the legend key instead, see below):

```
/tikz/data visualization/new legend=\langle legend name \rangle
```

(default main legend)

This key is used to create a new legend named $\langle legend\ name \rangle$. The legend is empty by default and further options are needed to add entries to it. When the key is called a second time for the same $\langle legend\ name \rangle$ nothing happens.

When a legend is created, a new key is created that can subsequently be used to configure the legend:

```
/tikz/data visualization/\langle legend name \rangle = \langle options \rangle
```

(no default)

When this key is used, the $\langle options \rangle$ are executed with the path prefix

```
/tikz/data visualization/legend options
```

The different keys with this path prefix allow you to change the position where the legend is shown and how it is organised (for instance, whether legend entries are shown in a row or in a column or in a square).

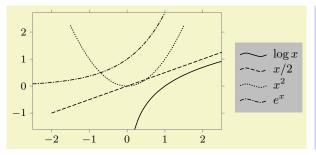
The different possible keys will be explained in the course of this section.

In the end, the legend is just a TikZ node, a matrix node, to be precise. The following key is used to style this node:

```
/tikz/data visualization/legend options/matrix node style=\langle options \rangle
```

(no default)

Adds the *(options)* to the list of options that will be executed when the legend's node is created.



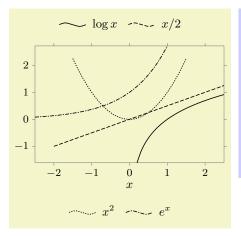
```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   legend={matrix node style={fill=black!25}},
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$}},
   squared={label in legend={text=$x^2$}},
   exp=      {label in legend={text=$x^2$}},
   style sheet=vary dashing]
data group {function classes};
```

The following style allows you to configure the default appearance of every newly created legend:

```
/tikz/data visualization/legend options/every new legend
```

(style, no value)

This key defaults to east outside, label style=text right. This means that by default a legend is placed to the right of the data visualization and that in the individual legend entries the text is to the right of the data set visualization.

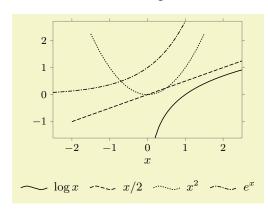


```
\tikz \datavisualization [
    scientific axes, x axis={label=$x$},
    visualize as smooth line/.list={log, lin, squared, exp},
    new legend={upper legend},
    new legend={lower legend},
    upper legend=above,
    lower legend=below,
    log=    {label in legend={text=$\log x$, legend=upper legend}},
    lin=    {label in legend={text=$x/2$, legend=upper legend}},
    squared={label in legend={text=$x^2$, legend=lower legend}},
    exp=    {label in legend={text=$x^2$, legend=lower legend}},
    style sheet=vary dashing]
data group {function classes};
```

/tikz/data visualization/legend=\langle options\rangle

(no default)

This is a shorthand for new legend=main legend, main legend= $\langle options \rangle$. In other words, this key creates a new main legend and immediately passes the configuration $\langle options \rangle$ to this legend.



```
\tikz \datavisualization [
   scientific axes, x axis={label=$x$},
   visualize as smooth line/.list={log, lin, squared, exp},
   legend=below,
   log= {label in legend={text=$\log x$}},
   lin= {label in legend={text=$x/2$}},
   squared={label in legend={text=$x^2$}},
   exp= {label in legend={text=$x^2$}},
   style sheet=vary dashing]
   data group {function classes};
```

As pointed out above, a legend is empty by default. In particular, the different data sets are not automatically inserted into the legend. Instead, the key label in legend must be used together with a data set:

```
/tikz/data visualization/visualizer options/label in legend=(options) (no default)
```

This key is passed to a data set, similar to options like pin in data or smooth line. The $\langle options \rangle$ are used to configure the following:

- The legend in which the data set should be visualized.
- The text that is to be shown in the legend for the data set.
- The appearance of the legend entries.

In detail, the $\langle options \rangle$ are executed with the path prefix

/tikz/data visualization/legend entry options

To configure in which legend the label should appear, use the following key:

```
/tikz/data visualization/legend entry options/legend=\langle name \rangle (no default, initially main legend)
```

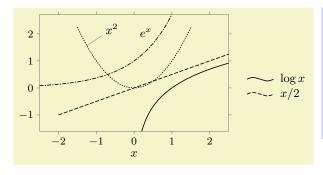
Set this key to the name of a legend that has previously been created using new legend. The label will then be shown in this legend.

In most cases, there is only one legend (namely main legend) and there is no need to set this key since it defaults to the main legend.

Also note that the legend $\langle name \rangle$ is automatically created if it nodes not yet exist.

```
tikz/data visualization/legend entry options/text=\langle text \rangle (no default)
```

Use this key to setup the $\langle text \rangle$ that is shown as the label of the data set.



```
\tikz \datavisualization [
   scientific axes, x axis={label=$x$},
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$}},
   squared={pin in data = {text=$x^2$, pos=0.1}},
   exp=      {label in data = {text=$e^x$}},
   style sheet=vary dashing]
data group {function classes};
```

In addition to the two keys described above, there are further keys that are described in Section 61.9.6.

Rows and Columns of Legend Entries

In a legend, the different legend entries are arranged in a matrix, which typically has only one row or one column. For the impatient reader: Say rows=1 to get everything in a row, say columns=1 to get everything in a single column, and skip the rest of this section.

The more patient reader will appreciate that when there are very many different data sets in a single visualization, it may be necessary to use more than one row or column inside the legend. TikZ comes with a rather powerful mechanism for distributing the multiple legend entries over the matrix.

The first thing to decide is in which "direction" the entries should be inserted into the matrix. Suppose we have a 3×3 matrix and our entries are a, b, c, and so on. Then, one might place the a in the upper left corner of the matrix, b in the upper middle position, c in the upper right position, and d in the middle left position. This is a "first right, then down" strategy. A different strategy might be to place the a in the upper left corner, but b in the middle left position, c in the lower left position, and d then in the upper middle position. This is a "first down, then right" strategy. In certain situations it might even make sense to place a in the lower right corner and then go "first up, then left".

All of these strategies are supported by the legend command. You can configure which strategy is used using the following keys:

```
/tikz/data visualization/legend options/down then right
```

(no value)

Causes the legend entries to fill the legend matrix first downward and, once a column is full, the next column is begun to the right of the previous one. This is the default.

```
\tikz \datavisualization [
 visualize as smooth line/.list={1,2,3,4,5,6,7,8},
 legend example, style sheet=vary hue,
main legend={down then right, columns=3}]
data group {sin functions};
```

In the example, the legend example is the following style:

```
\tikzdatavisualizationset {
  legend example/.style={
    scientific axes, all axes={length=1cm, ticks=none},
    1={label in legend={text=1}},
    2={label in legend={text=2}},
    3={label in legend={text=3}},
    4={label in legend={text=4}},
    5={label in legend={text=5}},
    6={label in legend={text=6}},
    7={label in legend={text=7}},
    8={label in legend={text=8}}
}
```

/tikz/data visualization/legend options/down then left

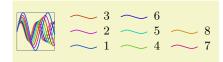
(no value)



```
\tikz \datavisualization [
tikz \datavisualization [
visualize as smooth line/.list={1,2,3,4,5,6,7,8},
legend example, style sheet=vary hue,
main legend={down then left, columns=3}]
                                        data group {sin functions};
```

/tikz/data visualization/legend options/up then right

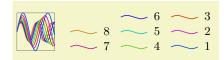
(no value)



```
\tikz \datavisualization [
 visualize as smooth line/.list={1,2,3,4,5,6,7,8},
 legend example, style sheet=vary hue,
 main legend={up then right, columns=3}]
data group {sin functions};
```

/tikz/data visualization/legend options/up then left

(no value)



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
 legend example, style sheet=vary hue,
main legend={up then left, columns=3}]
data group {sin functions};
```



```
data group {sin functions};
```

/tikz/data visualization/legend options/left then down

(no value)



```
\tikz \datavisualization [
visualize as smooth line/.list={1,2,3,4,5,6,7,8},
legend example, style sheet=vary hue,
main legend={left then down, columns=3}]
                                   data group {sin functions};
```

/tikz/data visualization/legend options/right then up

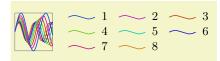
(no value)



```
\tikz \datavisualization [
7 8 \tikz \datavisualization [ visualize as smooth line/.list={1,2,3,4} legend example, style sheet=vary hue, main legend={right then up, columns=3}]
                                        visualize as smooth line/.list={1,2,3,4,5,6,7,8},
                                       data group {sin functions};
```

/tikz/data visualization/legend options/right then down

(no value)



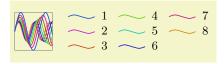
```
\tikz \datavisualization [
visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down, columns=3}]
data group {sin functions};
```

Having configured the directions in which the matrix is being filled, you must next setup the number of rows or columns that are to be shown. There are actually two different ways of doing so. The first way is to specify a maximum number of rows or columns. For instance, you might specify that there should be at most ten rows to a column and when there are more, a new column should be begun. This is achieved using the following keys:

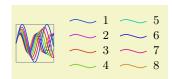
```
/tikz/data visualization/legend options/max rows=\langle number \rangle
```

(no default)

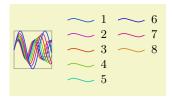
As the legend matrix is being filled, whenever the number of rows in the current column would exceed $\langle number \rangle$, a new column is started.



```
\tikz \datavisualization [
 visualize as smooth line/.list={1,2,3,4,5,6,7,8},
 legend example, style sheet=vary hue,
 main legend={max rows=3}]
data group {sin functions};
```



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
 legend example, style sheet=vary hue,
 main legend={max rows=4}]
data group {sin functions};
```

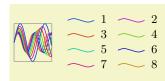


```
\tikz \datavisualization [
  visualize as smooth line/.list=\{1,2,3,4,5,6,7,8\},
  legend example, style sheet=vary hue,
 main legend={max rows=5}]
data group {sin functions};
```

/tikz/data visualization/legend options/max columns= $\langle number \rangle$

(no default)

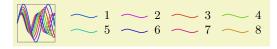
This key works like max rows, only now the number of columns is monitored. Note that this strategy only really makes sense when the when you use this key with a strategy that first goes left or right and then up or down.



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down, max columns=2}]
data group {sin functions};
```



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down,max columns=3}]
data group {sin functions};
```

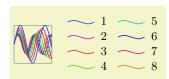


```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down,max columns=4}]
data group {sin functions};
```

The second way of specifying the number of entries in a row or column is to specify an "ideal number of rows or columns." The idea is as follows: Suppose that we use the standard strategy and would like to have everything in two columns. Then if there are eight entries, the first four should go to the first column, while the next four should go to the second column. If we have 20 entries, the first ten should go the first column and the next ten to the second, and so on. So, in general, the objective is to distribute the entries evenly so the this "ideal number of columns" is reached. Only when there are too few entries to achieve this or when the number of entries per column would exceed the max rows value, will the number of columns deviate from this ideal value.

```
/tikz/data visualization/legend options/ideal number of columns=\langle number \rangle (no default)
```

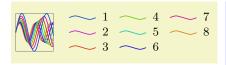
Specifies, that the entries should be split into $\langle number \rangle$ different columns, whenever possible. However, when there would be more than the max rows value of rows per column, more columns than the ideal number are created.



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of columns=2}]
data group {sin functions};
```



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of columns=4}]
data group {sin functions};
```



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={max rows=3,ideal number of columns=2}]
data group {sin functions};
```

/tikz/data visualization/legend options/rows= $\langle number \rangle$

(no default)

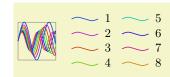
Shorthand for ideal number of rows= $\langle number \rangle$.

```
/tikz/data visualization/legend options/ideal number of rows=\langle number \rangle
```

Works like ideal number of columns.



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of rows=2}]
data group {sin functions};
```



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of rows=4}]
data group {sin functions};
```



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={max columns=3,ideal number of rows=2}]
data group {sin functions};
```

```
/tikz/data visualization/legend options/columns=\langle number \rangle
```

(no default)

(no default)

Shorthand for ideal number of columns= $\langle number \rangle$.

61.9.3 Legend Placement: The General Mechanism

A legend can either be placed next to the data visualization or inside the data visualization at some place where there are no data entries. Both approached have advantages: Placing the legend next to the visualization minimises the "cluttering" by keeping all the extra information apart from the actual data, while placing the legend inside the visualization minimises the distance between the data sets and their explanations, making it easier for the eye to connect them.

For both approaches there are options that make the placement easier, see Sections 61.9.4 and 61.9.5, but these options internally just map to the following two options:

```
/tikz/data visualization/legend options/anchor=\langle anchor \rangle
```

(no default)

The whole legend is a TikZ-matrix internally. Thus, in particular, it is stored in a node, which has anchors. Like for any other node, when the node is shown, the node is shifted in such a way that the $\langle anchor \rangle$ of the node lies at the current at position.

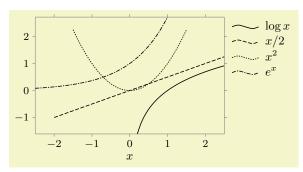
```
/tikz/data visualization/legend options/at=\(coordinate\)
```

(no default)

Configures the $\langle coordinate \rangle$ at which the $\langle anchor \rangle$ of the legend's node should lie.

It may seem hard to predict a good $\langle coordinate \rangle$ for a legend since, depending of the size of the axis, different positions need to the chosen for the legend. However, it turns out that one can often use the coordinates of the special nodes data bounding box and data visualization bounding box, documented in Section 57.6.

As an example, let us put a legend to the right of the visualization, but so that the first entry starts at the top of the visualization:



```
\tikz \datavisualization [
    scientific axes, x axis={label=$x$},
    visualize as smooth line/.list=
      {log, lin, squared, exp},
    legend={anchor=north west, at=
         (data visualization bounding box.north east)},
    log=      {label in legend={text=$\log x$}},
    lin=      {label in legend={text=$x/2$}},
    squared={label in legend={text=$x^2$}},
    exp=      {label in legend={text=$x^2$}},
    style sheet=vary dashing]
    data group {function classes};
```

As can be seen, a bit of an additional shift might have been in order, but the result is otherwise quite satisfactory.

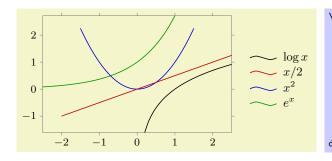
61.9.4 Legend Placement: Outside to the Data Visualization

The following keys make it easy to place a legend outside the data visualization.

/tikz/data visualization/legend options/east outside

(no value)

Placing the legend to the right of the data visualization is the default:



```
\tikz \datavisualization [
    scientific axes,
    visualize as smooth line/.list=
        {log, lin, squared, exp},
    legend=east outside,
    log=        {label in legend={text=$\log x$}},
    lin=        {label in legend={text=$x/2$}},
    squared={label in legend={text=$x^2$}},
    exp=        {label in legend={text=$x^2$}},
    style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/right

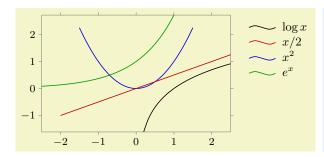
(no value)

This is an easier-to-remember alias.

/tikz/data visualization/legend options/north east outside

(no value)

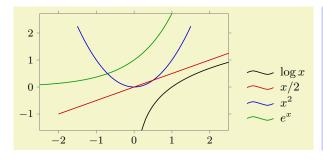
A variant, where the legend is to the right, but aligned with the northern end of the data visualization:



```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   legend=north east outside,
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$}},
   squared={label in legend={text=$x^2$}},
   exp=      {label in legend={text=$x^2$}},
   style sheet=strong colors]
   data group {function classes};
```

/tikz/data visualization/legend options/south east outside

(no value)

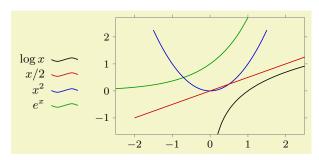


```
\tikz \datavisualization [
    scientific axes,
    visualize as smooth line/.list=
        {log, lin, squared, exp},
    legend=south east outside,
    log=        {label in legend={text=$\log x$}},
    lin=        {label in legend={text=$x/2$}},
    squared={label in legend={text=$x^2$}},
    exp=        {label in legend={text=$x^2$}},
    style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/west outside

(no value)

The legend is placed left. Note that the text also swaps its position.



```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   legend=west outside,
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$}},
   squared={label in legend={text=$x^2$}},
   exp=      {label in legend={text=$x^2$}},
   style sheet=strong colors]
   data group {function classes};
```

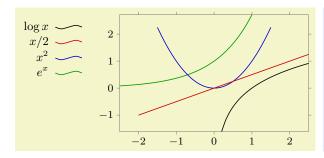
/tikz/data visualization/legend options/left

(no value)

This is an easier-to-remember alias.

/tikz/data visualization/legend options/north west outside

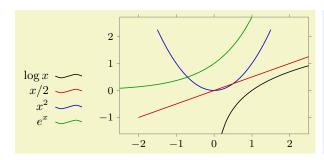
(no value)



```
\tikz \datavisualization [
    scientific axes,
    visualize as smooth line/.list=
        {log, lin, squared, exp},
    legend=north west outside,
    log=        {label in legend={text=$\log x$}},
    lin=        {label in legend={text=$x/2$}},
    squared={label in legend={text=$x^2$}},
    exp=        {label in legend={text=$x^2$}},
    style sheet=strong colors]
    data group {function classes};
```

/tikz/data visualization/legend options/south west outside

(no value)

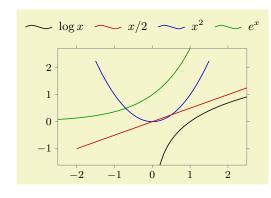


```
\tikz \datavisualization [
    scientific axes,
    visualize as smooth line/.list=
        {log, lin, squared, exp},
    legend=south west outside,
    log=        {label in legend={text=$\log x$}},
    lin=        {label in legend={text=$x/2$}},
    squared={label in legend={text=$x^2$}},
    exp=        {label in legend={text=$x^2$}},
    style sheet=strong colors]
    data group {function classes};
```

/tikz/data visualization/legend options/north outside

(no value)

The legend is placed above the data. Note that the legend entries now for a row rather than a column.



```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   legend=north outside,
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$}},
   squared={label in legend={text=$x^2$}},
   exp=      {label in legend={text=$x^2$}},
   style sheet=strong colors]
data group {function classes};
```

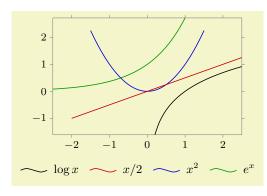
/tikz/data visualization/legend options/above

(no value)

This is an easier-to-remember alias.

/tikz/data visualization/legend options/south outside

(no value)



```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   legend=south outside,
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$}},
   squared={label in legend={text=$x^2$}},
   exp=      {label in legend={text=$x^2$}},
   style sheet=strong colors]
   data group {function classes};
```

/tikz/data visualization/legend options/below

(no value)

This is an easier-to-remember alias.

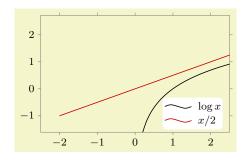
61.9.5 Legend Placement: Inside to the Data Visualization

There are two sets of options for placing a legend directly inside a data visualization: First, there are options for placing it inside, but next to some part of the border. Second, there are options for positioning it relative to a coordinate given by a certain data point.

/tikz/data visualization/legend options/south east inside

(no value)

Puts the legend in the upper right corner of the data.



```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin},
   legend=south east inside,
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$}},
   style sheet=strong colors]
data group {function classes};
```

Note that the text is now a little smaller since there tends to be much less space inside the data visualization than next to it. Also, the legend's node is filled in white by default to ensures that the legend is clearly legible even in the presence of, say, a grid or data points behind it. This behaviour is triggered by the following style key:

```
/tikz/data visualization/legend options/every legend inside
```

(style, no value)

Executed the keys opaque by default and sets the text size to the size of footnotes.

In order to further configure the default appearance of an inner legend, the following keys might be useful:

/tikz/data visualization/legend options/opaque= $\langle color \rangle$

(default white)

When this key is used, the legend's node will be filled with the $\langle color \rangle$ and its corners will be rounded. Additionally, the inner and outer separations will be set to sensible values.

/tikz/data visualization/legend options/transparent

(no value)

Sets the filling of the legend node to none.

The following keys work much the same way as south east inside:

/tikz/data visualization/legend options/east inside

(no value)

/tikz/data visualization/legend options/north east inside

(no value)

/tikz/data visualization/legend options/south west inside

(no value)

/tikz/data visualization/legend options/west inside

(no value)

/tikz/data visualization/legend options/north west inside

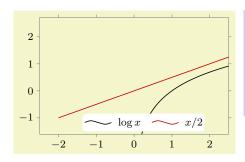
(no value)

The keys south inside and north inside are a bit different: They use a row rather than a column for the legend entries:

/tikz/data visualization/legend options/south inside

(no value)

Puts the legend in the upper right corner of the data. Note that the text is now a little smaller since there tends to be much less space inside the data visualization than next to it.



```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list={log, lin},
   legend=south inside,
   log= {label in legend={text=$\log x$}},
   lin= {label in legend={text=$x/2$}},
   style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/north inside

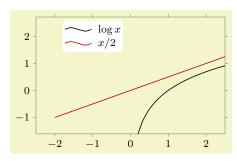
(no value)

As above

The above keys do not always give you as fine a control as you may need over the placement of the legend. In such cases, the following keys may help (or you can revert to directly setting the at and the anchor keys):

```
/tikz/data visualization/legend options/at values=\langle data \ point \rangle (no default)
```

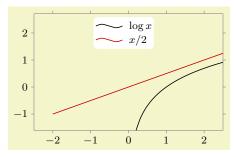
This key allows you to specify the desired center of the legend in terms of a data point. The $\langle data \ point \rangle$ should be a list of comma-separated key-value pairs that specify a data point. The legend will then be centered at this data point.



/tikz/data visualization/legend options/right of= $\langle data \; point \rangle$

(no default)

Works like at values, but the anchor is set to west:



The following keys work similarly:

```
\label{tikzdata} $$ / tikz/data \ visualization/legend \ options/above \ right \ of = \langle data \ point \rangle $$ (no \ default) $$ / tikz/data \ visualization/legend \ options/above \ left \ of = \langle data \ point \rangle $$ (no \ default) $$ / tikz/data \ visualization/legend \ options/left \ of = \langle data \ point \rangle $$ (no \ default) $$ / tikz/data \ visualization/legend \ options/below \ left \ of = \langle data \ point \rangle $$ (no \ default) $$ / tikz/data \ visualization/legend \ options/below \ of = \langle data \ point \rangle $$ (no \ default) $$ / tikz/data \ visualization/legend \ options/below \ right \ of = \langle data \ point \rangle $$ (no \ default) $$
```

61.9.6 Legend Entries: General Styling

The entries in a legend can be styled in several ways:

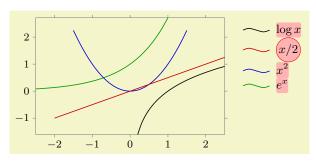
- You can configure the styling of the text node.
- You can configure the relative placement of the text node and the little picture depicting the data set's styling.
- You can configure how the data set's styling is depicted.

Before we have look at how each of these are configured, in detail, let us first have a look at the keys that allow us to save a set of such styles:

/tikz/data visualization/every label in legend

(style, no value)

This key is executed with every label in a legend. However, the options stored in this style are executed with the path prefix /tikz/data visualization/legend entry options. Thus, this key can use keys like node style to configure the styling of all text nodes:

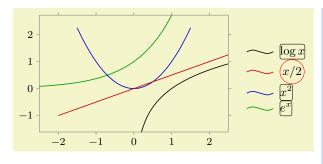


```
\tikz \datavisualization [
 scientific axes,
 every label in legend/.style={node style=
   {fill=red!30}},
 visualize as smooth line/.list=
   {log, lin, squared, exp},
 legend=north east outside,
         {label in legend={text=$\log x$}},
 log=
 lin=
          {label in legend={text=$x/2$,
     node style={circle, draw=red}}},
 squared={label in legend={text=$x^2$}},
         {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/label style=(options)

(no default)

This key can be used with a legend. It will simply add the $\langle options \rangle$ to the every label in legend style for the given legend.



```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   legend={label style={node style=draw}},
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$,
        node style={circle, draw=red}}},
   squared={label in legend={text=$x^2$}},
   exp=      {label in legend={text=$x^2$}},
   style sheet=strong colors]
   data group {function classes};
```

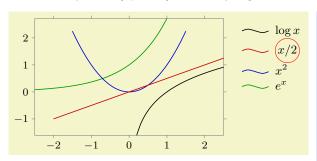
61.9.7 Legend Entries: Styling the Text Node

The appearance of the text nodes is easy to configure.

```
/tikz/data visualization/legend entry options/node style=\(\lambda ptions\)
```

(no default)

This key adds $\langle options \rangle$ to the styling of the text nodes of the label.

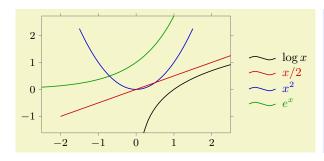


```
\tikz \datavisualization [
   scientific axes,
   visualize as smooth line/.list=
      {log, lin, squared, exp},
   legend=north east outside,
   log=      {label in legend={text=$\log x$}},
   lin=      {label in legend={text=$x/2$,
        node style={circle, draw=red}}},
   squared={label in legend={text=$x^2$}},
   exp=      {label in legend={text=$x^2$}},
   style sheet=strong colors]
   data group {function classes};
```

/tikz/data visualization/legend entry options/text colored

(no value)

Causes the node style to set the text color to visualizer color. The effect of this is that the label's text will have the same color as the data set to which it is attached.



```
\tikz \datavisualization [
    scientific axes,
    visualize as smooth line/.list=
        {log, lin, squared, exp},
    legend={label style=text colored},
    log=        {label in legend={text=$\log x$}},
    lin=        {label in legend={text=$x/2$}},
    squared={label in legend={text=$x^2$}},
    exp=        {label in legend={text=$x^2$}},
    style sheet=strong colors]
    data group {function classes};
```

61.9.8 Legend Entries: Text Placement

Three keys govern where the text will be placed relative to the data set style visualization.

/tikz/data visualization/legend entry options/text right

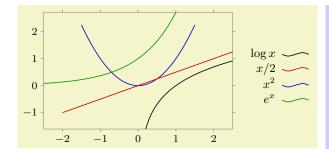
(no value)

Placed the text node to the right of the data set style visualization. This is the default for most, but not all, legends.

/tikz/data visualization/legend entry options/text left

(no value)

Placed the text node to the left of the data set style visualization.

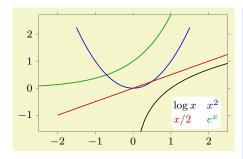


```
\tikz \datavisualization [
    scientific axes,
    visualize as smooth line/.list=
        {log, lin, squared, exp},
    legend={label style=text left},
    log=        {label in legend={text=$\log x$}},
    lin=        {label in legend={text=$x/2$}},
    squared={label in legend={text=$x^2$}},
    exp=        {label in legend={text=$x^2$}},
    style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend entry options/text only

(no value)

Shows only the text nodes and no data set style visualization at all. This options only makes sense in conjunction with the text colored options, which is why this options is also selected implicitly.



61.9.9 Advanced: Labels in Legends and Their Visualizers

The following explanations are important only for you if you intend to create a new visualizer and an accompanying label in legend visualizer; otherwise you can safely proceed with the next section.

A legend entry consists not only of some explaining text, but, even more importantly, of a visual representation of the style used for the data points, created by a *label in legend visualizer*. For instance, when data points are visualized as lines in different colors, the legend entry for the first line might consist of the text "first experiment" and a short line in black and the second entry might consist of "failed experiment"

and a short line in red – assuming, of course, that the style sheet makes the first line black and the second line blue. As another example, when data sets are visualized as clouds of plot marks, the texts in the legend would be accompanied by the plot marks used to visualize the data sets.

For every visualizer, the *label in legend visualizer* creates an appropriate visualization of the data set's styling. There may be more than one possible such label in legend visualizer that is appropriate, in which case options are used to choose between them.

Let us start with the key for creating a new legend entry. This key gets called for instance by label in legend:

/tikz/data visualization/new legend entry=\langle options \rangle

(no default)

This key will add a new entry to the legend that is identified by the $\langle options \rangle$. For this, the $\langle options \rangle$ are executed once with the path prefix /tikz/data visualization/legend entry options and the resulting setting of the legend key is used to pick which legend the new entry should belong to. Then, the $\langle options \rangle$ are stored away for the time being.

Later, when the legend is created, the $\langle options \rangle$ get executed once more. This time, however, the legend key is no longer important. Instead, the $\langle options \rangle$ that setup keys like text or visualizer in legend now play a role.

In detail, the following happens:

- For the legend entry, a little cell picture is created in the matrix of the legend (see Section 18.3 for details on cell pictures).
- Inside this picture, a node is created whose text is taken from the key

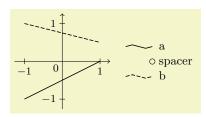
```
/tikz/data visualization/legend entry options/text
```

• Also inside the picture, the code stored in the following key gets executed:

```
/tikz/data visualization/legend entry options/visualizer in legend
```

(no value)

Set this key to some code that paints something in the cell picture. Typically, this will be a visual representation of the data set styling, but it could also be something different.



```
\tikz \datavisualization [
   school book axes, visualize as line/.list={a,b},
   style sheet=vary dashing,
   a={label in legend={text=a}},
   new legend entry={
      text=spacer,
      visualizer in legend={\draw[solid] (0,0) circle[radius=2pt];}
   },
   b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

The following styles are applied in the following order before the cell picture is filled:

- 1. /tikz/data visualization/every data set label with path /tikz/data visualization
- 2. /tikz/data visualization/every label in legend with path /tikz/data visualization/legend entry options.
- 3. The $\langle options \rangle$.
- 4. The code in the following key:

```
/tikz/data visualization/legend entry options/setup
```

(no value)

Some code to be executed at this point. Mostly, it is used to setup attributes for style sheets.

- 5. A styling signal is emitted.
- 6. Only for the node: The current value of node style.
- 7. Only for the visualizer in legend: The styling that has been accumulated by calls to the following key:

```
\label{eq:continuous} $$ \text{tikz/data visualization/legend entry options/visualizer in legend style=} $$ \{\langle options \rangle \}$$ (style, no default)
```

Calls to this key accumulate $\langle options \rangle$ that will be executed with the path prefix /tikz at this point.

As indicated earlier, the new legend entry key is called by the label in legend= $\langle options \rangle$ key internally. In this case, the following extra $\langle extra\ options \rangle$ are passed to new legend entry key:

- The styling of the visualizer.
- The /tikz/data visualization/every label in legend style.
- The /tikz/every label style with path /tikz.
- Setting setup to /data point/set= $\langle name\ of\ the\ visualizer \rangle$.
- The value of the label legend options that are stored in the visualizer. These options can be changed using the following key:

```
/tikz/data visualization/visualizer options/label in legend options=\langle options \rangle (no default)
```

Use this key with a visualizer to configure the label in legend options. Typically, this key is used only internally by a visualizer upon its creating to set the $\langle options \rangle$ to setup the visualizer in legend key.

61.9.10 Reference: Label in Legend Visualizers for Lines and Scatter Plots

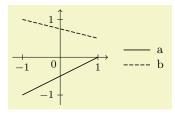
Visualizers like visualize as line or visualize as smooth line use a label in legend visualizer that draws a short line to represent the data set inside the legend. However, this line needs not be a simple straight line, but can be a little curve or a small circle – indeed, even the default line is not a simple straight line but rather a small zig-zag curve. To configure this line, the two keys are used, although you will only rarely use them directly, but rather use one of the predefined styles mentioned later on.

Before we go into the glorious details of all of these keys, let us first have a look at the keys you are most likely to use in practice: The keys for globally reconfiguring the default label in legend visualizers:

/tikz/data visualization/legend entry options/default label in legend path (style, no value)

This style is set, by default, to zig zag label in legend line. It is installed by the styles straight line, smooth line, and gap line, so changing this style will change the appearance of lines in legends. The main other sensible option for this key is straight label in legend line.

```
\tikz \datavisualization [
   school book axes, visualize as line/.list={a,b},
   style sheet=vary dashing,
   a={label in legend={text=a}}, b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```



```
\tikz \datavisualization [
   school book axes, visualize as line/.list={a,b},
   legend entry options/default label in legend path/.style=
        straight label in legend line,
   style sheet=vary dashing,
   a={label in legend={text=a}},   b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1, set=b]   data point [x=1, y=0.5, set=b];
```

/tikz/data visualization/legend entry options/default label in legend closed path (style, no value)

This style is executed by smooth cycle and straight cycle. There are (currently) no other predefined sets of coordinates that can be used instead of the default value circular label in legend line.

```
/tikz/data visualization/legend entry options/default label in legend mark (style, no value)
```

This style is executed by no lines and, implicitly, by scatter plots. The default is to use label in legend line one mark. Another possible value is label in legend line three marks.

```
\tikz \datavisualization [
visualize as scatter/.list={a,b,c},
style sheet=cross marks,
legend entry options/default label in legend mark/.style=
label in legend={text=example a},
b={label in legend={text=example b}},
c={label in legend={text=example c}};
```

/tikz/data visualization/legend entry options/label in legend line coordinates= $\{\langle list\ of\ coordinates \rangle\}$ (no default)

This key takes a $\langle list\ of\ coordinates \rangle$, which are TikZ-coordinates separated by commas like (0,0), (1,1). The effect of setting the key is the following: The label in legend visualizer used by, for instance, visualize as line will draw a path going through these points. When the line is drawn, the exact same style will be used as was used for the data set. For instance, if the smooth line key was used and also the style=red key, the line through the $\langle list\ of\ coordinates \rangle$ will also be red and smooth. When the straight cycle key was used, the coordinates will also be connected by a cycle, and so on.

When the line connecting the $\langle list\ of\ coordinates \rangle$ is drawn, the coordinate system will have been shifted and transformed in such a way that (0,0) lies to the left of the text and at half the height of the character "x". This means that the right-most-point in the list should usually be (0,0) and all other x-coordinates should usually be negative. When the text left options is used, the coordinate system will have been flipped, so the $\langle list\ of\ coordinates \rangle$ is independent of whether the text is to the right or to the left of the line.

Let us now have a look at a first, simple example. We create a legend entry that is just a straight line, so it should start somewhere to the left of the origin at height 0 and go to the origin:

```
\tikz \datavisualization [
    school book axes, visualize as line/.list={a,b},
    style sheet=vary dashing,
    a={label in legend={text=a,
        label in legend line coordinates={(-1em,0), (0,0)}}},
    b={label in legend line coordinates={(-2em,0), (0,0)}}}]
    data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
    data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

Now let us make this a bit more fancy and useful by using shifted lines:

```
htikz \datavisualization [
    school book axes, visualize as line/.list={a,b},
    legend={up then right}, style sheet=vary dashing,
    a={label in legend={text=a,
        label in legend line coordinates={(-2em,-.25ex), (0,0)}}},
    b={label in legend line coordinates={(-2em,.25ex), (0,0)}}}]
    data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
    data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

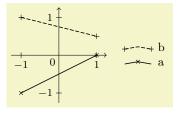
In the final example, we use a little "hat" to represent lines:

```
\tikz \datavisualization [
    school book axes, visualize as line/.list={a,b},
    legend={up then right}, style sheet=vary dashing,
    a={label in legend={text=a,
        label in legend line coordinates={
        (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)}}},
    b={label in legend line coordinates={
        (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)}}}]
    data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
    data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

```
/tikz/data visualization/legend entry options/label in legend mark coordinates= \{\langle list\ of\ coordinates \rangle\} (no default)
```

This key is similar to label in legend line coordinates, but now the $\langle list\ of\ coordinates \rangle$ is used as the positions where plot marks are shown. Naturally, plot marks are only shown there if they are also shown by the visualizer in the actual data – just like the line through the coordinates of the previous key is only shown when there is a line.

The $\langle list\ of\ coordinates \rangle$ may be the same as the one used for lines, but usually it is not. In general, it is better to have marks for instance not at the ends of the line.



```
\tikz \datavisualization [
 school book axes, visualize as line/.list={a,b},
 legend={up then right},
 style sheet=vary dashing,
 style sheet=cross marks,
 a={label in legend={text=a,
      label in legend line coordinates={
        (-2em, -.2ex), (-1em, .2ex), (0, -.2ex)
     label in legend mark coordinates={
        (-1em, .2ex)}},
 b={label in legend={text=b,
     label in legend line coordinates={
        (-2em, -.2ex), (-1em, .2ex), (0, -.2ex)
     label in legend mark coordinates={
        (-2em, -.2ex), (0, -.2ex)}
data point [x=-1, y=-1, set=a]
                                data point [x=1, y=0, set=a]
                                data point [x=1, y=0.5, set=b];
data point [x=-1, y=1, set=b]
```

Naturally, you typically will not give coordinates explicitly for each label, but use one of the following styles:

/tikz/data visualization/legend entry options/straight label in legend line (no value)

Just gives a straight line and two plot marks.

This style might seem like a good idea to use in general, but it does have a huge drawback: Some commonly used plot marks will be impossible to distinguish – even though there is no problem distinguishing them in a graph.

For this reason, this option is not the default, but rather the next one.

```
/tikz/data visualization/legend entry options/zig zag label in legend line (no value)
```

Uses a small up-down-up line as the label in legend visualizer. The two plot marks are at the extremal points of the line. It works pretty well in almost all situations and is the default.

```
better example a
better example b
better example c

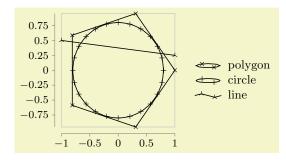
visualize as line=a,
visualize as smooth line/.list={b,c},
a={style={mark=+}, label in legend={text=better example a}},
b={style={mark=-}, label in legend={text=better example c}};
c={style={mark=|}, label in legend={text=better example c}};
```

Even though the above example shows that the marks are easier to distinguish than with a straight line, the chosen marks are still not optimal. This is the reason that the cross marks style uses different crosses:

```
tikz \datavisualization [
    visualize as line/.list={a,b},
    visualize as smooth line=c,
    style sheet=cross marks,
    a={label in legend={text=good example a}},
    b={label in legend={text=good example b}},
    c={gap line, label in legend={text=good example c}}];
```

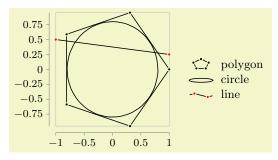
/tikz/data visualization/legend entry options/circular label in legend line (no value)

This style is especially tailored to represent lines that are closed. It is automatically selected for instance by the polygon or the smooth cycle styles.



```
\tikz \datavisualization [
  scientific axes={clean}, all axes={length=3cm},
  visualize as line/.list={a,b,c},
  a={polygon}, b={smooth cycle},
  style sheet=cross marks,
  a={label in legend={text=polygon}},
  b={label in legend={text=circle}},
  c={label in legend={text=line}}]
data [format=function, set=a] {
  var t : \{0,72,...,359\};
  func x = cos(\forall x);
  func y = sin(\value t);
data [format=function, set=b] {
  var t : [0:2*pi];
  func x = .8*cos(\forall x);
  func y = .8*sin(\forall x);
data point [x=-1, y=0.5, set=c]
data point [x=1, y=0.25, set=c];
```

/tikz/data visualization/legend entry options/gap circular label in legend line (no value) This style is especially tailored to for the gap cycle style and automatically selected by it:



```
\tikz \datavisualization [
  scientific axes={clean}, all axes={length=3cm},
  visualize as line/.list={a,b,c},
  a={gap cycle}, b={smooth cycle}, c={gap line},
  a={style={mark=*, mark size=0.5pt},
     label in legend={text=polygon}},
  b={label in legend={text=circle}},
  c={style={mark=*, mark size=0.5pt, mark options=red},
     label in legend={text=line}}]
data [format=function, set=a] {
  var t : \{0,72,...,359\};
  func x = cos(\forall x);
  func y = sin(\value t);
data [format=function, set=b] {
  var t : [0:352];
  func x = .8*cos(\forall x);
  func y = .8*sin(\forall t);
data point [x=-1, y=0.5, set=c]
data point [x=1, y=0.25, set=c];
```

/tikz/data visualization/legend entry options/label in legend one mark

(no value)

To be used with scatter plots, since no line is drawn. Just displays a single mark (this is the default with a scatter plot or when the no line is selected.

/tikz/data visualization/legend entry options/label in legend three marks

(no value)

An alternative to the previous style, where several marks are shown.

62 The Data Visualization Backend

62.1 Overview

The present section explains the mechanisms behind the data visualization engine.

Until it is documented properly, we will have to make do with the documentation in the source code.

62.2 The Rendering Pipeline

To be written...

62.3 Usage

To be written...

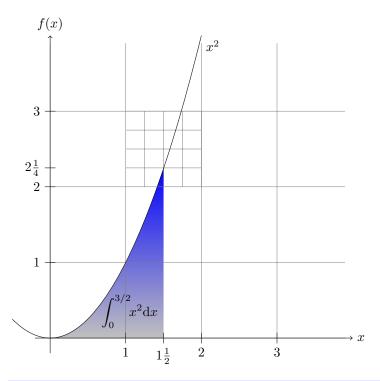
62.4 The Mathematical Micro-Kernel

To be written...

Part VII Utilities

by Till Tantau

The utility packages are not directly involved in creating graphics, but you may find them useful nonetheless. All of them either directly depend on PGF or they are designed to work well together with PGF even though they can be used in a stand-alone way.



63 Key Management

This section describes the package pgfkeys. It is loaded automatically by both PGF and TikZ.

```
\usepackage{pgfkeys} % MEX
\input pgfkeys.tex % plain TEX
\usemodule[pgfkeys] % ConTEXt
```

This package can be used independently of PGF. Note that no other package of PGF needs to be loaded (so neither the emulation layer nor the system layer is needed). The ConTEXt abbreviation is pgfkey if pgfmod is not loaded.

63.1 Introduction

63.1.1 Comparison to Other Packages

The pgfkeys package defines a key-value management system that is in some sense similar to the more light-weight keyval system and the improved xkeyval system. However, pgfkeys uses a slightly different philosophy than these systems and it will coexist peacefully with both of them.

The main differences between pgfkeys and xkeyval are the following:

- pgfkeys organizes keys in a tree, while keyval and xkeyval use families. In pgfkeys the families correspond to the root entries of the key tree.
- pgfkeys has no save-stack impact (you will have to read the TEXBook very carefully to appreciate this).
- pgfkeys is slightly slower than keyval, but not much.
- pgfkeys supports styles. This means that keys can just stand for other keys (which can stand for other keys in turn or which can also just execute some code). TikZ uses this mechanism heavily.
- pgfkeys supports multi-argument key code. This can, however, be emulated in keyval.
- pgfkeys supports handlers. These are call-backs that are called when a key is not known. They are very flexible, in fact even defining keys in different ways is handled by, well, handlers.

63.1.2 Quick Guide to Using the Key Mechanism

The following quick guide to PGF's key mechanism only treats the most commonly used features. For an in-depth discussion of what is going on, please consult the remainder of this section.

Keys are organized in a large tree that is reminiscent of the Unix file tree. A typical key might be, say, /tikz/coordinate system/x or just /x. Again as in Unix, when you specify keys you can provide the complete path of the key, but you usually just provide the name of the key (corresponding to the file name without any path) and the path is added automatically.

Typically (but not necessarily) some code is associated with a key. To execute this code, you use the $\protect\operatorname{\sc hypfkeys}$ command. This command takes a list of so-called key-value pairs. Each pair is of the form $\langle key \rangle = \langle value \rangle$. For each pair the $\protect\operatorname{\sc hypfkeys}$ command will execute the code stored for the $\langle key \rangle$ with its parameter set to $\langle value \rangle$.

Here is a typical example of how the \pgfkeys command is used:

```
\pgfkeys{/my key=hallo,/your keys/main key=something\strange, key name without path=something else}
```

Now, to set the code that is stored in a key you do not need to learn a new command. Rather, the \pgfkeys command can also be used to set the code of a key. This is done using so-called handlers. They look like keys whose names look like "hidden files in Unix" since they start with a dot. The handler for setting the code of a key is appropriately called /.code and it is used as follows:

```
The value is 'hi!'. \pgfkeys{/my key/.code=The value is '#1'.} \pgfkeys{/my key=hi!}
```

As you can see, in the first line we defined the code for the key /my key. In the second line we executed this code with the parameter set to hi!.

There are numerous handlers for defining a key. For instance, we can also define a key whose value actually consists of more than one parameter.

The values are 'a1' and 'a2'.

```
\pgfkeys{/my key/.code 2 args=The values are '#1' and '#2'.}
\pgfkeys{/my key={a1}{a2}}
```

We often want to have keys where the code is called with some default value if the user does not provide a value. Not surprisingly, this is also done using a handler, this time called /.default.

```
(hallo)(hello)
  \pgfkeys{/my key/.code=(#1)}
  \pgfkeys{/my key/.default=hello}
  \pgfkeys{/my key=hallo,/my key}
```

The other way round, it is also possible to specify that a value *must* be specified, using a handler called /.value required. Finally, you can also require that no value *may* be specified using /.value forbidden.

All keys for a package like, say, TikZ start with the path /tikz. We obviously do not like to write this path down every time we use a key (so we do not have to write things like \draw[/tikz/line width=1cm]). What we need is to somehow "change the default path to a specific location." This is done using the handler /.cd (for "change directory"). Once this handler has been used on a key, all subsequent keys in the current call of \pgfkeys only are automatically prefixed with this path, if necessary.

Here is an example:

```
\pgfkeys{/tikz/.cd,line width=1cm,line cap=round}
```

This makes it easy to define commands like \tikzset, which could be defined as follows (the actual definition is a bit faster, but the effect is the same):

```
\def\tikzset#1{\pgfkeys{/tikz/.cd,#1}}
```

When a key is handled, instead of executing some code, the key can also cause further keys to be executed. Such keys will be called *styles*. A style is, in essence, just a key list that should be executed whenever the style is executed. Here is an example:

```
(a:foo)(b:bar)(a:wow)

\text{pgfkeys{/a/.code=(a:#1)}}
\text{pgfkeys{/b/.code=(b:#1)}}
\text{pgfkeys{/my style/.style={/a=foo,/b=bar,/a=#1}}
\text{pgfkeys{/my style=wow}}
```

As the above example shows, styles can also be parametrized, just like the normal code keys.

As a typical use of styles, suppose we wish to set up the key /tikz so that it will change the default path to /tikz. This can be achieved as follows:

```
\pgfkeys{/tikz/.style=/tikz/.cd}
\pgfkeys{tikz,line width=1cm,draw=red}
```

Note that when \pgfkeys is executed, the default path is set to /. This means that the first tikz will be completed to /tikz. Then /tikz is a style and, thus, replaced by /tikz/.cd, which changes the default path to /tikz. Thus, the line width is correctly prefixed with /tikz.

63.2 The Key Tree

The pgfkeys package organizes keys in a so-called *key tree*. This tree will be familiar to anyone who has used a Unix operating system: A key is addressed by a path, which consists of different parts separated by slashes. A typical key might be /tikz/line width or just /tikz or something more complicated like /tikz/cs/x/.store in.

Let us fix some further terminology: Given a key like /a/b/c, we call the part leading up the last slash (/a/b) the *path* of the key. We call everything after the last slash (c) the *name* of the key (in a file system this would be the file name).

We do not always wish to specify keys completely. Instead, we usually specify only part of a key (typically only the name) and the *default path* is then added to the key at the front. So, when the default path is /tikz and you refer to the (partial) key line width, the actual key that is used is /tikz/line width. There is a simple rule for deciding whether a key is a partial key or a full key: If it starts with a slash, then it is a full key and it is not modified; if it does not start with a slash, then the default path is automatically prefixed.

Note that the default path is not the same as a search path. In particular, the default path is just a single path. When a partial key is given, only this single default path is prefixed; pgfkeys does not try to

look up the key in different parts of a search path. It is, however, possible to emulate search paths, but a much more complicated mechanism must be used.

When you set keys (to be explained in a moment), you can freely mix partial and full keys and you can change the default path. This makes it possible to temporarily use keys from another part of the key tree (this turns out to be a very useful feature).

Each key (may) store some tokens and there exist commands, described below, for setting, getting, and changing the tokens stored in a key. However, you will only very seldom use these commands directly. Rather, the standard way of using keys is the \pgfkeys command or some command that uses it internally like, say, \tikzset. So, you may wish to skip the following commands and continue with the next subsection.

$\protect\pro$

Stores the $\langle token\ text \rangle$ in the $\langle full\ key \rangle$. The $\langle full\ key \rangle$ may not be a partial key, so no default-path-adding is done. The $\langle token\ text \rangle$ can be arbitrary tokens and may even contain things like # or unbalanced T_FX-ifs.

```
Hello, world! \pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysvalueof{/my family/my key}
```

The setting of a key is always local to the current T_FX group.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu
```

Performs a \let statement so the the $\langle full \ key \rangle$ points to the contents of $\langle macro \rangle$.

```
Hello, world!
\def\helloworld{Hello, world!}
\pgfkeyslet{/my family/my key}{\helloworld}
\pgfkeysvalueof{/my family/my key}
```

You should never let a key be equal to \relax. Such a key may or may not be indistinguishable from an undefined key.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu
```

Retrieves the tokens stored in the $\langle full \; key \rangle$ and lets $\langle macro \rangle$ be equal to these tokens. If the key has not been set, the $\langle macro \rangle$ will be equal to $\$ relax.

```
Hello, world!
\pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysgetvalue{/my family/my key}{\helloworld}
\helloworld
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Inserts the value stored in $\langle full \ key \rangle$ at the current position into the text.

```
Hello, world! \pgfkeyssetvalue{/my family/my key}{Hello, world!} \pgfkeysvalueof{/my family/my key}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Checks whether this key was previously set using either \pgfkeyssetvalue or \pgfkeyslet. If so, the code in $\langle if \rangle$ is executed, otherwise the code in $\langle else \rangle$.

This command will use eTeX's \ifcsname command, if available, for efficiency. This means, however, that it may behave differently for TeX and for eTeX when you set keys to \relax. For this reason you should not do so.

```
yes
\pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysifdefined{/my family/my key}{yes}{no}
```

63.3 Setting Keys

Settings keys is done using a powerful command called $\protect\operatorname{\sc hey-value}$. This command takes a list of so-called $\protect\operatorname{\sc hey-value}$ pairs. These are pairs of the form $\protect\operatorname{\sc hey-value}$. The principle idea is the following: For each pair in the list, some action is taken. This action can be one of the following:

- 1. A command is executed whose argument(s) are $\langle value \rangle$. This command is stored in a special subkey of $\langle key \rangle$.
- 2. The $\langle value \rangle$ is stored in the $\langle key \rangle$ itself.
- 3. If the key's name (the part after the last slash) is a known *handler*, then this handler will take care of the key.
- 4. If the key is totally unknown, one of several possible unknown key handlers is called.

Additionally, if the $\langle value \rangle$ is missing, a default value may or may not be substituted. Before we plunge into all the details, let us have a quick look at the command itself.

$\protect\$ \pro

The $\langle key | list \rangle$ should be a list of key-value pairs, separated by commas. A key-value pair can have the following two forms: $\langle key \rangle = \langle value \rangle$ or just $\langle key \rangle$. Any spaces around the $\langle key \rangle$ or around the $\langle value \rangle$ are removed. It is permissible to surround both the $\langle key \rangle$ or the $\langle value \rangle$ in curly braces, which are also removed. Especially putting the $\langle value \rangle$ in curly braces needs to be done quite often, namely whenever the $\langle value \rangle$ contains an equal-sign or a comma.

The key-value pairs in the list are handled in the order they appear. How this handling is done, exactly, is described in the rest of this section.

If a $\langle key \rangle$ is a partial key, the current value of the default path is prefixed to the $\langle key \rangle$ and this "upgraded" key is then used. The default path is just the root path / when the first key is handled, but it may change later on. At the end of the command, the default path is reset to the value it had before this command was executed.

Calls of this command may be nested. Thus, it is permissible to call \pgfkeys inside the code that is executed for a key. Since the default path is restored after a call of \pgfkeys, the default path will not change when you call \pgfkeys while executing code for a key (which is exactly what you want).

$\protect\operatorname{pgfqkeys}\{\langle default\ path \rangle\}\{\langle key\ list \rangle\}$

This command has the same effect as $\psi {\langle default\ path \rangle / .cd, \langle key\ list \rangle \}}$, it is only marginally quicker. This command should not be used in user code, but rather in commands like \tikzset or $\psi fset$ that get called very often.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabula$

This command has execatly the same effect as \pgfkeys, only the default path is not modified before or after the keys are being set. This command is mainly intended to be called by the code that is being processed for a key.

This command has the same effect as $\psi {default path}/.cd, (key list)}$, it is only quicker. Changing the default path inside a $\psi {default path}$ is dangerous, so use with care. A rather safe place to call this command is at the beginning of a $\psi {default path}$ group.

63.3.1 First Char Syntax Detection

Usually, keys are of the form $\langle key \rangle = \langle value \rangle$ and how such keys are handled is discussed in the rest of this section. However, it is also possible to setup a different syntax for certain parts of the input to \pgfkeys. Since this is a rather advanced option, most readers may wish to skip the following discussion upon first reading; it is discussed here because this special syntax detection is the very first thing that is done when a key is processed, before any of the following operations are performed.

The \pgfkeys command and its variants decompose their input into a list of $\langle string \rangle$ s that are separated by commas. By default, each such $\langle string \rangle$ must either have the form $\langle key \rangle = \langle value \rangle$ or of the form $\langle key \rangle$ with the value-part missing. However, you might wish to interpret some of these strings differently. For instance, when a $\langle string \rangle$ has the form " $\langle text \rangle$ ", you might wish the $\langle string \rangle$ to be interpreted as if one had written label text={ $\langle text \rangle$ }. Then, people could write

\myset{red, "main valve", thick}

instead of the more cumbersome

```
\myset{red, label text=main valve, thick}
```

An example where such a syntax reinterpretation is done is the quotes library, which allows you to write things like

```
a \xrightarrow{1} b \xrightarrow{0} c \tikz \graph { a ->["1" red] b ->["0"] c };
```

instead of the somewhat longer

```
 a \xrightarrow{1} b \xrightarrow{0} c  \tikz \graph { a ->[edge node={node[red,auto]{1}}] b ->[edge label=0] c };
```

In order to detect whether a $\langle string \rangle$ has a special syntax, you can request that the *first character* of $\langle string \rangle$ is analysed by the key parser. If this first character matches a character that has been flagged as a special character, the $\langle string \rangle$ is not interpreted as a usual key-value pair. Instead, $\langle string \rangle$ is passed as a parameter to a special macro that should take care of the $\langle string \rangle$. After this macro has finished, the parsing continues with the $\langle next \ string \rangle$ in the list.

In order to setup a special syntax handling for $\langle strings \rangle$ that begin with a certain character, two things need to be done:

1. First, the whole first char syntax detection must be "switched on," since, by default, it is turned off for efficiency reasons (the overhead is rather small, however). This is done by setting the following key:

```
/handlers/first char syntax=\langle true \ or \ false \rangle (default true, initially false)
```

2. Second, in order to handle strings starting with a certain $\langle character \rangle$ in a special way, you need to store a macro in the following key:

```
/handlers/first char syntax/\(\lambda\) meaning of character\(\rangle\) (no value)
```

The $\langle meaning\ of\ character \rangle$ should be the text that TEX's command \meaning returns for a macro that has been \let to the $\langle character \rangle$. For instance, when strings starting with "should be treated in a special way, the $\langle meaning\ of\ character \rangle$ would be the string the character "since this is what TEX writes when you say

```
the character " \let\mycharacter=" \meaning\mycharacter
```

Now, the key /handlers/first char syntax/ $\langle meaning\ of\ character \rangle$ should be setup (using \pgfkeyssetvalue or using the .initial handler) to store a $\langle macro\ name \rangle$.

If this is the case and if $\langle string \rangle$ starts with the $\langle character \rangle$ (blanks at the beginning of $\langle string \rangle$ are deleted prior to this test), then $\langle macro\ name \rangle$ is called with $\langle string \rangle$ as its argument.

Let us now have a look at an example. We install two handlers, one for strings starting with " and one for strings starting with <.

```
Quoted: "foo". Pointed: <bar>.

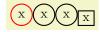
\pgfkeys{
   /handlers/first char syntax=true,
   /handlers/first char syntax/the character "/.initial=\myquotemacro,
   /handlers/first char syntax/the character </.initial=\mypointedmacro,
}

\def\myquotemacro#1{Quoted: #1. }
\def\mypointedmacro#1{Pointed: #1. }
\ttfamily \pgfkeys{"foo", <bar>}
```

Naturally, in the above examples, the two handling macros did not do something particularly exciting. In the next example, we setup a more elaborate macro that mimics a small part the behaviour of the quotes library, only for single quotes:

Note that in the above example, the macro \mysinglequotemacro gets passed the complete string, including the single quotes. It is the job of the macro to get rid of them, if this is necessary.

The first char syntax detection allows you to perform rather powerful transformations on the syntax of keys – provided you can "pin down" the syntax on the first character. In the following example, you can write expressions in parentheses in front of a key–value pair and the pair will only be executed when the expression evaluates to true:



```
\pgfkeys{
   /handlers/first char syntax=true,
   /handlers/first char syntax/the character (/.initial=\myparamacro)
}

\def\myparamacro#1{\myparaparser#1\someendtext}
\def\myparaparser(#1)#2\someendtext{
   \pgfmathparse{#1}
   \ifx\pgfmathresult\onetext
   \pgfkeysalso{#2}
   \fi
}
\def\onetext{1}

\foreach \i in {1,...,4}
   \tikz \node [draw, thick, rectangle, (pi>\i) circle, (pi>\i*2) draw=red] {x};
```

63.3.2 Default Arguments

The arguments of the \pgfkeys command can either be of the form $\langle key \rangle = \langle value \rangle$ or of the form $\langle key \rangle$ with the value-part missing. In the second case, the \pgfkeys will try to provide a default value for the $\langle value \rangle$. If such a default value is defined, it will be used as if you had written $\langle key \rangle = \langle default \ value \rangle$.

In the following, the details of how default values are determined is described; however, you should normally use the handlers /.default and /.value required as described in Section 63.4.2 and you may wish to skip the following details.

When $\protect\protec$

- 1. The input is replaced by $\langle key \rangle = \text{pgfkeysnovalue}$. In particular, the commands \pgfkeys{my key} and \pgfkeys{my key=\pgfkeysnovalue} have exactly the same effect and you can "simulate" a missing value by providing the value \pgfkeysnovalue, which is sometimes useful.
- 2. If the $\langle value \rangle$ is \pgfkeysnovalue , then it is checked whether the subkey $\langle key \rangle$ /.Qdef exists. For instance, if you write $\pgfkeys{/my key}$, then it is checked whether the key $\mbox{/my key}$.Qdef exists.
- 3. If the key $\langle key \rangle$ /. Qdef exists, then the tokens stored in this key are used as $\langle value \rangle$.
- 4. If the key does not exist, then $\protect\operatorname{pgfkeysnovalue}$ is used as the $\langle value \rangle$.
- 5. At the end, if the $\langle value \rangle$ is now equal to pgfkeysvaluerequired, then the code (or something fairly equivalent) $pgfkeys{/errors/value required=\langle key \rangle{}}$ is executed. Thus, by changing this key you can change the error message that is printed or you can handle the missing value in some other way.

63.3.3 Keys That Execute Commands

After the transformation process described in the previous subsection, we arrive at a key of the form $\langle key \rangle = \langle value \rangle$, where $\langle key \rangle$ is a full key. Different things can now happen, but always the macro \pgfkeyscurrentkey will have been set up to expand to the text of the $\langle key \rangle$ that is currently being processed.

The first things that is tested is whether the key $\langle key \rangle$ /. Qcmd exists. If this is the case, then it is assumed that this key stores the code of a macro and this macro is executed. The argument of this macro is $\langle value \rangle$ directly followed by $\langle pgfeov$, which stands for "end of value." The $\langle value \rangle$ is not surrounded by braces. After this code has been executed, $\langle pgfkeys$ continues with the next key in the $\langle key \ list \rangle$.

It may seem quite peculiar that the macro stored in the key $\langle key \rangle$ /. **@cmd** is not simply executed with the argument $\{\langle value \rangle\}$. However, the approach taken in the **pgfkeys** packages allows for more flexibility. For instance, assume that you have a key that expects a $\langle value \rangle$ of the form " $\langle text \rangle$ + $\langle more\ text \rangle$ " and wishes to store $\langle text \rangle$ and $\langle more\ text \rangle$ in two different macros. This can be achieved as follows:

```
\a is hello, \b is world.

\def\mystore#1+#2\pgfeov{\def\a{#1}\def\b{#2}}
\pgfkeyslet{/my key/.@cmd}{\mystore}
\pgfkeys{/my key=hello+world}

\a| is \a, |\b| is \b.
```

Naturally, defining the code to be stored in a key in the above manner is too awkward. The following commands simplify things a bit, but the usual manner of setting up code for a key is to use one of the handlers described in Section 63.4.3.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabula
```

This command temporarily defines a TeX-macro with the argument list #1\pgfeov and then lets $\langle key \rangle$ /.@cmd be equal to this macro. The net effect of all this is that you have then set up code for the key $\langle key \rangle$ so that when you write \pgfkeys{ $\langle key \rangle = \langle value \rangle$ }, then the $\langle code \rangle$ is executed with all occurrences of #1 in $\langle code \rangle$ being replaced by $\langle value \rangle$. (This behaviour is quite similar to the \define@key command of keyval and xkeyval).

```
hello, hello. \pgfkeysdef{/my key}{#1, #1.} \pgfkeys{/my key=hello}
```

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu
```

This command works like \pgfkeysdef, but it uses \edef rather than \def when defining the key macro. If you do not know the difference between the two, then you will not need this command; and if you know the difference, then you will know when you need this command.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command works like $\poline{pgfkeysdef}$, but it allows you to provide an arbitrary $\langle argument\ count \rangle$ between 0 and 9 (inclusive).

```
\a is 'hello', \b is 'world'.

\pgfkeysdefnargs{/my key}{2}{\def\a{#1}\def\b{#2}}
\pgfkeys{/my key=
{hello}
{world}}

\a| is '\a', \b| is '\b'.
```

The resulting key will expect exactly $\{\langle argument \ count \rangle\}$ arguments.

```
\protect\operatorname{\mathtt{pgfkeysedefnargs}}\{\langle key \rangle\}\{\langle argument\ count \rangle\}\{\langle code \rangle\}
```

The \edef version of \pgfkeysdefnargs.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command works like \pgfkeysdefnargs , but it allows you to provide an arbitrary $\langle argument \ pattern \rangle$ rather than just a number of arguments.

```
\a is hello, \b is world.

\pgfkeysdefargs{/my key}{#1+#2}{\def\a{#1}\def\b{#2}}
\pgfkeys{/my key=hello+world}

\a is \a, \b| is \b.
```

Note that $\protect\operatorname{heps}$ is better when it comes to simple argument $counts^{17}$.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

The \edef version of \pgfkeysdefargs.

¹⁷When the resulting keys are used, the defnargs variant allows spaces between arguments whereas the defargs variant does not; it considers the spaces as part of the argument.

63.3.4 Keys That Store Values

Let us continue with what happens when \pgfkeys processes the current key and the subkey $\langle key \rangle$ /.@cmd is not defined. Then it is checked whether the $\langle key \rangle$ itself exists (has been previously assigned a value using, for instance, \pgfkeyssetvalue). In this case, the tokens stored in $\langle key \rangle$ are replaced by $\langle value \rangle$ and \pgfkeys proceeds with the next key in the $\langle key | list \rangle$.

63.3.5 Keys That Are Handled

If neither the $\langle key \rangle$ itself nor the subkey $\langle key \rangle$ /. **@cmd** are defined, then the $\langle key \rangle$ cannot be processed "all by itself." Rather, a $\langle handler \rangle$ is needed for this key. Most of the power of pgfkeys comes from the proper use of such handlers.

Recall that the $\langle key \rangle$ is always a full key (if it was not originally, it has already been upgraded at this point to a full key). It decomposed into two parts:

- 1. The $\langle path \rangle$ of $\langle key \rangle$ (everything before the last slash) is stored in the macro \pgfkeyscurrentpath.
- 2. The $\langle name \rangle$ of $\langle key \rangle$ (everything after the last slash) is stored in the macro \pgfkeyscurrentname. It is recommended (but not necessary) that the name of a handler starts with a dot (but not with .0), so that they are easy to detect for the reader.

(For efficiency reasons, these two macros are only set up at this point; so when code is executed for a key in the "usual" manner than these macros are not set up.)

The \pgfkeys command now checks whether the key /handlers/ $\langle name \rangle$ /.@cmd exists. If so, it should store a command and this command is executed exactly in the same manner as described in Section 63.3.3. Thus, this code gets the $\langle value \rangle$ that was originally intended for $\langle key \rangle$ as its argument, followed by \pgfeov. It is the job of the handlers to do something useful with the $\langle value \rangle$.

For an example, let us write a handler that will output the value stored in a key to the log file. We call this handler /.print to log. The idea is that when someone tries to use the key /my key/.print to log, then this key will not be defined and the handler gets executed. The handler will then have access to the path-part of the key, which is /my key, via the macro \pgfkeyscurrentpath. It can then lookup which value is stored in this key and print it.

```
\pgfkeysdef{/handlers/.print to log}
{%
   \pgfkeysgetvalue{\pgfkeyscurrentpath}{\temp}
   \writetolog{\temp}
}
\pgfkeyssetvalue{/my key}{Hi!}
...
\pgfkeys{/my key/.print to log}
```

The above code will print Hi! in the log, provided the macro \writetolog is set up appropriately.

For a more interesting handler, let us program a handler that will set up a key so that when the key is used, some code is executed. This code is given as $\langle value \rangle$. All the handler must do is to call \pgfkeysdef for the path of the key (which misses the handler's name) and assign the parameter value to it.

```
(hallo) \pgfkeysdef{/handlers/.my code}{\pgfkeysdef{\pgfkeyscurrentpath}{#1}}
\pgfkeys{/my key/.my code=(#1)}
\pgfkeys{/my key=hallo}
```

There are some parameters for handled keys which prove to be useful in some (possibly rare) special cases:

```
/handler config=all|only existing|full or existing (no default, initially all)
```

Changes the initial configuration how key handlers will be used.

This configuration is for advanced users and rarely necessary.

all The preconfigured setting all works as described above and imposes no restriction on the key setting process.

only existing The value only existing modifies the algorithm for handled keys as follows: a handler $\langle key \; name \rangle / . \langle handler \rangle$ will be executed only if $\langle key \; name \rangle$ is either a key which stores its value directly or a command key for which /.@cmd exists. If $\langle key \; name \rangle$ does not exist already, the

complete string $\langle key \; name \rangle / . \langle handler \rangle$ is considered to be an unknown key and the procedure described in the next section applies (for the path of $\langle key \; name \rangle$).

Initial definition.Re-Definition.Unknown key ".

```
% Define a test key and error handlers:
\pgfkeys{/the/key/.code={Initial definition. }}
\pgfkeys{/handlers/.unknown/.code={Unknown key '\pgfkeyscurrentkey'. }}
% calling the test key yields 'Initial definition. ':
\pgfkeys{/the/key}
% Change configuration:
\pgfkeys{/handler config=only existing}
% allowed: key *re*-definition:
\pgfkeys{/the/key/.code={Re-Definition. }}
% calling the key yields 'Re-Definition. ':
\pgfkeys{/the/key}
% not allowed: definition of new keys:
% this checks for '/the/other key/.unknown'
% and '/handlers/.unknown'
% and yields finally
% 'Unknown key '/the/other key/.code',
\pgfkeys{/the/other key/.code={New definition. }}
```

It is necessary to exclude some key handlers from this procedure. Altogether, the detailed procedure is as follows:

- 1. If a handled key like /a path/a key/.a handler=value is encountered, it is checked whether the handler should be invoked. This is the case if
 - An exception from only existing for this key exists (see below),
 - The key /a path/a key exists already either directly as storage key or with the .@cmd suffix.
- 2. If the check passes, everything works as before.
- 3. If the check fails, the complete key will be considered to be unknown. In that case, the handling of unknown keys as described in the next section applies. There, the current key path will be set to /a path and the current key's name to key/.a handler.

A consequence of this configuration is to provide more meaningful processing of handled keys if a search path for keys is in effect, see section 63.3.6 for an example.

full or existing Finally, the choice full or existing is a variant of only existing: it works in the same way for keys which do not have a full key path. For example, the style

```
\pgfkeys{/my path/.cd,key/.style={...}}
```

can only be redefined: it doesn't have a full path, so the only existing mechanism applies. But the style

```
\pgfkeys{/my path/key/.style={...}}
```

will still work. This allows users to override the only existing feature if they know what they're doing (and provide full key paths).

```
/handler config/only existing/add exception=\{\langle key \ handler \ name \rangle\} (no default)
```

Allows to add exceptions to the /handler config=only existing feature. Initially exceptions for the key handlers /.cd, /.try, /.retry, /.lastretry and /.unknown are defined. The value $\{\langle key \ handler \ name \rangle\}$ should be the name of a key handler.

63.3.6 Keys That Are Unknown

For some keys, neither the key, nor its .@cmd subkey nor a handler is defined. In this case, it is checked whether the key $\langle current\ path \rangle$ /.unknown/.@cmd exists. Thus, when you try to use the key /tikz/strange, then it is checked whether /tikz/.unknown/.@cmd exists. If this key exists (which it does), it is executed. This code can then try to make sense of the key. For instance, the handler for TikZ will try to interpret the key's name as a color or as an arrow specification or as a PGF option.

You can set up unknown key handlers for your own keys by simply setting the code of the key $\langle my \; path \; prefix \rangle$ /.unknown. This also allows you to set up "search paths." The idea is that you would like keys to be searched not only in a single default path, but in several. Suppose, for instance, that you would like keys to be searched for in /a, /b, and /b/c. We set up a key /my search path for this:

In the above code, foo and bar will be searched for in the three directories /a, /b, and /b/c. Before you start implementing search paths using this pattern, consider the /.search also handler discussed below.

If the key $\langle current\ path \rangle$.unknown.@cmd does not exist, the handler /handlers/.unknown is invoked instead, which is always defined and which prints an error message by default.

63.3.7 Search Paths And Handled Keys

There is one special case which occurs in the search path example above. What happens if we want to change a style? For example,

```
\pgfkeys{/my search path/.cd,custom/.style={variables}}
```

could mean a style in /my search path/, /a/, /b/ or even /b/c/!

Due to the rules for handled keys, the answer is /my search path/custom/.style={variables}. It may be useful to modify this default behavior. One useful thing would be to search for existing styles named custom and redefine them. For example, if a style /b/custom exists, the assignment custom/.style={variables} should probably redefine /b/custom instead of /my search path/custom. This can be done using handler config:

This is '/b/custom'. This is '/b/custom'. Modified.

```
\pgfkeys{/my search path/.unknown/.code=
  {%
    \let\searchname=\pgfkeyscurrentname%
    \pgfkeysalso{%
      /a/\searchname/.try=#1,
      /b/\searchname/.retry=#1,
      /b/c/\searchname/.retry=#1%
 }%
% Let's define /b/custom here:
\pgfkeys{/b/custom/.code={This is '\pgfkeyscurrentkey'. }}
% Reconfigure treatment of key handlers:
\pgfkeys{/handler config=only existing}
% The search path procedure will find /b/custom
% -> leads to This is '/b/custom'
\pgfkeys{/my search path/.cd,custom}
% Due to the reconfiguration, this will find /b/custom instead of
% defining /my search path/custom:
\pgfkeys{/my search path/.cd,custom/.append code={Modified. }}
% So using the search path, we again find /b/custom which
% leads to This is '/b/custom' Modified
\pgfkeys{/my search path/.cd,custom}
```

A slightly different approach to search paths can be realized using the /.search also key handler, see below.

63.4 Key Handlers

We now describe which key handlers are defined by default. You can also define new ones as described in Section 63.3.5.

63.4.1 Handlers for Path Management

Key handler $\langle key \rangle / .cd$

This handler causes the default path to be set to $\langle key \rangle$. Note that the default path is reset at the beginning of each call to \pgfkeys to be equal to /.

Example: \pgfkeys{/tikz/.cd,...}

Key handler $\langle key \rangle$ /.is family

This handler sets up things such that when $\langle key \rangle$ is executed, then the current path is set to $\langle key \rangle$. A typical use is the following:

```
\pgfkeys{/tikz/.is family}
\pgfkeys{tikz,line width=1cm}
```

The effect of this handler is the same as if you had written $\langle key \rangle$ /.style= $\langle key \rangle$ /.cd, only the code produced by the /.is family handler is quicker.

63.4.2 Setting Defaults

Key handler $\langle key \rangle / .default = \langle value \rangle$

Sets the default value of $\langle key \rangle$ to $\langle value \rangle$. This means that whenever no value is provided in a call to pgfkeys, then this $\langle value \rangle$ will be used instead.

Example: \pgfkeys{/width/.default=1cm}

Key handler $\langle key \rangle$ /.value required

This handler causes the error message key /erros/value required to be issued whenever the $\langle key \rangle$ is used without a value.

Example: \pgfkeys{/width/.value required}

Key handler $\langle key \rangle$ /.value forbidden

This handler causes the error message key /erros/value forbidden to be issued whenever the $\langle key \rangle$ is used with a value.

This handler works be adding code to the code of the key. This means that you have to define the key first before you can use this handler.

```
\pgfkeys{/my key/.code=I do not want an argument!}
\pgfkeys{/my key/.value forbidden}
\pgfkeys{/my key}  %  Ok
\pgfkeys{/my key=foo} %  Error
```

63.4.3 Defining Key Codes

A number of handlers exist for defining the code of keys.

Key handler $\langle key \rangle / .code = \langle code \rangle$

This handler executes $\polinimes \polinimes \polinime$

It is permissible that $\langle code \rangle$ calls the command \pgfkeys. It is also permissible the $\langle code \rangle$ calls the command \pgfkeysalso, which is useful for styles, see below.

```
\pgfkeys{/par indent/.code={\parindent=#1},/par indent/.default=2em}
\pgfkeys{/par indent=1cm}
...
\pgfkeys{/par indent}
```

Key handler $\langle key \rangle / .ecode = \langle code \rangle$

This handler works like /.code, only the command \pgfkeysedef is used.

Key handler $\langle key \rangle$ /.code 2 args= $\langle code \rangle$

This handler works like /.code, only two arguments rather than one are expected when the $\langle code \rangle$ is executed. This means that when $\langle key \rangle = \langle value \rangle$ is encountered in a key list, the $\langle value \rangle$ should consist of two arguments. For instance, $\langle value \rangle$ could be {first}{second}. Then $\langle code \rangle$ is executed with any occurrence of #1 replaced first and any occurrence of #2 replaced by second.

```
\pgfkeys{/page size/.code 2 args={\paperheight=#2\paperwidth=#1}}
\pgfkeys{/page size={30cm}{20cm}}
```

The second argument is optional: if it is not provided, it will be the empty string.

Because of the special way the $\langle value \rangle$ is parsed, if you set $\langle value \rangle$ to, for instance, first (without any braces), then #1 will be set to f and #2 will be set to irst.

```
Key handler \langle key \rangle/.ecode 2 args=\langle code \rangle
```

This handler works like /.code 2 args, only an \edef is used rather than a \def to define the macro.

```
Key handler \langle key \rangle /.code n args={\langle argument\ count \rangle} {\langle code \rangle}
```

This handler also works like /.code, but you can now specify a number of arguments between 0 and 9 (inclusive).

```
First='A', Second='B' \pgfkeys{/a key/.code n args={2}{First='#1', Second='#2'}} \pgfkeys{/a key={A}{B}}
```

In contrast to $/.code\ 2$ args, there must be exactly $\langle argument\ count \rangle$ arguments, not more and not less and these arguments should be properly delimited.

```
Key handler \langle key \rangle /.ecode n args={\langle argument\ count \rangle} {\langle code \rangle}
```

This handler works like /.code n args, only an \edef is used rather than a \def to define the macro.

```
Key handler \langle key \rangle/.code args={\langle argument\ pattern \rangle}{\langle code \rangle}
```

This handler is the most flexible way to define a /.code key: you can now specify an arbitrary $\langle argument \ pattern \rangle$. Such a pattern is a usual TEX macro pattern. For instance, suppose $\langle argument \ pattern \rangle$ is (#1/#2) and $\langle key \rangle = \langle value \rangle$ is encountered in a key list with $\langle value \rangle$ being (first/second). Then $\langle code \rangle$ is executed with any occurrence of #1 replaced first and any occurrence of #2 replaced by second. So, the actual $\langle value \rangle$ is matched against the $\langle argument \ pattern \rangle$ in the standard TEX way.

```
\pgfkeys{/page size/.code args={#1 and #2}{\paperheight=#2\paperwidth=#1}}
\pgfkeys{/page size=30cm and 20cm}
```

Note that /.code n args should be preferred in case you need just a number of arguments (when the resulting keys are used, /.code n args gobbles spaces between the arguments whereas /.code args considers spaces to be part of the argument).

```
Key handler \langle key \rangle/.ecode args={\langle argument\ pattern \rangle}{\langle code \rangle}
```

This handler works like /.code args, only an \edef is used rather than a \def to define the macro.

There are also handlers for modifying existing keys.

```
Key handler \langle key \rangle /.add code={\langle prefix \ code \rangle}{\langle append \ code \rangle}
```

This handler adds code to an existing key. The $\langle prefix\ code \rangle$ is added to the code stored in $\langle key \rangle$ /. Ccmd at the beginning, the $\langle append\ code \rangle$ is added to this code at the end. Either can be empty. The argument list of $\langle code \rangle$ cannot be changed using this handler. Note that both $\langle prefix\ code \rangle$ and $\langle append\ code \rangle$ may contain parameters like #2.

Key handler $\langle key \rangle$ /.prefix code= $\langle prefix code \rangle$

This handler is a shortcut for $\langle key \rangle$ /.add code={ $\langle prefix \ code \rangle$ }{}. That is, this handler adds the $\langle prefix \ code \rangle$ at the beginning of the code stored in $\langle key \rangle$ /.Qcmd.

Key handler $\langle key \rangle$.append code= $\langle append \ code \rangle$

This handler is a shortcut for $\langle key \rangle$ /.add code={}{ $\langle append\ code \rangle$ }{}.

63.4.4 Defining Styles

The following handlers allow you to define *styles*. A style is a key list that is processed whenever the style is given as a key in a key list. Thus, a style "stands for" a certain key value list. Styles can be parameterized just like normal code.

```
Key handler \langle key \rangle / .style = \langle key \ list \rangle
```

This handler sets things up so that whenever $\langle key \rangle = \langle value \rangle$ is encountered in a key list, then the $\langle key | list \rangle$, with every occurrence of #1 replaced by $\langle value \rangle$, is processed instead. As always, if no $\langle value \rangle$ is given, the default value is used, if defined, or the special value \pgfkeysnovalue.

You can achieve the same effect by writing $\langle key \rangle$.code=\pgfkeysalso{ $\langle key | list \rangle$ }. This means, in particular, that the code of a key could also first execute some normal code and only then process some further keys.

```
\pgfkeys{/par indent/.code={\parindent=#1}}
\pgfkeys{/no indent/.style={/par indent=0pt}}
\pgfkeys{/normal indent/.style={/par indent=2em}}
\pgfkeys{/no indent}
...
\pgfkeys{/normal indent}
```

The following example shows a parameterized style "in action".

Key handler $\langle key \rangle$ / .estyle= $\langle key | list \rangle$

This handler works like /.style, only the $\langle code \rangle$ is set using \edef rather than \def. Thus, all macros in the $\langle code \rangle$ are expanded prior to saving the style.

For styles the corresponding handlers as for normal code exist:

```
Key handler \langle key \rangle/.style 2 args=\langle key \ list \rangle
```

This handler works like /.code 2 args, only for styles. Thus, the $\langle key \ list \rangle$ may contain occurrences of both #1 and #2 and when the style is used, two parameters must be given as $\langle value \rangle$.

```
\pgfkeys{/paper height/.code={\paperheight=#1},/paper width/.code={\paperwidth=#1}}
\pgfkeys{/page size/.style 2 args={/paper height=#1,/paper width=#2}}
\pgfkeys{/page size={30cm}{20cm}}
```

Key handler $\langle key \rangle$ / .estyle 2 args= $\langle key \ list \rangle$

This handler works like /.style 2 args, only an \edef is used rather than a \def to define the macro.

Key handler $\langle key \rangle$ /.style args={ $\langle argument\ pattern \rangle$ }{ $\langle key\ list \rangle$ }

This handler works like /.code args, only for styles.

```
Key handler \langle key \rangle /.estyle args={\langle argument\ pattern \rangle} {\langle code \rangle}
```

This handler works like /.ecode args, only for styles.

```
Key handler \langle key \rangle / .style \ n \ args = \{\langle argument \ count \rangle\} \langle key \ list \rangle
```

This handler works like /.code n args, only for styles. Here, $\langle key \; list \rangle$ may depend on all $\langle argument \; count \rangle$ parameters.

```
Key handler \langle key \rangle/.add style={\langle prefix \ key \ list \rangle}{\langle append \ key \ list \rangle}
```

This handler works like /.add code, only for styles. However, it is permissible to add styles to keys that have previously been set using /.code. (It is also permissible to add normal $\langle code \rangle$ to a key that has previously been set using /.style). When you add a style to a key that was previously set using /.code, the following happens: When $\langle key \rangle$ is processed, the $\langle prefix \ key \ list \rangle$ will be processed first, then the $\langle code \rangle$ that was previously stored in $\langle key \rangle$ /.@cmd, and then the keys in $\langle append \ key \ list \rangle$ are processed.

```
Key handler \langle key \rangle /.prefix style=\langle prefix key list \rangle
```

Works like /.add style, but only for the prefix key list.

```
Key handler \langle key \rangle /.append style=\langle append \ key \ list \rangle
```

Works like /.add style, but only for the append key list.

63.4.5 Defining Value-, Macro-, If- and Choice-Keys

For some keys, the code that should be executed for them is rather "specialized." For instance, it happens often that the code for a key just sets a certain T_EX -if to true or false. For these cases, predefined handlers make it easier to install the necessary code.

However, we start with some handlers that are used to manage the value that is directly stored in a key.

```
Key handler \langle key \rangle/.initial=\langle value \rangle
```

This handler sets the value of $\langle key \rangle$ to $\langle value \rangle$. Note that no subkeys are involved. After this handler has been used, by the rules governing keys, you can subsequently change the value of the $\langle key \rangle$ by just writing $\langle key \rangle = \langle value \rangle$. Thus, this handler is used to set the initial value of key.

```
\pgfkeys{/my key/.initial=red}
% "/my key" now stores the value "red"
\pgfkeys{/my key=blue}
% "/my key" now stores the value "blue"
```

Note that with this configuration, writing \pgfkeys{/my key} will not have the effect you might expect (namely that blue is inserted into the main text). Rather, /my key will be promoted to /my key=\pgfkeysnovalue and, thus, \pgfkeysnovalue will be stored in /my key.

To retrieve the value stored in a key, the handler /.get is used.

```
Key handler \langle key \rangle / .get = \langle macro \rangle
```

Executes a \let command so that $\langle macro \rangle$ contains the contents stored in $\langle key \rangle$.

```
blue \pgfkeys{/my key/.initial=red}
  \pgfkeys{/my key=blue}
  \pgfkeys{/my key/.get=\mymacro}
  \mymacro
```

```
Key handler \langle key \rangle / .add = {\langle prefix value \rangle} {\langle append value \rangle}
```

Adds the $\langle prefix\ value \rangle$ and the beginning and the $\langle append\ value \rangle$ at the end of the value stored in $\langle key \rangle$.

```
Key handler \langle key \rangle / .prefix={\langle prefix \ value \rangle}
```

Adds the $\langle prefix \ value \rangle$ and the beginning of the value stored in $\langle key \rangle$.

```
Key handler \langle key \rangle / .append = \{ \langle append \ value \rangle \}
```

Adds the $\langle append\ value \rangle$ at the end of the value stored in $\langle key \rangle$.

```
Key handler \langle key \rangle / .link = \langle another key \rangle
```

Stores the value $\protect\operatorname{\sc Npgfkeysvalueof}\{\another\ key\}\}\$ in the $\another\ key\}$. The idea is that when you expand the $\another\ key\}$, the value of $\another\ key\}$ is expanded instead. This corresponds loosely to the notion of soft links in Unix, hence the name.

The next handler is useful for the common situation where $\langle key \rangle = \langle value \rangle$ should cause the $\langle value \rangle$ to be stored in some macro. Note that, typically, you could just as well store the value in the key itself.

```
Key handler \langle key \rangle /.store in=\langle macro \rangle
```

This handler has the following effect: When you write $\langle key \rangle = \langle value \rangle$, the code $\backslash def(macro) \{ \langle value \rangle \}$ is executed. Thus, the given value is "stored" in the $\langle macro \rangle$.

```
Hello Gruffalo!
\pgfkeys{/text/.store in=\mytext}
\def\a{world}
\pgfkeys{/text=Hello \a!}
\def\a{Gruffalo}
\mytext
```

Key handler $\langle key \rangle$ /.estore in= $\langle macro \rangle$

This handler is similar to /.store in, only the code $\ensuremath{\mbox{edef}\langle macro\rangle}\{\langle value\rangle\}$ is used. Thus, the macro-expanded version of $\langle value\rangle$ is stored in the $\langle macro\rangle$.

```
Hello world!
    \pgfkeys{/text/.estore in=\mytext}
    \def\a{\world}
    \pgfkeys{/text=Hello \a!}
    \def\a{Gruffalo}
    \mytext
```

In another common situation a key is used to set a TEX-if to true or false.

```
Key handler \langle key \rangle / .is if=\langle T_E X - if name \rangle
```

This handler has the following effect: When you write $\langle key \rangle = \langle value \rangle$, it is first checked that $\langle value \rangle$ is true or false (the default is true if no $\langle value \rangle$ is given). If this is not the case, the error key /errors/boolean expected is executed. Otherwise, the code $\langle TEX-if\ name \rangle \langle value \rangle$ is executed, which sets the TEX-if accordingly.

```
Round? \newif\iftheworldisflat \pgfkeys{/flat world/.is if=theworldisflat} \pgfkeys{/flat world=false} \iftheworldisflat Flat \else Round? \fi
```

The next handler deals with the problem when a $\langle key \rangle = \langle value \rangle$ makes sense only for a small set of possible $\langle value \rangle$ s. For instance, the line cap can only be rounded or rect or butt, but nothing else. For this situation the following handler is useful.

```
Key handler \langle key \rangle/.is choice
```

This handler sets things up so that writing $\langle key \rangle = \langle value \rangle$ will cause the subkey $\langle key \rangle / \langle value \rangle$ to be executed. So, each of the different possible choices should be given by a subkey of $\langle key \rangle$.

```
\pgfkeys{/line cap/.is choice}
\pgfkeys{/line cap/round/.style={\pgfsetbuttcap}}
\pgfkeys{/line cap/butt/.style={\pgfsetroundcap}}
\pgfkeys{/line cap/rect/.style={\pgfsetrectcap}}
\pgfkeys{/line cap/rectangle/.style={/line cap=rect}}
...
\draw [/line cap=butt] ...
```

If the subkey $\langle key \rangle / \langle value \rangle$ does not exist, the error key /errors/unknown choice value is executed.

63.4.6 Expanded and Multiple Values

When you write $\langle key \rangle = \langle value \rangle$, you usually wish to use the $\langle value \rangle$ "as is." Indeed, great care is taken to ensure that you can even use things like #1 or unbalanced TeX-ifs inside $\langle value \rangle$. However, sometimes you want the $\langle value \rangle$ to be expanded before it is used. For instance, $\langle value \rangle$ might be a macro name like \mymacro and you do not want \mymacro to be used as the macro, but rather the *contents* of \mymacro. Thus, instead of using $\langle value \rangle$, you wish to use whatever $\langle value \rangle$ expands to. Instead of using some fancy \expandafter hackery, you can use the following handlers:

Key handler $\langle key \rangle$ / .expand once= $\langle value \rangle$

This handler expands $\langle value \rangle$ once (more precisely, it executes an \expandafter command on the first token of $\langle value \rangle$) and then process the resulting $\langle result \rangle$ as if you had written $\langle key \rangle = \langle result \rangle$. Note that if $\langle key \rangle$ contains a handler itself, this handler will be called normally.

```
Key 1: \c
Key 2: \b
Key 3: \a
Key 4: bottom
```

```
\def\a{bottom}
\left( \frac{b}{a} \right)
\left( \left( b\right) \right)
\pgfkeys{/key1/.initial=\c}
\pgfkeys{/key2/.initial/.expand once=\c}
\pgfkeys{/key3/.initial/.expand twice=\c}
\pgfkeys{/key4/.initial/.expanded=\c}
\def\a{{\ttfamily\string\a}}
\def\b{{\ttfamily\string\b}}
\def\c{{\ttfamily\string\c}}
\begin{tabular}{11}
Key 1:& \pgfkeys{/key1} \\
Key 2:& \pgfkeys{/key2} \\
Key 3:& \pgfkeys{/key3} \\
Key 4:& \pgfkeys{/key4}
\end{tabular}
```

Key handler $\langle key \rangle$ / .expand twice= $\langle value \rangle$

This handler works like saying $\langle key \rangle$.expand once / expand once = $\langle value \rangle$.

Key handler $\langle key \rangle$ / .expanded= $\langle value \rangle$

This handler will completely expand $\langle value \rangle$ (using \edef) before processing $\langle key \rangle = \langle result \rangle$.

Key handler $\langle key \rangle$ /.list= $\langle comma\text{-}separated\ list\ of\ values \rangle$

This handler causes the key to be used repeatedly, namely once for every element of the list of values. Note that the list of values should typically be surrounded by braces since, otherwise, TEX will not be able to tell whether a comma starts a new key or a new value.

The $\langle list\ of\ values \rangle$ is processed using the \foreach statement, so you can use the ... notation.

63.4.7 Handlers for Forwarding

Key handler $\langle key \rangle$ /.forward to= $\langle another key \rangle$

This handler causes the $\langle key \rangle$ to "forward" its argument to $\langle another \ key \rangle$. When the $\langle key \rangle$ is used, its normal code will be executed first. Then, the value is (additionally) passed to $\langle another \ key \rangle$. If the

 $\langle key \rangle$ has not yet been defined prior to the use of .forward to, it will be defined then (and do nothing by itself, expect for forwarding it to $\langle key \ name \rangle$). The $\langle another \ key \rangle$ must be a fully qualified key name.

```
(b:1)(a:1) (a:2) 

\pgfkeys{
    /a/.code=(a:#1),
    /b/.code=(b:#1),
    /b/.forward to=/a,
    /c/.forward to=/a
}
\pgfkeys{/b=1} \pgfkeys{/c=2}
```

Key handler $\langle key \rangle$ /.search also={ $\langle path \ list \rangle$ }

A style which installs a /.unknown handler into $\langle key \rangle$. This /.unknown handler will then search for unknown keys in every path provided in $\{\langle path \ list \rangle\}$.

Invoking /secondary path/option with 'value'

```
% define a key:
\pgfkeys{/secondary path/option/.code={Invoking /secondary path/option with '#1'}}
% set up a search path:
\pgfkeys{/main path/.search also={/secondary path}}
% try searching for 'option=value' in '/main path':
% -> this finds '/secondary path/option'!
\pgfkeys{/main path/.cd,option=value}
```

The /.search also handler follows the strategy

- 1. If a user provides a fully qualified key which could not be found, for example the full string /main path/option, it assume that the user knew what she is doing and does not continue searching for an option in {\langle path \ list \rangle \}.
- 2. If a user provides only the key's name, for example option and option cannot be found in the current default path (which is /main path in our example above), the current default path is set to the next element in {\path list\} (which is /secondary path here) and \pgfkeys will be restarted. This will be iterated until either a match has been found or all elements in {\path list\} have been tested.
- 3. If all elements in {\(\text{path list} \) \} have been checked and the key is still unknown, the fall-back handler \(\text{/handlers/.unknown} \) will be invoked.

Invoking /secondary path/option with 'value' Found unknown option /main path/option=value!

```
% define a key:
\pgfkeys{/secondary path/option/.code={Invoking /secondary path/option with '#1'}}

% set up a search path:
\pgfkeys{/main path/.search also={/secondary path}}

% try searching for 'option=value' in '/main path':
% -> this finds '/secondary path/option'!
\pgfkeys{/main path/.cd,option=value}

% negative example:
% try searching for fully qualified key /main path/option.
% This won't be handled by .search also.
\pgfkeys{/handlers/.unknown/.code={Found unknown option \pgfkeyscurrentkeyRAW={#1}!}}%
\pgfkeys{/main path/.cd,/main path/option=value}
```

Please note that the strategy of /.search also is different from the first example provided in section 63.3.6 "Unknown Keys" because /.search also only applies to keys that are not fully qualified. For those who are familiar with \pgfkeys, the actual implementation of /.search also might be interesting:

1. \pgfkeys{/path/.search also={/tikz}} is equivalent to

```
\pgfkeys{/path/.unknown/.code={%
      \ifpgfkeysaddeddefaultpath
          % only process keys for which no full path has been
          % provided:
          \pgfkeyssuccessfalse
          \let\pgfkeys@searchalso@name =\pgfkeyscurrentkeyRAW
          \ifpgfkeyssuccess
          \else
             % search with /tikz as default path:
             \pgfqkeys{/tikz}{\pgfkeys@searchalso@name={#1}}%
         \fi
      \else
          \pgfutilnext
      \fi
   }
```

2. \pgfkeys{/path/.search also={/tikz,/pgf}} is equivalent to

```
\pgfkeys{/path/.unknown/.code={%
        \ifpgfkeysaddeddefaultpath
            \pgfkeyssuccessfalse
            \let\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
            \ifpgfkeyssuccess
            \else
                % step 1: search in /tikz with .try:
                \pgfqkeys{/tikz}{\pgfkeys@searchalso@name/.try={#1}}%
            \fi
            \ifpgfkeyssuccess
            \else
                % step 2: search in /pgf (without .try!):
                \pgfqkeys{/pgf}{\pgfkeys@searchalso@name={#1}}%
            \fi
            \def\pgfutilnext{\pgfkeysvalueof {/handlers/.unknown/.@cmd}#1\pgfeov}%
            \pgfutilnext
        \fi
```

To also enable searching for styles (or other handled keys), consider changing the configuration for handled keys to /hander config=full or existing when you use /.search also, that is, use

```
\pgfkeys{
  /main path/.search also={/secondary path},
  /handler config=full or existing}
```

63.4.8 Handlers for Testing Keys

Key handler $\langle key \rangle / .try = \langle value \rangle$

This handler causes the same things to be done as if $\langle key \rangle = \langle value \rangle$ had been written instead. However, if neither $\langle key \rangle / .0$ cmd nor the key itself is defined, no handlers will be called. Instead, the execution of the key just stops. Thus, this handler will "try" to use the key, but no further action is taken when the key is not defined.

The TEX-if \ifpgfkeyssuccess will be set according to whether the $\langle key \rangle$ was successfully executed or not

```
(a:hallo)(b:welt)
\pgfkeys{/a/.code=(a:#1)}
\pgfkeys{/b/.code=(b:#1)}
\pgfkeys{/x/.try=hmm,/a/.try=hallo,/b/.try=welt}
```

Key handler $\langle key \rangle / .retry = \langle value \rangle$

This handler works just like /.try, only it will not do anything if \ifpgfkeyssuccess is false. Thus, this handler will only retry to set a key if "the last attempt failed".

```
(a:hallo) \pgfkeys{/a/.code=(a:#1)} \pgfkeys{/b/.code=(b:#1)} \pgfkeys{/x/.try=hmm,/a/.retry=hallo,/b/.retry=welt}
```

Key handler $\langle key \rangle$ /.lastretry= $\langle value \rangle$

This handler works like /.retry, only it will invoke the usual handlers for unknowns keys if \ifpgfkeyssuccess is false. Thus, this handler will only try to set a key if "the last attempt failed". Furthermore, this here is the last such attempt.

63.4.9 Handlers for Key Inspection

Key handler $\langle key \rangle$ /.show value

This handler executes a \show command on the value stored in $\langle key \rangle$. This is useful mostly for debugging.

Example: \pgfkeys{/my/obscure key/.show value}

Key handler $\langle key \rangle$ /.show code

This handler executes a \show command on the code stored in $\langle key \rangle$ /.@cmd. This is useful mostly for debugging.

Example: \pgfkeys{/my/obscure key/.show code}

The following key is not a handler, but it also commonly used for inspecting things:

/utils/exec= $\langle code \rangle$ (no default)

This key will simply execute the given $\langle code \rangle$.

Example: \pgfkeys{some key=some value,/utils/exec=\show\hallo,obscure key=obscure}

63.5 Error Keys

In certain situations errors can occur, like using an undefined key. In these situations error keys are executed. They should store a macro that gets two arguments: The first is the offending key (possibly only after macro expansion), the second is the value that was passed as a parameter (also possibly only after macro expansion).

Currently, error keys are simply executed. In the future it might be a good idea to have different subkeys that are executed depending on the language currently set so that users get a localized error message.

```
/errors/value required=\{\langle offending \ key \rangle\} \{\langle value \rangle\} (no default)
```

This key is executed whenever an $\langle offending \ key \rangle$ is used without a value when a value is actually required.

```
/errors/value forbidden=\{\langle offending \ key \rangle\} \{\langle value \rangle\} (no default)
```

This key is executed whenever a key is used with a value when a value is actually forbidden.

```
/errors/boolean expected=\{\langle offending \ key \rangle\} \{\langle value \rangle\} (no default)
```

This key is executed whenever a key set up using /.is if gets called with a $\langle value \rangle$ other than true or false.

```
/errors/unknown choice value=\{\langle offending\ key \rangle\} \{\langle value \rangle\} (no default)
```

This key is executed whenever a choice is used as a $\langle value \rangle$ for a key set up using the /.is choice handler that is not defined.

```
/errors/unknown key=\{\langle offending \ key \rangle\} \{\langle value \rangle\} (no default)
```

This key is executed whenever a key is unknown and no specific /.unknown handler is found.

63.6 Key Filtering

An extension by Christian Feuersänger

Normally, a call to \pgfkeys sets all keys provided in its argument list. This is usually what users expect it to do. However, implementations of different packages or PGF-libraries may need more control over the key setting procedure: library A may want to set its options directly and communicate all remaining ones to library B.

This section describes key filtering methods of PGF, including options for family groupings. If you merely want to use PGF (or its libraries), you can skip this section. It is addressed to package (or library) authors.

63.6.1 Starting With An Example

Users of xkeyval are familiar with the concept of key families: keys belong to groups and those keys can be 'filtered' out of other options. PGF supports family groupings and more abstract key selection mechanism with \pgfkeysfiltered, a variant of \pgfkeys. Suppose we have the example key grouping

```
\pgfkeys{
/my group/A1/.code=(A1:#1),
/my group/A2/.code=(A2:#1),
/my group/A3/.code=(A3:#1),
/my group/B/.code=(B:#1),
/my group/C/.code=(B:#1),
}
```

and we want to set options A1, A2 and A3 only. A call to \pgfkeys yields

```
(A1:a1)(A2:a2)(B:b)(B:c) \pgfkeys{/my group/A1=a1, /my group/A2=a2, /my group/B=b, /my group/C=c}
```

because all those command option are processed consecutively.

Now, let's define a family named A which contains A1, A2 and A3 and set only family members of A. We prepare our key settings with

```
\pgfkeys{
/my group/A/.is family,
/my group/A1/.belongs to family=/my group/A,
/my group/A2/.belongs to family=/my group/A,
/my group/A3/.belongs to family=/my group/A,
}
```

and

```
\pgfkeys{/pgf/key filters/active families/.install key filter}
```

After this preparation, we can use \pgfkeysfiltered with

```
(A1:a1)(A2:a2) \pgfkeys{/my group/A/.activate family} \pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2, /my group/B=b, /my group/C=c}
```

or

```
(A1:a1)(A2:a2)(A3:a3) \pgfkeys{/my group/A/.activate family} \pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2, /my group/B=b, /my group/C=c, /tikz/color=blue, /my group/A3=a3}
```

to set only keys which belong to an 'active' family – in our case, only family A was active, so the remaining options have not been processed. The family processing is quite fast and allows an arbitrary number of active key families.

Unprocessed options can be collected into a macro (similar to xkeyval's \xkv@rm), discarded or handled manually. The details of key selection and family declaration are described in the following sections.

63.6.2 Setting Filters

The command \pgfkeysfiltered is the main tool to process only selected options. It works as follows.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

Processes all options in exactly the same way as $\texttt{pgfkeys}\{\langle key\text{-}value\text{-}list\rangle\}$, but a key filter is considered as soon as key identification is complete.

The key filter tells \pgfkeysfiltered whether it should continue to apply the current option (return value is 'true') or whether something different shall be done (filter returns 'false').

There is exactly one key filter in effect, and it is installed by the .install key filter handler or by \pgfkeysinstallkeyfilter.

If the key filter returns 'false', a unique key filter handler gets control. This handler is installed by the .install key filter handler method and has access to the key's full name, value and (possibly) path.

Key filtering applies to any (possibly nested) call to pgfkeys, pgfkeysalso, pgfqkeys and pgfqkeysalso during the evaluation of $\{\langle key\text{-}value\text{-}list\rangle\}$. It does not apply to routines like pgfkeyssetvalue or pgfkeysgetvalue. Furthermore, keys belonging to pgfkeysgetvalue routines can't be nested: you can't combine different key filters automatically.

$\protect\operatorname{pgfqkeysfiltered}(\langle \operatorname{default-path}\rangle) \{\langle \operatorname{key-value-list}\rangle\}$

A variant of \pgfkeysfiltered which uses the 'quick' search path setting. It is the \pgfqkeys variant of \pgfkeysfiltered, see the documentation for \pgfqkeys for more details.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

A variant of $\protect\operatorname{\belowdex}$ as which loads its key list from $\{\langle macro \rangle\}$.

It is useful in conjunction with the pgf/key filter handlers/append filtered to=macro handler.

The following example uses the same settings as in the intro section 63.6.1.

```
(A1:a1)(A2:a2)(A3:a3)Remaining: '/my group/B=b,/my group/C=c,/tikz/color=blue'.(B:b)(B:c)
```

```
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=\remainingoptions}
\def\remainingoptions{}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b, /my group/C=c, /tikz/color=blue, /my group/A3=a3}

Remaining: '\remainingoptions'.
\pgfkeysalsofrom{\remainingoptions}
```

$\protect\operatorname{pgfkeysalsofiltered}\{\langle key-value-list\rangle\}$

This command works as \pgfkeysfiltered, but it does not change the current default path. See the documentation of \pgfkeysalso for more details.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

A variant of \pgfkeysalsofiltered which loads its key list from $\{\langle macro \rangle\}$.

```
Key handler \langle key \rangle/.install key filter=\langle optional \ arguments \rangle
```

This handler installs a key filter. A key filter is a command key which sets the TEX-boolean \ifpgfkeysfiltercontinue, that means a key with existing '/.@cmd' suffix. A simple example is a key filter which returns always true:

```
\pgfkeys{/foo/bar/true key filter/.code={\pgfkeysfiltercontinuetrue}}
\pgfkeys{/foo/bar/true key filter/.install key filter}
```

If key filters require arguments, they are installed by .install key filter as well. An example is the /pgf/key filters/equals handler:

```
(A1:a1) \pgfkeys{/pgf/key filters/equals/.install key filter={/my group/A1}} \pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2, /my group/B=b, /my group/C=c, /tikz/color=blue, /my group/A3=a3}
```

If a key filter requires more than one argument, you need to provide the complete argument list in braces like {{first}{second}}.

You can also use $\protect\operatorname{pgfkeysinstallkeyfilter}\langle full\ key\rangle\langle optional\ arguments\rangle$, it has the same effect. See section 63.6.7 for how to write key filters.

Key handler $\langle key \rangle$ /.install key filter handler= $\langle optional \ arguments \rangle$

This handler installs the routine which will be invoked for every *unprocessed* option, that means any option for which the key filter returned 'false'.

The .install key filter handler is used in the same way as .install key filter. There exists a macro version, $\protect\prote$

See section 63.6.7 for how to write key filter handlers.

63.6.3 Handlers For Unprocessed Keys

Each option for which key filters decided to skip them is handed over to a 'key filter handler'. There are several predefined key filter handlers.

```
/pgf/key filter handlers/append filtered to=\{\langle macro \rangle\}
```

(no default)

Install this filter handler to append any unprocessed options to macro $\{\langle macro \rangle\}$.

```
(A1:a1)(A2:a2)Remaining options: '/my group/B=b,/my group/C=c,/tikz/color=blue'.
```

```
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=\remainingoptions}
\def\remainingoptions{}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b, /my group/C=c, /tikz/color=blue}
Remaining options: '\remainingoptions'.
```

This example uses the same keys as defined in the intro section 63.6.1.

```
/pgf/key filter handlers/ignore
```

(no value)

Install this filter handler if you simply want to ignore any unprocessed option. This is the default.

```
/pgf/key filter handlers/log
```

(no value

This key filter handler writes messages for any unprocessed option to your logfile (and terminal).

63.6.4 Family Support

PGF supports a family concept: every option can be associated with (at most) one family. Families form loose key groups which are independent of the key hierarchy. For example, /my tree/key1 can belong to family /tikz.

It is possible to 'activate' or 'deactivate' single families. Furthermore, it is possible to set only keys which belong to active families using appropriate key filter handlers.

The family support is fast: if there are N options in a key-value-list and there are K active families, the runtime for $\operatorname{pgfkeysfiltered}$ is O(N+K) (activate every family O(K)), check every option O(N), deactivate every family O(K)).

```
Key handler \langle key \rangle/.is family
```

Defines a new family. This option has already been described in section 63.4.1 on page 641.

```
Key handler \langle key \rangle /.activate family
```

Activates a family. The family needs to be defined, otherwise /errors/family unknown will be raised. Activation means a T_FX-boolean will be set to true, indicating that a family should be processed.

You can also use $\protect\operatorname{\sc high same}$ to get the same effect. Furthermore, you can use $\protect\operatorname{\sc high same}$ to activate a list of families (see section 63.6.6).

```
Key handler \langle key \rangle/.deactivate family
```

Deactivates a family. The family needs to be defined, otherwise /errors/family unknown will be raised.

You can also use $\protect\pr$

```
Key handler \langle key \rangle /.belongs to family={\langle family \ name \rangle}
```

Associates the current option with $\{\langle family\ name \rangle\}$, which is expected to be a full path of a family.

```
\pgfkeys{/foo/bar/.is family}
\pgfkeys{
/foo/a/.belongs to family=/foo/bar,
/foo/b/.belongs to family=/foo/bar
}
```

Each option can have up to one family, .belongs to family overwrites any old setting.

/pgf/key filters/active families

(no value)

Install this key filter if \pgfkeysfiltered should only process activated families. If a key does not belong to any family, it is not processed. If a key is completely unknown within the default path, the normal 'unknown' handlers of \pgfkeys are invoked.

```
/pgf/key filters/active families or no family=\{\langle key \ filter \ 1 \rangle\} \{\langle key \ filter \ 2 \rangle\} (no default)
```

This key filter configures \pgfkeysfiltered to work as follows.

- If the current key belongs to a family, set \ifpgfkeysfiltercontinue to true if and only if its family is active.
- 2. If the current key does *not* belong to a family, assign \ifpgfkeysfiltercontinue as result of $\{\langle key \ filter \ 1 \rangle\}$.
- 3. If the current key is unknown within the default path, assign \ifpgfkeysfiltercontinue as result of $\{\langle key \ filter \ 2 \rangle\}$.

The arguments $\{\langle key \text{ filter } 1 \rangle\}$ and $\{\langle key \text{ filter } 2 \rangle\}$ are other key filters (possibly with options) and allow fine-grained control over the filtering process.

```
\pgfkeysinstallkeyfilter
{/pgf/key filters/active families or no family}
{{/pgf/key filters/is descendant of=/tikz}% for keys without family
{/pgf/key filters/false}% for unknown keys
}%
```

This key filter will return true for any option with active family. If an option has no family, the return value is true if and only if it belongs to /tikz. If the option is unknown, the return value is false and unknown handlers won't be called.

```
/pgf/key filters/active families or no family DEBUG=\{\langle key \ filter \ 1 \rangle\}\{\langle key \ filter \ 2 \rangle\} (no default) A variant of active families or no family which protocols each action on your terminal (log-file).
```

```
/pgf/key filters/active families and known (no value)
```

A fast alias for

```
/pgf/key filters/active families or no family=
  {/pgf/keys filters/false}
  {/pgf/keys filters/false}.
```

```
/pgf/key filters/active families or descendants of=\{\langle path \ prefix \rangle\} (no default)
```

A fast alias for

```
/pgf/key filters/active families or no family= {/pgf/keys filters/is descendant of={\langle path\ prefix \rangle}} {/pgf/keys filters/false}.
```

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\be
```

A simple shortcut macro which activates any family in the comma separated $\{\langle family\ list \rangle\}$, invokes $\protect\operatorname{pgfkeysfiltered}\langle key\text{-}value\text{-}list \rangle$ and deactivates the families afterwards.

Please note that you will need to install a family key filter, otherwise family activation has no effect.

```
\verb|\pgfqkeysactivatefamiliesandfilteroptions{|\langle family list \rangle} {\langle default path \rangle} {\langle key-value-list \rangle}
```

The 'quick' default path variant of \pgfkeysactivatefamiliesandfilteroptions.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

A shortcut macro which activates a single family and invokes \pgfkeysfiltered.

Please note that you will need to install a family key filter, otherwise family activation has no effect.

```
\verb|\pgfqkeysactivatesinglefamilyandfilteroptions{|\langle family\ name \rangle} {\langle default\ path \rangle} {\langle key-value-list \rangle} |
```

The 'quick' default path variant of \pgfkeysactivatesinglefamilyandfilteroptions.

63.6.5 Other Key Filters

There are some more key filters which have nothing to do with family handling.

```
/pgf/key filters/is descendant of=\{\langle path \rangle\}
```

(no default)

Install this key filter to process only options belonging to the key tree $\langle path \rangle$. It returns true for every key whose key path is equal to $\langle path \rangle$. It also returns true for any unknown key, that means unknown keys are processed using the standard unknown handlers of PGF.

```
(A:a)(B:b) \pgfkeys{
   /group 1/A/.code={(A:#1)},
   /group 1/foo/bar/B/.code={(B:#1)},
   /group 2/C/.code={(C:#1)},
   /pgf/key filters/is descendant of/.install key filter=/group 1}
   \pgfkeysfiltered{/group 1/A=a,/group 1/foo/bar/B=b,/group 2/C=c}
```

```
/pgf/key filters/equals=\{\langle full \ key \rangle\}
```

(no default)

Install this key filter to process only the fully qualified option $\{\langle full\ key\rangle\}$. The filter returns true for any unknown key or if the key equals $\{\langle full\ key\rangle\}$.

```
(A:a) \pgfkeys{
   /group 1/A/.code={(A:#1)},
   /group 1/B/.code={(B:#1)},
   /pgf/key filters/equals/.install key filter=/group 1/A}
   \pgfqkeysfiltered{/group 1}{A=a,B=b}
```

```
/pgf/key filters/not=\{\langle key \ filter \rangle\}
```

(no default)

This key filter logically inverts the result of $\{\langle key \ filter \rangle\}$.

```
(C:c) \pgfkeys{
   /group 1/A/.code={(A:#1)},
   /group 1/foo/bar/B/.code={(B:#1)},
   /group 2/C/.code={(C:#1)},
   /pgf/key filters/not/.install key filter=
   {/pgf/key filters/is descendant of=/group 1}}
   \pgfkeysfiltered{/group 1/A=a,/group 1/foo/bar/B=b,/group 2/C=c}
```

Please note that unknown keys will be handed to the usual unknown handlers.

```
/pgf/key filters/and=\{\langle key \ filter \ 1 \rangle\} \{\langle key \ filter \ 2 \rangle\}
```

(no default)

This key filter returns true if and only if both, $\{\langle key \ filter \ 1 \rangle\}$ and $\{\langle key \ filter \ 2 \rangle\}$ return true.

```
/pgf/key filters/or={\langle key filter 1 \rangle}{\langle key filter 2 \rangle}
```

(no default)

This key filter returns true if one of $\{\langle key \text{ filter } 1 \rangle\}$ and $\{\langle key \text{ filter } 2 \rangle\}$ returns true.

```
/pgf/key filters/true
```

(no value)

This key filter returns always true.

/pgf/key filters/false

(no value)

This key filter returns always false (including unknown keys).

/pgf/key filters/defined

(no value)

This key filter returns false if the current key is unknown, which avoids calling the unknown handlers.

63.6.6 Programmer Interface

\pgfkeysinterruptkeyfilter

 $\langle environment\ contents \rangle$

\endpgfkeysinterruptkeyfilter

Temporarily disables key filtering inside the environment. If key filtering is not active, this has no effect at all.

Please note that no T_EX-group is introduced.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Creates $\{\langle macro \rangle\}$ which contains commands to re-activate the current key filter and key filter handler. It can be used to temporarily switch the key filter.

$\protect\operatorname{pgfkeysinstallkeyfilter} \{\langle full\ key \rangle\} \{\langle optional\ arguments \rangle\}$

The command $\pgfkeysinstallkeyfilter{\langle full key \rangle}{\langle optional arguments \rangle}$ has the same effect as $\pgfkeys{\langle full key \rangle}$.install key filter={ $\langle optional arguments \rangle}$ }.

$\protect\pro$

The command $\pgfkeysinstallkeyfilterhandler{\langle full key \rangle} {\langle optional arguments \rangle}$ has the same effect as $\pgfkeys{\langle full key \rangle}$.install key filter handler={ $\langle optional arguments \rangle}$ }.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Equivalent to $pgfkeys{\langle family \ name \rangle / .activate family}.$

$\protect\operatorname{\mathtt{Npgfkeysdeactivatefamily}} \$

Equivalent to $pgfkeys{\langle family \ name \rangle}/.deactivate family}$.

$\protect\operatorname{\mathtt{f pgfkeysactivatefamilies}} \{\langle family\ list \rangle\} \{\langle deactivate\ macro\ name \rangle\}$

Activates each family in $\langle family \ list \rangle$ and creates a macro $\langle deactivate \ macro \ name \rangle$ which deactivates each family in $\langle family \ list \rangle$.

```
\pgfkeysactivatefamilies{/family 1,/family 2,/family 3}{\deactivatename}
\pgfkeysfiltered{foo,bar}
\deactivatename
```

$\protect\pro$

Checks whether the full key $\langle family \rangle$ is a family and executes either $\langle true\ case \rangle$ or $\langle false\ case \rangle$.

$\protect\operatorname{\begin{tabular}{l} \protect\protec$

Sets the TeX-boolean \ifpgfkeysfiltercontinue to whether $\langle family \rangle$ is active or not.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\b$

Returns the family associated to a full key $\langle key \rangle$ into macro $\langle resultmacro \rangle$.

$\protect\pro$

The command $\pgfkeyssetfamily{\langle full\ key\rangle}{\langle family\rangle}$ has the same effect as $\pgfkeys{\langle full\ key\rangle}$. belongs to family= $\{\langle family\rangle\}$ }.

63.6.7 Defining Own Filters Or Filter Handlers

During \pgfkeysfiltered, the key filter code will be invoked. At this time, the full key path including key name is available as \pgfkeyscurrentkey, the key name before default paths have been considered as \pgfkeyscurrentkeyRAW and the values as \pgfkeyscurrentvalue.

Furthermore, the macro \pgfkeyscasenumber contains the current key's type as an integer:

- $\langle 1 \rangle$ The key is a command key (i.e. .../.@cmd exists).
- $\langle 2 \rangle$ The key contains its value directly.

- \(3\)\) The key is handled (for example it is .code or .cd).
 In this case, the macros \pgfkeyscurrentname and \pgfkeyscurrentpath are set to the handlers name and path, respectively. Invoke \pgfkeyssplitpath{} to extract these values for non-handled keys.
- $\langle \theta \rangle$ The key is unknown.

Any key filter or key filter handler can access these variables. Key filters are expected to set the TeX-boolean \ifpgfkeysfiltercontinue to whether the current key shall be processed or not.

 $\verb|\pgfkeysevalkeyfilterwith{|\langle full\ key\rangle|} = \{\langle filter\ arguments\rangle\}|$

Evaluates a fully qualified key filter $\langle full \ key \rangle$ with argument(s) $\langle filter \ arguments \rangle$.

\pgfkeysevalkeyfilterwith{/pgf/key filters/equals=/tikz}

64 Repeating Things: The Foreach Statement

This section describes the package pgffor, which is loaded automatically by TikZ, but not by PGF:

```
\usepackage{pgffor} % MEX
\input pgffor.tex % plain TEX
\usemodule[pgffor] % ConTEXt
```

This package can be used independently of PGF, but works particularly well together with PGF and TikZ. It defines two new commands: \foreach and \breakforeach.

```
\langle variables \rangle \ [\langle options \rangle] \ in \ \langle list \rangle \ \langle commands \rangle
```

The syntax of this command is a bit complicated, so let us go through it step-by-step.

In the easiest case, $\langle variables \rangle$ is a single T_EX-command like \x or \point. (If you want to have some fun, you can also use active characters. If you do not know what active characters are, you are blessed.) Still in the easiest case, $\langle options \rangle$ will be omitted. The keys for customizing this command will be discussed below.

Again, in the easiest case, $\langle list \rangle$ is either a comma-separated list of values surrounded by curly braces or it is the name of a macro that contain such a list of values. Anything can be used as a value, but numbers are most likely.

Finally, in the easiest case, (commands) is some T_FX-text in curly braces.

With all these assumptions, the \foreach statement will execute the $\langle commands \rangle$ repeatedly, once for every element of the $\langle list \rangle$. Each time the $\langle commands \rangle$ are executed, the $\langle variable \rangle$ will be set to the current value of the list item.

Note that in each execution of $\langle commands \rangle$ the $\langle commands \rangle$ are put in a TEX group. This means that local changes to counters inside $\langle commands \rangle$ do not persist till the next iteration. For instance, if you add 1 to a counter inside $\langle commands \rangle$ locally, then in the next iteration the counter will have the same value it had at the beginning of the first iteration. You have to add \global if you wish changes to persist from iteration to iteration.

Syntax for the commands. Let us move on to a more complicated setting. The first complication occurs when the $\langle commands \rangle$ are not some text in curly braces. If the \foreach statement does not encounter an opening brace, it will instead scan everything up to the next semicolon and use this as $\langle commands \rangle$. This is most useful in situations like the following:

```
\tikz
\foreach \x in \{0,1,2,3\}
\draw (\x,0) circle (0.2cm);
```

However, the "reading till the next semicolon" is not the whole truth. There is another rule: If a \foreach statement is directly followed by another \foreach statement, this second foreach statement is collected as \foreach . This allows you to write the following:

The dots notation. The second complication concerns the $\langle list \rangle$. If this $\langle list \rangle$ contains the list item "...", this list item is replaced by the "missing values." More precisely, the following happens:

Normally, when a list item . . . is encountered, there should already have been two list items before it, which where numbers. Examples of numbers are 1, -10, or -0.24. Let us call these numbers x and y and let d := y - x be their difference. Next, there should also be one number following the three dots, let us call this number z.

In this situation, the part of the list reading " x,y,\ldots,z " is replaced by " $x,x+d,x+2d,x+3d,\ldots,x+md$," where the last dots are semantic dots, not syntactic dots. The value m is the largest number such that $x+md \leq z$ if d is positive or such that $x+md \geq z$ if d is negative.

Perhaps it is best to explain this by some examples: The following $\langle list \rangle$ have the same effects:

```
\foreach \x in \{1,2,...,6\} {\x, } yields 1, 2, 3, 4, 5, 6,
```

\foreach \x in $\{1,2,3,...,6\}$ {\x, } yields 1, 2, 3, 4, 5, 6,

\foreach \x in $\{1,3,...,11\}$ {\x, } yields 1, 3, 5, 7, 9, 11,

\foreach \x in $\{1,3,...,10\}$ {\x, } yields 1, 3, 5, 7, 9,

\foreach \x in $\{0,0.1,\ldots,0.5\}$ $\{\x, \}$ yields 0,0.1,0.20001,0.30002,0.40002,

\foreach \x in {a,b,9,8,...,1,2,2.125,...,2.5} {\x, } yields a, b, 9, 8, 7, 6, 5, 4, 3, 2, 1, 2, 2.125, 2.25, 2.375, 2.5,

As can be seen, for fractional steps that are not multiples of 2^{-n} for some small n, rounding errors can occur pretty easily. Thus, in the second last case, 0.5 should probably be replaced by 0.501 for robustness.

There is another special case for the ... statement: If the ... is used right after the first item in the list, that is, if there is an x, but no y, the difference d obviously cannot be computed and is set to 1 if the number z following the dots is larger than x and is set to -1 if z is smaller:

```
\foreach \x in \{1,...,6\} {\x, } yields 1, 2, 3, 4, 5, 6,
```

\foreach \x in $\{9,...,3.5\}$ {\x, } yields 9, 8, 7, 6, 5, 4,

There is a yet another special case for the ... statement, in that it can indicate an alphabetic character sequence:

```
\foreach \x in \{Z,X,\ldots,M\} \{\x,\ \} yields Z,X,V,T,R,P,N,
```

A final special case for the ... statement is contextual replacement. If the ... is used in some context, for example, sin(...), this context will be interpreted correctly, provided that the list items prior to the ... statement have *exactly* the same pattern, except that, instead of dots, they have a number or a character:

```
\foreach \x in \{A_1, ..., H_1\} {\$\x\$, } yields A_1, B_1, C_1, D_1, E_1, F_1, G_1, H_1,
```

Special handling of pairs. Different list items are separated by commas. However, this causes a problem when the list items contain commas themselves as pairs like (0,1) do. In this case, you should put the items containing commas in braces as in {(0,1)}. However, since pairs are such a natural and useful case, they get a special treatment by the \foreach statement. When a list item starts with a (everything up to the next) is made part of the item. Thus, we can write things like the following:

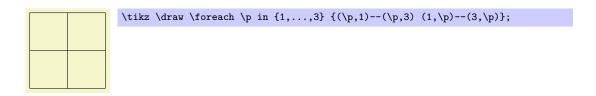
```
\tikz
\foreach \position in {(0,0), (1,1), (2,0), (3,1)}
\draw \position rectangle +(.25,.5);
```

Using the foreach-statement inside paths. TikZ allows you to use a \foreach statement inside a path construction. In such a case, the $\langle commands \rangle$ must be path construction commands. Here are two examples:

```
\tikz
\draw (0,0)
\foreach \x in {1,...,3}

{ -- (\x,1) -- (\x,0) }

;
```



Multiple variables. You will often wish to iterate over two variables at the same time. Since you can nest \foreach loops, this is normally straight-forward. However, you sometimes wish variables to iterate "simultaneously." For example, we might be given a list of edges that connect two coordinates and might wish to iterate over these edges. While doing so, we would like the source and target of the edges to be set to two different variables.

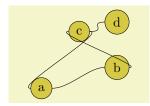
To achieve this, you can use the following syntax: The $\langle variables \rangle$ may not only be a single T_EX-variable. Instead, it can also be a list of variables separated by slashes (/). In this case the list items can also be lists of values separated by slashes.

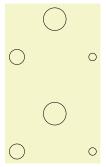
Assuming that the $\langle variables \rangle$ and the list items are lists of values, each time the $\langle commands \rangle$ are executed, each of the variables in $\langle variables \rangle$ is set to one part of the list making up the current list item. Here is an example to clarify this:

Example: $\$ in {1/2,a/b} {''\x\ and \y''} yields "1 and 2" "a and b".

If some entry in the $\langle list \rangle$ does not have "enough" slashes, the last entry will be repeated. Here is an example:

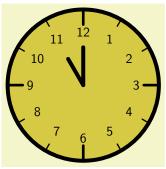
Here are more useful examples:





```
\begin{tikzpicture}
  % Let's draw circles at interesting points:
  \foreach \x / \y / \diameter in {0 / 0 / 2mm, 1 / 1 / 3mm, 2 / 0 / 1mm}
  \draw (\x,\y) circle (\diameter);

  % Same effect
  \foreach \center/\diameter in {{(0,0)/2mm}, {(1,1)/3mm}, {(2,0)/1mm}}
  \draw[yshift=2.5cm] \center circle (\diameter);
\end{tikzpicture}
```





```
\tikz[shading=ball]
\foreach \x / \cola in {0/red,1/green,2/blue,3/yellow}
\foreach \y / \colb in {0/red,1/green,2/blue,3/yellow}
\shade[ball color=\cola!50!\colb] (\x,\y) circle (0.4cm);
```

Options to customize the foreach-statement.

The keys described below can be used in the $\langle options \rangle$ argument to the \foreach command. They all have the path /pgf/foreach/, however, the path is set automatically when $\langle options \rangle$ are parsed, so it does not have to be explicitly stated.

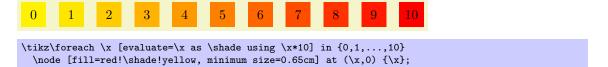
```
/pgf/foreach/var=\langle variable \rangle (no default)
```

This key provides an alternative way to specify variables: \foreach [var=\x,var=\y] is the same as \foreach \x/\y. If used, this key should be used before the other keys.

```
/pgf/foreach/evaluate = \langle variable \rangle as \langle macro \rangle using \langle formula \rangle (no default)
```

By default, list items are not evaluated: 1+2, yields 1+2, not 3. This key allows a variable to be evaluated using the mathematical engine. The variable must have been specified either using the var key or in the $\langle variables \rangle$ argument of the foreach command. By default, the result of the evaluation will be stored in $\langle variable \rangle$. However, the optional as $\langle macro \rangle$ statement can be used to store the result in $\langle macro \rangle$.

The optional using $\langle formula \rangle$ statement means an evaluation does not have to be explicitly stated for each item in $\langle list \rangle$. The $\langle formula \rangle$ should contain at least one reference to $\langle variable \rangle$.



```
/pgf/foreach/remember = \langle variable \rangle as \langle macro \rangle (initially \langle value \rangle) (no default)
```

This key allows the item value stored in $\langle variable \rangle$ to be remembered during the next iteration, stored in $\langle macro \rangle$. If a variable is evaluated, the result of this evaluation is remembered. By default the value of $\langle variable \rangle$ is zero for the first iteration, however, the optional (initially $\langle value \rangle$) statement, allows the $\langle macro \rangle$ to be initially defined as $\langle value \rangle$.

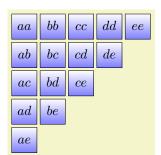
```
\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CD}, \overrightarrow{DE}, \overrightarrow{EF}, \overrightarrow{FG}, \overrightarrow{GH},
```

```
\foreach \x [remember=\x as \lastx (initially A)] in {B,...,H}{$\overrightarrow{\lastx\x}$, }
```

```
/pgf/foreach/count = \langle macro \rangle from \langle value \rangle
```

(no default)

This key allows $\langle macro \rangle$ to hold the position in the list of the current item. The optional from $\langle value \rangle$ statement allows the counting to begin from $\langle value \rangle$.



```
\tikz[x=0.75cm,y=0.75cm]
\foreach \x [count=\xi] in {a,...,e}
\foreach \y [count=\yi] in {\x,...,e}
\node [draw, top color=white, bottom color=blue!50, minimum size=0.666cm]
at (\xi,-\yi) {$\mathstrut\x\y$};
```

\breakforeach

If this command is given inside a \foreach command, no further executions of the $\langle commands \rangle$ will occur. However, the current execution of the $\langle commands \rangle$ is continued normally, so it is probably best to use this command only at the end of a \foreach command.

65 Date and Calendar Utility Macros

This section describes the package pgfcalendar.

```
\usepackage{pgfcalendar} % MEX
\input pgfcalendar.tex % plain TEX
\usemodule[pgfcalendar] % ConTEXt
```

This package can be used independently of PGF. It has two purposes:

- 1. It provides functions for working with dates. Most noticeably, it can convert a date in ISO-standard format (like 1975-12-26) to a so-called Julian day number, which is defined in Wikipedia as follows: "The Julian day or Julian day number is the (integer) number of days that have elapsed since the initial epoch at noon Universal Time (UT) Monday, January 1, 4713 BC in the proleptic Julian calendar." The package also provides a function for converting a Julian day number to an ISO-format date.
 - Julian day numbers make it very easy to work with days. For example, the date ten days in the future of 2008-02-20 can be computed by converting this date to a Julian day number, adding 10, and then converting it back. Also, the day of week of a given date can be computed by taking the Julian day number modulo 7.
- 2. It provides a macro for typesetting a calendar. This macro is highly configurable and flexible (for example, it can produce both plain text calendars and also complicated TikZ-based calendars), but most users will not use the macro directly. It is the job of a frontend to provide useful configurations for typesetting calendars based on this command.

65.1 Handling Dates

65.1.1 Conversions Between Date Types

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This macro converts a date in a format to be described in a moment to the Julian day number in the Gregorian calendar. The $\langle date \rangle$ should expand to a string of the following form:

- 1. It should start with a number representing the year. Use \year for the current year, that is, the year the file is being typeset.
- 2. The year must be followed by a hyphen.
- 3. Next should come a number representing the month. Use \month for the current month. You can, but need not, use leading zeros. For example, 02 represents February, just like 2.
- 4. The month must also be followed by a hyphen.
- 5. Next you must either provide a day of month (again, a number and, again, \day yields the current day of month) or the keyword last. This keyword refers to the last day of the month, which is automatically computed (and which is a bit tricky to compute, especially for February).
- 6. Optionally, you can next provide a plus sign followed by positive or negative number. This number of days will be added to the computed date.

Here are some examples:

- 2006-01-01 refers to the first day of 2006.
- 2006-02-last refers to February 28, 2006.
- \year-\month-\day refers to today.
- \bullet 2006-01-01+2 refers to January 3, 2006.
- \year-\month-\day+1 refers to tomorrow.
- \year-\month-\day+-1 refers to yesterday.

The conversion method is taken from the English Wikipedia entry on Julian days.

Example: \pgfcalendardatetojulian{2007-01-14}{\mycount} sets \mycount to 2454115.

This command converts a Julian day number to an ISO-date. The $\langle Julian\ day \rangle$ must be a number or TEX counter, the $\langle year\ macro \rangle$, $\langle month\ macro \rangle$ and $\langle day\ macro \rangle$ must be TEX macro names. They will be set to numbers representing the year, month, and day of the given Julian day in the Gregorian calendar.

The $\langle year\ macro \rangle$ will be assigned the year without leading zeros. Note that this macro will produce year 0 (as opposed to other calendars, where year 0 does not exist). However, if you really need calendars for before the year 1, it is expected that you know what you are doing anyway.

The $\langle month\ macro \rangle$ gets assigned a two-digit number representing the month (with a leading zero, if necessary). Thus, the macro is set to 01 for January.

The $\langle day \ macro \rangle$ gets assigned a two-digit number representing the day of the month (again, possibly with a leading zero).

To convert a Julian day number to an ISO-date you use code like the following:

 $\propto learned and a tetojulian {2454115} {\myyear} {$

The above code sets \isodate to 2007-01-14.

$\protect\pro$

This command converts a Julian day to a week day by computing the day modulo 7. The $\langle week \ day \ counter \rangle$ must be a T_EX counter. It will be set to 0 for a Monday, to 1 for a Tuesday, and so on.

Example: \pgfcalendarjuliantoweekday{2454115}{\mycount} sets \mycount to 6 (it was a Sunday).

65.1.2 Checking Dates

$\protect{pgfcalendarifdate} \langle date \rangle \} \{\langle tests \rangle \} \{\langle code \rangle \} \{\langle else\ code \rangle \}$

This command is used to execute code based on properties of $\langle date \rangle$. The $\langle date \rangle$ must be a date in ISO-format. For this date, the $\langle tests \rangle$ are checked (to be detailed later) and if one of the tests succeeds, the $\langle code \rangle$ is executed. If none of the tests succeeds, the $\langle else\ code \rangle$ is executed.

The $\langle tests \rangle$ is a comma-separated list of key-value pairs. The following are defined by default:

- all This test is passed by all dates.
- Monday This test is passed by all dates that are Mondays.
- Tuesday as above.
- Wednesday as above.
- Thursday as above.
- Friday as above.
- Saturday as above.
- Sunday as above.
- workday Passed by Mondays, Tuesdays, Wednesdays, Thursdays, and Fridays.
- weekend Passed by Saturdays and Sundays.
- equals= $\langle reference \rangle$ The $\langle reference \rangle$ can be in one of two forms: Either, it is a full ISO format date like 2007-01-01 or the year may be missing as in 12-31. In the first case, the test is passed if $\langle date \rangle$ is the same as $\langle reference \rangle$. In the second case, the test is passed if the month and day part of $\langle date \rangle$ is the same as $\langle reference \rangle$.

For example, the test equals=2007-01-10 will only be passed by this particular date. The test equals=05-01 will be passed by every first of May on any year.

• at least=\langle reference \rangle. This test works similarly to the equals test, only it is checked whether \langle date \rangle is equal to \langle reference \rangle or to any later date. Again, the \langle reference \rangle can be a full date like 2007-01-01 or a short version like 07-01. For example, at least=07-01 is true for every day in the second half of any year.

- at most= $\langle reference \rangle$ as above.
- between=\(\start \text{ reference}\) and \(\langle end \text{ reference}\) This test checks whether the current date lies between the two given reference dates. Both full and short version may be given.

For example between=2007-01-01 and 2007-02-28 is true for the days in January and February of 2007.

For another example, between=05-01 and 05-07 is true for the days of the first week of May of any year.

- day of month= $\langle number \rangle$ Passed by the day of month of the $\langle date \rangle$ that is $\langle number \rangle$. For example, the test day of month=1 is passed by every first of every month.
- end of month= $\langle number \rangle$ Passed by the day of month of the $\langle date \rangle$ that is $\langle number \rangle$ from the end of the month. For example, the test end of month=1 is passed by the last day of every month, the test end of month=2 is passed by the second last day of every month. If $\langle number \rangle$ is omitted, it is assumed to be 1.

In addition to the above checks, you can also define new checks. To do so, you must add a new key to the key-value group pgfcalendar using \define@key. The job of the code of this new key is to possibly set the T_EX -if \ifpgfcalendarmatches to true (if it is already true, no action should be taken) to indicate that the $\langle date \rangle$ passes the test setup by this new key.

In order to perform the test, the key code needs to know the date that should be checked. The date is available through a macro, but a whole bunch of additional information about this date is also available through the following macros:

- \pgfcalendarifdatejulian is the Julian day number of the $\langle date \rangle$ to be checked.
- \pgfcalendarifdateweekday is the weekday of the $\langle date \rangle$ to be checked.
- \pgfcalendarifdateyear is the year of the $\langle date \rangle$ to be checked.
- \pgfcalendarifdatemonth is the month of the $\langle date \rangle$ to be checked.
- \pgfcalendarifdateday is the day of month of the $\langle date \rangle$ to be checked.

For example, let us define a new key that checks whether the $\langle date \rangle$ is a Workers day (May 1st). This can be done as follows:

```
\define@key{pgfcalendar}{workers day}[]
{
  \ifnum\pgfcalendarifdatemonth=5\relax
   \ifnum\pgfcalendarifdateday=1\relax
   \pgfcalendarmatchestrue
  \fi
  \fi
}
```

65.1.3 Typesetting Dates

$\protect\operatorname{\mathtt{f pgfcalendarweekdayname}} \$

This command expands to a textual representation of the day of week, given by the $\langle week \ day \ number \rangle$. Thus, $\protect\ protect\ pr$

Example: \pgfcalendarweekdayname{2} yields Wednesday.

```
\protect\operatorname{\mathtt{Npgfcalendarweekdayshortname}} \
```

This command works similarly to the previous command, only an abbreviated version of the week day is produced.

Example: \pgfcalendarweekdayshortname{2} yields Wed.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

This command expands to a textual representation of the month, which is given by the $\langle month \ number \rangle$.

Example: \pgfcalendarmonthname{12} yields December.

$\protect\operatorname{\mathtt{Nonth}}\ number\$

As above, only an abbreviated version is produced.

Example: \pgfcalendarmonthshortname{12} yields Dec.

65.1.4 Localization

All textual representations of week days or months (like "Monday" or "February") are wrapped with \translate commands from the translator package (it this package is not loaded, no translation takes place). Furthermore, the pgfcalendar package will try to load the translator-months-dictionary, if the translator package is loaded.

The net effect of all this is that all dates will be translated to the current language setup in the translator package. See the documentation of this package for more details.

65.2 Typesetting Calendars

```
\proonup \
```

This command can be used to typeset a calendar. It is a very general command, the actual work has to be done by giving clever implementations of $\langle rendering\ code \rangle$. Note that this macro need *not* be called inside a {pgfpicture} environment (even though it typically will be) and you can use it to typeset calendars in normal TeX or using packages other than PGF.

Basic typesetting process. A calendar is typeset as follows: The $\langle start \ date \rangle$ and $\langle end \ date \rangle$ specify a range of dates. For each date in this range the $\langle rendering \ code \rangle$ is executed with certain macros setup to yield information about the *current date* (the current date in the enumeration of dates of the range). Typically, the $\langle rendering \ code \rangle$ places nodes inside a picture, but it can do other things as well. Note that it is also the job of the $\langle rendering \ code \rangle$ to position the calendar correctly.

The different calls of the $\langle rending\ code \rangle$ are not surrounded by TEX groups (though you can do so yourself, of course). This means that settings can accumulate between different calls, which is often desirable and useful.

Information about the current date. Inside the $\langle rendering \ code \rangle$, different macros can be access:

- \pgfcalendarprefix The \langle prefix \rangle parameter. This prefix is recommended for nodes inside the calendar, but you have to use it yourself explicitly.
- \pgfcalendarbeginiso The \(start date \) of range being typeset in ISO format (like 2006-01-10).
- \pgfcalendarbeginjulian Julian day number of $\langle start \ date \rangle$.
- \pgfcalendarendiso The $\langle end \ date \rangle$ of range being typeset in ISO format.
- \pgfcalendarendjulian Julian day number of $\langle end \ date \rangle$.
- \pgfcalendarcurrentjulian This TEX count holds the Julian day number of the day currently being rendered.
- \pgfcalendarcurrentweekday The weekday (a number with zero representing Monday) of the current date.
- \pgfcalendarcurrentyear The year of the current date.
- \pgfcalendarcurrentmonth The month of the current date (always two digits with a leading zero, if necessary).
- \pgfcalendarcurrentday The day of month of the current date (always two digits).

The \ifdate command. Inside the \pgfcalendar the macro \ifdate is available locally:

This command has the same effect as calling \pgfcalendarifdate for the current date.

Examples. In a first example, let us create a very simple calendar: It just lists the dates in a certain range.

20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10

$\pgfcalendar{cal}{2007-01-20}{2007-02-10}{\pgfcalendarcurrentday} \ \}$

Let us now make this a little more interesting: Let us add a line break after each Sunday.

```
20
    21
22
    23
         24
             25
                  26
                       27
                           28
29
    30
         31
             01
                  02
                      03
                           04
05
    06 07 08
                 09
                      10
\pgfcalendar{cal}{2007-01-20}{2007-02-10}
  \pgfcalendarcurrentday\
  \ifdate{Sunday}{\par}{}
```

We now want to have all Mondays to be aligned on a column. For this, different approaches work. Here is one based positioning each day horizontally using a skip.

```
20
                                                                                                                                                                                                                                                                                                                                                                                                                             21
22
                                                                     23
                                                                                                                                                                                                                                                                                    26
                                                                                                                                         24
                                                                                                                                                                                                              25
                                                                                                                                                                                                                                                                                                                                                         27
                                                                                                                                                                                                                                                                                                                                                                                                                             28
29
                                                                                                                                         31
                                                                     30
                                                                                                                                                                                                            01
                                                                                                                                                                                                                                                                                  02
                                                                                                                                                                                                                                                                                                                                                         03
                                                                                                                                                                                                                                                                                                                                                                                                                             04
                                                                     06
                                                                                                                                         07
                                                                                                                                                                                                            08
                                                                                                                                                                                                                                                                                    09
                                                                                                                                                                                                                                                                                                                                                         10
 \pgfcalendar{cal}{2007-01-20}{2007-02-10}
                    \leavevmode%
                     \verb|\hbox to0pt{\hskip\pgfcalendarcurrentweekday cm\pgfcalendarcurrentday\hss}||% \cite{Allowedge}| = \cit
                     \left( Sunday \right) {\par}{}%
```

Let us now typeset two complete months.

```
January
 1
       2
              3
                     4
                            5
                                  6
                                         7
 8
       9
             10
                    11
                           12
                                 13
                                        14
15
      16
             17
                    18
                           19
                                 20
                                        21
22
      23
             24
                    25
                           26
                                        28
29
      30
             31
                February
                     1
                            2
                                  3
                                         4
       6
              7
                     8
                            9
 5
                                 10
                                        11
12
      13
             14
                    15
                           16
                                 17
                                        18
19
      20
             21
                    22
                           23
                                 24
                                        25
26
      27
             28
```

```
\pgfcalendar{cal}{2007-01-01}{2007-02-28}{%
  \ifdate{day of month=1}{
    \par\bigskip\hbox to7.5cm{\itshape\hss\pgfcalendarshorthand mt\hss}\par
}{}%
  \leavevmode%
{%
    \ifdate{weekend}{\color{black!50}}{\color{black}}%
    \hbox to0pt{%
    \hskip\pgfcalendarcurrentweekday cm%
    \hbox to1cm{\hss\pgfcalendarshorthand d-}\hss%
    }%
}%
\ifdate{Sunday}{\par}{}%
```

For our final example, we use a {tikzpicture}.

```
20
                                          21
                                   27
                                          28
22
       23
              24
                    25
                            26
29
      30
              31
                    01
                           02
                                   03
                                          04
05
      06
              07
                    08
                           09
                                   10
```

```
\begin{tikzpicture}
\pgfcalendar{cal}{2007-01-20}{2007-02-10}{%
  \ifdate{workday}
    {\tikzset{filling/.style={fill=blue!20}}}
    {\tikzset{filling/.style={fill=red!20}}}
    \node (\pgfcalendarsuggestedname) at (\pgfcalendarcurrentweekday,0)
        [anchor=base,circle,filling] {\pgfcalendarcurrentday};
    \ifdate{Sunday}{\pgftransformyshift{-3em}}{}%
    }
    \draw (cal-2007-01-21) -- (cal-2007-02-03);
\end{tikzpicture}
```

$\protect\operatorname{\mathtt{f pgfcalendarshorthand}} \{\langle kind \rangle\} \{\langle representation \rangle\}$

This command can be used inside a \pgfcalendar , where it will expand to a representation of the current day, month, year or day of week, depending on whether $\langle kind \rangle$ is d, m, y or w. The $\langle representation \rangle$ can be one of the following: \neg , =, 0, ., and t. They have the following meanings:

- The minus sign selects the shortest numerical representation possible (no leading zeros).
- The equal sign also selects the shortest numerical representation, but a space is added to single digit days and months (thereby ensuring that they have the same length as other days).
- The zero digit selects a two-digit numerical representation for days and months. For years it is allowed, but has no effect.
- The letter t selects a textual representation.
- The dot selects an abbreviated textual representation.

Normally, you should say \let\%=\pgfcalendarshorthand locally, so that you can write \%wt instead of the much more cumbersome \pgfcalendarshorthand{w}{t}.

```
ISO form: 2007-01-20, long form: Saturday, January 20, 2007
```

```
\let\%=\pgfcalendarshorthand
\pgfcalendar{cal}{2007-01-20}{2007-01-20}
{ ISO form: \%y0-\%m0-\%d0, long form: \%wt, \%mt \%d-, \%y0}
```

\pgfcalendarsuggestedname

This macro expands to a suggested name for nodes representing days in a calendar. If the $\langle prefix \rangle$ is empty, it expands to the empty string, otherwise it expands to the $\langle prefix \rangle$ of the calendar, followed by a hyphen, followed by the ISO format version of the date. Thus, when the date 2007-01-01 is typeset in a calendar for the prefix mycal, the macro expands to mycal-2007-01-01.

66 Page Management

This section describes the pgfpages package. Although this package is not concerned with creating pictures, its implementation relies so heavily on PGF that it is documented here. Currently, pgfpages only works with LATEX, but if you are adventurous, feel free to hack the code so that it also works with plain TEX.

The aim of pgfpages is to provide a flexible way of putting multiple pages on a single page *inside T_EX*. Thus, pgfpages is quite different from useful tools like psnup or pdfnup insofar as it creates its output in a single pass. Furthermore, it works uniformly with both latex and pdflatex, making it easy to put multiple pages on a single page without any fuss.

A word of warning: using pgfpages will destroy hyperlinks. Actually, the hyperlinks are not destroyed, only they will appear at totally wrong positions on the final output. This is due to a fundamental flaw in the PDF specification: In PDF the bounding rectangle of a hyperlink is given in "absolute page coordinates" and translations or rotations do not affect them. Thus, the transformations applied by pgfpages to put the pages where you want them are (cannot, even) be applied to the coordinates of hyperlinks. It is unlikely that this will change in the foreseeable future.

66.1 Basic Usage

The internals of pgfpages are complex since the package can do all sorts of interesting tricks. For this reason, so-called *layouts* are predefined that set up all option in appropriate ways.

You use a layout as follows:

```
\documentclass{article}
\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]
\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The layout 2 on 1 puts two pages on a single page. The option a4paper tells pgfpages that the resulting page (called the *physical* page in the following) should be a4paper and it should be landscape (which is quite logical since putting two portrait pages next to each other gives a landscape page). Normally, the *logical* pages, that is, the pages that TEX "thinks" that it is typesetting, will have the same sizes, but this need not be the case. pgfpages will automatically scale down the logical pages such that two logical pages fit next to each other inside a DIN A4 page.

The border shrink tells pgfpages that it should add an additional 5mm to the shrinking such that a 5mm-wide border is shown around the resulting logical pages.

As a second example, let us put two pages produced by the BEAMER class on a single page:

```
\documentclass{beamer}
\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,border shrink=5mm]
\begin{document}
\begin{frame}
   This text is shown at the top.
\end{frame}
\begin{frame}
   This text is shown at the bottom.
\end{frame}
\begin{frame}
   This text is shown at the bottom.
\end{frame}
\end{document}
```

Note that we do not use the landscape option since BEAMER's logical pages are already in landscape mode and putting two landscape pages on top of each other results in a portrait page. However, if you had used the 4 on 1 layout, you would have had to add landscape once more, using the 8 on 1 you must not, using 16 on 1 you need it yet again. And, no, there is no 32 on 1 layout.

Another word of caution: using pgfpages will produce wrong page numbers in the .aux file. The reason is that TEX instantiates the page numbers when writing an .aux file only when the physical page is shipped out. Fortunately, this problem is easy to fix: First, typeset our file normally without using the \pgfpagesuselayout command (just put the comment marker % before it) Then, rerun TEX with the

\pgfpagesuselayout command included and add the command \nofiles. This command ensures that the .aux file is not modified, which is exactly what you want. So, to typeset the above example, you should actually first TFX the following file:

```
\documentclass{article}
\usepackage{pgfpages}
%%\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]
%%\nofiles
\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

and then typeset

```
\documentclass{article}
\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]
\nofiles
\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The final basic example is the resize to layout (it works a bit like a hypothetical 1 on 1 layout). This layout resizes the logical page such that is fits the specified physical size. Since this does not change the page numbering, you need not worry about the .aux files with this layout. For example, adding the following lines will ensure that the physical output will fit on DIN A4 paper:

```
\usepackage{pgfpages}
\pgfpagesuselayout{resize to}[a4paper]
```

This can be very useful when you have to handle lots of papers that are typeset for, say, letter paper and you have an A4 printer or the other way round. For example, the following article will be fit for printing on letter paper:

```
\documentclass[a4paper]{article}
%% a4 is currently the logical size and also the physical size

\usepackage{pgfpages}
\pgfpagesuselayout{resize to}[letterpaper]
%% a4 is still the logical size, but letter is the physical one

\begin{document}
   \title{My Great Article}
...
\end{document}
```

66.2 The Predefined Layouts

This section explains the predefined layouts in more detail. You select a layout using the following command:

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

Installs the specified $\langle layout \rangle$ with the given $\langle options \rangle$. The predefined layouts and their permissible options are explained below.

If this function is called multiple times, only the last call "wins." You can thereby overwrite any previous settings. In particular, layouts do not accumulate.

Example: \pgfpagesuselayout{resize to}[a4paper]

```
\protect\operatorname{\mathtt{pgfpagesuselayout\{resize\ to\}}}[\langle \mathit{options} \rangle]
```

This layout is used to resize every logical page to a specified physical size. To determine the target size, the following options may be given:

- physical paper height= $\langle size \rangle$ sets the height of the physical page size to $\langle size \rangle$.
- physical paper width= $\langle size \rangle$ sets the width of the physical Pappe size to $\langle size \rangle$.
- a0paper sets the physical page size to DIN A0 paper.
- alpaper sets the physical page size to DIN A1 paper.
- a2paper sets the physical page size to DIN A2 paper.
- a3paper sets the physical page size to DIN A3 paper.
- a4paper sets the physical page size to DIN A4 paper.
- a5paper sets the physical page size to DIN A5 paper.
- a6paper sets the physical page size to DIN A6 paper.
- letterpaper sets the physical page size to the American letter paper size.
- legalpaper sets the physical page size to the American legal paper size.
- executivepaper sets the physical page size to the American executive paper size.
- landscape swaps the height and the width of the physical paper.
- border shrink= $\langle size \rangle$ additionally reduces the size of the logical page on the physical page by $\langle size \rangle$.

$\protect\operatorname{\mathtt{Npgfpagesuselayout\{2\ on\ 1\}}}[\langle \mathit{options}\rangle]$

Puts two logical pages alongside each other on each physical page if the logical height is larger than the logical width (logical pages are in portrait mode). Otherwise, two logical pages are put on top of each other (logical pages are in landscape mode). When using this layout, it is advisable to use the \nofiles command, but this is not done automatically.

The same (options) as for the resize to layout can be used, plus the following option:

• odd numbered pages right places the first page on the right.

$\protect\pro$

Puts four logical pages on a single physical page. The same $\langle options \rangle$ as for the resize to layout can be used.

\pgfpagesuselayout{8 on 1} [$\langle options \rangle$]

Puts eight logical pages on a single physical page. As for 2 on 1 and 4 on 1, the orientation depends on whether the logical pages are in landscape mode or in portrait mode.

\pgfpagesuselayout{16 on 1}[$\langle options \rangle$]

This is for the CEO.

$\protect\operatorname{\mathtt{pgfpagesuselayout\{rounded\ corners\}[\langle options\rangle]}}$

This layout adds "rounded corners" to every page, which, supposedly, looks nicer during presentations with projectors (personally, I doubt this). This is done by (possibly) resizing the page to the physical page size. Then four black rectangles are drawn in each corner. Next, a clipping region is set up that contains all of the logical page except for little rounded corners. Finally, the logical page is drawn and clipped against the clipping region.

Note that every logical page should fill its background for this to work.

In addition to the *(options)* that can be given to resize to, the following options may be given.

• corner width= $\langle size \rangle$ specifies the size of the corner.

```
\documentclass{beamer}
\usepackage{pgfpages}
\pgfpagesuselayout{rounded corners}[corner width=5pt]
\begin{document}
...
\end{document}
```

```
\pgfpagesuselayout{two screens with lagging second} [\langle options \rangle]
```

This layout puts two logical pages alongside each other. The second page always shows what the main page showed on the previous physical page. Thus, the second page "lags behind" the main page. This can be useful when you have two projectors attached to your computer and can show different parts of a physical page on different projectors.

The following $\langle options \rangle$ may be given:

- second right puts the second page right of the main page. This will make the physical pages twice as wide as the logical pages, but it will retain the height.
- second left puts the second page left, otherwise it behaves the same as second right.
- second bottom puts the second page below the main page. This make the physical pages twice as high as the logical ones.
- second top works like second bottom.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This layout works similarly to two screens with lagging second. The difference is that the contents of the second screen only changes when one of the commands $\position page{2}{\langle box\rangle}$ or $\position page{2}$ is called. The first puts the given $\position beta become page$. The second specifies that the current page should be put there, once it is finished.

The same options as for two screens with lagging second may be given.

You can define your own predefined layouts using the following command:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command predefines a $\langle layout \rangle$ that can later be installed using the \pgfpagesuselayout command.

When $\protect\protec$

Here is an example:

```
\pgfpagesdeclarelayout{resize to}
{
  \def\pgfpageoptionborder{Opt}}
}
{
  \pgfpagesphysicalpageoptions
{%
    logical pages=1,%
    physical height=\pgfpageoptionheight,%
    physical width=\pgfpageoptionwidth%
}
\pgfpageslogicalpageoptions{1}
{%
    resized width=\pgfphysicalwidth,%
    resized height=\pgfphysicalheight,%
    border shrink=\pgfpageoptionborder,%
    center=\pgfpoint{.5\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
}
```

66.3 Defining a Layout

If none of the predefined layouts meets your problem or if you wish to modify them, you can create layouts from scratch. This section explains how this is done.

Basically, pgfpages hooks into TEX's \shipout function. This function is called whenever TEX has completed typesetting a page and wishes to send this page to the .dvi or .pdf file. The pgfpages package redefines this command. Instead of sending the page to the output file, pgfpages stores it in an internal box and then acts as if the page had been output. When TEX tries to output the next page using \shipout, this call is once more intercepted and the page is stored in another box. These boxes are called *logical pages*.

At some point, enough logical pages have been accumulated such that a *physical page* can be output. When this happens, pgfpages possibly scales, rotates, and translates the logical pages (and possibly even does further modifications) and then puts them at certain positions of the *physical* page. Once this page is fully assembled, the "real" or "original" \shipout is called to send the physical page to the output file.

In reality, things are slightly more complicated. First, once a physical page has been shipped out, the logical pages are usually voided, but this need not be the case. Instead, it is possible that certain logical pages just retain their contents after the physical page has been shipped out and these pages need not be filled once more before a physical shipout can occur. However, the contents of these logical pages can still be changed using special commands. It is also possible that after a shipout certain logical pages are filled with the contents of *other* logical pages.

A *layout* defines for each logical page where it will go on the physical page and which further modifications should be done. The following two commands are used to define the layout:

$\verb|\pgfpagesphysicalpageoptions{|} \langle options \rangle|$

This command sets the characteristics of the "physical" page. For example, it is used to specify how many logical pages there are and how many logical pages must be accumulated before a physical page is shipped out. How each individual logical page is typeset is specified using the command <code>\pgfpageslogicalpageoptions</code>, described later.

Example: A layout for putting two portrait pages on a single landscape page:

```
\pgfpagesphysicalpageoptions
{%
  logical pages=2,%
  physical height=\paperwidth,%
  physical width=\paperheight,%
}

\pgfpageslogicalpageoptions{1}
{%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%

\pgfpageslogicalpageoptions{2}
{%
  resized width=.5\pgfphysicalwidth,%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.75\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
```

The following $\langle options \rangle$ may be set:

- logical pages=\langle logical pages \rangle specified how many logical pages there are, in total. These are numbered 1 to \langle logical pages \rangle.
- first logical shipout= $\langle first \rangle$. See the next option. By default, $\langle first \rangle$ is 1.
- last logical shipout= $\langle last \rangle$. Together with the previous option, these two options define an interval of pages inside the range 1 to $\langle logical\ pages \rangle$. Only this range is used to store the pages that are shipped out by TeX. This means that after a physical shipout has just occurred (or at the beginning), the first time TeX wishes to perform a shipout, the page to be shipped out is stored in logical page $\langle first \rangle$. The next time TeX performs a shipout, the page is stored in logical page $\langle first \rangle + 1$ and so on, until the logical page $\langle last \rangle$ is also filled. Once this happens, a physical shipout occurs and the process starts once more.

Note that logical pages that lie outside the interval between $\langle first \rangle$ and $\langle last \rangle$ are filled only indirectly or when special commands are used.

By default, $\langle last \rangle$ equals $\langle logical\ pages \rangle$.

• current logical shipout= $\langle current \rangle$ changes an internal counter such that TEX's next logical shipout will be stored in logical page $\langle current \rangle$.

This option can be used to "warp" the logical page filling mechanism to a certain page. You can both skip logical pages and overwrite already filled logical pages. After the logical page $\langle current \rangle$ has been filled, the internal counter is incremented normally as if the logical page $\langle current \rangle$ had been "reached" normally. If you specify a $\langle current \rangle$ larger than $\langle last \rangle$, a physical shipout will occur after the logical page $\langle current \rangle$ has been filled.

- physical height=\langle height\rangle specifies the height of the physical pages. This height is typically different from the normal \paperheight, which is used by TEX for its typesetting and page breaking purposes.
- physical width=\langle width\rangle specifies the physical width.

$\protect\pro$

This command is used to specify where the logical page number $\langle logical\ page\ number \rangle$ will be placed on the physical page. In addition, this command can be used to install additional "code" to be executed when this page is put on the physical page.

The number $\langle logical\ page\ number \rangle$ should be between 1 and $\langle logical\ pages \rangle$, which has previously been installed using the $\protect\operatorname{pagesphysicalpageoptions}$ command.

The following $\langle options \rangle$ may be given:

• center=\(\rangle pgf point \rangle\) specifies the center of the logical page inside the physical page as a PGF-point. The origin of the coordinate system of the physical page is at the lower left corner.

```
\pgfpageslogicalpageoptions{1}
{% center logical page on middle of left side
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
}
```

- resized width=\langle size \rangle specifies the width that the logical page should have at most on the physical page. To achieve this width, the pages is scaled down appropriately or more. The "or more" part can happen if the resize height option is also used. In this case, the scaling is chosen such that both the specified height and width are met. The aspect ratio of a logical page is not modified.
- resized height= $\langle height \rangle$ specifies the maximum height of the logical page.
- original width=\langle width \rangle specifies the width the TeX "thinks" that the logical page has. This width is \paperwidth at the point of invocation, by default. Note that setting this width to something different from \paperwidth does not change the \pagewidth during TeX's typesetting. You have to do that yourself.

You need this option only for special logical pages that have a height or width different from the normal one and for which you will (later on) set these sizes yourself.

- original height=\langle height \rangle works like original width.
- scale= $\langle factor \rangle$ scales the page by at least the given $\langle factor \rangle$. A $\langle factor \rangle$ of 0.5 will half the size of the page, a factor or 2 will double the size. "At least" means that if options like resize height are given and if the scaling required to meet that option is less than $\langle factor \rangle$, that other scaling is used instead.
- $xscale=\langle factor \rangle$ scales the logical page along the x-axis by the given $\langle factor \rangle$. This scaling is done independently of any other scaling. Mostly, this option is useful for a factor of -1, which flips the page along the y-axis. The aspect ratio is not kept.
- yscale= $\langle factor \rangle$ works like xscale, only for the y-axis.
- rotation= $\langle degree \rangle$ rotates the page by $\langle degree \rangle$ around its center. Use a degree of 90 or -90 to go from portrait to landscape and back. The rotation need not be a multiple of 90.
- copy from=\langle logical page number \rangle. Normally, after a physical shipout has occurred, all logical pages are voided in a loop. However, if this option is given, the current logical page is filled with the contents of the old logical page number \langle logical page number \rangle.

Example: Have logical page 2 retain its contents:

```
\pgfpageslogicalpageoptions{2}{copy from=2}
```

Example: Let logical page 2 show what logical page 1 showed on the just-shipped-out physical page:

```
\pgfpageslogicalpageoptions{2}{copy from=1}
```

- border shrink=(size) specifies an additional reduction of the size to which the page is scaled.
- border code=\(\circ code\)\. When this option is given, the \(\circ code\)\ is executed before the page box is inserted with a path preinstalled that is a rectangle around the current logical page. Thus, setting \(\circ code\)\ to \pgfstroke draws a rectangle around the logical page. Setting \(\circ code\)\ to \pgfsetlinewidth{3pt}\pgfstroke results in a thick (ugly) frame. Adding dashes and filling can result in arbitrarily funky and distracting borders.

You can also call \pgfdiscardpath and add your own path construction code (for example to paint a rectangle with rounded corners). The coordinate system is set up in such a way that a rectangle starting at the origin and having the height and width of TeX-box 0 will result in a rectangle filling exactly the logical page currently being put on the physical page. The logical page is inserted after these commands have been executed.

Example: Add a rectangle around the page:

```
\pgfpageslogicalpageoptions{1}{border code=\pgfstroke}
```

• corner width=\(size\) adds black "rounded corners" to the page. See the description of the predefined layout rounded corners on page 670.

66.4 Creating Logical Pages

Logical pages are created whenever TeX thinks that a page is full and performs a \shipout command. This will cause pgfpages to store the box that was supposed to be shipped out internally until enough logical pages have been collected such that a physical shipout can occur.

Normally, whenever a logical shipout occurs, that current page is stored in logical page number $\langle current \ logical \ page \rangle$. This counter is then incremented, until it is larger than $\langle last \ logical \ shipout \rangle$. You can, however, directly change the value of $\langle current \ logical \ page \rangle$ by calling \pgfpagesphysicalpageoptions.

Another way to set the contents of a logical page is to use the following command:

```
\protect\operatorname{\begin{tabular}{l} pgfpagesshipoutlogicalpage{\langle number\rangle}\langle box\rangle. \protect\begin{tabular}{l} \protect\begin{tabular}{l}
```

This command sets to logical page $\langle number \rangle$ to $\langle box \rangle$. The $\langle box \rangle$ should be the code of a TEX box command. This command does not influence the counter $\langle current\ logical\ page \rangle$ and does not cause a physical shipout.

```
\pgfpagesshipoutlogicalpage{0}\vbox{Hi!}
```

This command can be used to set the contents of logical pages that are normally not filled.

The final way of setting a logical page is using the following command:

```
\protect\operatorname{\mathtt{Number}}
```

When the current TEX page has been typeset, it will be become the given logical page $\langle number \rangle$. This command "interrupts" the normal order of logical pages, that is, it behaves like the previous command and does not update the $\langle current\ logical\ page \rangle$ counter.

```
\pgfpagesuselayout{two screens with optional second}
...
Text for main page.
\clearpage

\pgfpagescurrentpagewillbelogicalpage{2}
Text that goes to second page
\clearpage

Text for main page.
```

67 Extended Color Support

This section documents the package xxcolor, which is currently distributed as part of PGF. This package extends the xcolor package, written by Uwe Kern, which in turn extends the color package. I hope that the commands in xxcolor will some day migrate to xcolor, such that this package becomes superfluous.

The main aim of the xxcolor package is to provide an environment inside which all colors are "washed out" or "dimmed." This is useful in numerous situations and must typically be achieved in a roundabout manner if such an environment is not available.

The mix-in specification is applied to all colors inside the environment. At the beginning of the environment, the mix-in is applied to the current color, i. e., the color that was in effect before the environment started. A mix-in specification is a number between 0 and 100 followed by an exclamation mark and a color name. When a \color command is encountered inside a mix-in environment, the number states what percentage of the desired color should be used. The rest is "filled up" with the color given in the mix-in specification. Thus, a mix-in specification like 90!blue will mix in 10% of blue into everything, whereas 25!white will make everything nearly white.

Red text, washed-out red text, washed-out blue text, dark washed-out blue text, dark washed-out green text, back to washed-out blue text and back to red.

```
\begin{minipage}{3.5cm}\raggedright
\color{red}Red text,%
\begin{colormixin}{25!white}
  washed-out red text,
  \color{blue} washed-out blue text,
  \begin{colormixin}{25!black}
    dark washed-out blue text,
  \color{green} dark washed-out green text,%
  \end{colormixin}
  back to washed-out blue text,
  \end{colormixin}
  and back to red.
  \end{minipage}%
```

Note that the environment only changes colors that have been installed using the standard LATEX \color command. In particular, the colors in images are not changed. There is, however, some support offered by the commands \pgfuseimage and \pgfuseshading. If the first command is invoked inside a colormixin environment with the parameter, say, 50!black on an image with the name foo, the command will first check whether there is also a defined image with the name foo.!50!black. If so, this image is used instead. This allows you to provide a different image for this case. If you nest colormixin environments, the different mixins are all appended. For example, inside the inner environment of the above example, \pgfuseimage{foo} would first check whether there exists an image named foo.!50!white!25!black.

\colorcurrentmixin

Expands to the current accumulated mix-in. Each nesting of a colormixin adds a mix-in to this list.

```
!75!white should be "!75!white"
!75!black!75!white should be "!50!white!75!black!75!white"

\begin{minipage}{\linewidth-6pt}\raggedright
\begin{colormixin}{75!white}
\colorcurrentmixin\ should be ''!75!white''\par
\begin{colormixin}{75!black}
\colorcurrentmixin\ should be ''!75!black!75!white''\par
\begin{colormixin}{50!white}
\colorcurrentmixin\ should be ''!50!white!75!black!75!white''\par
\begin{colormixin}{50!colormixin}\ should be ''!50!white!75!black!75!white''\par
\end{colormixin}
\end{colormixin}
\end{colormixin}
\end{colormixin}
\end{minipage}
```

68 Parser Module

```
\usepgfmodule{parser} % MEX and plain TeX and pure pgf \usepgfmodule[parser] % ConTeXt and pure pgf
```

This module defines some commands for creating a simple letter-by-letter parser.

This module provides commands for defining a parser that scans some given text letter-by-letter. For each letter, some code is executed and, possibly a state-switch occurs. The parsing process ends when a final state has been reached.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu
```

This command is used to parse the $\langle text \rangle$ using the (previously defined) parser named $\langle parser\ name \rangle$.

The $\langle text \rangle$ is not contained in curly braces, rather it is all the text that follows. The end of the text is determined implicitly, namely when the final state of the parser has been reached.

The parser works as follows: At any moment, it is in a certain state, initially this state is called initial. Then, the first letter of the $\langle text \rangle$ is examined (using the \futurlet command). For each possible state and each possible letter, some action code is stored in the parser in a table. This code is then executed. This code may, but need not, trigger a state switch, causing a new state to be set. The parser then moves on to the next character of the text and repeats the whole procedure, unless it is in the state final, which causes the parsing process to stop immediately.

In the following example, the parser counts the number of a's in the text, ignoring any b's. The $\langle text \rangle$ ends with the first c.

cccThere are 9 a's.

```
\newcount\mycount
\pgfparserdef{myparser}{initial}{the letter a}
{\advance\mycount by 1\relax}
\pgfparserdef{myparser}{initial}{the letter b}
{} % do nothing
\pgfparserdef{myparser}{initial}{the letter c}
{\pgfparserswitch{final}}% done!
\pgfparserparse{myparser}aabaababbbbbabaabcccc
There are \the\mycount\ a's.
```

```
\protect{pgfparserdef{\langle parser\ name \rangle}{\langle state \rangle}}{\langle symbol\ meaning \rangle}{\langle action \rangle}
```

This command should be used repeatedly to define a parser named $\langle parser\ name \rangle$. With a call to this command you specify that the $\langle parser\ name \rangle$ should do the following: When it is in state $\langle state \rangle$ and reads the letter $\langle symbol\ meaning \rangle$, perform the code stored in $\langle action \rangle$.

The $\langle symbol\ meaning \rangle$ must be the text that results from applying the T_EX command \meaning to the given character. For instance, \meaning a yields the letter a, while \meaning 1 yields the character 1. A space yields blank space.

Inside the $\langle action \rangle$ you can perform almost any kind of code. This code will not be surrounded by a scope, so its effect persists after the parsing is done. However, each time after the $\langle action \rangle$ is executed, control goes back to the parser. You should not launch a parser inside the $\langle action \rangle$ code, unless you put it in a scope.

When you set the $\langle state \rangle$ to all, the state $\langle action \rangle$ is performed in all states as a fallback, whenever $\langle symbol\ meaning \rangle$ is encountered. This means that when you do not specify anything explicitly for a state and a letter, but you do specify something for all and this letter, then the specified $\langle action \rangle$ will be used.

When the parser encounters a letter for which nothing is specified in the current state (neither directly nor indirectly via all), an error occurs.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command can be called inside the action code of a parser to cause a state switch to (state).

Part VIII

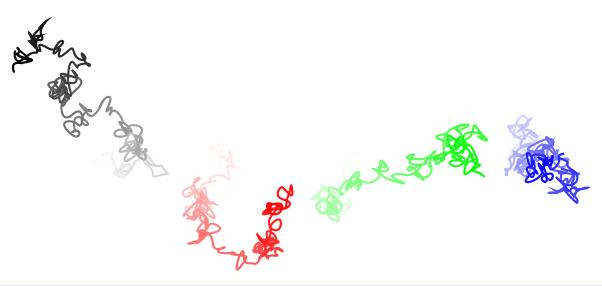
Mathematical and Object-Oriented Engines

by Mark Wibrow and Till Tantau

PGF comes with two useful engines: One for doing mathematics, one for doing object-oriented programming. Both engines can be used independently of the main PGF.

The job of the mathematical engine is to support mathematical operations like addition, subtraction, multiplication and division, using both integers and non-integers, but also functions such as square-roots, sine, cosine, and generate pseudo-random numbers. Mostly, you will use the mathematical facilities of PGF indirectly, namely when you write a coordinate like (5cm*3,6cm/4), but the mathematical engine can also be used independently of PGF and TikZ.

The job of the object-oriented engine is to support simple object-oriented programming in TeX. It allows the definition of classes (without inheritance), methods, attributes and objects.



69 Design Principles

PGF needs to perform many computations while typesetting a picture. For this, PGF relies on a mathematical engine, which can also be used independently of PGF, but which is distributed as part of the PGF package nevertheless. Basically, the engine provides a parsing mechanism similar to the CALC package so that expressions like 2*3cm+5cm can be parsed; but the PGF engine is more powerful and can be extended and enhanced.

PGF provides enhanced functionality, which permits the parsing of mathematical operations involving integers and non-integers with or without units. Furthermore, various functions, including trigonometric functions and random number generators can also be parsed (see Section 70.1). The CALC macros \setlength and friends have PGF versions which can parse these operations and functions (see Section 70.1). Additionally, each operation and function has an independent PGF command associated with it (see Section 72), and can be accessed outside the parser.

The mathematical engine of PGF is implicitly used whenever you specify a number or dimension in a higher-level macro. For instance, you can write \pgfpoint{2cm+4cm/2}{3cm*sin(30)} or suchlike. However, the mathematical engine can also be used independently of the PGF core, that is, you can also just load it to get access to a mathematical parser.

69.1 Loading the Mathematical Engine

The mathematical engine of PGF is loaded automatically by PGF, but if you wish to use the mathematical engine but you do not need PGF itself, you can load the following package:

```
\usepackage{pgfmath} % MEX
\input pgfmath.tex % plain TEX
\usemodule[pgfmath] % ConTEXt
```

This command will load the mathematical engine of PGF, but not PGF itself. It defines commands like \pgfmathparse.

69.2 Layers of the Mathematical Engine

Like PGF itself, the mathematical engine is also structured into different layers:

- 1. The top layer, which you will typically use directly, provides the command \pgfmathparse. This command parses a mathematical expression and evaluates it.
 - Additionally, the top layer also defines some additional functions similar to the macros of the calc package for setting dimensions and counters. These macros are just wrappers around the \pgfmathparse macro.
- 2. The calculation layer provides macros for performing one specific computation like computing a reciprocal or a multiplication. The parser uses these macros for the actual computation.
- 3. The implementation layer provides the actual implementations of the computations. These can be changed (and possibly be made more efficient) without affecting the higher layers.

69.3 Efficiency and Accuracy of the Mathematical Engine

Currently, the mathematical algorithms are all implemented in TEX. This poses some intriguing programming challenges as TEX is a language for typesetting, rather than for general mathematics, and as with any programming language, there is a trade-off between accuracy and efficiency. If you find the level of accuracy insufficient for your purposes, you will have to replace the algorithms in the implementation layer.

All the fancy mathematical "bells-and-whistles" that the parser provides, come with an additional processing cost, and in some instances, such as simply setting a length to 1cm, with no other operations involved, the additional processing time is undesirable. To overcome this, the following feature is implemented: when no mathematical operations are required, an expression can be preceded by +. This will bypass the parsing process and the assignment will be orders of magnitude faster. This feature *only* works with the macros for setting registers described in Section 70.1.

```
\pgfmathsetlength\mydimen{1cm} % parsed : slower.
\pgfmathsetlength\mydimen{+1cm} % not parsed : much faster.
```

70 Evaluating Mathematical Expressions

The easiest way of using PGF's mathematical engine is to provide a mathematical expression given in familiar infix notation, for example, 1cm+4*2cm/5.5 or 2*3+3*sin(30). This expression can be parsed by the mathematical engine and the result can be placed in a dimension register, a counter, or a macro.

It should be noted that all calculations must not exceed ± 16383.99999 at any point, because the underlying computations rely on T_EX dimensions. This means that many of the underlying computations are necessarily approximate and, in addition, not very fast. T_EX is, after all, a typesetting language and not ideally suited to relatively advanced mathematical operations. However, it is possible to change the computations as described in Section 73.

In the present section, the high-level macros for parsing an expression are explained first, then the syntax for expression is explained.

70.1 Commands for Parsing Expressions

The basic command for invoking the parser of PGF's mathematical engine is the following:

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

This macro parses (expression) and returns the result without units in the macro \pgfmathresult.

Example: \pgfmathparse{2pt+3.5pt} will set \pgfmathresult to the text 5.5.

In the following, the special properties of this command are explained. The exact syntax of mathematical expressions is explained in Section 71.

- The result stored in the macro \pgfmathresult is a decimal without units. This is true regardless of whether the \(\lambda expression \rangle \) contains any unit specification. All numbers with units are converted to points first.
- You can check whether an expression contained a unit using the TEX-if \ifpgfmathunitsdeclared. After a call of \pgfmathparse this if will be true exactly if some unit was encountered in the expression.
- The parser can recognize TEX registers and box dimensions, so \mydimen, 0.5\mydimen, \wd\mybox, 0.5\dp\mybox, \mycount\mydimen and so on can be parsed.
- Parenthesis can be used to change the order of the evaluation.
- Various functions are recognized, so it is possible to parse $sin(.5*pi\ r)*60$, which means "the sine of 0.5 times π radians, multiplied by 60". The argument of functions can be any expression.
- Scientific notation in the form 1.234e+4 is recognized (but the restriction on the range of values still applies). The exponent symbol can be upper or lower case (i.e., E or e).
- An integer with a zero-prefix (excluding, of course zero itself), is interpreted as an octal number and is automatically converted to base 10.
- An integer with prefix 0x or 0X is interpreted as a hexadecimal number and is automatically converted to base 10. Alphabetic digits can be in uppercase or lowercase.
- An integer with prefix 0b or 0B is interpreted as a binary number and is automatically converted to base 10.
- An expression (or part of an expression) surrounded with double quotes (i.e., the character ") will not be evaluated. Obviously this should be used with great care.

$\protect\operatorname{\mathtt{ar{pgfmathqparse}}}$

This macro is similar to \pgfmathparse: it parses \(\lambda expression \rangle \) and returns the result in the macro \pgfmathresult. It differs in two respects. Firstly, \pgfmathqparse does not parse functions, scientific notation, the prefixes for binary octal, or hexadecimal numbers, nor does it accept the special use of ", ? or: characters. Secondly, numbers in \(\lambda expression \rangle must specify a TeX unit (except in such instances as 0.5\pgf@x), which greatly simplifies the problem of parsing real numbers. As a result of these restrictions \pgfmathqparse is about twice as fast as \pgfmathparse. Note that the result will still be a number without units.

\pgfmathpostparse

At the end of the parse this command is executed, allowing some custom action to be performed on the result of the parse. When this command is executed, the macro \pgfmathresult will hold the result of the parse (as always, without units). The result of the custom action should be used to redefine \pgfmathresult appropriately. By default, this command is equivalent to \relax. This differs from previous versions, where, if the parsed expression contained no units, the result of the parse was scaled according to the value in \pgfmathresultunitscale (which by default was 1).

This scaling can be turned on again using: \let\pgfmathpostparse=\pgfmathscaleresult. Note, however that by scaling the result, the base conversion functions will not work, and the " character should not be used to quote parts of an expression.

Instead of the \pgfmathparse macro you can also use wrapper commands, whose usage is very similar to their cousins in the CALC package. The only difference is that the expressions can be any expression that is handled by \pgfmathparse. For all of the following commands, if \(\lambda expression \rangle \) starts with +, no parsing is done and a simple assignment or increment is done using normal TeX assignments or increments. This will be orders of magnitude faster than calling the parser.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                     Sets the length of the T_{EX} (dimension register), to the value (in points) specified by (expression). The
                                                     ⟨expression⟩ will be parsed using \pgfmathparse.
\protect\operatorname{\mathtt{Npgfmathaddtolength}} \{\langle dimension\ register \rangle\} \{\langle expression \rangle\}
                                                     Adds the value (in points) of \langle expression \rangle to the T<sub>F</sub>X \langle dimension \ register \rangle.
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                     Sets the value of the T_{FX} (count register), to the truncated value specified by (expression).
\protect\operatorname{\mathtt{ightheta}} \{\langle count\ register \rangle\} \{\langle expression \rangle\}
                                                       Adds the truncated value of \langle expression \rangle to the T<sub>F</sub>X \langle count\ register \rangle.
\protect\operatorname{\counter} {\counter} {\counter} {\counter} {\counter} {\counter} {\counter}
                                                     Sets the value of the AT_{FX} \langle counter \rangle to the truncated value specified by \langle expression \rangle.
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\be
                                                     Adds the truncated value of \langle expression \rangle to \langle counter \rangle.
\protect{\protect} \protect{\p
                                                     Defines \langle macro \rangle as the value of \langle expression \rangle. The result is a decimal without units.
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                     Defines \langle macro \rangle as the value of \langle expression \rangle LATEX in points.
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{
                                                     Defines \langle macro \rangle as the truncated value of \langle expression \rangle.
```

71 Syntax for mathematical expressions

The syntax for the expressions recognized by \pgfmathparse and friends is straightforward, and the following sections describe the operators and functions that are recognized by default.

71.1 Operators

The following operators (presented in the context in which they are used) are recognized:

```
x + y (infix operator; uses the add function)
Adds x to y.
```

```
(infix operator; uses the subtract function)
x - y
    Subtracts y from x.
-x
                                                                      (prefix operator; uses the neg function)
    Reverses the sign of x.
                                                                 (infix operator; uses the multiply function)
x * y
    Multiples x by y.
x / y
                                                                   (infix operator; uses the divide function)
    Divides x by y. An error will result if y is 0, or if the result of the division is too big for the mathematical
    engine. Please remember when using this command that accurate (and reasonably quick) division of
    real numbers that are not integers is particularly tricky in T<sub>F</sub>X.
x \, \hat{} \, y
                                                                       (infix operator; uses the pow function)
    Raises x to the power y.
x!
                                                             (postfix operator; uses the factorial function)
    Calculates the factorial of x.
                                                                    (postfix operator; uses the deg function)
xr
    Converts x to degrees (x is assumed to be in radians). This operator has the same precedence as
    multiplication.
x ? y : z
                                                       (conditional operators; use the ifthenelse function)
    ? and : are special operators which can be used as a shorthand for if x then y else z inside the parser.
    The expression x is taken to be true if it evaluates to any non-zero value.
x == y
                                                                    (infix operator; uses the equal function)
    Returns 1 if x=y, 0 otherwise.
x > y
                                                                  (infix operator; uses the greater function)
    Returns 1 if x>y, 0 otherwise.
                                                                      (infix operator; uses the less function)
x < y
    Returns 1 if x < y, 0 otherwise.
                                                                 (infix operator; uses the notequal function)
x != y
    Returns 1 if x \neq y, 0 otherwise.
                                                                  (infix operator; uses the notless function)
    Returns 1 if x \ge y, 0 otherwise.
                                                              (infix operator; uses the notgreater function)
    Returns 1 if x \leq y, 0 otherwise.
                                                                       (infix operator; uses the and function)
x & y
    Returns 1 if both x and y evaluate to some non-zero value. Both arguments are evaluated.
                                                                        (infix operator; uses the or function)
x \mid \mid y
    Returns 1 if either x or y evaluate to some non-zero value.
!x
                                                                     (prefix operator; uses the not function)
    Returns 1 if x evaluates to zero, 0 otherwise.
(x)
                                                                                            (group operators)
    These operators act in the usual way, that is, to control the order in which operators are executed, for
    example, (1+2)*3. This includes the grouping of arguments for functions, for example, sin(30*10) or
    mod(72,3) (the comma character is also treated as an operator).
    Parentheses for functions with one argument are not always necessary, sin 30 (note the space) is
```

sin(30)*10.

the same as sin(30). However, functions have the highest precedence so, sin 30*10 is the same as

 $\{x\}$ (array operators)

These operators are used to process array-like structures (within an expression these characters do not act like TeX grouping tokens). The $\langle array \ specification \rangle$ consists of comma separated elements, for example, {1, 2, 3, 4, 5}. Each element in the array will be evaluated as it is parsed, so expressions can be used. In addition, an element of an array can be an array itself, allowing multiple dimension arrays to be simulated: {1, {2,3}, {4,5}, 6}. When storing an array in a macro, do not forget the surrounding braces: \def\myarray{{1,2,3}} not \def\myarray{{1,2,3}}.

```
1, two, 3.0, IV, cinq, sechs, 7.0,

\def\myarray{{1,"two",2+1,"IV","cinq","sechs",sin(\i*5)*14}}
\foreach \i in {0,...,6}{\pgfmathparse{\myarray[\i]}\pgfmathresult, }
```

[x] (array access operators; use the array function)

[and] are two operators used in one particular circumstance: to access an array (specified using the $\{$ and $\}$ operators) using the index x. Indexing starts from zero, so, if the index is greater than, or equal to, the number of values in the array, an error will occur, and zero will be returned.

```
-9.0 \def\myarray{{7,-3,4,-9,11}} \pgfmathparrse{\myarray[3]} \pgfmathresult
```

If the array is defined to have multiple dimensions, then the array access operators can be immediately repeated.

"x" (group operators)

These operators are used to quote x. However, as every expression is expanded with $\ensuremath{\verb|cdef|}$ before it is parsed, macros (e.g., font commands like $\t t$ or $\t Wuge$) may need to be "protected" from this expansion (e.g., $\ensuremath{\verb|cned|}$ huge). Ideally, you should avoid such macros anyway. Obviously, these operators should be used with great care as further calculations are unlikely to be possible with the result.

```
5 is Bigger than 0. 5 is smaller than 10.

\def\x{5}
\foreach \y in {0,10}{
\pgfmathparse{\x > \y ? "\noexpand\Large Bigger" : "\noexpand\tiny smaller"}
\x\ is \pgfmathresult\ than \y.
}
```

71.2 Functions

The following functions are recognized:

abs	cosec	floor	isprime	not	real	veclen
acos	cosh	frac	less	notequal	rnd	width
add	cot	gcd	ln	notgreater	round	
and	deg	greater	log10	notless	sec	
array	depth	height	log2	oct	sin	
asin	div	hex	max	or	sinh	
atan	divide	Hex	min	pi	sqrt	
atan2	е	int	mod	pow	subtract	
bin	equal	ifthenelse	Mod	rad	tan	
ceil	factorial	iseven	multiply	rand	tanh	
cos	false	isodd	neg	random	true	

Each function has a PGF command associated with it (which is also shown with the function below). In general, the command is simply the name of the function prefixed with \pgfmath, for example, \pgfmathadd, but there are some notable exceptions.

71.2.1 Basic arithmetic functions

```
add(x,y)
 \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
                                                  Adds x and y.
                                                                                                                                                                                                                                                                                                                        \pgfmathparse{add(75,6)} \pgfmathresult
 subtract(x,y)
 \protect{pgfmathsubtract}{x}{y}
                                                  Subtract x from y.
                                                                                                                                                                                                                                                                                                                        \pgfmathparse{subtract(75,6)} \pgfmathresult
neg(x)
\protect\operatorname{pgfmathneg}\{x\}
                                                  This returns -x.
                                                                                                                                                                                                                                           -50.0 \pgfmathparse{neg(50)} \pgfmathresult
multiply(x,y)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                  Multiply x by y.
                                                                                                                                                                                                                                      450.0 \pgfmathparse{multiply(75,6)} \pgfmathresult
divide(x,y)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                Divide x by y.
                                                                                                                                                                                                                                                   12.5 \pgfmathparse{divide(75,6)} \pgfmathresult
div(x,y)
\protect\operatorname{\mathsf{pgfmathdiv}}\{x\}\{y\}
                                                Divide x by y and round to the nearest integer
                                                                                                                                                                                                                                                                                     8 \pgfmathparse{div(75,9)} \pgfmathresult
 factorial(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                  Return x!.
                                                                                                                                                                                                                                      120.0 \pgfmathparse{factorial(5)} \pgfmathresult
 sqrt(x)
\verb|\pgfmathsqrt{}(x)|
                                                Calculates \sqrt{x}.
                                                                                                                                                                                                                                                                                                                        \pgfmathparse{sqrt(10)} \pgfmathresult
                                                                                                                                                                                                               3.16227
                                                                                                                                                                                                      93.62388 \pgfmathparse{sqrt(8765.432)} \pgfmathresult
```

```
pow(x, y)
 \protect\operatorname{pgfmathpow}\{x\}\{y\}
                                                Raises x to the power y. For greatest accuracy, y should be an integer. If y is not an integer, the actual
                                                calculation will be an approximation of e^{y\ln(x)}.
                                                                                                                                                                                                                             128.0 \pgfmathparse{pow(2,7)} \pgfmathresult
 е
 \pgfmathe
                                                Returns the value 2.718281828.
                                                                                                                                                                                                       3.62685 \pgfmathparse{(e^2-e^-2)/2} \pgfmathresult
 exp(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                              Maclaurin series for e^x.
                                                                                                                                                                                                       2.71825 \pgfmathparse{exp(1)} \pgfmathresult
                                                                                                                                                                                           10.38083 \pgfmathparse{exp(2.34)} \pgfmathresult
ln(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                An approximation for ln(x). This uses an algorithm of Rouben Rostamian, and coefficients suggested
                                              by Alain Matthes.
                                                                                                                                                                                                         2.30257 \pgfmathparse{ln(10)} \pgfmathresult
                                                                                                                                                                                                      4.99997 \pgfmathparse{ln(exp(5))} \pgfmathresult
log10(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                An approximation for \log_{10}(x).
                                                                                                                                                                                                      1.99997 \pgfmathparse{log10(100)} \pgfmathresult
log2(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                An approximation for \log_2(x).
                                                                                                                                                                                                         6.99994 \pgfmathparse{log2(128)} \pgfmathresult
 abs(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                Evaluates the absolute value of x.
                                                                                                                                                                                                                                                                                                            \pgfmathparse{abs(-5)} \pgfmathresult
                                                                                                                                                                                                                                   -12.0 \pgfmathparse{-abs(4*-3)} \pgfmathresult
```

mod(x,y)

```
\protect\operatorname{\mathsf{pgfmathmod}}\{x\}\{y\}
```

This evaluates x modulo y, using truncated division. The sign of the result is the same as the sign of $\frac{x}{y}$.

2.0 \pgfmathparse{mod(20,6)} \pgfmathresult

-10.0 \pgfmathparse{mod(-100,30)} \pgfmathresult

Mod(x, y)

 $\protect\operatorname{\mathsf{Mod}}\{x\}\{y\}$

This evaluates x modulo y, using floored division. The sign of the result is never negative.

20.0 \pgfmathparse{Mod(-100,30)} \pgfmathresult

71.2.2 Rounding functions

round(x)

 $\protect\pro$

Rounds x to the nearest integer. It uses "asymmetric half-up" rounding. So 1.5 is rounded to 2, but -1.5 is rounded to -2 (not -1).

2.0 \pgfmathparse{round(32.5/17)} \pgfmathresult

33.0 \pgfmathparse{round(398/12)} \pgfmathresult

floor(x)

 $\protect\$

Rounds x down to the nearest integer.

1.0 \pgfmathparse{floor(32.5/17)} \pgfmathresult

33.0 \pgfmathparse{floor(398/12)} \pgfmathresult

-34.0 \pgfmathparse{floor(-398/12)} \pgfmathresult

ceil(x)

 $\protect\$

Rounds x up to the nearest integer.

 $\frac{2.0}{2.0}$ \pgfmathparse{ceil(32.5/17)} \pgfmathresult

34.0 \pgfmathparse{ceil(398/12)} \pgfmathresult

-33.0 \pgfmathparse{ceil(-398/12)} \pgfmathresult

int(x)

\pgfmathint{x}

Returns the integer part of x.

1 \pgfmathparse{int(32.5/17)} \pgfmathresult

frac(x)

```
\pgfmathfrac{x}
                    Returns the fractional part of x.
                                                                                      0.91176 \pgfmathparse{frac(32.5/17)} \pgfmathresult
real(x)
\protect\operatorname{pgfmathreal}{x}
                    Ensures x contains a decimal point.
                                                                                                         4.0 \pgfmathparse{real(4)} \pgfmathresult
71.2.3 Integer arithmetics functions
gcd(x,y)
\protect\operatorname{\mathsf{pgfmathgcd}}\{x\}\{y\}
                    Computes the greatest common divider of x and y.
                                                                                                           14 \pgfmathparse{gcd(42,56)} \pgfmathresult
isodd(x)
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                    Returns 1 if the integer part of x is odd. Otherwise, returns 0.
                                                                                                                                    \pgfmathparse{isodd(2)} \pgfmathresult,
                                                                                                                                     \pgfmathparse{isodd(3)} \pgfmathresult
iseven(x)
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                    Returns 1 if the integer part of x is even. Otherwise, returns 0.
                                                                                                                                   \pgfmathparse{iseven(2)} \pgfmathresult,
                                                                                                                                     \pgfmathparse{iseven(3)} \pgfmathresult
isprime(x)
\pgfmathisprime{x}
                    Returns 1 if the integer part of x is prime. Otherwise, returns 0.
                                                                                           0,1,1,0
                                                                                                                                   \pgfmathparse{isprime(1)} \pgfmathresult,
                                                                                                                                      \pgfmathparse{isprime(2)} \pgfmathresult,
                                                                                                                                      \pgfmathparse{isprime(31)} \pgfmathresult,
                                                                                                                                     \pgfmathparse{isprime(64)} \pgfmathresult
71.2.4 Trigonometric functions
рi
\pgfmathpi
                    Returns the value \pi = 3.141592654.
                                                                 3.141592654 \pgfmathparse{pi} \pgfmathresult
                                                                          179.99962 \pgfmathparse{pi r} \pgfmathresult
```

rad(x)

 $\protect\operatorname{pgfmathrad}\{x\}$

Convert x to radians. x is assumed to be in degrees.

1.57079 \pgfmathparse{rad(90)} \pgfmathresult

```
deg(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                   Convert x to degrees. x is assumed to be in radians.
                                                                                                                                                        269.999 \pgfmathparse{deg(3*pi/2)} \pgfmathresult
 sin(x)
 \protect\operatorname{pgfmathsin}\{x\}
                                    Sine of x. By employing the r operator, x can be in radians.
                                                                                                                                                        0.86603 \pgfmathparse{\sin(60)} \pgfmathresult
                                                                                                                                                        0.86601 \pgfmathparse{\sin(pi/3 r)} \pgfmathresult
 cos(x)
 \protect\operatorname{\mathsf{pgfmathcos}}\{x\}
                                    Cosine of x. By employing the r operator, x can be in radians.
                                                                                                                                                                                            0.5 \pgfmathparse{cos(60)} \pgfmathresult
                                                                                                                                                       0.49998 \pgfmathparse{cos(pi/3 r)} \pgfmathresult
 tan(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                    Tangent of x. By employing the \mathbf{r} operator, x can be in radians.
                                                                                                                                                       1.00005 \pgfmathparse{tan(45)} \pgfmathresult
                                                                                                                                                                                          1.0 \pgfmathparse{tan(2*pi/8 r)} \pgfmathresult
 sec(x)
 \protect\
                                    Secant of x. By employing the \mathbf{r} operator, x can be in radians.
                                                                                                                                                       1.41429 \pgfmathparse{sec(45)} \pgfmathresult
 cosec(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                    Cosecant of x. By employing the \mathbf{r} operator, x can be in radians.
                                                                                                                                                                                          2.0 \pgfmathparse{cosec(30)} \pgfmathresult
 \cot(x)
 \pgfmathcot{x}
                                    Cotangent of x. By employing the \mathbf{r} operator, x can be in radians.
                                                                                                                                                       3.73215 \pgfmathparse{cot(15)} \pgfmathresult
asin(x)
 \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                    Arcsine of x. The result is in degrees and in the range \pm 90^{\circ}.
```

44.99945 \pgfmathparse{asin(0.7071)} \pgfmathresult

```
acos(x)
  \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                        Arccosine of x in degrees. The result is in the range [0^{\circ}, 180^{\circ}].
                                                                                                                                                                                                                                                                                     60.0 \pgfmathparse{acos(0.5)} \pgfmathresult
  atan(x)
  \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                        Arctangent of x in degrees.
                                                                                                                                                                                                                                                                                45.0 \pgfmathparse{atan(1)} \pgfmathresult
  atan2(x,y)
  \protect\operatorname{\mathsf{pgfmathatantwo}}\{x\}\{y\}
                                                        Arctangent of y \div x in degrees. This also takes into account the quadrants.
                                                                                                                                                                                                         143.13011 \pgfmathparse{atan2(-4,3)} \pgfmathresult
  71.2.5 Comparison and logical functions
  equal(x,y)
  This returns 1 if x = y and 0 otherwise.
                                                                                                                                                                                                                                                                                                                    1 \pgfmathparse{equal(20,20)} \pgfmathresult
greater(x, y)
  \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu
                                                        This returns 1 if x > y and 0 otherwise.
                                                                                                                                                                                                                                                                                                                        () \pgfmathparse{greater(20,25)} \pgfmathresult
  less(x,y)
  \protect\operatorname{\mathsf{pgfmathless}}\{x\}\{y\}
                                                        This returns 1 if x < y and 0 otherwise.
                                                                                                                                                                                                                                                                                                                      notequal(x, y)
  \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                                                        This returns 0 if x = y and 1 otherwise.
                                                                                                                                                                                                                                                                                                                    1 \pgfmathparse{notequal(20,25)} \pgfmathresult
  notgreater(x, y)
  \protect\operatorname{\begin{subarray}{c} \operatorname{\begin{subarray}{c} \operatorname{\begin{sub
                                                        This returns 1 if x \leq y and 0 otherwise.
                                                                                                                                                                                                                                                                                                                    1 \pgfmathparse{notgreater(20,25)} \pgfmathresult
notless(x,y)
  \protect\operatorname{\begin{tabular}{l} pgfmathnotless{x}{y}} \protect\protect\operatorname{\begin{tabular}{l} pgfmathnotless{x}{y}} \protect\protect\operatorname{\begin{tabular}{l} pgfmathnotless{x}{y}} \protect\protect\operatorname{\begin{tabular}{l} pgfmathnotless{x}{y}} \protect\protect\operatorname{\begin{tabular}{l} pgfmathnotless{x}{y}} \protect\protect\operatorname{\begin{tabular}{l} pgfmathnotless{x}{y}} \protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\p
                                                        This returns 1 if x \geq y and 0 otherwise.
                                                                                                                                                                                                                                                                                                                      () \pgfmathparse{notless(20,25)} \pgfmathresult
```

```
and(x,y)
\protect\operatorname{\mathsf{pgfmathand}}\{x\}\{y\}
                   This returns 1 if x and y both evaluate to non-zero values. Otherwise 0 is returned.
                                                                                                           () \pgfmathparse{and(5>4,6>7)} \pgfmathresult
or(x,y)
\protect\ \pgfmathor{x}{y}
                   This returns 1 if either x or y evaluate to non-zero values. Otherwise 0 is returned.
                                                                                                            () \pgfmathparse{and(5>4,6>7)} \pgfmathresult
not(x)
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
                   This returns 1 if x = 0, otherwise 0.
                                                                                                          ifthenelse(x,y,z)
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu
                   This returns y if x evaluates to some non-zero value, otherwise z is returned.
                                                                                                    no \pgfmathparse{ifthenelse(5==4,"yes","no")} \pgfmathresult
true
\pgfmathtrue
                   This evaluates to 1.
                                                                                                                         \pgfmathparse{true ? "yes" : "no"} \pgfmathresult
false
\pgfmathfalse
                   This evaluates to 0.
                                                                                                                         \pgfmathparse{false ? "yes" : "no"} \pgfmathresult
                                     Pseudo-random functions
71.2.6
rnd
\pgfmathrnd
                   Generates a pseudo-random number between 0 and 1 with a uniform distribution.
                     0.35255, 0.52394, 0.9081, 0.07314, 0.01263, 0.86674, 0.42123, 0.63577, 0.0252, 0.1887,
                    \foreach \x in \{1,...,10\}{\pgfmathparse\{rnd\}\pgfmathresult, }
rand
\pgfmathrand
                   Generates a pseudo-random number between -1 and 1 with a uniform distribution.
                     -0.171, -0.6502, 0.80553, 0.81995, 0.18893, 0.25227, 0.83768, -0.39989, -0.43884, 0.51918,
```

random(x,y)

$\protect\$

This function takes zero, one or two arguments. If there are zero arguments, a uniform random number between 0 and 1 is generated. If there is one argument x, a random integer between 1 and x is generated. Finally, if there are two arguments, a random integer between x and y is generated. If there are no arguments, the PGF command should be called as follows: $pgfmathrandom{}$

```
0.67236, 0.3279, 0.01428, 0.76451, 0.26901, 0.00323, 0.03891, 0.40141, 0.8682, 0.51057, \foreach \x in {1,...,10}{\pgfmathparse{random()}\pgfmathresult, }
```

52, 2, 17, 39, 1, 53, 85, 42, 100, 39,

619, 426, 467, 758, 569, 493, 674, 626, 614, 400,

 $\label{local-prop} $$ \operatorname{x in } \{1,\ldots,10\} \operatorname{product} (232,762) \right) $$$

71.2.7 Base conversion functions

hex(x)

$\protect\pro$

Convert x (assumed to be an integer in base 10) to a hexadecimal representation, using lower case alphabetic digits. No further calculation will be possible with the result.

ffff \pgfmathparse{hex(65535)} \pgfmathresult

Hex(x)

$\protect\pro$

Convert x (assumed to be an integer in base 10) to a hexadecimal representation, using upper case alphabetic digits. No further calculation will be possible with the result.

FFFF \pgfmathparse{Hex(65535)} \pgfmathresult

oct(x)

$\protect\{x\}$

Convert x (assumed to be an integer in base 10) to an octal representation. No further calculation should be attempted with the result, as the parser can only process numbers converted to base 10.

77 \pgfmathparse{oct(63)} \pgfmathresult

bin(x)

$\protect\operatorname{pgfmathbin}\{x\}$

Convert x (assumed to be an integer in base 10) to a binary representation. No further calculation should be attempted with the result, as the parser can only process numbers converted to base 10.

10111001 \pgfmathparse{bin(185)} \pgfmathresult

71.2.8 Miscellaneous functions

```
\min(x_1, x_2, \dots, x_n)
\pgfmathmin\{x_1, x_2, \dots\} \{\dots, x_{n-1}, x_n\}
```

Return the minimum value from $x_1
ldots x_n$. For historical reasons, the command \pgfmathmin takes two arguments, but each of these can contain an arbitrary number of comma separated values.

-8.0 \pgfmathparse{min(3,4,-2,250,-8,100)} \pgfmathresult

```
\max(x_1, x_2, ..., x_n)
\pgfmathmax{x_1, x_2, ...}{..., x_{n-1}, x_n}
```

Return the maximum value from $x_1 cdots x_n$. Again, for historical reasons, the command \pgfmathmax takes two arguments, but each of these can contain an arbitrary number of comma separated values.

250.0 \pgfmathparse{max(3,4,-2,250,-8,100)} \pgfmathresult

veclen(x,y) \pgfmathveclen{x}{y}

Calculates $\sqrt{(x^2+y^2)}$. This uses a polynomial approximation, based on ideas of Rouben Rostamian

12.99976 \pgfmathparse{veclen(12,5)} \pgfmathresult

array(x,y)
\pgfmatharray{x}{y}

This accesses the array x at the index y. The array must begin and end with braces (e.g., $\{1,2,3,4\}$) and array indexing starts at 0.

17 \pgfmathparse{array({9,13,17,21},2)} \pgfmathresult

The following hyperbolic functions were adapted from code suggested by Martin Heller:

sinh(x)
\pgfmathsinh{x}

The hyperbolic sine of x

0.52103 \pgfmathparse{sinh(0.5)} \pgfmathresult

cosh(x)

 $\protect\pro$

The hyperbolic cosine of x

1.12767 \pgfmathparse{cosh(0.5)} \pgfmathresult

tanh(x)

 $\protect\pro$

The hyperbolic tangent of x

0.462 \pgfmathparse{tanh(0.5)} \pgfmathresult

width("x")

 $\protect\operatorname{pgfmathwidth}\{"x"\}$

Return the width of a T_{EX} (horizontal) box containing x. The quote characters are necessary to prevent x from being parsed. It is important to remember that any expression is expanded with $\ensuremath{\mbox{edef}}$ being parsed, so any macros (e.g., font commands like $\t \t$ or $\t \t$ Wuge) will need to be "protected" (e.g., $\t \t$ noexpand $\t \t$ Wuge is usually sufficient).

78.47237 \pgfmathparse{width("Some Lovely Text")} \pgfmathresult

Note that results of this method are provided in points.

height("x")

 $\protect\pro$

Return the height of a box containing x.

6.94444 \pgfmathparse{height("Some Lovely Text")} \pgfmathresult

```
depth("x")
\pgfmathdepth{"x"}
```

Returns the depth of a box containing x.

1.94444 \pgfmathparse{depth("Some Lovely Text")} \pgfmathresult

72 Additional Mathematical Commands

Instead of parsing and evaluating complex expressions, you can also use the mathematical engine to evaluate a single mathematical operation. The macros used for many of these computations are listed above in Section 71.2. PGF also provides some additional commands which are shown below:

72.1 Basic arithmetic functions

In addition to the commands described in Section 71.2.1, the following command is provided:

```
\protect\
```

Defines \pgfmathresult as $1 \div \langle x \rangle$. This provides greatest accuracy when x is small.

72.2 Comparison and logical functions

In addition to the commands described in Section 71.2.5, the following command was provided by Christian Feuersänger:

```
\protect\ \pgfmathapproxequal to \{\langle x \rangle\} \{\langle y \rangle\}
```

Defines \pgfmathresult 1.0 if $|\langle x \rangle - \langle y \rangle| < 0.0001$, but 0.0 otherwise. As a side-effect, the global boolean \ifpgfmathcomparison will be set accordingly.

72.3 Pseudo-Random Numbers

In addition to the commands described in Section 71.2.6, the following commands are provided:

\pgfmathgeneratepseudorandomnumber

Defines \pgfmathresult as a pseudo-random integer between 1 and $2^{31} - 1$. This uses a linear congruency generator, based on ideas of Erich Janka.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

This defines $\langle macro \rangle$ as a pseudo-randomly generated integer from the range $\langle maximum \rangle$ to $\langle minimum \rangle$ (inclusive).

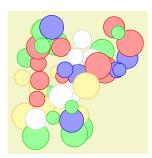
```
\begin{pgfpicture}
  \foreach \x in {1,...,50}{
    \pgfmathrandominteger{\a}{1}{50}
    \pgfmathrandominteger{\b}{1}{50}
    \pgfpathcircle{\pgfpoint{+\a pt}{+\b pt}}{+2pt}
    \color{blue!40!white}
    \pgfsetstrokecolor{blue!80!black}
    \pgfusepath{stroke, fill}
  }
\end{pgfpicture}
```

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This creates a list of items with the name $\langle list \ name \rangle$.

```
\verb|\pgfmathrandomitem{|\langle macro\rangle|} {\langle list\ name\rangle}|
```

Select an item from a random list $\langle list \ name \rangle$. The selected item is placed in $\langle macro \rangle$.



```
\begin{pgfpicture}
  \pgfmathdeclarerandomlist{color}{{red}{blue}{green}{yellow}{white}}
  \foreach \a in {1,...,50}{
    \pgfmathrandominteger{\x}{1}{85}}
  \pgfmathrandominteger{\y}{1}{85}}
  \pgfmathrandominteger{\r}{5}{10}
  \pgfmathrandominteger{\r}{5}{10}
  \pgfmathrandomitem{\c}{color}
  \pgfpathcircle{\pgfpoint{+\x pt}{+\y pt}}{+\r pt}
  \color{\c!40!white}
  \pgfsetstrokecolor{\c!80!black}
  \pgfusepath{stroke, fill}
}
\end{pgfpicture}
```

$\protect\operatorname{\mathtt{ightheta}}$

Explicitly sets the seed for the pseudo-random number generator. By default it is set to the value of \time×\year.

72.4 Base Conversion

PGF provides limited support for conversion between representations of numbers. Currently the numbers must be positive integers in the range 0 to $2^{31} - 1$, and the bases in the range 2 to 36. All digits representing numbers greater than 9 (in base ten), are alphabetic, but may be upper or lower case.

In addition to the commands described in Section 71.2.7, the following commands are provided:

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Defines $\langle macro \rangle$ as the result of converting $\langle number \rangle$ from base $\langle base \rangle$ to base 10. Alphabetic digits can be upper or lower case.

4223 \pgfmathbasetodec\mynumber{107f}{16} \mynumber

Note that, as usual in T_EX, the braces around an argument can be omitted if the argument is just a single token (a macro name is a single token).

25512 \pgfmathbasetodec\mynumber{33FC}{20} \mynumber

$\protect\pro$

Defines $\langle macro \rangle$ as the result of converting $\langle number \rangle$ from base 10 to base $\langle base \rangle$. Any resulting alphabetic digits are in *lower case*.

ffff \pgfmathdectobase\mynumber{65535}{16} \mynumber

$\protectoBase{\langle macro \rangle} {\langle number \rangle} {\langle base \rangle}$

Defines $\langle macro \rangle$ as the result of converting $\langle number \rangle$ from base 10 to base $\langle base \rangle$. Any resulting alphabetic digits are in *upper case*.

FFFF \pgfmathdectoBase\mynumber{65535}{16} \mynumber

$\verb|\pgfmathbasetobase{|\langle macro\rangle|}{|\langle number\rangle|}{|\langle base-1\rangle|}{|\langle base-2\rangle|}$

Defines $\langle macro \rangle$ as the result of converting $\langle number \rangle$ from base $\langle base-1 \rangle$ to base $\langle base-2 \rangle$. Alphabetic digits in $\langle number \rangle$ can be upper or lower case, but any resulting alphabetic digits are in *lower case*.

db \pgfmathbasetobase\mynumber{11011011}{2}{16} \mynumber

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \$

Defines $\langle macro \rangle$ as the result of converting $\langle number \rangle$ from base $\langle base-1 \rangle$ to base $\langle base-2 \rangle$. Alphabetic digits in $\langle number \rangle$ can be upper or lower case, but any resulting alphabetic digits are in *upper case*.

31B \pgfmathbasetoBase\mynumber{121212}{3}{12} \mynumber

$\protect\pro$

Sets the number of digits in the result of a base conversion to $\langle integer \rangle$. If the result of a conversion has less digits than this number, it is prefixed with zeros.

00001111 \pgfmathsetbasenumberlength{8} \pgfmathdectobase\mynumber{15}{2} \mynumber

73 Customizing the Mathematical Engine

Perhaps you have a desire for some function that PGF does not provide. Perhaps you are not happy with the accuracy or efficiency of some of the algorithms that are implemented in PGF. In these cases you will want to add a function to the parser or replace the current implementations of the algorithms with your own code.

The mathematical engine was designed with such customization in mind. It is possible to add new functions, or modify the code for existing functions. Note, however, that whilst adding new operators is possible, it can be a rather tricky business and is only recommended for adventurous users.

To add a new function to the math engine the following command can be used:

$\protect\pro$

This will set up the parser to recognize a function called $\langle name \rangle$. The name of the function can consist of, uppercase or lowercase letters, numbers or the underscore _. In line with many programming languages, a function name cannot begin with a number or contain any spaces.

The $\langle number\ of\ arguments \rangle$ can be any positive integer, zero, or the value ..., which indicates a variable number of arguments. PGF treats constants, such as pi and e, as functions with zero arguments. Functions with more than nine arguments or with a variable number of arguments are a "bit special" and are discussed below.

The effect of $\langle code \rangle$ should be to set the macro \pgfmathresult to the correct value (namely to the result of the computation without units). Furthermore, the function should have no other side effects, that is, it should not change any global values. As an example, consider the creation of a new function double, which takes one argument, and returns the value of that argument times two.

```
\makeatletter
\pgfmathdeclarefunction{double}{1}{
   \begingroup
   \pgf@x=#1pt\relax
   \multiply\pgf@x by2\relax
   \pgfmathreturn\pgf@x
   \endgroup
}
\makeatother
\pgfmathparse{double(44.3)}\pgfmathresult
```

The macro \pgfmathreturn $\langle tokens \rangle$ must be directly followed by an \endgroup and will save the result of the computation, by defining \pgfmathresult as the expansion of $\langle tokens \rangle$ (without units) outside the group, so $\langle tokens \rangle$ must be something that can be assigned to a dimension register.

Alternatively, the $\protect\operatorname{pgfmathsmuggle}(macro)$ can be used. This must also be directly followed by an $\protect\operatorname{pgfmathsmuggle}$ "smuggle" the definition of (macro) outside the T_EX -group.

By performing computations within a TeX-group, PGF registers such as \pgf@x, \pgf@y and \c@pgf@counta, \c@pgfcountb, and so forth, can be used at will.

Beyond setting up the parser, this command also defines two macros which provide access to the function independently of the parser:

• $\protect\$ \p

This macro will provide a "public" interface for the function $\langle function \ name \rangle$ allowing the function to be called independently of the parser. All arguments passed to this macro are evaluated using $\protect\operatorname{pgfmathparse}$ and then passed on to the following macro:

\pgfmath\\(function name \)\@

This macro is the "private" implementation of the function's algorithm (but note that, for speed, the parser calls this macro rather than the "public" one). Arguments passed to this macro are expected to be numbers without units. It is defined using $\langle code \rangle$, but need not be self-contained.

For functions that are declared with less than ten arguments, the public macro is defined in the same way as normal TEX macros using, for example, $\def\pgfmathNoArgs\{\langle code\rangle\}\$ for a function with no arguments, or $\def\pgfmathThreeArgs#1#2#3\{\langle code\rangle\}\$ for a function with three arguments. The private macro is defined in the same way, and each argument can therefore be accessed in $\langle code\rangle$ using #1, #2 and so on.

For functions with more than nine arguments, or functions with a variable number of arguments, these macros are only defined as taking *one* argument. The public macro expects its arguments to be comma separated, for example, \pgfmathVariableArgs{1.1,3.5,-1.5,2.6}. Each argument is parsed and passed on to the private macro as follows: \pgfmathVariableArgs@{{1.1}-{3.5}-1.5}{2.6}}. This means that some "extra work" will be required to access each argument (although it is a fairly simple task).

Note that there are two exceptions to this arrangement: the public versions of the min and max functions still take two arguments for compatibility with older versions, but each of these arguments can take several comma separated values.

To redefine a function use the following command:

$\protection{\langle function name \rangle} {\langle algorithm code \rangle}$

This command redefines the $\protect\operatorname{pgfmath}(function\ name)@$ macro with the new $(algorithm\ code)$. See the description of the $\protect\operatorname{pgfmathdeclarefunction}$ for details. You cannot change the number of arguments for an existing function.

PGF uses the last known definition of a function within the prevailing scope, so it is possible for a function to be redefined locally. You should also remember that any .sty or .tex file containing any re-implementations should be loaded after pgfmath.

In addition to the above commands, the following key is provided to quickly create simple ad hoc functions which can greatly improve the readability of code, and is particularly useful in TikZ:

```
/pgf/declare function=\langle function definitions\rangle
```

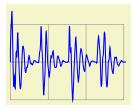
(no default)

This key allows simple functions to be created locally. Its use is perhaps best illustrated by an example:

```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw [blue, thick, x=0.0085cm, y=1cm,
    declare function={
      sines(\t,\a,\b)=1 + 0.5*(sin(\t)+sin(\t*\a)+sin(\t*\b));
    }]
    plot [domain=0:360, samples=144, smooth] (\x,{sines(\x,3,5)});
  \end{tikzpicture}
```

Each definition in $\langle function \ definitions \rangle$ takes the form $\langle name \rangle$ ($\langle arguments \rangle$) = $\langle definition \rangle$; (note the semicolon at the end, this is very important). If multiple functions are being defined, the semicolon is used to separate them (not a comma). The function $\langle name \rangle$ can be any name that is not already a function name in the current scope. The list of $\langle arguments \rangle$ are commands such as $\xspace x$, or $\xspace y$ (it is not possible to declare functions that take variable numbers of arguments using this key). If the function takes no arguments, then the parentheses need not be used. The $\langle definition \rangle$ should be an expression that can be parsed by the mathematical engine and should use the commands specified in $\langle arguments \rangle$.

When specifying multiple functions, functions that appear later on in $\langle function \ definitions \rangle$ can refer to earlier functions:



74 Number Printing

An extension by Christian Feuersänger

PGF supports number printing in different styles and rounds to arbitrary precision.

```
\protect\
```

Generates pretty-printed output for the (real) number $\langle x \rangle$. The input number $\langle x \rangle$ is parsed using \pgfmathfloatparsenumber which allows arbitrary precision.

Numbers are typeset in math mode using the current set of number printing options, see below. Optional arguments can also be provided using $\protect\operatorname{pgfmathprintnumber}[\langle options \rangle]\langle x \rangle$.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{
```

Returns the resulting number into $\langle macro \rangle$ instead of typesetting it directly.

/pgf/number format/fixed

(no value)

Configures \pgfmathprintnumber to round the number to a fixed number of digits after the period, discarding any trailing zeros.

```
4.57 0 0.1 24,415.98 123,456.12
```

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See section 74.1 for how to change the appearance.

```
/pgf/number format/fixed zerofill=\{\langle boolean \rangle\}
```

(default true)

Enables or disables zero filling for any number drawn in fixed point format.

```
4.57 0.00 0.10 24,415.98 123,456.12
```

```
\pgfkeys{/pgf/number format/.cd,fixed,fixed zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

This key affects numbers drawn with fixed or std styles (the latter only if no scientific format is chosen).

```
4.57 \quad 5 \cdot 10^{-5} \quad 1.00 \quad 1.23 \cdot 10^{5}
```

```
\pgfkeys{/pgf/number format/.cd,std,fixed zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-05}\hspace{1em}
\pgfmathprintnumber{1}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See section 74.1 for how to change the appearance.

/pgf/number format/sci

(no value)

Configures \pgfmathprintnumber to display numbers in scientific format, that means sign, mantissa and exponent (basis 10). The mantissa is rounded to the desired precision (or sci precision, see below).

```
4.57 \cdot 10^{0} \quad 5 \cdot 10^{-4} \quad 1 \cdot 10^{-1} \quad 2.44 \cdot 10^{4} \quad 1.23 \cdot 10^{5}
```

```
\pgfkeys{/pgf/number format/.cd,sci,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See section 74.1 for how to change the exponential display style.

```
/pgf/number format/sci zerofill=\{\langle boolean \rangle\}
```

(default true)

Enables or disables zero filling for any number drawn in scientific format.

```
4.57 \cdot 10^{0} 5.00 \cdot 10^{-4} 1.00 \cdot 10^{-1} 2.44 \cdot 10^{4} 1.23 \cdot 10^{5}
```

```
\pgfkeys{/pgf/number format/.cd,sci,sci zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

As with fixed zerofill, this option does only affect numbers drawn in sci format (or std if the scientific format is chosen).

See section 74.1 for how to change the exponential display style.

```
/pgf/number format/zerofill=\{\langle boolean \rangle\}
```

(style, default true)

Sets both fixed zerofill and sci zerofill at once.

```
\begin{tabular}{ll} \beg
```

Configures \pgfmathprintnumber to a standard algorithm. It chooses either fixed or sci, depending on the order of magnitude. Let $n = s \cdot m \cdot 10^e$ be the input number and p the current precision. If $-p/2 \le e \le 4$, the number is displayed using fixed format. Otherwise, it is displayed using sci format.

```
4.57 \quad 5 \cdot 10^{-4} \quad 0.1 \quad 24,415.98 \quad 1.23 \cdot 10^{5}
```

```
\pgfkeys{/pgf/number format/.cd,std,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

The parameters can be customized using the optional integer argument(s): if $\langle lower \ e \rangle \leq e \leq \langle upper \ e \rangle$, the number is displayed in fixed format, otherwise in sci format. Note that $\langle lower \ e \rangle$ should be negative for useful results. The precision used for the scientific format can be adjusted with sci precision if necessary.

```
/pgf/number format/relative*=\langle exponent \ base \ 10 \rangle
```

(no default)

Configures \pgfmathprintnumber to format numbers relative to an order of magnitude, 10^r , where r is an integer number.

This key addresses different use-cases.

First use-case: provide a unified format for a sequence of numbers. Consider the following test:

```
0 1.2 6 20.6 87 \pgfkeys{/pgf/number format/relative*={1}} \pgfmathprintnumber{6.42e-16}\hspace{1em} \pgfmathprintnumber{1.2}\hspace{1em} \pgfmathprintnumber{6}\hspace{1em} \pgfmathprintnumber{20.6}\hspace{1em} \pgfmathprintnumber{87}
```

With any other style, the 6.42e-16 would have been formatted as an isolated number. Here, it is rounded to 0 because when viewed relative to 10¹ (the exponent 1 is the argument for relative), it has no significant digits.

```
123 0 0 \pgfkeys{/pgf/number format/relative*={2}} \pgfmathprintnumber{123.345}\hspace{1em} \pgfmathprintnumber{0.0012}\hspace{1em} \pgfmathprintnumber{0.0014}\hspace{1em}
```

The example above applies the initial precision=2 to 123.345 – relative to 100. Two significant digits of 123.345 relative to 100 are 123. Note that the "2 significant digits of 123.345" translates to "round 1.2345 to 2 digits", which would yield 1.2300. Similarly, the other two numbers are 0 compared to 100 using the given precision.

```
123.345 \quad 1.2 \cdot 10^{-3} \quad 1.4 \cdot 10^{-3}
```

```
\pgfkeys{/pgf/number format/relative*={-3}}
\pgfmathprintnumber{123.345}\hspace{1em}
\pgfmathprintnumber{0.0012}\hspace{1em}
\pgfmathprintnumber{0.0014}\hspace{1em}
```

Second use-case: improve rounding in the presence of *inaccurate* numbers. Let us suppose that some limited-precision arithmetics resulted in the result 123456999 (like the fpu of PGF). You know that its precision is about five or six significant digits. And you want to provide a fixed point output. In this case, the trailing digits999 are a numerical artifact due to the limited precision. Use relative*=3,precision=0 to eliminate the artifacts:

```
1.23457 \cdot 10^8 \quad 1.23457 \cdot 10^8
```

```
\pgfkeys{/pgf/number format/.cd,relative*={3},precision=0}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

Here, precision=0 means that we inspect 123456.999 and round that number to 0 digits. Finally, we move the period back to its initial position. Adding relative style=fixed results in fixed point output format:

```
 \begin{array}{lll} 123,\!457,\!000 & 123,\!457,\!000 \\ & & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ & & & \\ &
```

Note that there is another alternative for this use-case which is discussed later: the fixed relative style.

```
123,457,000 123,457,000 \pgfkeys{/pgf/number format/.cd,fixed relative,precision=6} \pgfmathprintnumber{123456999}\hspace{1em} \pgfmathprintnumber{123456999.12}
```

You might wonder why there is an asterisk in the key's name. The short answer is: there is also a /pgf/number format/relative number printer which does unexpected things. The key relative* repairs this. Existing code will still use the old behavior.

Technically, the key works as follows: as already explained above, relative*=3 key applied to 123456999.12 moves the period by three positions and analyzes 123456.99912. Mathematically speaking, we are given a number $x = \pm m \cdot 10^e$ and we attempt to apply relative*=r. The method then rounds $x/10^r$ to precision digits. Afterwards, it multiplies the result by 10^r and typesets it.

```
/pgf/number format/every relative
```

(style, no value)

A style which configures how the relative method finally displays its results.

The initial configuration is

```
\pgfkeys{/pgf/number format/every relative/.style=std}
```

Note that rounding is turned off when the resulting style is being evaluated (since relative already rounded the number).

Although supported, I discourage from using fixed zerofill or sci zerofill in this context – it may lead to a suggestion of higher precision than is actually used (because fixed zerofill might simply add .00 although there was a different information before relative rounded the result).

```
/pgf/number format/relative style=\{\langle options \rangle\}
```

(no default)

The same as every relative/.append style= $\{\langle options \rangle\}$.

/pgf/number format/fixed relative

(no value)

Configures \pgfmathprintnumber to format numbers in a similar way to the fixed style, but the precision is interpreted relatively to the number's exponent.

The motivation is to get the same rounding effect as for sci, but to display the number in the fixed style:

1,000 100 0.00001 0.0101 1.24 1,000 1,010

```
\pgfkeys{/pgf/number format/.cd,fixed relative,precision=3}
\pgfmathprintnumber{1000.0123}\hspace{1em}
\pgfmathprintnumber{100.0567}\hspace{1em}
\pgfmathprintnumber{0.000010003452}\hspace{1em}
\pgfmathprintnumber{0.010073452}\hspace{1em}
\pgfmathprintnumber{1.23567}\hspace{1em}
\pgfmathprintnumber{1.033.75}\hspace{1em}
\pgfmathprintnumber{1006.75}\hspace{1em}
```

The effect of fixed relative is that the number is rounded to exactly the first $\langle precision \rangle$ non-zero digits, no matter how many leading zeros the number might have.

Use fixed relative if you want fixed and if you know that only the first n digits are correct. Use sci if you need a scientific display style and only the first n digits are correct.

Note that fixed relative ignores the fixed zerofill flag.

See also the relative* key. Note that the relative={\lambda exponent\rangle} key explicitly moves the period to some designated position before it attempts to round the number. Afterwards, it "rounds from the right", i.e. it rounds to that explicitly chosen digit position. In contrast to that, fixed relative "rounds from the left": it takes the first non-zero digit, temporarily places the period after this digit, and rounds that number. The rounding style fixed leaves the period where it is, and rounds everything behind that digit. The sci style is similar to fixed relative.

/pgf/number format/int detect

(no value)

Configures \pgfmathprintnumber to detect integers automatically. If the input number is an integer, no period is displayed at all. If not, the scientific format is chosen.

```
15 \quad 20 \quad 2.04 \cdot 10^{1} \quad 1 \cdot 10^{-2} \quad 0
```

```
\pgfkeys{/pgf/number format/.cd,int detect,precision=2}
\pgfmathprintnumber{15}\hspace{1em}
\pgfmathprintnumber{20}\hspace{1em}
\pgfmathprintnumber{20.4}\hspace{1em}
\pgfmathprintnumber{0.01}\hspace{1em}
\pgfmathprintnumber{0.01}\hspace{1em}
\pgfmathprintnumber{0}
```

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

A command which does the same check as int detect, but it invokes $\langle true\ code \rangle$ if the $\langle number\ constant \rangle$ actually is an integer and the $\langle false\ code \rangle$ if not.

As a side-effect, \pgfretval will contain the parsed number, either in integer format or as parsed floating point number.

The argument (number constant) will be parsed with \pgfmathfloatparsenumber.

```
15 is an int: 15. 15.5 is no int
```

```
15 \pgfmathifisint{15}{is an int: \pgfretval.}{is no int}\hspace{1em}
15.5 \pgfmathifisint{15.5}{is an int: \pgfretval.}{is no int}
```

/pgf/number format/int trunc

(no value)

Truncates every number to integers (discards any digit after the period).

```
4 0 0 24,415 123,456
```

```
\pgfkeys{/pgf/number format/.cd,int trunc}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

/pgf/number format/frac

(no value)

Displays numbers as fractionals.

```
/pgf/number format/frac TeX=\{\langle nacro \rangle\}
```

(no default, initially \frac)

Allows to use a different implementation for \frac inside of the frac display type.

```
/pgf/number format/frac denom=\langle int \rangle
```

\pgfmathprintnumber{-6}

(no default, initially empty)

Allows to provide a custom denominator for frac.

```
\frac{1}{10} \frac{5}{10} 1\frac{2}{10} -\frac{6}{10} -1\frac{4}{10}
```

```
\pgfkeys{/pgf/number format/.cd,frac, frac denom=10}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{0.5}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{-0.6}\hspace{1em}
\pgfmathprintnumber{-1.4}\hspace{1em}
```

/pgf/number format/frac whole=true|false

(no default, initially true)

Configures whether complete integer parts shall be placed in front of the fractional part. In this case, the fractional part will be less then 1. Use frac whole=false to avoid whole number parts.

```
201 11 2 6 -28 -75

\pgfkeys{/pgf/number format/.cd,frac, frac whole=false}

\pgfmathprintnumber{20.1}\hspace{1em}

\pgfmathprintnumber{5.5}\hspace{1em}

\pgfmathprintnumber{1.2}\hspace{1em}

\pgfmathprintnumber{-5.6}\hspace{1em}

\pgfmathprintnumber{-1.4}\hspace{1em}
```

```
/pgf/number format/frac shift=\{\langle integer \rangle\}
```

(no default, initially 4)

In case you experience problems because of stability problems, try experimenting with a different frac shift. Higher shift values k yield higher sensitivity to inaccurate data or inaccurate arithmetics.

Technically, the following happens. If r < 1 is the fractional part of the mantissa, then a scale $i = 1/r \cdot 10^k$ is computed where k is the shift; fractional parts of i are neglected. The value 1/r is computed internally, its error is amplified.

If you still experience stability problems, use \space{fp} in your preamble. The frac style will then automatically employ the higher absolute precision of fp for the computation of 1/r.

```
/pgf/number format/precision=\{\langle number \rangle\}
```

(no default)

Sets the desired rounding precision for any display operation. For scientific format, this affects the mantissa.

```
/pgf/number format/sci precision=\langle number or empty\rangle
```

(no default, initially empty)

Sets the desired rounding precision only for sci styles.

Use sci precision={} to restore the initial configuration (which uses the argument provided to precision for all number styles).

74.1 Changing display styles

You can change the way how numbers are displayed. For example, if you use the 'fixed' style, the input number is rounded to the desired precision and the current fixed point display style is used to typeset the number. The same is applied to any other format: first, rounding routines are used to get the correct digits, afterwards a display style generates proper TeX-code.

```
/pgf/number format/set decimal separator=\{\langle text \rangle\}
```

(no default)

Assigns $\{\langle text \rangle\}$ as decimal separator for any fixed point numbers (including the mantissa in sci format). Use $\protect{\protect}$ use $\protect{\protect}$ value to get the current separator into $\protect{\protect}$ value.

```
/pgf/number format/dec sep=\{\langle text \rangle\}
```

(style, no default)

Just another name for set decimal separator.

```
/pgf/number format/set thousands separator=\{\langle text \rangle\}
```

(no default)

Assigns $\{\langle text \rangle\}$ as thousands separator for any fixed point numbers (including the mantissa in sci format).

```
1234.56 \pgfkeys{/pgf/number format/.cd, fixed, fixed zerofill, precision=2, set thousands separator={}} \pgfmathprintnumber{1234.56}
```

```
1234567890.00
```

\pgfkeys{/pgf/number format/.cd,
fixed,
fixed zerofill,
precision=2,
set thousands separator={}}
\pgfmathprintnumber{1234567890}

```
1.234.567.890.00
```

\pgfkeys{/pgf/number format/.cd,
fixed,
fixed zerofill,
precision=2,
set thousands separator={.}}
\pgfmathprintnumber{1234567890}

1,234,567,890.00

\pgfkeys{/pgf/number format/.cd,
fixed,
fixed zerofill,
precision=2,
set thousands separator={,}}
\pgfmathprintnumber{1234567890}

1,234,567,890.00

\pgfkeys{/pgf/number format/.cd,
fixed,
fixed zerofill,
precision=2,
set thousands separator={{{,}}}}
\pgfmathprintnumber{1234567890}

The last example employs commas and disables the default comma-spacing.

Use \pgfkeysgetvalue{/pgf/number format/set thousands separator}\value to get the current separator into \value.

```
/pgf/number format/1000 sep=\{\langle text \rangle\}
```

(style, no default)

Just another name for set thousands separator.

```
/pgf/number format/1000 sep in fractionals=\{\langle boolean \rangle\}
```

(no default, initially false)

Configures whether the fractional part should also be grouped into groups of three digits.

The value true will active the 1000 sep for both, integer and fractional parts. The value false will active 1000 sep only for the integer part.

```
1234.123 4567 

\text{pgfkeys{/pgf/number format/.cd,} fixed,} precision=999, set thousands separator={\,}, 1000 sep in fractionals,} \text{pgfmathprintnumber}{1234.1234567}
```

```
1234.123 456 700 \pgfkeys{/pgf/number format/.cd, fixed,fixed zerofill, precision=9, set thousands separator={\,}, 1000 sep in fractionals, } \pgfmathprintnumber{1234.1234567}
```

```
/pgf/number format/min exponent for 1000 sep=\{\langle number \rangle\}
```

(no default, initially 0)

Defines the smallest exponent in scientific notation which is required to draw thousand separators. The exponent is the number of digits minus one, so $\langle number \rangle = 4$ will use thousand separators starting with 1e4 = 10000.

```
5000; 1000000 \pgfkeys{/pgf/number format/.cd, int detect, 1000 sep={\,}, min exponent for 1000 sep=0} \pgfmathprintnumber{5000}; \pgfmathprintnumber{1000000}
```

```
1000; 5000 \pgfkeys{/pgf/number format/.cd, int detect, 1000 sep={\,}, min exponent for 1000 sep=4} \pgfmathprintnumber{1000}; \pgfmathprintnumber{5000}
```

```
10 000; 1 000 000 \pgfkeys{/pgf/number format/.cd, int detect, 1000 sep={\,}, min exponent for 1000 sep=4} \pgfmathprintnumber{100000}; \pgfmathprintnumber{1000000}
```

A value of 0 disables this feature (negative values are ignored).

/pgf/number format/use period

(no value)

A predefined style which installs periods '.' as decimal separators and commas ',' as thousands separators. This style is the default.

```
12.35 \pgfkeys{/pgf/number format/.cd,fixed,precision=2,use period} \pgfmathprintnumber{12.3456}
```

```
1,234.56 \pgfkeys{/pgf/number format/.cd,fixed,precision=2,use period} \pgfmathprintnumber{1234.56}
```

/pgf/number format/use comma

(no value)

A predefined style which installs commas ',' as decimal separators and periods '.' as thousands separators.

12,35 \pgfkeys{/pgf/number format/.cd,fixed,precision=2,use comma} \pgfmathprintnumber{12.3456}

1.234,56 \pgfkeys{/pgf/number format/.cd,fixed,precision=2,use comma} \pgfmathprintnumber{1234.56}

/pgf/number format/skip $0.=\{\langle boolean \rangle\}$

(no default, initially false)

Configures whether numbers like 0.1 shall be typeset as .1 or not.

.56 \pgfkeys{/pgf/number format/.cd, fixed, fixed zerofill,precision=2, skip 0.} \pgfmathprintnumber{0.56}

0.56
 \pgfkeys{/pgf/number format/.cd,
 fixed,
 fixed zerofill,precision=2,
 skip 0.=false}
 \pgfmathprintnumber{0.56}

/pgf/number format/showpos= $\{\langle boolean \rangle\}$

(no default, initially false)

Enables or disables the display of plus signs for non-negative numbers.

+12.35 \pgfkeys{/pgf/number format/showpos} \pgfmathprintnumber{12.345}

12.35 \pgfkeys{/pgf/number format/showpos=false} \pgfmathprintnumber{12.345}

 $+1.23 \cdot 10^{1} \quad \texttt{\pgfkeys{/pgf/number format/.cd,showpos,sci}} \\ \texttt{\pgfmathprintnumber{12.345}}$

/pgf/number format/print sign= $\{\langle boolean \rangle\}$

(style, no default)

A style which is simply an alias for showpos= $\{\langle boolean \rangle\}$.

/pgf/number format/sci 10e

(no value)

Uses $m \cdot 10^e$ for any number displayed in scientific format.

1.23.101 \pgfkeys{/pgf/number format/.cd,sci,sci 10e} \pgfmathprintnumber{12.345}

/pgf/number format/sci 10^e

(no value)

The same as 'sci 10e'.

/pgf/number format/sci e

(no value)

Uses the '1e+0' format which is generated by common scientific tools for any number displayed in scientific format.

1.23e+1 \pgfkeys{/pgf/number format/.cd,sci,sci e} \pgfmathprintnumber{12.345}

/pgf/number format/sci E

(no value)

The same with an uppercase 'E'.

```
\frac{1.23E+1}{\text{pgfmathprintnumber}\{12.345\}} \label{eq:local_constraints}
```

/pgf/number format/sci subscript

(no value)

Typesets the exponent as subscript for any number displayed in scientific format. This style requires very little space.

```
1.23<sub>1</sub> \pgfkeys{/pgf/number format/.cd,sci,sci subscript} \pgfmathprintnumber{12.345}
```

/pgf/number format/sci superscript

(no value)

Typesets the exponent as superscript for any number displayed in scientific format. This style requires very little space.

```
1.231 \pgfkeys{/pgf/number format/.cd,sci,sci superscript} \pgfmathprintnumber{12.345}
```

```
/pgf/number format/sci generic=\{\langle keys \rangle\}
```

(no default)

Allows to define an own number style for the scientific format. Here, $\langle keys \rangle$ can be one of the following choices (omit the long key prefix):

```
/pgf/number format/sci generic/mantissa sep=\{\langle text \rangle\} (no default, initially empty)
```

Provides the separator between a mantissa and the exponent. It might be \cdot, for example,

```
/pgf/number format/sci generic/exponent={\langle text\rangle}
(no default, initially empty)
```

Provides text to format the exponent. The actual exponent is available as argument #1 (see below).

```
\frac{1.23\times10^{1};1.23\times10^{-4}}{\text{pgf/number format/.cd,}} $$ \text{sci,} $$ \text{sci generic=\{mantissa sep=\times,exponent=\{10^{\#1}\}\}} $$ \text{pgfmathprintnumber\{12.345\};} $$ \text{pgfmathprintnumber\{0.00012345\}}
```

The $\langle keys \rangle$ can depend on three parameters, namely on #1 which is the exponent, #2 containing the flags entity of the floating point number and #3 is the (unprocessed and unformatted) mantissa.

Note that sci generic is *not* suitable to modify the appearance of fixed point numbers, nor can it be used to format the mantissa (which is typeset like fixed point numbers). Use dec sep, 1000 sep and print sign to customize the mantissa.

```
/pgf/number format/@dec sep mark=\{\langle text \rangle\} (no default)
```

Will be placed right before the place where a decimal separator belongs to. However, $\{\langle text \rangle\}$ will be inserted even if there is no decimal separator. It is intended as place-holder for auxiliary routines to find alignment positions.

This key should never be used to change the decimal separator! Use dec sep instead.

```
/pgf/number format/0sci exponent mark=\{\langle text \rangle\} (no default)
```

Will be placed right before exponents in scientific notation. It is intended as place-holder for auxiliary routines to find alignment positions.

This key should never be used to change the exponent!

```
/pgf/number format/assume math mode=\{\langle boolean \rangle\} (default true)
```

Set this to true if you don't want any checks for math mode. The initial setting checks whether math mode is active using \pgfutilensuremath for each final number.

Use assume math mode=true if you know that math mode is active. In that case, the final number is typeset as-is, no further checking is performed.

/pgf/number format/verbatim

(style, no value)

A style which configures the number printer to produce verbatim text output, i.e. it doesn't contain T_{EX} macros.

```
1.23e1;1.23e-4;3.27e6

\pgfkeys{
\/pgf/fpu,
\/pgf/number format/.cd,
\sci,
\verbatim}
\\pgfmathprintnumber{12.345};
\\pgfmathprintnumber{0.00012345};
\\pgfmathprintnumber{exp(15)}
\\pgfmathprintnumber{\pgfmathresult}}
```

The style resets 1000 sep, dec sep, print sign, skip 0. and sets assume math mode. Furthermore, it installs a sci generic format for verbatim output of scientific numbers.

However, it will still respect precision, fixed zerofill, sci zerofill and the overall styles fixed, sci, int detect (and their variants). It might be useful if you intend to write output files.

75 Object-Oriented Programming

This section describes the oo module.

```
\usepgfmodule{oo} % MTEX and plain TEX and pure pgf \usepgfmodule[oo] % ConTEXt and pure pgf
```

This module defines a relatively small set of TEX commands for defining classes, methods, attributes and objects in the sense of object-oriented programming.

In this chapter it is assumed that you are familiar with the basics of a typical object-oriented programming language like Java, C++ or Eiffel.

75.1 Overview

TEX does not support object-oriented programming, presumably because it was written at a time when this style of programming was not yet "en vogue." When one is used to the object-oriented style of thinking, some programming constructs in TEX often seem overly complicated. The object-oriented programming module of PGF may help here. It is written completely using simple TEX macros and is, thus, perfectly portable. This also means, however, that it is not particularly fast (but not too slow either), so you should use it only for non-time-critical things.

Basically, the oo-system supports classes (in the object-oriented sense, this has nothing to do with LATEX-classes), methods, constructors, attributes, objects, and object identities. It (currently) does not support either inheritance, overloading, destructors, or class nesting.

The first step is to define a class, using the macro \pgfooclass (all normal macros in PGF's object-oriented system start with \pgfoo). This macro gets the name of a class and in its body a number of methods are defined. These are defined using the \method macro (which is defined only inside such a class definition) and they look a bit like method definitions in, say, Java. Object attributes are declared using the \attribute command, which is also defined only inside a class definition.

Once a class has been defined, you can create objects of this class. Objects are created using \pgfoonew. Such an object has many characteristics of objects in a normal object-oriented programming language: Each object has a *unique identity*, so when you create another object, this object is completely distinct from all other objects. Each object also has a set of private attributes, which may change over time. Suppose, for instance, that we have a point class. Then creating a new object (called an instance) of this class would typically have an x-attribute and a y-attribute. These can be changed over time. Creating another instance of the point class creates another object with its own x- and y-attributes.

Given an object, you can call a method for this object. Inside the method the attributes of the object for which the method is being called can be accessed.

The life of an object always ends with the end of the TEX scope in which it was created. However, changes to attribute values are not local to scopes, so when you change an attribute anywhere, this change persists till the end of the life of the object or until the attribute is changed again.

75.2 A Running Example: The Stamp Class

As a running example we will develop a stamp class and stamp objects. The idea is that a stamp object is able to "stamp something" on a picture. This means that a stamp object has an attribute storing the "stamp text" and there is a method that asks the object to place this text somewhere on a canvas. The method can be called repeatedly and there can be several different stamp objects, each producing a different text. Stamp objects can either be created dynamically when needed or a library might define many such objects in an outer scope.

Such stamps are similar to many things present in PGF such as arrow tips, patterns, or shadings and, indeed, these could all have been implemented in this object-oriented fashion (which might have been better, but the object-oriented subsystem is a fairly new addition to PGF).

75.3 Classes

We start with the definition of the stamp class. This is done using the \pgfooclass macro:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command defines a class named $\langle class\ name \rangle$. The name of the class can contain spaces and most other characters, but no periods. So, valid class names are MyClass or my class or Class_C++_emulation??1.

The $\langle body \rangle$ is actually just executed, so any normal TEX-code is permissible here. However, while the $\langle body \rangle$ is being executed, the macros \method and \attribute are set up so that they can be used to define methods and attributes for this class (the original meanings are restored afterward).

The definition of a class is local to the scope where the class has been defined.

```
\pgfooclass{stamp}{
    % This is the class stamp

\attribute text;
\attribute rotation angle=20;

\method stamp(#1) { % The constructor
    ...
}

\method apply(#1,#2) { % Causes the stamp to be shown at coordinate (#1,#2)
    ...
}

% We can now create objects of type "stamp"
```

The $\langle body \rangle$ of a class usually just consists of calls to the macros \attribute and \method, which will be discussed in more detail in later sections.

75.4 Objects

Once a class has been declared, we can start creating objects for this class. For this the \pgfoonew command can be used, which has a peculiar syntax:

```
\protect{pgfoonew} (object\ handle\ or\ attribute) = new\ \langle class\ name \rangle (\langle constructor\ arguments \rangle)
```

Causes a new object to be created. The class of the object will be $\langle class \ name \rangle$, which must previously have been declared using \pgfooclass. Once the object has been created, the constructor method of the object will be called with the parameter list set to $\langle constructor \ arguments \rangle$.

The resulting object is stored internally and its lifetime will end exactly at the end of the current scope. Here is an example in which three stamp objects are created.

```
\pgfoonew \firststamp = new stamp()
\pgfoonew \secondstamp = new stamp()
{
  \pgfoonew \thirdstamp = new stamp()
  ...
}
% \thirdstamp no longer exists, but \firststamp and \secondstamp do
% even if you try to store \thirdstamp in a global variable, trying
% to access it will result in an error.
```

The optional $\langle object\ handle\ or\ attribute\rangle$ can either be an $\langle object\ handle\rangle$ or an $\langle attribute\rangle$. When an $\langle object\ handle\rangle$ is given, it must be a normal TEX macro name that will "point" to the object (handles are discussed in more detail in Section 75.7). You can use this macro to call methods of the object as discussed in the following section. When an $\langle attribute\rangle$ is given, it must be given in curly braces (the curly braces are used to detect the presence of an attribute). In this case, a handle to the newly created object is stored in this attribute.

```
\pgfooclass{foo}
{
  \attribute stamp obj;
  \attribute another object;

\method \foo() {
  \pgfoonew{stamp obj}=new stamp()
  \pgfoonew{another object}=new bar()
}
...
}
```

\pgfoogc

This command causes the "garbage collector" to be invoked. The job of this garbage collector is to free the global TeX-macros that are used by "dead" objects (objects whose life-time has ended). This macro is called automatically after every scope in which an object has been created, so you normally do not need to call this macro yourself.

75.5 Methods

Methods are defined inside the body of classes using the following command:

```
\mbox{\ensuremath{\mbox{\sf Method}}} (method\ name) (\mbox{\ensuremath{\mbox{\it (}}} (parameter\ list)) {\ensuremath{\mbox{\it (}}} (method\ body))}
```

This macro, which is only defined inside a class definition, defines a new method named $\langle method\ name \rangle$. Just like class names, method names can contain spaces and other characters, so $\langle method\ names \rangle$ like put_stamp_here or put stamp here are both legal.

Three method names are special: First, a method having the same name as the class is called the *constructor* of the class. There must be such a method, even if its body is empty. There are (currently) no destructors; objects simply become "undefined" at the end of the scope in which they have been created. The other two methods are called **get id** and **get handle**, which are always automatically defined and which you cannot redefine. They are discussed in Section 75.7.

Overloading of methods is not possible, that is, it is illegal to have two methods inside a single class with the same name (despite possibly different parameter lists). However, two different classes may contain a method with the same name, that is, classes form namespaces for methods.

The $\langle method\ name \rangle$ must be followed by a $\langle parameter\ list \rangle$ in parentheses, which must be present even when the $\langle parameter\ list \rangle$ is empty. The $\langle parameter\ list \rangle$ is actually a normal TeX parameter list that will be matched against the parameters inside the parentheses upon method invocation and, thus, could be something like #1#2 foo #3 bar., but a list like #1,#2,#3 is more customary. By setting the parameter list to just #1 and then calling, say, \pgfkeys{#1} at the beginning of a method, you can implement Objective-C-like named parameters.

When a method is called, the $\langle body \rangle$ of the method will be executed. The main difference to a normal macro is that while the $\langle body \rangle$ is executed, a special macro called **\pgfoothis** is set up in such a way that it references that object for which the method is executed.

In order to call a method for an object, you first need to create the object and you need a handle for this object. In order to invoke a method for this object, a special syntax is used that is similar to Java or C++ syntax:

```
\langle object\ handle \rangle. \langle method\ name \rangle (\langle parameters \rangle)
```

This causes the method $\langle method\ name \rangle$ to be called for the object referenced by the $\langle object\ handle \rangle$. Naturally, the method is the one defined in the class of the object. The $\langle parameters \rangle$ are matched against the parameters of the method and, then, the method body is executed. The execution of the method body is not done inside a scope, so the effects of a method body persist.

```
\method stamp() { % The constructor
}
\method apply(#1,#2) { % Causes the stamp to be shown at coordinate (#1,#2)
    % Draw the stamp:
    \node [rotate=20,font=\huge] at (#1,#2) {Passed};
}
}
\method apply(#1,#2) { % Causes the stamp to be shown at coordinate (#1,#2)
    % Draw the stamp:
    \node [rotate=20,font=\huge] at (#1,#2) {Passed};
}
\method [rotate=20,
```

Inside a method, you can call other methods. If you have a handle for another object, you can simply call it in the manner described above. In order to call a method of the current object, you can use the special object handle \pgfoothis.

\pgfoothis

This object handle is defined only when a method is being executed. There, it is then set to point to the object for which the method is being called, which allows you to call another method for the same object.

```
\pgfooclass{stamp}{
    % This is the class stamp

\method stamp() {}

\method apply(#1,#2) {
    \pgfoothis.shift origin(#1,#2)

    % Draw the stamp:
    \node [rotate=20,font=\huge] {Passed};
}

% Private method:
  \method shift origin(#1,#2) {
    \tikzset{xshift=#1,yshift=#2}}
}
```

75.6 Attributes

Every object has a set of attributes, which may change over time. Attributes are declared using the \attribute command, which, like the \method command, is defined only inside the scope of \pgfooclass. Attributes can be modified (only) by methods. To take the stamp example, an attribute of a stamp object might be the text that should be stamped when the apply method is called.

When an attribute is changed, this change is *not* local to the current TEX group. Changes will persist till the end of the object's life or until the attribute is changed once more.

To declare an attribute you should use the \attribute command:

```
\attribute \( attribute \ name \) = \( initial \ value \);
```

This command can only be given inside the body of an \pgfooclass command. It declares the attribute named \(\lambda attribute name \rangle \). This name, like method or class names, can be quite arbitrary, but should not contain periods. Valid names are an_attribute? or my attribute.

You can optionally specify an (*initial value*) for the attribute; if none is given, the empty string is used automatically. The initial value is the value that the attribute will have just after the object has been created and before the constructor is called.

```
\pgfooclass{stamp}{
    % This is the class stamp

\attribute text;
\attribute rotation angle = 20;

\method stamp(#1) {
    \pgfooset{text}{#1} % Set the text
}

\method apply(#1,#2) {
    \pgfoothis.shift origin(#1,#2)

    % Draw the stamp:
    \node [rotate=\pgfoovalueof{rotation angle},font=\huge]
         {\pgfoovalueof{text}};
}

\method shift origin(#1,#2) { ... }

\method set rotation (#1) {
    \pgfooset{rotation angle}{#1}
}
```

Attributes can be set and read only inside methods, it is not possible to do so using an object handle. Spoken in terms of traditional object-oriented programming, attributes are always private. You need to define getter and setter methods if you wish to read or modify attributes.

Reading and writing attributes is not done using the "dot-notation" that is used for method calls. This is mostly due to efficiency reasons. Instead, a set of special macros is used, all of which can *only be used inside methods*.

$\protect{\langle attribute \rangle} {\langle value \rangle}$

Sets the $\langle attribute \rangle$ of the current object to $\langle value \rangle$.

```
\method set rotation (#1) {
  \pgfooset{rotation angle}{#1}
}
```

$\protect{pgfoolet}{\langle attribute \rangle} {\langle macro \rangle}$

Sets the $\langle attribute \rangle$ of the current value to the current value of $\langle macro \rangle$ using TEX's \left\text{let} command.

```
\method foo () {
  \pgfoolet{my func}\myfunc
  % Changing \myfunc now has no effect on the value of attribute my func
}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Expands (eventually) to the current value of $\langle attribute \rangle$ of the current object.

```
\method apply(#1,#2) {
  \pgfoothis.shift origin(#1,#2)

\node [rotate=\pgfoovalueof{rotation angle},font=\huge]
  {\pgfoovalueof{text}};
}
```

$\proptoget{\langle attribute \rangle} {\langle macro \rangle}$

Reads the current value of $\langle attribute \rangle$ and stores the result in $\langle macro \rangle$.

```
...
\method get rotation (#1) {
   \pgfooget{rotation angle}{#1}
}
...
\mystamp.get rotation(\therotation)
''\therotation'' is now ''20'' (or whatever).
```

75.7 Identities

Every object has a unique identity, which is simply an integer. It is possible to retrieve the object id using the get id method (discussed below), but normally you will not need to do so because the id itself cannot be used to access an object. Rather, you access objects via their methods and these, in turn, can only be called via object handles.

Object handles can be created in four ways:

- 1. Calling \pgfoonew(object handle)=... will cause (object handle) to be a handle to the newly created object.
- 2. Using \let to create an alias of an existing object handle: If \mystamp is a handle, saying \let\myotherstamp=\mystamp creates a second handle to the same object.
- 3. \pgfooobj $\{\langle id \rangle\}$ can be used as an object handle to the object with the given $\langle id \rangle$.
- 4. Using the get handle method to create a handle to a given object.

Let us have a look at the last two methods.

$\protect\pro$

Provided that $\langle id \rangle$ is the id of an existing object (an object whose life-time has not expired), calling this command yields a handle to this object. The handle can then be used to call methods:

```
% Create a new object:
\pgfoonew \mystamp=new stamp()

% Get the object's id and store it in \myid:
\mystamp.get id(\myid)

% The following two calls have the same effect:
\mystamp.apply(1,1)
\pgfooobj{\myid}.apply(1,1)
```

The get id method can be used to retrieve the id of an object. This method is predefined for every class and you should not try to define a method of this name yourself.

```
Method get id(\langle macro \rangle)
```

(predefined for all classes)

Calling $\langle obj \rangle$ get $id(\langle macro \rangle)$ stores the $id\langle obj \rangle$ in $\langle macro \rangle$. This is mainly useful when you wish to store an object for a longer time and you cannot guarantee that any handle that you happen to have for this object will be available later on.

The only way to use the retrieved id later on is to call \pgfooobj.

Different object that are alive (that are still within the scope in which they were created) will always have different ids, so you can use the id to test for equality of objects. However, after an object has been destroyed because its scope has ended, the same id may be used again for newly created objects.

Here is a typical application where you need to call this method: You wish to collect a list of objects for which you wish to call a specific method from time to time. For the collection process you wish to offer a macro called \addtoobjectlist, which takes an object handle as parameter. It is quite easy to store this handle somewhere, but a handle is, well, just a handle. Typically, shortly after the call to \addtoobjectlist the handle will no longer be valid or even exist, even though the object still exists. In this case, you wish to store the object id somewhere instead of the handle. Thus, for the object passed to \addtoobjectlist you call the get id method and store the resulting id, rather than the handle.

There is a second predefined method, called get handle, which is also used to create object handles.

```
Method get handle(\{\langle macro\ name \rangle\})
```

(predefined for all classes)

Calling this method for an object will cause $\langle macro\ name \rangle$ to become a handle to the given object. For any object handle \o - other than \p -pfoothis - the following two have the same effect:

- 1. $\langle macro\ name \rangle = \langle obj \rangle$
- 2. \obj.get handle($\langle macro\ name \rangle$)

The first method is simpler and faster. However, for $\protect\protect\operatorname{handle}(\langle macro\ name \rangle)$ will cause $\langle macro\ name \rangle$ to be an object handle to the current object and will persist to be so even after the method is done. By comparison, $\protect\protect\operatorname{handle}(\langle macro\ name \rangle) = \protect\pr$

75.8 The Signal Class

The object-oriented module predefines, in addition to the basic mechanism for defining and using classes and object, one class: signal. This class implements a so-called signal—slot mechanism.

Class signal

This class is used to implement a simple signal—slot mechanism. The idea is the following: From time to time special things happen about which a number of objects need to be informed. Different things can happen and different object will be interested in these things. A signal object can be used to signal that such special things of a certain kind have happened. For example, one signal object might be used to signal the event that "a page has been shipped out." Another signal might be used to signal that "a figure is about to be typeset," and so on.

Objects can "tune in" to signals. They do so by *connecting* one of their methods (then called a *slot*) to the signal. Then, whenever the signal is *emitted*, the method of the connected object(s) get called. Different objects can connect different slots to the same signal as long as the argument lists will fit. For example, the object that is used to signal the "end of page has been reached" might emit signals that have, say, the box number in which the finished page can be found as a parameter (actually, the finished page is always in box 255). Then one object could connect a method handle page(#1) to this signal, another might connect the method emergency action(#1) to this signal, and so on.

Currently, it is not possible to "unregister" or "detach" a slot from a signal, that is, once an object has been connect to a signal, it will continue to receive emissions of this signal till the end of the life-time of the signal. This is even true when the object no longer exists (but the signal does), so care must be taken that signal objects are always created before the objects that are listening to them.

Constructor signal()

The constructor does nothing.

```
Method connect (\langle object\ handle \rangle, \langle method\ name \rangle)
```

This method gets an $\langle object\ handle \rangle$ as parameter and a $\langle method\ name \rangle$ of this object. It will queue the object-method pair in an internal list and each time the signal emits something, this object's method is called.

Be careful not to pass \pgfoothis as \(object handle \). This would cause the signal object to connect to itself. Rather, if you wish to connect a signal to a method of the current object you first need to create an alias using the get handle method:

```
\pgfooclass{some class}{
  \method some class() {
    \pgfoothis.get handle(\me)
    \somesignal.connect(\me,foo)
    \anothersignal.connect(\me,bar)
}
  \method foo () {}
  \method bar (#1,#2) {}
}
\pgfoonew \objA=new some class()
\pgfoonew \objB=new some class()
```

Method emit($\langle arguments \rangle$)

This method emits a signal to all connected slots. This means that for all objects that have previously been connected via a call of connect, the method (slot) that was specified during the call of connect is invoked with given \(\langle arguments \rangle \).

```
\anothersignal.emit(1,2)
% will call \objA.bar(1,2) and \objB.bar(1,2)
```

75.9 Implementation Notes

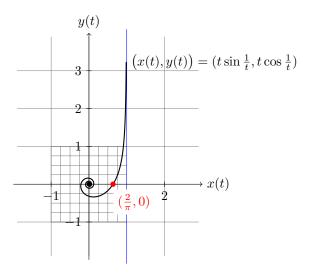
For the curious, here are some notes on how the oo-system is implemented:

- There is an object id counter that gets incremented each time an object is created. However, this counter is local to the current scope, which means that it is reset at the end of each scope, corresponding to the fact that at the end of a scope all objects created in this scope become invalid. Newly created objects will then have the same id as "deleted" objects.
- Attributes are stored globally. For each attribute of each object there is a macro whose name is composed of the object's id and the attribute name. Changes to object attributes are always global.
- A call to the garbage collector causes a loop to be executed that tries to find objects whose object number is larger than the current maximum alive objects. The global attributes of these objects are then freed (set to \relax) by calling a special internal method of these (dead) objects.
 - The garbage collector is automatically called after each group in which an object was created using \aftergroup.
- When a method is called, before the method call some code is executed that sets a global counter storing the current object id to the object id of the object being called. After the method call some code is inserted that restores the global counter to its original value. This is done without scopes, so some tricky \expandafter magic is needed. Note that, because of this process, you cannot use commands like \pgfutil@ifnextchar at the end of a method.
- An object handle contains just the code to set up and restore the current object number to the number of the object being called.

Part IX

The Basic Layer

by Till Tantau



```
\begin{tikzpicture}
 \draw[gray,very thin] (-1.9,-1.9) grid (2.9,3.9)
 [step=0.25cm] (-1,-1) grid (1,1);
\draw[blue] (1,-2.1) -- (1,4.1); % asymptote
 foreach pos in {-1,2}
   \draw[shift={(\pos,0)}] (Opt,2pt) -- (Opt,-2pt) node[below] {$\pos$};
 foreach pos in {-1,1,2,3}
   \label{left} $$ \displaystyle \int_{0,\infty} (2pt,0pt) -- (-2pt,0pt) \ node[left] {\scriptstyle s\ };
 \fill (0,0) circle (0.064cm);
 \draw[thick,parametric,domain=0.4:1.5,samples=200]
   \mbox{\%} The plot is reparameterised such that there are more samples
   \mbox{\ensuremath{\mbox{\%}}} near the center.
   \verb|plot[id=asymptotic-example]| function{(t*t*t)*sin(1/(t*t*t)),(t*t*t)*cos(1/(t*t*t))}|
   \fill[red] (0.63662,0) circle (2pt)
   node \ [below \ right,fill=white,yshift=-4pt] \ \{\$(\frac{2}{\pi},0)\$\};
\end{tikzpicture}
```

76 Design Principles

This section describes the basic layer of PGF. This layer is built on top of the system layer. Whereas the system layer just provides the absolute minimum for drawing graphics, the basic layer provides numerous commands that make it possible to create sophisticated graphics easily and also quickly.

The basic layer does not provide a convenient syntax for describing graphics, which is left to frontends like TikZ. For this reason, the basic layer is typically used only by "other programs." For example, the BEAMER package uses the basic layer extensively, but does not need a convenient input syntax. Rather, speed and flexibility are needed when BEAMER creates graphics.

The following basic design principles underlie the basic layer:

- 1. Structuring into a core and modules.
- 2. Consistently named T_FX macros for all graphics commands.
- 3. Path-centered description of graphics.
- 4. Coordinate transformation system.

76.1 Core and Modules

The basic layer consists of a *core package*, called pgfcore, which provides the most basic commands, and several *modules* like commands for plotting (in the plot module). Modules are loaded using the \usepgfmodule command

If you say \usepackage{pgf} or \input pgf.tex or \usemodule[pgf], the plot and shapes modules are preloaded (as well as the core and the system layer).

76.2 Communicating with the Basic Layer via Macros

In order to "communicate" with the basic layer you use long sequences of commands that start with \pgf. You are only allowed to give these commands inside a {pgfpicture} environment. (Note that {tikzpicture} opens a {pgfpicture} internally, so you can freely mix PGF commands and TikZ commands inside a {tikzpicture}.) It is possible to "do other things" between the commands. For example, you might use one command to move to a certain point, then have a complicated computation of the next point, and then move there.



```
\newdimen\myypos
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{\myypos}}
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \advance \myypos by 1cm
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}
```

The following naming conventions are used in the basic layer:

- 1. All commands and environments start with pgf.
- 2. All commands that specify a point (a coordinate) start with \pgfpoint.
- 3. All commands that extend the current path start with \pgfpath.
- 4. All commands that set/change a graphics parameter start with \pgfset.
- 5. All commands that use a previously declared object (like a path, image or shading) start with \pgfuse.
- 6. All commands having to do with coordinate transformations start with \pgftransform.
- 7. All commands having to do with arrow tips start with \pgfarrows.
- 8. All commands for "quickly" extending or drawing a path start with \pgfpathq or \pgfusepathq.
- 9. All commands having to do with matrices start with \pgfmatrix.

76.3 Path-Centered Approach

In PGF the most important entity is the *path*. All graphics are composed of numerous paths that can be stroked, filled, shaded, or clipped against. Paths can be closed or open, they can self-intersect and consist of unconnected parts.

Paths are first *constructed* and then *used*. In order to construct a path, you can use commands starting with \pgfpath. Each time such a command is called, the current path is extended in some way.

Once a path has been completely constructed, you can use it using the command \pgfusepath. Depending on the parameters given to this command, the path will be stroked (drawn) or filled or subsequent drawings will be clipped against this path.

76.4 Coordinate Versus Canvas Transformations

PGF provides two transformation systems: PGF's own *coordinate* transformation matrix and PDF's or PostScript's *canvas* transformation matrix. These two systems are quite different. Whereas a scaling by a factor of, say, 2 of the canvas causes *everything* to be scaled by this factor (including the thickness of lines and text), a scaling of two in the coordinate system causes only the *coordinates* to be scaled, but not the line width nor text.

By default, all transformations only apply to the coordinate transformation system. However, using the command \pgflowlevel it is possible to apply a transformation to the canvas.

Coordinate transformations are often preferable over canvas transformations. Text and lines that are transformed using canvas transformations suffer from differing sizes and lines whose thickness differs depending on whether the line is horizontal or vertical. To appreciate the difference, consider the following two "circles" both of which have been scaled in the x-direction by a factor of 3 and by a factor of 0.5 in the y-direction. The left circle uses a canvas transformation, the right uses PGF's coordinate transformation (some viewers will render the left graphic incorrectly since they do no apply the low-level transformation the way they should):





77 Hierarchical Structures: Package, Environments, Scopes, and Text

77.1 Overview

PGF uses two kinds of hierarchical structuring: First, the package itself is structured hierarchically, consisting of different packages that are built on top of each other. Second, PGF allows you to structure your graphics hierarchically using environments and scopes.

77.1.1 The Hierarchical Structure of the Package

The PGF system consists of several layers:

System layer. The lowest layer is called the *system layer*, though it might also be called "driver layer" or perhaps "backend layer." Its job is to provide an abstraction of the details of which driver is used to transform the .dvi file. The system layer is implemented by the package pgfsys, which will load appropriate driver files as needed.

The system layer is documented in Part X.

Basic layer. The basic layer is loaded by the package pgfcore and subsequent use of the command \usepffmodule to load additional modules of the basic layer.

The basic layer is documented in the present part.

Frontend layer. The frontend layer is not loaded by a single package. Rather, different packages, like TikZ or PGFPICT2E, are different frontends to the basic layer.

The TikZ frontend is documented in Part III.

Each layer will automatically load the necessary files of the layers below it.

In addition to the packages of these layers, there are also some library packages. These packages provide additional definitions of things like new arrow tips or new plot handlers.

The library packages are documented in Part V.

77.1.2 The Hierarchical Structure of Graphics

Graphics in PGF are typically structured hierarchically. Hierarchical structuring can be used to identify groups of graphical elements that are to be treated "in the same way." For example, you might group together a number of paths, all of which are to be drawn in red. Then, when you decide later on that you like them to be drawn in, say, blue, all you have to do is to change the color once.

The general mechanism underlying hierarchical structuring is known as *scoping* in computer science. The idea is that all changes to the general "state" of the graphic that are done inside a scope are local to that scope. So, if you change the color inside a scope, this does not affect the color used outside the scope. Likewise, when you change the line width in a scope, the line width outside is not changed, and so on.

There are different ways of starting and ending scopes of graphic parameters. Unfortunately, these scopes are sometimes "in conflict" with each other and it is sometimes not immediately clear which scopes apply. In essence, the following scoping mechanisms are available:

1. The "outermost" scope supported by PGF is the {pgfpicture} environment. All changes to the graphic state done inside a {pgfpicture} are local to that picture.

In general, it is *not* possible to set graphic parameters globally outside any {pgfpicture} environments. Thus, you can *not* say \pgfsetlinewidth{1pt} at the beginning of your document to have a default line width of one point. Rather, you have to (re)set all graphic parameters inside each {pgfpicture}. (If this is too bothersome, try defining some macro that does the job for you.)

2. Inside a {pgfpicture} you can use a {pgfscope} environment to keep changes of the graphic state local to that environment.

The effect of commands that change the graphic state are local to the current {pgfscope}, but not always to the current TeX group. Thus, if you open a TeX group (some text in curly braces) inside a {pgfscope}, and if you change, for example, the dash pattern, the effect of this changed dash pattern will persist till the end of the {pgfscope}.

Unfortunately, this is not always the case. *Some* graphic parameters only persist till the end of the current TEX group. For example, when you use \pgfsetarrows to set the arrow tip inside a TEX group, the effect lasts only till the end of the current TEX group.

- 3. Some graphic parameters are not scoped by {pgfscope} but "already" by TEX groups. For example, the effect of coordinate transformation commands is always local to the current TEX group.
 - Since every {pgfscope} automatically creates a TeX group, all graphic parameters that are local to the current TeX group are also local to the current {pgfscope}.
- 4. Some graphic parameters can only be scoped using TEX groups, since in some situations it is not possible to introduce a {pgfscope}. For example, a path always has to be completely constructed and used in the same {pgfscope}. However, we might wish to have different coordinate transformations apply to different points on the path. In this case, we can use TEX groups to keep the effect local, but we could not use {pgfscope}.
- 5. The \pgftext command can be used to create a scope in which TEX "escapes back" to normal TEX mode. The text passed to the \pgftext is "heavily guarded" against having any effect on the scope in which it is used. For example, it is possible to use another {pgfpicture} environment inside the argument of \pgftext.

Most of the complications can be avoided if you stick to the following rules:

- Give graphic commands only inside {pgfpicture} environments.
- Use {pgfscope} to structure graphics.
- Do not use T_EX groups inside graphics, *except* for keeping the effect of coordinate transformations local.

77.2 The Hierarchical Structure of the Package

Before we come to the structuring commands provided by PGF to structure your graphics, let us first have a look at the structure of the package itself.

77.2.1 The Core Package

To use PGF, include the following package:

```
\usepackage{pgfcore} % MEX
\input pgfcore.tex % plain TeX
\usemodule[pgfcore] % ConTeXt
```

This package loads the complete core of the "basic layer" of PGF, but not any modules. That is, it will load all of the commands described in the current part of this manual, but it will not load frontends like TikZ. It will also load the system layer. To load additional modules, use the \usepfmodule command explained below.

The following package is just a convenience.

```
\usepackage{pgf} % MEX
\input pgf.tex % plain TEX
\usemodule[pgf] % ConTEXt
```

This package loads the pgfcore and the two modules shapes and plot.

In LATEX, the package takes two options:

```
\usepackage[draft]{pgf}
```

When this option is set, all images will be replaced by empty rectangles. This can speedup compilation.

```
\usepackage[version=\(version\)] {pgf}
```

Indicates that the commands of version $\langle version \rangle$ need to be defined. If you set $\langle version \rangle$ to 0.65, then a large bunch of "compatibility commands" are loaded. If you set $\langle version \rangle$ to 0.96, then these compatibility commands will not be loaded.

If this option is not given at all, then the commands of all versions are defined.

77.2.2 The Modules

$\usepgflibrary{\langle module \ names \rangle}$

Once the core has been loaded, you can use this command to load further modules. The modules in the $\langle module\ names \rangle$ list should be separated by commas. Instead of curly braces, you can also use square brackets, which is something ConTEXt users will like. If you try to load a module a second time, nothing will happen.

Example: \usepgfmodule{matrix,shapes}

What this command does is to load the file $pgfmodule \ module \ code.tex$ for each $\ module \ in$ the list of $\ module \ names \$. Thus, to write your own module, all you need to do is to place a file of the appropriate name somewhere T_EX can find it. LaTeX, plain T_EX , and $ConT_EXt$ users can then use your library.

The following modules are available for use with pgfcore:

- The plot module provides commands for plotting functions. The commands are explained in Section 89.
- The shapes module provides commands for drawing shapes and nodes. These commands are explained in Section 83.
- The decorations module provides commands for adding decorations to paths. These commands are explained in Section 80.
- The matrix module provides the \pgfmatrix command. The commands are documented in Section 84.

77.2.3 The Library Packages

There is a special command for loading library packages. The difference between a library and module is the following: A library just defines additional objects using the basic layer, whereas a module adds completely new functionality. For instance, a decoration library defines additional decorations, while a decoration module defines the whole code for handling decorations.

$\usepgflibrary{\langle list\ of\ libraries\rangle}$

Use this command to load further libraries. The list of libraries should contain the names of libraries separated by commas. Instead of curly braces, you can also use square brackets. If you try to load a library a second time, nothing will happen.

Example: \usepgflibrary{arrows}

This command causes the file pgflibrary $\langle library \rangle$.code.tex to be loaded for each $\langle library \rangle$ in the $\langle list$ of $libraries \rangle$. This means that in order to write your own library file, place a file of the appropriate name somewhere where TeX can find it. LaTeX, plain TeX, and ConTeXt users can then use your library.

You should also consider adding a TikZ library that simply includes your PGF library.

77.3 The Hierarchical Structure of the Graphics

77.3.1 The Main Environment

Most, but not all, commands of the PGF package must be given within a {pgfpicture} environment. The only commands that (must) be given outside are commands having to do with including images (like \pgfuseimage) and with inserting complete shadings (like \pgfuseshading). However, just to keep life entertaining, the \pgfshadepath command must be given *inside* a {pgfpicture} environment.

```
\begin{pgfpicture}
      ⟨environment contents⟩
\end{pgfpicture}
```

This environment will insert a TeX box containing the graphic drawn by the $\langle environment \ contents \rangle$ at the current position.

The size of the bounding box. The size of the box is determined in the following manner: While PGF parses the $\langle environment \ contents \rangle$, it keeps track of a bounding box for the graphic. Essentially,

this bounding box is the smallest box that contains all coordinates mentioned in the graphics. Some coordinates may be "mentioned" by PGF itself; for example, when you add circle to the current path, the support points of the curve making up the circle are also "mentioned" despite the fact that you will not "see" them in your code.

Once the $\langle environment\ contents \rangle$ have been parsed completely, a TeX box is created whose size is the size of the computed bounding box and this box is inserted at the current position.

Sometimes, you may need more fine-grained control over the size of the bounding box. For example, the computed bounding box may be too large or you intensionally wish the box to be "too small." In these cases, you can use the command \pgfusepath{use as bounding box}, as described in Section 81.5.

The baseline of the bounding box. When the box containing the graphic is inserted into the normal text, the baseline of the graphic is normally at the bottom of the graphic. For this reason, the following two sets of code lines have the same effect, despite the fact that the second graphic uses "higher" coordinates than the first:

```
Rectangles \begin{pgfpicture} \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}} \pgfusepath{stroke} \end{pgfpicture} and \begin{pgfpicture} \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}} \pgfusepath{stroke} \end{pgfpicture}.
```

You can change the baseline using the \pgfsetbaseline command, see below.

```
Rectangles \begin{pgfpicture}
\pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
\pgfusepath{stroke}
\pgfsetbaseline{0pt}
\end{pgfpicture} and \begin{pgfpicture}
\pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
\pgfusepath{stroke}
\pgfsetbaseline{0pt}
\end{pgfpicture}.
```

Including text and images in a picture. You cannot directly include text and images in a picture. Thus, you should *not* simply write some text in a {pgfpicture} or use a command like \includegraphics or even \pgfimage. In all these cases, you need to place the text inside a \pgftext command. This will "escape back" to normal TEX mode, see Section 77.3.3 for details.

Remembering a picture position for later reference. After a picture has been typeset, its position on the page is normally forgotten by PGF and also by TEX. This means that is not possible to reference a node in this picture later on. In particular, it is normally impossible to draw lines between nodes in different pictures automatically.

In order to make PGF "remember" a picture, the TEX-if \ifpgfrememberpicturepositiononpage should be set to true. It is only important that this TEX-if is true at the end of the {pgfpicture}-environment, so you can switch it on inside the environment. However, you can also just switch it on globally, then the positions of all pictures are remembered.

There are several reasons why the remembering is not switched on by default. First, it does not work for all backend drivers (currently, it works only for pdfTeX). Second, it requires two passes of TeX over the file; on the first pass all positions will be wrong. Third, for every remembered picture a line is added to the .aux-file, which may result in a large number of extra lines.

Despite all these "problems," for documents that are processed with pdfTEX and in which there is only a small number of pictures (less than a hundred or so), you can switch on this option globally, it will not cause any significant slowing of TEX.

```
\langle environment \ contents \rangle
```

\endpgfpicture

The plain T_EX version of the environment. Note that in this version, also, a T_EX group is created around the environment.

\startpgfpicture

 $\langle environment\ contents \rangle$

\stoppgfpicture

This is the ConT_EXt version of the environment.

\ifpgfrememberpicturepositiononpage

Determines whether the position of pictures on the page should be recorded. The value of this TEX-if at the end of a {pgfpicture} environment is important, not the value at the beginning.

If this option is set to true of a picture, PGF will attempt to record the position of the picture on the page. (This attempt will fail with most drivers and when it works, it typically requires two runs of TeX.) The position is not directly accessible. Rather, the nodes mechanism will use this position if you access a node from another picture. See Sections 83.3.2 and 16.13 for more details.

$\protect\pro$

This command specifies a y-coordinate of the picture that should be used as the baseline of the whole picture. When a PGF picture has been typeset completely, PGF must decide at which height the baseline of the picture should lie. Normally, the baseline is set to the y-coordinate of the bottom of the picture, but it is often desirable to use a different height.

```
Text \bigcirc, \bigcirc, \bigcirc, \bigcirc.
```

```
Text \tikz{\pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
    \tikz{\pgfsetbaseline{0pt}
    \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
    \tikz{\pgfsetbaseline{.5ex}
    \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
    \tikz{\pgfsetbaseline{-1ex}
    \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}}.
```

$\protect\pro$

This command specifies the baseline indirectly, namely as the y-coordinate that the given $\langle point \rangle$ has when the command is called.

$\protect\pro$

This command also specifies the baseline indirectly, but the y-coordinate of the given $\langle point \rangle$ is only computed at the end of the picture.

```
Hello world.
```

```
Hello
\tikz{
  \pgfsetbaselinepointlater{\pgfpointanchor{X}{base}}
  % Note: no shape X, yet
  \node [cross out,draw] (X) {world.};
}
```

77.3.2 Graphic Scope Environments

Inside a {pgfpicture} environment you can substructure your picture using the following environment:

```
\begin{pgfscope}
     ⟨environment contents⟩
\end{pgfscope}
```

nd (bgrpcobc)

All changes to the graphic state done inside this environment are local to the environment. The graphic state includes the following:

- The line width.
- The stroke and fill colors.
- The dash pattern.

- The line join and cap.
- The miter limit.
- The canvas transformation matrix.
- The clipping path.

Other parameters may also influence how graphics are rendered, but they are *not* part of the graphic state. For example, the arrow tip kind is not part of the graphic state and the effect of commands setting the arrow tip kind are local to the current TEX group, not to the current {pgfscope}. However, since {pgfscope} starts and ends a TEX group automatically, a {pgfscope} can be used to limit the effect of, say, commands that set the arrow tip kind.

```
\begin{pgfpicture}
\begin{pgfscope}
{
    \pgfsetlinewidth{2pt}
    \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{2ex}}
    \pgfusepath{stroke}
}
\pgfpathrectangle{\pgfpoint{3ex}{0ex}}{\pgfpoint{2ex}{2ex}}
\pgfusepath{stroke}
\end{pgfscope}
\pgfpathrectangle{\pgfpoint{6ex}{0ex}}{\pgfpoint{2ex}{2ex}}
\pgfusepath{stroke}
\end{pgfscope}
\pgfusepath{stroke}
\end{pgfpicture}
```

At the start of the scope, the current path must be empty, that is, you cannot open a scope while constructing a path.

It is usually a good idea not to introduce TeX groups inside a {pgfscope} environment.

```
\pgfscope
\( \langle environment contents \rangle \)
\endpgfscope
```

Plain TEX version of the {pgfscope} environment.

```
\startpgfscope
\langle environment contents\rangle
\stoppgfscope
```

This is the ConTeXt version of the environment.

The following scopes also encapsulate certain properties of the graphic state. However, they are typically not used directly by the user.

```
\begin{pgfinterruptpath}
     ⟨environment contents⟩
\end{pgfinterruptpath}
```

This environment can be used to temporarily interrupt the construction of the current path. The effect will be that the path currently under construction will be "stored away" and restored at the end of the environment. Inside the environment you can construct a new path and do something with it.

An example application of this environment is the arrow tip caching. Suppose you ask PGF to use a specific arrow tip kind. When the arrow tip needs to be rendered for the first time, PGF will "cache" the path that makes up the arrow tip. To do so, it interrupts the current path construction and then protocols the path of the arrow tip. The {pgfinterruptpath} environment is used to ensure that this does not interfere with the path to which the arrow tips should be attached.

This command does *not* install a {pgfscope}. In particular, it does not call any \pgfsys@ commands at all, which would, indeed, be dangerous in the middle of a path construction.

```
\pgfinterruptpath \( \langle environment contents \rangle \)
```

\endpgfinterruptpath

Plain TEX version of the environment.

\startpgfinterruptpath

 $\langle environment\ contents \rangle$

\stoppgfinterruptpath

ConTEXt version of the environment.

\begin{pgfinterruptpicture}

 $\langle environment\ contents \rangle$

\end{pgfinterruptpicture}

This environment can be used to temporarily interrupt a {pgfpicture}. However, the environment is intended only to be used at the beginning and end of a box that is (later) inserted into a {pgfpicture} using \pgfqbox. You cannot use this environment directly inside a {pgfpicture}.

```
Sub--picture.
```

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}} % In the middle of path, now
\newbox\mybox
\setbox\mybox=\hbox{
  \begin{pgfinterruptpicture}
    Sub-\begin{pgfpicture} % a subpicture
        \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
        \pgfpathlineto{\pgfpoint{1cm}{1cm}}
        \pgfusepath{stroke}
        \end{pgfpicture}-picture.
  \end{pgfinterruptpicture}
}
\pgfqbox{\mybox}%
\pgfpathlineto{\pgfpoint{0cm}{1cm}}
\pgfusepath{stroke}
\end{pgfpicture}-picture.
```

\pgfinterruptpicture

 $\langle environment\ contents \rangle$

\endpgfinterruptpicture

Plain T_FX version of the environment.

\startpgfinterruptpicture

 $\langle environment\ contents \rangle$

\stoppgfinterruptpicture

ConT_EXt version of the environment.

\begin{pgfinterruptboundingbox}

 $\langle environment\ contents \rangle$

\end{pgfinterruptboundingbox}

This environment temporarily interrupts the computation of the bounding box and sets up a new bounding box. At the beginning of the environment the old bounding box is saved and an empty bounding box is installed. After the environment the original bounding box is reinstalled as if nothing has happened.

\pgfinterruptboundingbox

```
\langle environment\ contents \rangle
```

\endpgfinterruptboundingbox

Plain T_EX version of the environment.

\startpgfinterruptboundingbox

 $\langle environment\ contents \rangle$

\stoppgfinterruptboundingbox

ConTFXt version of the environment.

77.3.3 Inserting Text and Images

Often, you may wish to add normal TEX text at a certain point inside a {pgfpicture}. You cannot do so "directly," that is, you cannot simply write this text inside the {pgfpicture} environment. Rather, you must pass the text as an argument to the \pgftext command.

You must also use the \pgftext command to insert an image or a shading into a {pgfpicture}.

$\protect[\langle options \rangle] \{\langle text \rangle\}$

This command will typeset $\langle text \rangle$ in normal TeX mode and insert the resulting box into the {pgfpicture}. The bounding box of the graphic will be updated so that all of the text box is inside. By default, the text box is centered at the origin, but this can be changed either by giving appropriate $\langle options \rangle$ or by applying an appropriate coordinate transformation beforehand.

The $\langle text \rangle$ may contain verbatim text. (In other words, the $\langle text \rangle$ "argument" is not a normal argument, but is put in a box and some **\aftergroup** hackery is used to find the end of the box.)

PGF's current (high-level) coordinate transformation is synchronized with the canvas transformation matrix temporarily when the text box is inserted. The effect is that if there is currently a high-level rotation of, say, 30 degrees, the $\langle text \rangle$ will also be rotated by thirty degrees. If you do not want this effect, you have to (possibly temporarily) reset the high-level transformation matrix.

The (options) keys are used with the path /pgf/text/. The following keys are defined for this path:

/pgf/text/left (no value)

The key causes the text box to be placed such that its left border is on the origin.

```
lovely \tikz{\draw[help lines] (-1,-.5) grid (1,.5); \pgftext[left] {lovely}}
```

/pgf/text/right (no value)

The key causes the text box to be placed such that its right border is on the origin.

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5); \pgftext[right] {lovely}}
```

/pgf/text/right (no value)

This key causes the text box to be placed such that its top is on the origin. This option can be used together with the left or right option.

/pgf/text/bottom (no value)

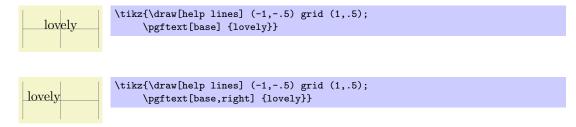
This key causes the text box to be placed such that its bottom is on the origin.

```
lovely \tikz{\draw[help lines] (-1,-.5) grid (1,.5); \pgftext[bottom] {lovely}}

\tikz{\draw[help lines] (-1,-.5) grid (1,.5); \pgftext[bottom,right] {lovely}}
```

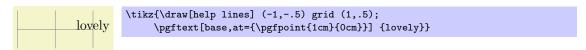
/pgf/text/base (no value)

This key causes the text box to be placed such that its baseline is on the origin.



 $/pgf/text/at = \langle point \rangle$ (no default)

Translates the origin (that is, the point where the text is shown) to (point).



 $/pgf/text/x = \langle dimension \rangle$ (no default)

Translates the origin by $\langle dimension \rangle$ along the x-axis.

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
\pgftext[base,x=1cm,y=-0.5cm] {lovely}}
```

 $pgf/text/y = \langle dimension \rangle$ (no default)

This key works like the x option.

```
/pgf/text/rotate = \langle degree \rangle (no default)
```

Rotates the coordinate system by $\langle degree \rangle$. This will also rotate the text box.

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
\pgftext[base,x=1cm,y=-0.5cm,rotate=30] {lovely}}
```

78 Specifying Coordinates

78.1 Overview

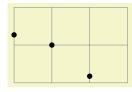
Most PGF commands expect you to provide the coordinates of a *point* (also called *coordinate*) inside your picture. Points are always "local" to your picture, that is, they never refer to an absolute position on the page, but to a position inside the current {pgfpicture} environment. To specify a coordinate you can use commands that start with \pgfpoint.

78.2 Basic Coordinate Commands

The following commands are the most basic for specifying a coordinate.

 $\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Yields a point location. The coordinates are given as TEX dimensions.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpoint{1cm}{1cm}} {2pt}
  \pgfpathcircle{\pgfpoint{2cm}{5pt}} {2pt}
  \pgfpathcircle{\pgfpoint{0pt}{.5in}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

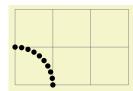
\pgfpointorigin

Yields the origin. Same as \pgfpoint{0pt}{0pt}.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

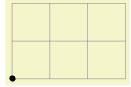
Yields a point location given in polar coordinates. You can specify the angle only in degrees, radians are not supported, currently.

If the optional $\langle y\text{-}radius \rangle$ is given, the polar coordinate is actually a coordinate on an ellipse whose x-radius is given by $\langle radius \rangle$ and whose y-radius is given by $\langle y\text{-}radius \rangle$.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm/2cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```

78.3 Coordinates in the XY-Coordinate System

Coordinates can also be specified as multiples of an x-vector and a y-vector. Normally, the x-vector points one centimeter in the x-direction and the y-vector points one centimeter in the y-direction, but using the commands pgfsetxvec and pgfsetyvec they can be changed. Note that the x- and y-vector do not necessarily point "horizontally" and "vertically."

```
\pgfpointxy\{\langle s_x \rangle\}\{\langle s_y \rangle\}
```

Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector.

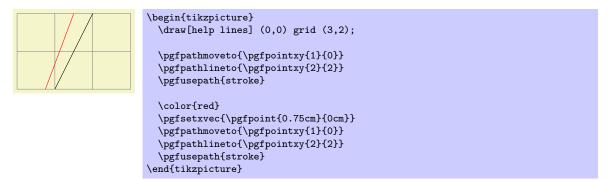


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
  \end{tikzpicture}
```

$\protect\operatorname{\protect}(point)$

Sets that current x-vector for usage in the xyz-coordinate system.

Example:



$\protect\pro$

Works like \pgfsetxvec.

This command is similar to the $\protect\operatorname{pgfpointpolar}$ command, but the $\langle radius \rangle$ is now a factor to be interpreted in the xy-coordinate system. This means that a degree of 0 is the same as the x-vector of the xy-coordinate system times $\langle radius \rangle$ and a degree of 90 is the y-vector times $\langle radius \rangle$. As for $\protect\operatorname{pgfpointpolar}$, a $\langle radius \rangle$ can also be a pair separated by a slash. In this case, the x- and y-vectors are multiplied by different factors.

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\begin{scope}[x={(1cm,-5mm)},y=1.5cm]
  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolarxy{\angle}{1}}{2pt}}
  \pgfusepath{fill}
  \end{scope}
  \end{tikzpicture}
```

78.4 Three Dimensional Coordinates

It is also possible to specify a point as a multiple of three vectors, the x-, y-, and z-vector. This is useful for creating simple three dimensional graphics.

```
\protect\operatorname{\mathsf{Npgfpointxyz}}\{\langle s_x \rangle\}\{\langle s_y \rangle\}\{\langle s_z \rangle\}
```

Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector plus s_z times the z-vector.

```
\begin{pgfpicture}
  \pgfsetarrowsend{to}

\pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{0}{0}{1}}

\pgfusepath{stroke}
  \pgfpathlineto{\pgfpointxyz{0}{1}{0}}

\pgfusepathlineto{\pgfpointxyz{0}{1}{0}}

\pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin}

\pgfpathlineto{\pgfpointorigin}

\pgfpathlineto{\pgfpointxyz{1}{0}{0}}

\pgfusepath{stroke}

\end{pgfpicture}
```

$\protect\pro$

Works like \pgfsetxvec.

Inside the xyz-coordinate system, you can also specify points using spherical and cylindrical coordinates.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command yields the same as

$\protect\pro$

This command yields a point "on the surface of the earth" specified by the $\langle longitude \rangle$ and the $\langle latitude \rangle$. The radius of the earth is given by $\langle radius \rangle$. The equator lies in the xy-plane.



```
\begin{tikzpicture}
  \pgfsetfillcolor{lightgray}

\foreach \latitude in {-90,-75,...,30} {
      \foreach \longitude in {0,20,...,360} {
            \pgfpathmoveto{\pgfpointspherical{\longitude}{\latitude}{1}}
            \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude}{1}}
            \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude+15}{1}}
            \pgfpathlineto{\pgfpointspherical{\longitude}{\latitude+15}{1}}
            \pgfpathlineto{\pgfpointspherical{\longitude}{\latitude+15}{1}}
            \pgfpathclose
            }
            \pgfusepath{fill,stroke}
        }
    }
\end{tikzpicture}
```

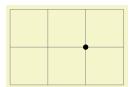
78.5 Building Coordinates From Other Coordinates

Many commands allow you to construct a coordinate in terms of other coordinates.

78.5.1 Basic Manipulations of Coordinates

\pgfpointadd $\{\langle v_1 \rangle\}\{\langle v_2 \rangle\}$

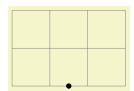
Returns the sum vector $\langle v_1 \rangle + \langle v_2 \rangle$.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointadd{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

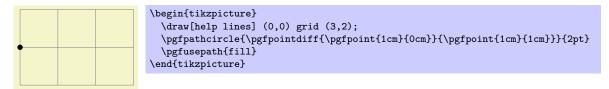
Returns the vector $\langle factor \rangle \langle coordinate \rangle$.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointscale{1.5}{\pgfpoint{1cm}{0cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

$\protect\pro$

Returns the difference vector $\langle end \rangle - \langle start \rangle$.



$\protect\operatorname{\begin{tabular}{l} pgfpointnormalised {\langle point \rangle }}$

This command returns a normalised version of $\langle point \rangle$, that is, a vector of length 1pt pointing in the direction of $\langle point \rangle$. If $\langle point \rangle$ is the 0-vector or extremely short, a vector of length 1pt pointing upwards is returned.

This command is *not* implemented by calculating the length of the vector, but rather by calculating the angle of the vector and then using (something equivalent to) the $\protect\p$

```
begin{tikzpicture}
    \draw[help lines] (0,0) grid (3,2);
    \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
    \pgfpathcircle{\pgfpointscale{20}}
      {\pgfpointnormalised{\pgfpoint{2cm}{1cm}}}{2pt}
    \pgfusepath{fill}
    \end{tikzpicture}
```

78.5.2 Points Traveling along Lines and Curves

The commands in this section allow you to specify points on a line or a curve. Imagine a point "traveling" along a curve from some point p to another point q. At time t=0 the point is at p and at time t=1 it is at q and at time, say, t=1/2 it is "somewhere in the middle." The exact location at time t=1/2 will not necessarily be the "halfway point," that is, the point whose distance on the curve from p and q is equal. Rather, the exact location will depend on the "speed" at which the point is traveling, which in turn depends on the lengths of the support vectors in a complicated manner. If you are interested in the details, please see a good book on Bézier curves.

\pgfpointlineattime $\{\langle time\ t \rangle\}\{\langle point\ p \rangle\}\{\langle point\ q \rangle\}$

Yields a point that is the tth fraction between p and q, that is, p + t(q - p). For t = 1/2 this is the middle of p and q.

$\protect\pro$

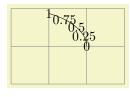
Yields a point that is located $\langle distance \rangle$ many units away from the start point in the direction of the end point. In other words, this is the point that results if we travel $\langle distance \rangle$ steps from $\langle start\ point \rangle$ towards $\langle end\ point \rangle$.

Example:

```
70pt
40pt
20pt
0pt
```

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \d in {0pt,20pt,40pt,70pt}
    {\pgftext[at=
     \pgfpointlineatdistance{\d}{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}]{\d}}
  \end{tikzpicture}
```

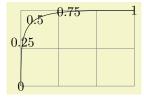
Yields a point on the arc between $\langle start\ angle \rangle$ and $\langle end\ angle \rangle$ on an ellipse whose center is at $\langle center \rangle$ and whose two principal axes are $\langle 0\text{-}degree\ axis} \rangle$ and $\langle 90\text{-}degree\ axis} \rangle$. For t=0 the point at the $\langle start\ angle \rangle$ is returned and for t=1 the point at the $\langle end\ angle \rangle$.



```
\draw[help lines] (0,0) grid (3,2);
\pgfpathmoveto{\pgfpoint{2cm}{1cm}}
\pgfpatharcaxes{0}{60}{\pgfpoint{2cm}{0cm}}{\pgfpoint{0cm}{1cm}}
\pgfusepath{stroke}
\foreach \t in {0,0.25,0.5,0.75,1}
    {\pgftext[at=\pgfpointarcaxesattime{\t}{\pgfpoint{0cm}{1cm}}}
    {\pgfpoint{2cm}{0cm}}{\pgfpoint{0cm}{1cm}}{\table}
\end{tikzpicture}
```

```
\propto | \pro
```

Yields a point that is on the Bézier curve from p to q with the support points s_1 and s_2 . The time t is used to determine the location, where t = 0 yields p and t = 1 yields q.



78.5.3 Points on Borders of Objects

The following commands are useful for specifying a point that lies on the border of special shapes. They are used, for example, by the shape mechanism to determine border points of shapes.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command returns a point that lies on the intersection of a line starting at the origin and going towards the point $\langle direction\ point \rangle$ and a rectangle whose center is in the origin and whose upper right corner is at $\langle corner \rangle$.

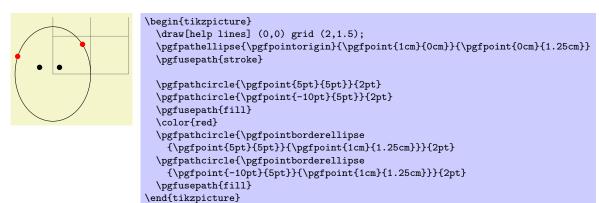
The $\langle direction\ point \rangle$ should have length "about 1pt," but it will be normalized automatically. Nevertheless, the "nearer" the length is to 1pt, the less rounding errors.

```
\begin\{tikzpicture\}
\draw[help lines] (0,0) grid (2,1.5);
\pgfpathrectanglecorners\\pgfpoint\{-1cm\}\{-1.25cm\}\\pgfpoint\{1.25cm\}\\
\pgfusepath\{stroke\}

\pgfpathcircle\\pgfpoint\{-10pt\}\{5pt\}\{2pt\}\\pgfusepath\{fill\}\\color\{red\}\\pgfpoint\{1.25cm\}\}\{2pt\}\\pgfpoint\{5pt\}\{5pt\}\{\pgfpoint\{1.25cm\}\}\{2pt\}\\pgfpoint\{color\{red\}\}\\pgfpoint\{5pt\}\{\pgfpoint\{1.25cm\}\}\{2pt\}\\pgfpoint\{-10pt\}\{5pt\}\\pgfpoint\{1.25cm\}\}\{2pt\}\\pgfpoint\{-10pt\}\{5pt\}\\pgfpoint\{1.25cm\}\}\{2pt\}\\pgfpoint\{-10pt\}\{5pt\}\\pgfpoint\{1.25cm\}\}\\pgfusepath\{fill\}\\pgfpoint\{1.25cm\}\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\\pgfusepath\{fill\}\pgfusepath\{fill\}\pgfusepath\{fill\}\pgfusepath\{fill\}\pgfusepath\{fill\}\pgfusepa
```

$\protect{\protection point}{\protection point}{\protection point}{\protection point}$

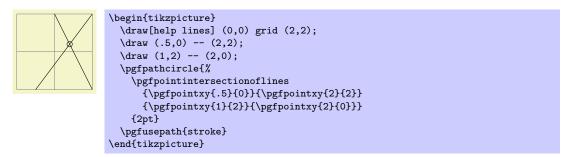
This command works like the corresponding command for rectangles, only this time the $\langle corner \rangle$ is the corner of the bounding rectangle of an ellipse.



78.5.4 Points on the Intersection of Lines

\pgfpointintersectionoflines $\{\langle p \rangle\}\{\langle q \rangle\}\{\langle s \rangle\}\{\langle t \rangle\}$

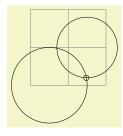
This command returns the intersection of a line going through p and q and a line going through s and t. If the lines do not intersection, an arithmetic overflow will occur.



78.5.5 Points on the Intersection of Two Circles

```
\protection of circles \{\langle p_1 \rangle\} \{\langle p_2 \rangle\} \{\langle r_1 \rangle\} \{\langle r_2 \rangle\} \{\langle solution \rangle\}
```

This command returns the intersection of the two circles centered at p_1 and p_2 with radii r_1 and r_2 . If $\langle solution \rangle$ is 1, the first intersection is returned, otherwise the second one is returned.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \draw (0.5,0) circle (1);
  \draw (1.5,1) circle (.8);
  \pgfpathcircle{%
    \pgfpointintersectionofcircles
      {\pgfpointxy{.5}{0}}{\pgfpointxy{1.5}{1}}
      {1cm}{0.8cm}{1}}
  \pgfusepath{stroke}
  \end{tikzpicture}
```

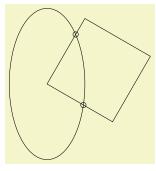
78.5.6 Points on the Intersection of Two Paths

```
\usepgflibrary{intersections} % MEX and plain TeX and pure pgf \usepgflibrary[intersections] % ConTeXt and pure pgf \usetikzlibrary{intersections} % MEX and plain TeX when using TikZ \usetikzlibrary[intersections] % ConTeXt when using TikZ
```

This library defines the below command and allows you to calculate the intersections of two arbitrary paths. However, due to the low accuracy of T_EX, the paths should not be "too complicated". In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths.

\pgfintersectionofpaths $\{\langle path \ 1 \rangle\}\{\langle path \ 2 \rangle\}$

This command finds the intersection points on the paths $\langle path \ 1 \rangle$ and $\langle path \ 2 \rangle$. The number of intersection points ("solutions") that are found will be stored, and each point can be accessed afterward. The code for $\langle path \ 1 \rangle$ and $\langle path \ 2 \rangle$ is executed within a TEX group and so can contain transformations (which will be in addition to any existing transformations). The code should not use the path in any way, unless the path is saved first and restored afterward. PGF will regard solutions as "a bit special", in that the points returned will be "absolute" and unaffected by any further transformations.



```
\begin{pgfpicture}
\pgfintersectionofpaths
{
  \pgfpathellipse{\pgfpointxy{0}{0}}{\pgfpointxy{1}{0}}{\pgfpointxy{0}{2}}
  \pgfgetpath\temppath
  \pgfusepath{stroke}
  \pgfsetpath\temppath
}
{
  \pgftransformrotate{-30}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpointxy{2}{2}}
  \pgfgetpath\temppath
  \pgfusepath\temppath
  \pgfusepath\temppath
  \pgfusepath\temppath
  \pgfusepath\temppath
}
\foreach \s in {1,...,\pgfintersectionsolutions}
  {\pgfpathcircle{\pgfpointintersectionsolution{\s}}{2pt}}
\pgfusepath{stroke}
\end{pgfpicture}
```

\pgfintersectionsolutions

After using the $\protection of paths command$, this T_EX -macro will indicate the number of solutions found.

$\verb|\pgfpointintersectionsolution{|} \langle number \rangle \}$

After using the $\protection of paths command$, this command will return the point for solution $\langle number \rangle$ or the origin if this solution was not found. By default, the intersections are simply returned in the order that the intersection algorithm finds them. Unfortunately, this is not necessarily a "helpful" ordering. However the following two commands can be used to order the solutions more helpfully.

\pgfintersectionsortbyfirstpath

Using this command will mean the solutions will be sorted along $\langle path 1 \rangle$.

```
\pgfintersectionsortbysecondpath
```

Using this command will mean the solutions will be sorted along $\langle path 2 \rangle$.

78.6 Extracting Coordinates

There are two commands that can be used to "extract" the x- or y-coordinate of a coordinate.

```
\protection \pro
```

Sets the $T_FX-\langle dimension \rangle$ to the x-coordinate of the point.

```
\newdimen\mydim
\pgfextractx{\mydim}{\pgfpoint{2cm}{4pt}}
%% \mydim is now 2cm
```

```
\protecty{\langle dimension \rangle}{\langle point \rangle}
```

Like $\protect\protec$

```
\protect\operatorname{\for} x = \{ (macro for x) \} \{ (macro for y) \}
```

Stores the most recently used (x, y) coordinates into two macros.

```
Macro x is '56.9055pt' and macro y is '113.81102pt'.

\pgfpoint{2cm}{4cm}
\pgfgetlastxy{\macrox}{\macroy}
```

Since (x, y) coordinates are usually assigned globally, it is safe to use this command after path operations.

78.7 Internals of How Point Commands Work

Macro \$x\$ is '\macrox' and macro \$y\$ is '\macroy'.

As a normal user of PGF you do not need to read this section. It is relevant only if you need to understand how the point commands work internally.

When a command like \pgfpoint{1cm}{2pt} is called, all that happens is that the two TeX-dimension variables \pgf@x and \pgf@y are set to 1cm and 2pt, respectively. These variables belong to the set of internal PGF registers, see section 93 for details. A command like \pgfpathmoveto that takes a coordinate as parameter will just execute this parameter and then use the values of \pgf@x and \pgf@y as the coordinates to which it will move the pen on the current path.

Since commands like \pgfpointnormalised modify other variables besides \pgf@x and \pgf@y during the computation of the final values of \pgf@x and \pgf@y, it is a good idea to enclose a call of a command like \pgfpoint in a TeX-scope and then make the changes of \pgf@x and \pgf@y global as in the following example:

```
{ % open scope
  \pgfpointnormalised{\pgfpoint{1cm}{1cm}}
  \global\pgf@x=\pgf@x % make the change of \pgf@x persist past the scope
  \global\pgf@y=\pgf@y % make the change of \pgf@y persist past the scope
}
% \pgf@x and \pgf@y are now set correctly, all other variables are
% unchanged
```

Since this situation arises very often, the macro \pgf@process can be used to perform the above code:

```
\process{\langle code \rangle}
```

Executes the $\langle code \rangle$ in a scope and then makes \pgf@x and \pgf@y global.

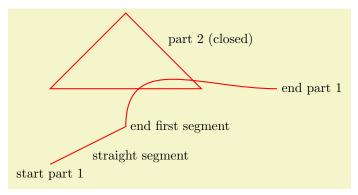
Note that this macro is used often internally. For this reason, it is not a good idea to keep anything important in the variables \pgf@x and \pgf@y since they will be overwritten and changed frequently. Instead, intermediate values can ge stored in the TeX-dimensions \pgf@xa, \pgf@xb, \pgf@xc and their y-counterparts \pgf@ya, \pgf@yb, pgf@yc. For example, here is the code of the command \pgfpointadd:

\def\pgfpointadd#1#2{%
\pgf@process{#1}%
\pgf@xa=\pgf@x%
\pgf@ya=\pgf@y%
\pgf@process{#2}%
\advance\pgf@x by\pgf@xa%
\advance\pgf@y by\pgf@ya}

79 Constructing Paths

79.1 Overview

The "basic entity of drawing" in PGF is the *path*. A path consists of several parts, each of which is either a closed or open curve. An open curve has a starting point and an end point and, in between, consists of several *segments*, each of which is either a straight line or a Bézier curve. Here is an example of a path (in red) consisting of two parts, one open, one closed:



```
\begin{tikzpicture} [scale=2]
  \draw[thick,red]
      (0,0) coordinate (a)
  -- coordinate (ab) (1,.5) coordinate (b)
      .. coordinate (bc) controls +(up:1cm) and +(left:1cm) .. (3,1) coordinate (c)
      (0,1) -- (2,1) -- coordinate (x) (1,2) -- cycle;

\draw (a) node[below] {start part 1}
      (ab) node[below right] {straight segment}
      (b) node[right] {end first segment}
      (c) node[right] {end part 1}
      (x) node[above right] {part 2 (closed)};
\end{tikzpicture}
```

A path, by itself, has no "effect," that is, it does not leave any marks on the page. It is just a set of points on the plane. However, you can *use* a path in different ways. The most natural actions are *stroking* (also known as *drawing*) and *filling*. Stroking can be imagined as picking up a pen of a certain diameter and "moving it along the path." Filling means that everything "inside" the path is filled with a uniform color. Naturally, the open parts of a path must first be closed before a path can be filled.

In PGF, there are numerous commands for constructing paths, all of which start with \pgfpath. There are also commands for *using* paths, though most operations can be performed by calling \pgfusepath with an appropriate parameter.

As a side-effect, the path construction commands keep track of two bounding boxes. One is the bounding box for the current path, the other is a bounding box for all paths in the current picture. See Section 79.13 for more details.

Each path construction command extends the current path in some way. The "current path" is a global entity that persists across TeX groups. Thus, between calls to the path construction commands you can perform arbitrary computations and even open and close TeX groups. The current path only gets "flushed" when the \pgfusepath command is called (or when the soft-path subsystem is used directly, see Section 97).

79.2 The Move-To Path Operation

The most basic operation is the move-to operation. It must be given at the beginning of paths, though some path construction command (like \pgfpathrectangle) generate move-tos implicitly. A move-to operation can also be used to start a new part of a path.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

This command expects a PGF-coordinate like \pgfpointorigin as its parameter. When the current path is empty, this operation will start the path at the given \(\langle coordinate \rangle \). If a path has already been partly constructed, this command will end the current part of the path and start a new one.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}} % New part
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{examplefill}
```

The command will apply the current coordinate transformation matrix to $\langle coordinate \rangle$ before using it. It will update the bounding box of the current path and picture, if necessary.

\pgfusepath{fill,stroke}

\end{pgfpicture}

79.3 The Line-To Path Operation

$\protect\pro$

This command extends the current path in a straight line to the given $\langle coordinate \rangle$. If this command is given at the beginning of path without any other path construction command given before (in particular without a move-to operation), the TEX file may compile without an error message, but a viewer application may display an error message when trying to render the picture.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
  \end{pgfpicture}
```

The command will apply the current coordinate transformation matrix to $\langle coordinate \rangle$ before using it. It will update the bounding box of the current path and picture, if necessary.

79.4 The Curve-To Path Operations

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command extends the current path with a Bézier curve from the last point of the path to $\langle coordinate \rangle$. The $\langle support \ 1 \rangle$ and $\langle support \ 2 \rangle$ are the first and second support point of the Bézier curve. For more information on Bézier curves, please consult a standard textbook on computer graphics.

Like the line-to command, this command may not be the first path construction command in a path.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{1cm}{1cm}{\pgfpoint{2cm}{1cm}}{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
  \end{pgfpicture}
```

The command will apply the current coordinate transformation matrix to $\langle coordinate \rangle$ before using it. It will update the bounding box of the current path and picture, if necessary. However, the bounding box is simply made large enough such that it encompasses all of the support points and the $\langle coordinate \rangle$. This will guarantee that the curve is completely inside the bounding box, but the bounding box will

typically be quite a bit too large. It is not clear (to me) how this can be avoided without resorting to "some serious math" in order to calculate a precise bounding box.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command works like \pgfpathcurveto, only it uses a quadratic Bézier curve rather than a cubic one. This means that only one support point is needed.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathquadraticcurveto
      {\pgfpoint{1cm}{1cm}{\pgfpoint{2cm}{0cm}}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
  \end{pgfpicture}
```

Internally, the quadratic curve is converted into a cubic curve. The only noticeable effect of this is that the points used for computing the bounding box are the control points of the converted curve rather than $\langle support \rangle$. The main effect of this is that the bounding box will be a bit tighter than might be expected. In particular, $\langle support \rangle$ will not always be part of the bounding box.

There exist two commands to draw only part of a cubic Bézier curve:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

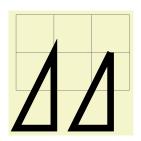
This command draws the part of the curve described by p, s_1 , s_2 and q between the times t_1 and t_2 . A time value of 0 indicates the point p and a time value of 1 indicates point q. This command includes a moveto operation to the first point.

\pgfpathcurvebetweentimecontinue{ $\langle time\ t_1 \rangle$ }{ $\langle time\ t_2 \rangle$ }{ $\langle point\ p \rangle$ }{ $\langle point\ s_1 \rangle$ }{ $\langle point\ s_2 \rangle$ }{ $\langle point\ q \rangle$ }
This command works like \pgfpathcurvebetweentime, except that a moveto operation is *not* made to the first point.

79.5 The Close Path Operation

\pgfpathclose

This command closes the current part of the path by appending a straight line to the start point of the current part. Note that there is a difference between closing a path and using the line-to operation to add a straight line to the start of the current path. The difference is demonstrated by the upper corners of the triangles in the following example:



```
\draw[help lines] (0,0) grid (3,2);
\pgfsetlinewidth{5pt}
\pgfpathmoveto{\pgfpoint{1cm}{1cm}}
\pgfpathlineto{\pgfpoint{0cm}{-1cm}}
\pgfpathlineto{\pgfpoint{1cm}{-1cm}}
\pgfpathlineto{\pgfpoint{2.5cm}{1cm}}
\pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
\pgfpathlineto{\pgfpoint{1.5cm}{-1cm}}
\pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
\pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
\pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
\pgfpathlineto{\pgfpoint{2.5cm}{1cm}}
\pgfpathlineto{\pgfpoint{2.5cm}{1cm}}
\pgfusepath{stroke}
\end{tikzpicture}
```

79.6 Arc, Ellipse and Circle Path Operations

The path construction commands that we have discussed up to now are sufficient to create all paths that can be created "at all." However, it is useful to have special commands to create certain shapes, like circles, that arise often in practice.

In the following, the commands for adding (parts of) (transformed) circles to a path are described.

```
\verb|\pgfpatharc{| \langle start\ angle \rangle}{ \langle end\ angle \rangle}{ \langle radius \rangle \ and \ \langle y\text{-}radius \rangle}}
```

This command appends a part of a circle (or an ellipse) to the current path. Imagine the curve between $\langle start\ angle \rangle$ and $\langle end\ angle \rangle$ on a circle of radius $\langle radius \rangle$ (if $\langle start\ angle \rangle < \langle end\ angle \rangle$, the curve goes around the circle counterclockwise, otherwise clockwise). This curve is now moved such that the point where the curve starts is the previous last point of the path. Note that this command will not start a new part of the path, which is important for example for filling purposes.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfpatharc{180}{90}{.5cm}
  \pgfpathlineto{\pgfpoint{3cm}{1.5cm}}
  \pgfpatharc{90}{-45}{.5cm}
  \pgfusepath{fill}
  \end{tikzpicture}
```

Saying \pgfpatharc{0}{360}{1cm} "nearly" gives you a full circle. The "nearly" refers to the fact that the circle will not be closed. You can close it using \pgfpathclose.

If the optional $\langle y\text{-}radius \rangle$ is given, the $\langle radius \rangle$ is the x-radius and the $\langle y\text{-}radius \rangle$ the y-radius of the ellipse from which the curve is taken:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm and 1cm}
  \pgfusepath{draw}
  \end{tikzpicture}
```

The axes of the circle or ellipse from which the arc is "taken" always point up and right. However, the current coordinate transformation matrix will have an effect on the arc. This can be used to, say, rotate an arc:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm and 1cm}
  \pgfusepath{draw}
  \end{tikzpicture}
```

The command will update the bounding box of the current path and picture, if necessary. Unless rotation or shearing transformations are applied, the bounding box will be tight.

```
\protect{\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pr
```

This command is similar to \pgfpatharc. The main difference is how the ellipse or circle is specified from which the arc is taken. The two parameters $\langle first~axis \rangle$ and $\langle second~axis \rangle$ are the 0°-axis and the 90°-axis of the ellipse from which the path is taken. Thus, \pgfpatharc{0}{90}{1cm and 2cm} has the same effect as



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2cm,5mm) (0,0) -- (0cm,1cm);

  \pgfpathmoveto{\pgfpoint{2cm}{5mm}}
  \pgfpatharcaxes{0}{90}{\pgfpoint{2cm}{5mm}}{\pgfpoint{0cm}{1cm}}
  \pgfusepath{draw}
  \end{tikzpicture}
```

```
\protect{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protection{\protec
```

This command (which directly corresponds to the arc-path command of SVG) is used to add an arc to the path that starts at the current point and ends at $\langle target\ point \rangle$. This arc is part of an ellipse that is determined in the following way: Imagine an ellipse with radii $\langle x\text{-}radius \rangle$ and $\langle y\text{-}radius \rangle$ that is rotated around its center by $\langle rotation \rangle$ degrees. When you move this ellipse around in the plane, there will be exactly two positions such that the two current point and the target point lie on the border of the ellipse (excluding pathological cases). The flags $\langle large\ arc\ flag \rangle$ and $\langle clockwise\ flag \rangle$ are then used to decide which of these ellipses should be picked and which arc on the picked ellipsis should be used.

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

\pgfpathmoveto{\pgfpoint{0mm}{20mm}}
  \pgfpatharcto{3cm}{1cm}{0}{0}{0}{\pgfpoint{3cm}{1cm}}
  \pgfusepath{draw}
  \end{tikzpicture}
```

Both flags are considered to be false exactly if they evaluate to 0, otherwise they are true. If the $\langle large\ arc\ flag\rangle$ is true, then the angle spanned by the arc will be greater than 180°, otherwise it will be less than 180°. The $\langle clockwise\ flag\rangle$ is used to determine which of the two ellipses should be used: if the flag is true, then the arc goes from the current point to the target point in a counterclockwise direction, otherwise in a clockwise fashion.



```
\begin{tikzpicture}
  \pgfsetlinewidth{2pt}
  % Flags 0 0: red
  \pgfsetstrokecolor{red}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{0}{0}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
  % Flags 0 1: blue
  \pgfsetstrokecolor{blue}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{0}{1}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
  % Flags 1 0: orange
  \pgfsetstrokecolor{orange}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{1}{0}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
  % Flags 1 1: black
  \pgfsetstrokecolor{black}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{1}{1}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

Warning: The internal computations necessary for this command are numerically very unstable. In particular, the arc will not always really end at the $\langle target\ coordinate \rangle$, but may be off by up to several points. A more precise positioning is currently infeasible due to TEX's numerical weaknesses. The only case it works quite nicely is when the resulting angle is a multiple of 90° .

```
\propto $$ \operatorname{center point} {\langle center point \rangle} {\langle
```

A specialized arc operation which is fast and numerically stable, provided a lot of information is given in advance.

In contrast to \pgfpatharc, it explicitly interpolates start and end points.

In contrast to \pgfpatharcto, this routine is numerically stable and quite fast since it relies on a lot of available information.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \def\cx{1.5cm}% center x
  \def\cy{1cm}% center y
  \def\startangle{0}%
  \def\endangle{270}%
  \def\a{1.5cm}% xradius
  \def\b{0.5cm}% yradius
  \pgfmathparse{\a/\b}\let\abratio=\pgfmathresult
  \pgfmathparse{\b/\a}\let\baratio=\pgfmathresult
 % start point:
  \pgfpathmoveto{\pgfpoint{\cx+\a*cos(\startangle)}{\cy+\b*sin(\startangle)}}\%
  \pgfpatharctoprecomputed
    {\pgfpoint{\cx}{\cy}}
    {\startangle}
    {\endangle}
    {\bf \{\pgfpoint\{\cx+\a*cos(\endangle)\}\{\cy+\b*sin(\endangle)\}\}} \ \ {\it end point}
    {\b}
    {\abratio}
    {\baratio}
  \pgfusepath{draw}
\end{tikzpicture}
```

\pgfpatharctomaxstepsize

The quality of arc approximation taken by \pgfpatharctoprecomputed by means of Bézier splines is controlled by a mesh width, which is initially

\def\pgfpatharctoprecomputed{45}.

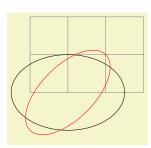
The mesh width is provided in (full!) degrees. The smaller the mesh width, the more precise the arc approximation.

Use an empty value to disable spline approximation (uses a single cubic polynomial for the complete arc).

The value must be an integer!

$\protect\pro$

The effect of this command is to append an ellipse to the current path (if the path is not empty, a new part is started). The ellipse's center will be $\langle center \rangle$ and $\langle first \ axis \rangle$ and $\langle second \ axis \rangle$ are the axis vectors. The same effect as this command can also be achieved using an appropriate sequence of move-to, arc, and close operations, but this command is easier and faster.



The command will apply coordinate transformations to all coordinates of the ellipse. However, the coordinate transformations are applied only after the ellipse is "finished conceptually." Thus, a transformation of 1cm to the right will simply shift the ellipse one centimeter to the right; it will not add 1cm to the x-coordinates of the two axis vectors.

The command will update the bounding box of the current path and picture, if necessary.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

A shorthand for \pgfpathellipse applied to $\langle center \rangle$ and the two axis vectors $(\langle radius \rangle, 0)$ and $(0, \langle radius \rangle)$.

79.7 Rectangle Path Operations

Another shape that arises frequently is the rectangle. Two commands can be used to add a rectangle to the current path. Both commands will start a new part of the path.

```
\protection \pro
```

Adds a rectangle to the path whose one corner is $\langle corner \rangle$ and whose opposite corner is given by $\langle corner \rangle + \langle diagonal\ vector \rangle$.

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectangle{\pgfpoint{1.5cm}{\log1cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1.5cm}{\log1cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{2cm}{0.5cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
  \end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

```
\protection \pro
```

Adds a rectangle to the path whose two opposing corners are $\langle corner \rangle$ and $\langle opposite\ corner \rangle$.

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectanglecorners{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
  \end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

79.8 The Grid Path Operation

```
\protect\operatorname{\mathtt{pgfpathgrid}}[\langle options \rangle] \{\langle lower\ left \rangle\} \{\langle upper\ right \rangle\}
```

Appends a grid to the current path. That is, a (possibly large) number of parts are added to the path, each part consisting of a single horizontal or vertical straight line segment.

Conceptually, the origin is part of the grid and the grid is clipped to the rectangle specified by the $\langle lower\ left \rangle$ and the $\langle upper\ right \rangle$ corner. However, no clipping occurs (this command just adds parts to the current path). Rather, the points where the lines enter and leave the "clipping area" are computed and used to add simple lines to the current path.

The following keys influence the grid:

 $/pgf/step=\langle vector \rangle$ (no default)

Sets the horizontal stepping to the x-coordinate of $\langle vector \rangle$ and the vertical stepping to its y-coordinate.

The command will apply coordinate transformations and update the bounding boxes tightly. As for ellipses, the transformations are applied to the "conceptually finished" grid.

```
\begin{pgfpicture}
  \pgftransformrotate{10}
  \pgfpathgrid[stepx=1mm,stepy=2mm]{\pgfpoint{0mm}{0mm}}{\pgfusepath{stroke}}
  \end{pgfpicture}
```

79.9 The Parabola Path Operation

 $\protect\operatorname{\mathtt{f pgfpathparabola}}\{\langle \mathit{bend}\;\mathit{vector}\rangle\}\{\langle \mathit{end}\;\mathit{vector}\rangle\}$

This command appends two half-parabolas to the current path. The first starts at the current point and ends at the current point plus $\langle bend\ vector \rangle$. At this point, it has its bend. The second half parabola starts at that bend point and ends at point that is given by the bend plus $\langle end\ vector \rangle$.

If you set $\langle end\ vector \rangle$ to the null vector, you append only a half parabola that goes from the current point to the bend; by setting $\langle bend\ vector \rangle$ to the null vector, you append only a half parabola that goes through the current point and $\langle end\ vector \rangle$ and has its bend at the current point.

It is not possible to use this command to draw a part of a parabola that does not contain the bend.

```
\begin{pgfpicture}
  % Half-parabola going ''up and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpointorigin}{\pgfpoint{2cm}{4cm}}
  \color{red}
  \pgfusepath{stroke}
  % Half-parabola going ''down and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpoint{-2cm}{4cm}}{\pgfpointorigin}
  \color{blue}
  \pgfusepath{stroke}
  % Full parabola
  \pgfpathmoveto{\pgfpoint{-2cm}{2cm}}
  \pgfpathparabola{\pgfpoint{1cm}{-1cm}}{\pgfpoint{2cm}{4cm}}
  \color{orange}
  \pgfusepath{stroke}
\end{pgfpicture}
```

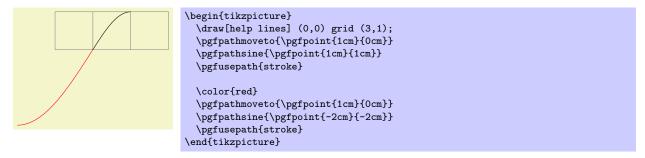
The command will apply coordinate transformations and update the bounding boxes.

79.10 Sine and Cosine Path Operations

Sine and cosine curves often need to be drawn and the following commands may help with this. However, they only allow you to append sine and cosine curves in intervals that are multiples of $\pi/2$.

```
\protect\
```

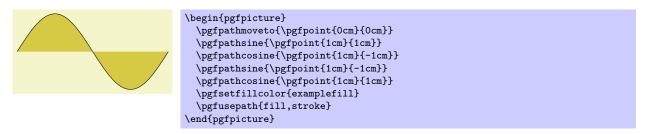
This command appends a sine curve in the interval $[0, \pi/2]$ to the current path. The sine curve is squeezed or stretched such that the curve starts at the current point and ends at the current point plus $\langle vector \rangle$.



The command will apply coordinate transformations and update the bounding boxes.

$\protect\pro$

This command appends a cosine curve in the interval $[0, \pi/2]$ to the current path. The curve is squeezed or stretched such that the curve starts at the current point and ends at the current point plus $\langle vector \rangle$. Using several sine and cosine operations in sequence allows you to produce a complete sine or cosine curve



The command will apply coordinate transformations and update the bounding boxes.

79.11 Plot Path Operations

There exist several commands for appending plots to a path. These commands are available through the module plot. They are documented in Section 89.

79.12 Rounded Corners

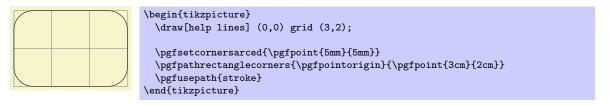
Normally, when you connect two straight line segments or when you connect two curves that end and start "at different angles," you get "sharp corners" between the lines or curves. In some cases it is desirable to produce "rounded corners" instead. Thus, the lines or curves should be shortened a bit and then connected by arcs.

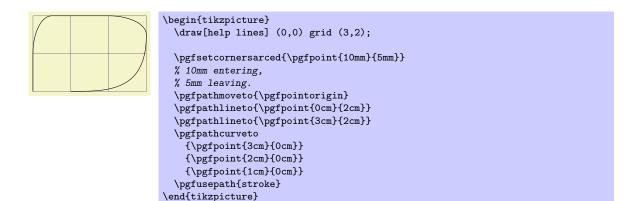
PGF offers an easy way to achieve this effect, by calling the following two commands.

$\protect\pro$

This command causes all subsequent corners to be replaced by little arcs. The effect of this command lasts till the end of the current T_FX scope.

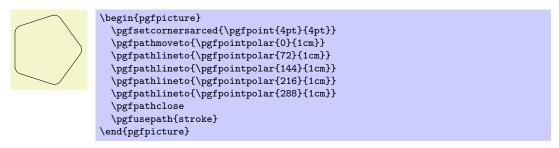
The $\langle point \rangle$ dictates how large the corner arc will be. Consider a corner made by two lines l and r and assume that the line l comes first on the path. The x-dimension of the $\langle point \rangle$ decides by how much the line l will be shortened, the y-dimension of $\langle point \rangle$ decides by how much the line r will be shortened. Then, the shortened lines are connected by an arc.





If the x- and y-coordinates of $\langle point \rangle$ are the same and the corner is a right angle, you will get a perfect quarter circle (well, not quite perfect, but perfect up to six decimals). When the angle is not 90°, you only get a fair approximation.

More or less "all" corners will be rounded, even the corner generated by a \pgfpathclose command. (The author is a bit proud of this feature.)



To return to normal (unrounded) corners, use \pgfsetcornersarced(\pgfpointorigin).

Note that the rounding will produce strange and undesirable effects if the lines at the corners are too short. In this case the shortening may cause the lines to "suddenly extend over the other end" which is rarely desirable.

79.13 Internal Tracking of Bounding Boxes for Paths and Pictures

The path construction commands keep track of two bounding boxes: One for the current path, which is reset whenever the path is used and thereby flushed, and a bounding box for the current {pgfpicture}.

\pgfresetboundingbox

Resets the picture's bounding box. The picture will simply forget any previous bounding box updates and start collecting from scratch.

You can use this together with \pgfusepath{use as bounding box} to replace the bounding box by the one of a particular path (ignoring subsequent paths).

The bounding boxes are not accessible by "normal" macros. Rather, two sets of four dimension variables are used for this, all of which contain the letter @.

\pgf@pathminx

The minimum x-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

\pgf@pathmaxx

The maximum x-coordinate "mentioned" in the current path. Initially, this is set to -16000pt.

\pgf@pathminy

The minimum y-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

\pgf@pathmaxy

The maximum y-coordinate "mentioned" in the current path. Initially, this is set to -16000pt.

\pgf@picminx

The minimum x-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

\pgf@picmaxx

The maximum x-coordinate "mentioned" in the current picture. Initially, this is set to -16000pt.

\pgf@picminy

The minimum y-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

\pgf@picmaxy

The maximum y-coordinate "mentioned" in the current picture. Initially, this is set to -16000pt.

Each time a path construction command is called, the above variables are (globally) updated. To facilitate this, you can use the following command:

Updates all of the above dimensions in such a way that the point specified by the two arguments is inside both bounding boxes. For the picture's bounding box this updating occurs only if \ipgf@relevantforpicturesize is true, see below.

For the bounding box of the picture it is not always desirable that every path construction command affects this bounding box. For example, if you have just used a clip command, you do not want anything outside the clipping area to affect the bounding box. For this reason, there exists a special "TEX if" that (locally) decides whether updating should be applied to the picture's bounding box. Clipping will set this if to false, as will certain other commands.

\pgf@relevantforpicturesizefalse

Suppresses updating of the picture's bounding box.

\pgf@relevantforpicturesizetrue

Causes updating of the picture's bounding box.

80 Decorations

```
\usepgfmodule{decorations} % MEX and plain TeX and pure pgf \usepgfmodule[decorations] % ConTeXt and pure pgf
```

The commands for creating decorations are defined in this module, so you need to load this module to use decorations. This module is automatically loaded by the different decoration libraries.

80.1 Overview

Decorations are a general way of creating graphics by "moving along" a path and, while doing so, either drawing something or constructing a new path. This could be as simple as extending a path with a "zigzagged" line...

```
\tikz \draw decorate[decoration=zigzag] {(0,0) -- (3,0)};
```

... but could also be as complex as typesetting text along a path:

```
\tikz \path decorate [decoration={text along path, text={Some text along a path}}] { (0,2) ... controls (2,2) and (1,0) ... (3,0) };
```

The workflow for using decorations is the following:

- 1. You define a decoration using the \pgfdeclaredecoration command. Different useful decorations are already declared in libraries like decorations.shapes.
- 2. You use normal path construction commands like \pgfpathlineto to construct a path. Let us call this path the to-be-decorated path.
- 3. You place the path construction commands inside the environment {pgfdecoration}. This environment takes the name of a previously declared decoration as a parameter. It will then start "walking along" the to-be-decorated path. As it does this, a special finite automaton called a decoration automaton produces new path commands as its output (or even other outputs). These outputs replace the to-be-decorated path; indeed, after the to-be-decorated path has been fully walked along it is thrown away, only the output of the automaton persists.

In the present section the process of how decoration automata work is explained first. Then the command(s) for declaring decoration automata and for using them are covered.

80.2 Decoration Automata

Decoration automata (and the closely related meta-decoration automata) are a general concept for creating graphics "along paths." For straight lines, this idea was first proposed by Till Tantau in an earlier version of PGF, the idea to extend this to arbitrary path was proposed and implemented by Mark Wibrow. Further versatility is provided by "meta-decorations". These are automata that decorate a path with decorations.

In the present subsection the different ideas underlying decoration automata are presented.

80.2.1 The Different Paths

In order to prevent confusion with different types of paths, such as those that are extended, those that are decorated and those that are created, the following conventions will be used:

- The *preexisting* path refers to the current path in existence before a decoration environment. (Possibly this path has been created by another decoration used earlier, but we will still call this path the preexisting path also in this case.)
- The *input* path refers to the to-be-decorated path that the decoration automaton moves along. The input path may consist of many line and curve input segments (for example, a circle or an ellipse consists of four curves). It is specified inside the decoration environment.

• The *output* path refers to the path that the decoration creates. Depending on the decoration, this path may or may not be empty (a decoration can also choose to use side-effects instead of producing an output path). The input path is always consumed by the decoration automaton, that is, it is no longer available in any way after the decoration automaton has finished.

The effect of a decoration environment is the following: The input path, which is specified inside the environment, is constructed and stored. This process does not alter the preexisting path in any way. Then the decoration automaton is started (as described later) and it produces an output path (possibly empty). Whenever part of the output path is produced, it is concatenated with the preexisting path. After the environment, the current path will equal the original preexisting path followed by the output path.

It is permissible that a decoration issues a \pgfusepath command. As usual, this causes the current path to be filled or stroked or some other action to be taken and the current path is set to the empty path. As described above, when the decoration automaton starts, the current path is the preexisting path and as the automaton progresses, the current path is constantly being extended by the output path. This means that first time a \pgfusepath command is used on a decoration, the preexisting path is part of the path this command operates on; in subsequent calls only the part of the output path constructed since the last \pgfusepath command will be used.

You can use this mechanism to stroke or fill different parts of the output path in different colors, line widths, fills and shades; all within the same decoration. Alternatively, a decoration can choose to produce no output path at all: the text decoration simply typesets text along a path.

80.2.2 Segments and States

The most common use of a decoration is to "repeat something along a path" (for example, the zigzag decoration repeats \sim along a path). However, it not necessarily the case that only one thing is repeated: a decoration can consist of different parts, or *segments*, repeated in a particular order.

When you declare a decoration, you provide a description of how their different segments will be rendered. The description of each segment should be given in a way as if the "x-axis" of the segment is the tangent to the path at a particular point, and that point is the origin of the segment. Thus, for example, the segment of the zigzag decoration might be defined using the following code:

```
\pgfpathlineto{\pgfpoint{5pt}{5pt}}
\pgfpathlineto{\pgfpoint{15pt}{-5pt}}
\pgfpathlineto{\pgfpoint{20pt}{0pt}}
```

PGF will ensure that an appropriate coordinate transformation is in place when the segment is rendered such that the segment actually points in the right direction. Also, subsequent segments will be transformed such that they are "further along the path" toward the end of the path. All transformations are set up automatically.

Note that we did not use a \pgfpathmoveto{\pgfpointorigin} at the beginning of the segment code. Doing so would subdivide the path into numerous subpaths. Rather, we assume that the previous segment caused the current point to be at the origin.

The width of a segment can (and must) be specified explicitly. PGF will use this width to find out the start point of the next segment and the correct rotation. The width the you provide need not be the "real" width of the segment, which allows decoration segments to overlap or to be spaced far apart.

The **zigzag** decoration only has one segment that is repeated again and again. However, we might also like to have *different* segments and use rules to describe which segment should be used where. For example, we might have special segments at the start and at the end.

Decorations use a mechanism known in theoretical in computer science as *finite state automata* to describe which segment is used at a particular point. The idea is the following: For the first segment we start in a special state called the *initial state*. In this state, and also in all other states later, PGF first computes how much space is left on the input path. That is, PGF keeps track of the distance to the end of the input path. Attached to each state there is a set of rules of the following form: "If the remaining distance on the input path is less than x, switch to state q." PGF checks for each of these rules whether it applies and, if so, immediately switches to state q.

Only if none of the rules tell us to switch to another state, PGF will execute the state's code. This code will (typically) add a segment to the output path. In addition to the rules there is also a width parameter attached to each state. PGF then translates the coordinate system by this width and reduces the remaining distance on the input path. Then, PGF either stays in the current state or switches to another state, depending on yet another property attached of the state.

The whole process stops when a special state called **final** is reached. The segment of this state is immediately added to the output path (it is often empty, though) and the process ends.

80.3 Declaring Decorations

The following command is used to declare a decoration. Essentially, this command describes the decoration automaton.

$\verb|\pgfdeclaredecoration{|} \langle name \rangle \} \{ \langle initial \ state \rangle \} \{ \langle states \rangle \}$

This command declares a new decoration called $\langle name \rangle$. The $\langle states \rangle$ argument contains a description of the decoration automaton's states and the transitions between them. The $\langle initial \ state \rangle$ is the state in which the automaton starts.

When the automaton is later applied to an input path, it keeps track of a certain position on the input path. This current point will "travel along the path," each time being moved along by a certain distance. This will also work if the path is not a straight line. That is, it is permissible that the path curves are veers at a sharp angle. It is also permissible that while traveling along the input path, the current input segment ends and a new input segment starts. In this case, the remaining distance on the first input segment is subtracted from the $\langle dimension \rangle$ and then we travel along the second input segment for the remaining distance. This input segment may also end early, in which case we travel along the next input segment, and so on. Note that it cannot happen that we travel past the end of the input path since this would have caused an immediate switch to the final state.

Note that the computation of the path lengths has only a low accuracy because of TEX's small math capabilities. Do not expect high accuracy alignments when using decorations (unless the input path consists only of horizontal and vertical lines).

The $\langle states \rangle$ argument should consist of \state commands, one for each state of the decoration automaton. The \state command is defined only when the $\langle states \rangle$ argument is executed.

This command declares a new state inside the current decoration automaton. The state is named $\langle name \rangle$.

When the decoration automaton is in state $\langle name \rangle$, the following things happen:

- 1. The $\langle options \rangle$ are parsed. This may lead to a state switch, see below. When this happens, the following steps are not executed. The $\langle options \rangle$ are executed one after the other in the given order. If an option causes a state switch, the switch is immediate, even if later options might cause a different state switch.
- 2. The $\langle code \rangle$ is executed in a T_EX-group with the current transformation matrix set up in such a way that the origin is on the input path at the current point (the point at the distance traveled up to now) and the coordinate system is rotated in such a way that the positive x-axis points in the direction of the tangent to the input path at the current point, while the positive y-axis points to the left of this tangent.

As described earlier, the $\langle code \rangle$ can have two different effects: If it just contains path construction commands, the decoration will produce an output path, that is, it extends the preexisting path. Here is an example:



```
\pgfdeclaredecoration{example}{initial}
{
   \state{initial}[width=10pt]
   {
   \pgfpathlineto{\pgfpoint{0pt}{5pt}}
   \pgfpathlineto{\pgfpoint{5pt}{-5pt}}
   \pgfpathlineto{\pgfpoint{10pt}{-5pt}}
   \pgfpathlineto{\pgfpoint{10pt}{0pt}}
   \pgfpathlineto{\pgfpoint{10pt}{0pt}}
}
}
\state{final}
   {
   \pgfpathlineto{\pgfpointdecoratedpathlast}}
}
}
\tikz[decoration=example]
{
   \draw [decorate] (0,0) -- (3,0);
   \draw [red,decorate] (0,0) to [out=45,in=135] (3,0);
}
```

Alternatively, the $\langle code \rangle$ can also contain the **\pgfusepath** command. This will use the path in the usual manner, where "the path" is the preexisting path plus a part of the output path for the first invocation and the different parts of the rest of the output path for the following invocation. Here is an example:



3. After the $\langle code \rangle$ has been executed (possibly more than once, if the repeat state option is used), the state switches to whatever state has been specified inside the $\langle options \rangle$ using the next state option. If no next state has been specified, the state stays the same.

The $\langle options \rangle$ are executed with the key path set to /pgf/decoration automaton. The following keys are defined:

```
/pgf/decoration automaton/switch if less than=\langle dimension \rangle to \langle new\ state \rangle (no default)
When this key is encountered, PGF checks whether the remaining distance to the end of the input path is less than \langle dimension \rangle. If so, an immediate state switch to \langle new\ state \rangle occurs.
```

```
/pgf/decoration automaton/switch if input segment less than= \langle \mathit{dimension} \rangle \text{ to } \langle \mathit{new state} \rangle \tag{no default}
```

When this key is encountered, PGF checks whether the remaining distance to the end of the current input segment of the input path is less than $\langle dimension \rangle$. If so, an immediate state switch to $\langle new \ state \rangle$ occurs.

```
/pgf/decoration automaton/width=\langle dimension \rangle (no default)
```

First, this option causes an immediate switch to the state final if the remaining distance on the input path is less than $\langle dimension \rangle$. The effect is the same as if you had said switch if less than= $\langle dimension \rangle$ to final just before the width option.

If no switch occurs, this option tells PGF the width of the segment. The current point will travel along the input path (as described earlier) by this distance.

/pgf/decoration automaton/repeat state=\(\repetitions\) (no default, initially 0)

Tells PGF how long the automaton stays "normally" in the current state. This count is reset to $\langle repetitions \rangle$ each time one of the switch if keys causes a state switch. If no state switches occur, the $\langle code \rangle$ is executed and the repetition counter is decreased. Then, there is once more a chance of a state change caused by any of the $\langle options \rangle$. If no repetition occurs, the $\langle code \rangle$ is executed once more and the counter is decreased once more. When the counter reaches zero, the $\langle code \rangle$ is executed once more, but, then, a different state is entered, as specified by the next state option.

Note, that the maximum number of times the state will be executed is $\langle repetitions \rangle + 1$.

pgf/decoration automaton/next state = (new state) (no default)

After the $\langle code \rangle$ for state has been executed for the last time, a state switch to $\langle new \ state \rangle$ is performed. If this option is not given, the next state is the same as the current state.

/pgf/decoration automaton/if input segment is closepath=\langle options \rangle (no default)

This key checks whether the current input segment is a closepath operation. If so, the $\langle options \rangle$ get executed; otherwise nothing happens. You can use this option to handle a closepath in some special way, for instance, switching to a new state in which \pgfpathclose is executed.

$/pgf/decoration automaton/auto end on length=\langle dimension \rangle$ (no default)

This key is just included for convenience, it does nothing that cannot be achieved using the previous options. The effect is the following: If the remaining input path's length is at most $\langle dimension \rangle$, the decorated path is ended with a straight line to the end of the input path and, possibly, it is closed, namely if the input path ended with a closepath operation. Otherwise, it is checked whether the current input segment is a closepath segment and whether the remaining distance on the current input segment is at most $\langle distance \rangle$. If so, then a closepath operation is used to close the decorated path and the automaton continues with the next subpath, remaining in the current state.

In all other cases, nothing happens.

/pgf/decoration automaton/auto corner on length= $\langle dimension \rangle$ (no default)

This key has the following effect: Firstly, in case the T_EX -if \ifpgfdecoratepathhascorners is false, nothing happens. Otherwise, it is tested whether the remaining distance on the current input segment is at most $\langle dimension \rangle$. If so, a lineto operation is used to reach the end of this input segment and the automaton continues with the next input segment, but remains in the current state.

The main idea behind this option is to avoid having decoration segments "overshoot" past a corner.

You may sometimes wish to do computations outside the transformational TEX-group of the current segment, so that these results of these computations are available in the next state. For this, the following two options are useful:

/pgf/decoration automaton/persistent precomputation= $\langle precode \rangle$ (no default)

If the $\langle code \rangle$ of the state is executed, the $\langle precode \rangle$ is executed first and it is executed outside the TeX-group of the $\langle code \rangle$. Note that when the $\langle precode \rangle$ is executed, the transformation matrix is not set up.

/pgf/decoration automaton/persistent postcomputation= $\langle postcode \rangle$ (no default)

Works like the persistent precomputation option, only the $\langle postcode \rangle$ is executed after (and also outside) the T_EX-group of the main $\langle code \rangle$.

There are a number of macros and dimensions which may be useful inside a decoration automaton. The following macros are available:

\pgfdecoratedpathlength

The length of the input path. If the input path consists of several input segments, this number is the sum of the lengths of the input segments.

\pgfdecoratedinputsegmentlength

The length of the current input segment of the input path. "Current input segment" refers to the input segment on which the current point lies.

\pgfpointdecoratedpathlast

The final point of the input path.

\pgfpointdecoratedinputsegmentlast

The final point of the current input segment of the input path.

\pgfdecoratedangle

The angle of the tangent to the decorated path at the *origin* of the current segment. The transformation matrix applied at the beginning of a state includes a rotation equivalent to this angle.

The following TEX dimension registers are also available inside the automaton:

\pgfdecoratedremainingdistance

The remaining distance on the input path.

\pgfdecoratedcompleteddistance

The completed distance on the input path.

\pgfdecoratedinputsegmentremainingdistance

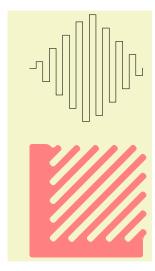
The remaining distance on the current input segment of the input path.

\pgfdecoratedinputsegmentcompleteddistance

The completed distance on the current input segment of the input path.

Further keys and macros are defined and used by the decoration libraries, see Section 31.

The following example shows how these options can be used:



```
\pgfdeclaredecoration{complicated example decoration}{initial}
  \state{initial}[width=5pt,next state=up]
 { \pgfpathlineto{\pgfpoint{5pt}{0pt}} }
  \state{up}[width=5pt,next state=down]
    \ifdim\pgfdecoratedremainingdistance>\pgfdecoratedcompleteddistance
     % Growing
      \pgfpathlineto{\pgfpoint{0pt}{\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \else
      \pgfpathlineto{\pgfpoint{0pt}{\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
  \state{down}[width=5pt,next state=up]
   \ifdim\pgfdecoratedremainingdistance>\pgfdecoratedcompleteddistance
      \pgfpathlineto{\pgfpoint{0pt}{-\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{-\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \else
      \pgfpathlineto{\pgfpoint{0pt}{-\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{-\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \fi%
 \state{final}
    \pgfpathlineto{\pgfpointdecoratedpathlast}
\begin{tikzpicture}[decoration=complicated example decoration]
 \frac{(0,0) -- (3,0)}{;}
 \fill [red!50,rounded corners=2pt]
   decorate \{(.5,-2) -- ++(2.5,-2.5)\} -- (3,-5) -| (0,-2) -- cycle;
\end{tikzpicture}
```

80.3.1 Predefined Decorations

The three decorations moveto, lineto, and curveto are predefined and "always available." They are mostly useful in conjunction with meta-decorations. They are documented in Section 31 alongside the other decorations.

80.4 Using Decorations

Once a decoration has been declared, it can be used.

```
\begin{pgfdecoration} \{\langle decoration \ list \rangle\} \\ \langle environment \ contents \rangle \\ \begin{pgfdecoration} \end{pgfdecoration} \end{pgfdecoration} \end{pdfdecoration}
```

The $\langle environment\ contents \rangle$ should contain commands for creating an path. This path is the basis for the input paths for the decorations in the $\langle decoration\ list \rangle$. In detail, the following happens:

- 1. The preexisting unused path is saved.
- 2. The path commands specified in $\langle environment\ contents \rangle$ are executed and this resulting path is saved. The path is then divided into different input paths as follows: The format for each item in $\{\langle decoration\ list \rangle\}$ is

```
{\langle decoration \rangle} {\langle length \rangle} {\langle before \ code \rangle} {\langle after \ code \rangle}
```

The $\langle before\ code \rangle$ and the $\langle after\ code \rangle$ are optional. The input path is divided into input paths as follows: The first input path consists of the first lines of the path specified in the $\langle environment \rangle$

 $contents\rangle$ until the $\langle length\rangle$ of the first element of the $\langle decoration\ list\rangle$ is reached. If this length is reached in the middle of a line, the line is broken up at this exact position. Then the second input path has the $\langle length\rangle$ of the second element in the $\langle decoration\ list\rangle$ and consists of the lines making up the following $\langle length\rangle$ part of the path in the $\langle environment\ contents\rangle$, and so on.

If the lengths in the $\langle decoration \ list \rangle$ do not add up to the total length of the path in the $\langle environment \ contents \rangle$, either some decorations are dropped (if their lengths add up to more than the length of the $\langle environment \ contents \rangle$) or the input path is not fully used (if their lengths add up to less).

- 3. The preexisting path is reinstalled.
- 4. The decoration automata move along the input paths, thus creating (and possibly using) the output paths. These output paths extend the current path (unless they are used).

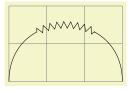
Some important points should be noted regarding the use of this environment:

- If \(\langle environment contents \rangle \) does not begin with \pgfpathmoveto, the last known point on the preexisting path is assumed as the starting point.
- All except the last of any sequence of consecutive move-to commands in $\langle environment\ contents \rangle$ are discarded.
- Any move-to commands at the end of $\langle environment \ contents \rangle$ are ignored.
- Any close-path commands on the input path are interpreted as straight lines. Internally, something a little more complicated is going on, however, a closed path on the input path has no effect on the output path, other than causing the automaton to travel in a straight line towards the location of the last move-to command on the input path.
- Although tangent computations for the input path work with the last point on the preexisting path, no automatic move-to operations are issued for the output path. If an output path starts with a line-to or curve-to when the existing path is empty, an appropriate move-to command should be inserted before the decoration starts.
- If a decoration uses its own path, the first time this happens the preexisting path is part of the path that is used at this point.

Before the automata start to "work on" their respective inputs paths, $\langle before\ code \rangle$ is executed. After the decoration automaton has finished, $\langle after\ code \rangle$ is executed.

```
\begin{tikzp \draw [hel \begin{pgfpath \pgfpath \pgfpath \pgfpath \quad \pgfpath \quad \pgfpath \quad \pgfpath \quad \qquad \quad \quad \quad \quad \q
```

When the lengths are evaluated, the dimension \pgfdecoratedremainingdistance holds the remaining distance on the entire decorated path, and \pgfdecoratedpathlength holds the total length of the path. Thus, it is possible to specify lengths like \pgfdecoratedpathlength/3.



When $\langle before\ code \rangle$ is executed, the following macro is useful:

\pgfpointdecoratedpathfirst

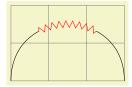
Returns the point corresponding to the start of the current input path.

When $\langle after\ code \rangle$ is executed, the following macro can be used:

\pgfpointdecoratedpathlast

Returns the point corresponding to the end of the current input path.

This means that if decorations do not use their own path, it is possible to do something with them and continue from the correct place.



```
\begin{tikzpicture}
             \draw [help lines] grid (3,2);
            \begin{pgfdecoration}{
                                    {curveto}{\pgfdecoratedpathlength/3}
                                                 \pgfusepath{stroke}
                                    {\tt zigzag} \{ \texttt{\pgfdecorated} \\ \texttt{\pgfdecorated} 
                                                 \pgfpathmoveto{\pgfpointdecoratedpathfirst}
                                                 \pgfdecorationsegmentlength=5pt
                                                  \pgfsetstrokecolor{red}
                                                  \pgfusepath{stroke}
                                                 \pgfpathmoveto{\pgfpointdecoratedpathlast}
                                                 \pgfsetstrokecolor{black}
                                    {curveto}{\pgfdecoratedremainingdistance}
                        \pgfpathmoveto{\pgfpointorigin}
                         \pgfpathcurveto
                                     {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}}
             \end{pgfdecoration}
             \pgfusepath{stroke}
\end{tikzpicture}
```

After the {decoration} environment has finished, the following macros are available:

\pgfdecorateexistingpath

The preexisting path before the environment was entered.

\pgfdecoratedpath

The (total) input path (that is, the path created by the environment contents).

\pgfdecorationpath

The output path. If the path is used, this macro contains only the last unused part of the output path.

\pgfpointdecoratedpathlast

The final point of the input path.

\pgfpointdecorationpathlast

The final point of the output path.

The following style is executed each time a decoration is used. You may use it to set up default options for decorations.

/pgf/every decoration

(style, initially empty)

This style is executed for every decoration.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

\endpgfdecoration

The plain-T_FX version of the {pgfdecorate} environment.

```
\startpgfdecoration{\langle name \rangle} \ \langle environment\ contents \rangle
\startpgfdecoration
```

The ConT_EXt version of the {pgfdecoration} environment.

For convenience, the following macros provide a "shorthand" for decorations (internally, they all use the {pgfdecoration} environment).

```
\protect\operatorname{\mathtt{f path}}\ (\operatorname{\it name})\ {\langle path \commands \rangle}
```

Decorate the path described by $\langle path\ commands \rangle$ with the decoration $\langle name \rangle$. This is equivalent to

$\protect\pro$

Decorate the preexisting path with the decoration $\langle name \rangle$.

Both the above commands use the current definitions of the following macros:

\pgfdecoratebeforecode

Code executed as \(\lambda before \code \rangle \), see the description of \pgfdecorate.

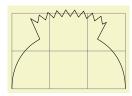
\pgfdecorateaftercode

Code executed as $\langle after\ code \rangle$, see the description of \pgfdecorate.

It may sometimes be useful to add an additional transformation for each segment of a decoration. The following command allows you to define such a "last minute transformation."

\pgfsetdecorationsegmenttransformation $\{\langle code \rangle\}$

The $\langle code \rangle$ will be executed at the very beginning of each segment. Note when applying multiple decorations, this will be reset between decorations, so it needs to be specified for each segment.



80.5 Meta-Decorations

A meta-decoration provides an alternative way to decorate a path with multiple decorations. It is, in essence, an automaton that decorates an input path with decoration automatons. In general, however, the end effect is still that a path is decorated with other paths, and the input path should be thought of as being divided into sub-input-paths, each with their own decoration. Like ordinary decorations, a meta-decoration must be declared before it can be used.

80.5.1 Declaring Meta-Decorations

$\protect\pro$

This command declares a new meta-decoration called $\langle name \rangle$. The $\langle states \rangle$ argument contains a description of the meta-decoration automaton's states and the transitions between them. The $\langle initial state \rangle$ is the state in which the automaton starts.

The \state command is similar to the one found in decoration declarations, and takes the same form:

Declares the state $\langle name \rangle$ inside the current meta-decoration automaton. Unlike decorations, states in meta-decorations are not executed within a group, which makes the persistent computation options superfluous. Consider using an initial state with width=0pt to do precalculations that could speed the execution of the meta-decoration.

The $\langle options \rangle$ are executed with the key path set to /pgf/meta-decorations automaton/, and the following keys are defined for this path:

```
/pgf/meta-decoration automaton/switch if less than=\langle dimension \rangle to \langle new \ state \rangle (no default)
```

This causes PGF to check whether the remaining distance to the end of the input path is less than $\langle dimension \rangle$, and, if so, to immediately switch to the state $\langle new \ state \rangle$. When this key is evaluated, the macro \pgfmetadecoratedpathlength will be defined as the total length of the decoration path, allowing for values such as \pgfmetadecoratedpathlength/8.

$/pgf/meta-decoration automaton/width=\langle dimension \rangle$ (no default)

As always, this option will cause an immediate switch to the state final if the remaining distance on the input path is less than $\langle dimension \rangle$.

Otherwise, this option tells PGF the width of the "meta-segment", that is, the length of the sub-input-path which the decoration automaton specified in $\langle code \rangle$ will decorate.

```
/pgf/meta-decoration automaton/next state=\langle new state \rangle (no default)
```

After the code for a state has been executed, a state switch to $\langle new \ state \rangle$ is performed. If this option is not given, the next state is the same as the current state.

The code in $\langle code \rangle$ is quite different from the code in a decoration state. In almost all cases only the following three macros will be required:

$\decoration{\langle name \rangle}$

This sets the decoration for the current state to $\langle name \rangle$. If this command is omitted, the moveto decoration will be used.

$\begin{tabular}{l} \begin{tabular}{l} \begin{tabu$

Defines $\langle before\ code \rangle$ as (typically) PGF commands to be executed before the decoration is applied to the current segment. This command can be omitted. If you wish to set up some decoration specific parameters such as segment length, or segment amplitude, then they can be set in $\langle before\ code \rangle$.

\arrangle \afterdecoration {\langle after code \rangle}

Defines $\langle after\ code \rangle$ as commands to be executed after the decoration has been applied to the current segment. This command can be omitted.

There are some macros that may be useful when creating meta-decorations (note that they are all macros):

\pgfpointmetadecoratedpathfirst

When the $\langle before\ code \rangle$ is executed, this macro stores the first point on the current sub-inputpath.

\pgfpointmetadecoratedpathlast

When the $\langle after\ code \rangle$ is executed, this macro stores the last point on the current sub-inputpath.

\pgfmetadecoratedpathlength

The entire length of the entire input path.

\pgfmetadecoratedcompleteddistance

The completed distance on the entire input path.

\pgfmetadecoratedremainingdistance

The remaining distance on the entire input path.

\pgfmetadecoratedinputsegmentcompleteddistance

The completed distance on the current input segment of the entire input path.

\pgfmetadecoratedinputsegmentremainingdistance

The remaining distance on the current input segment of the entire input path.

Here is a complete example of a meta-decoration:

```
\pgfdeclaremetadecoration{arrows}{initial}{
  \state{initial}[width=0pt, next state=arrow]
    \pgfmathdivide{100}{\pgfmetadecoratedpathlength}
    \let\factor\pgfmathresult
    \pgfsetlinewidth{1pt}
    \pgfset{/pgf/decoration/segment length=4pt}
  \state{arrow}[
    switch if less than=\pgfmetadecorationsegmentlength to final,
    width=\pgfmetadecorationsegmentlength/3,
    next state=zigzag]
    \decoration{curveto}
    \beforedecoration
      \pgfmathparse{\pgfmetadecoratedcompleteddistance*\factor}
      \pgfsetcolor{red!\pgfmathresult!yellow}
      \pgfpathmoveto{\pgfpointmetadecoratedpathfirst}
  \state{zigzag}[width=\pgfmetadecorationsegmentlength/3, next state=end arrow]
   \decoration{zigzag}
  \state{end arrow}[width=\pgfmetadecorationsegmentlength/3, next state=move]
    \decoration{curveto}
    \beforedecoration{\pgfpathmoveto{\pgfpointmetadecoratedpathfirst}}
    \afterdecoration
      \pgfsetarrowsend{to}
      \pgfusepath{stroke}
  \state{move}[width=\pgfmetadecorationsegmentlength/2, next state=arrow]{}
  \state{final}{}
\tikz\draw[decorate,decoration={arrows,meta-segment length=2cm}]
  (0,0) ... controls (0,2) and (3,2)
                                        .. (3,0)
        .. controls (3,-2) and (0,-2) .. (0,-4)
```

80.5.2 Predefined Meta-decorations

There are no predefined meta-decorations loaded with PGF.

80.5.3 Using Meta-Decorations

Using meta-decorations is "simpler" than using decorations, because you can only use one meta-decoration per path.

.. controls (0,-6) and (3,-6) .. (3,-8) .. controls (3,-10) and (0,-10) .. (0,-8);

 $\begin{pgfmetadecoration} {\langle name \rangle} \end{page}$

```
\langle environment\ contents \rangle \end{pgfmetadecoration}
```

This environment decorates the input path described in $\langle environment\ contents \rangle$, with the meta-decoration $\langle name \rangle$.

$\label{eq:continuous} $$ \begin{array}{c} \operatorname{pgfmetadecoration} \{\langle name \rangle \} \\ \langle environment\ contents \rangle \\ \operatorname{endpgfmetadecoration} \\ \end{array} $$$

The plain $T_{\hbox{\footnotesize E}}X$ version of the $\{{\tt pgfmetadecoration}\}$ environment.

```
\label{eq:continuous} $$ \left\langle environment\ contents \right\rangle $$ \stoppgfmetadecoration
```

The ConTEXt version of the {pgfmetadecoration} environment.

81 Using Paths

81.1 Overview

Once a path has been constructed, it can be *used* in different ways. For example, you can draw the path or fill it or use it for clipping.

Numerous graph parameters influence how a path will be rendered. For example, when you draw a path, the line width is important as well as the dashing pattern. The options that govern how paths are rendered can all be set with commands starting with \pgfset. All options that influence how a path is rendered always influence the complete path. Thus, it is not possible to draw part of a path using, say, a red color and drawing another part using a green color. To achieve such an effect, you must use two paths.

In detail, paths can be used in the following ways:

- 1. You can *stroke* (also known as *draw*) a path.
- 2. You can fill a path with a uniform color.
- 3. You can *clip* subsequent renderings against the path.
- 4. You can shade a path.
- 5. You can use the path as bounding box for the whole picture.

You can also perform any combination of the above, though it makes no sense to fill and shade a path at the same time.

To perform (a combination of) the first three actions, you can use the following command:

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\b$

Applies the given $\langle actions \rangle$ to the current path. Afterwards, the current path is (globally) empty. The following actions are possible:

• fill fills the path. See Section 81.3 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{fill}
\end{pgfpicture}
```

• stroke strokes the path. See Section 81.2 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
  \end{pgfpicture}
```

 \bullet $\verb"clip"$ clips all subsequent drawings against the path. See Section 81.4 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke,clip}
  \pgfpathcircle{\pgfpoint{1cm}{1cm}}{0.5cm}
  \pgfusepath{fill}
  \end{pgfpicture}
```

• discard discards the path, that is, it is not used at all. Giving this option (alone) has the same effect as giving an empty options list.

When more than one of the first three actions are given, they are applied in the above ordering, regardless of their ordering in \(\langle actions \rangle \). Thus, \(\stroke, fill \) and \(\fill, stroke \) have the same effect.

To shade a path, use the \pgfshadepath command, which is explained in Section 91.

81.2 Stroking a Path

When you use \pgfusepath{stroke} to stroke a path, several graphic parameters influence how the path is drawn. The commands for setting these parameters are explained in the following.

Note that all graphic parameters apply to the path as a whole, never only to a part of it.

All graphic parameters are local to the current {pgfscope}, but they persists past TEX groups, except for the interior rule (even-odd or nonzero) and the arrow tip kinds. The latter graphic parameters only persist till the end of the current TEX group, but this may change in the future, so do not count on this.

81.2.1 Graphic Parameter: Line Width

$\protect\pro$

This command sets the line width for subsequent strokes (in the current pgfscope). The line width is given as a normal TeX dimension like 0.4pt or 1mm.

\pgflinewidth

You can access the current line width via the TEX dimension \pgflinewidth. It will be set to the correct line width, that is, even when a TEX group closed, the value will be correct since it is set globally, but when a {pgfscope} closes, the value is set to the correct value it had before the scope.

81.2.2 Graphic Parameter: Caps and Joins

\pgfsetbuttcap

Sets the line cap to a butt cap. See Section 15.3.1 for an explanation of what this is.

\pgfsetroundcap

Sets the line cap to a round cap. See again Section 15.3.1.

\pgfsetrectcap

Sets the line cap to a square cap. See again Section 15.3.1.

\pgfsetroundjoin

Sets the line join to a round join. See again Section 15.3.1.

\pgfsetbeveljoin

Sets the line join to a bevel join. See again Section 15.3.1.

\pgfsetmiterjoin

Sets the line join to a miter join. See again Section 15.3.1.

$\protect\operatorname{pgfsetmiterlimit}\{\langle miter\ limit\ factor\rangle\}\$

Sets the miter limit to $\langle miter\ limit\ factor \rangle$. See again Section 15.3.1.

81.2.3 Graphic Parameter: Dashing

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

Sets the dashing of a line. The first entry in the list specifies the length of the first solid part of the list. The second entry specifies the length of the following gap. Then comes the length of the second solid part, following by the length of the second gap, and so on. The $\langle phase \rangle$ specifies where the first solid part starts relative to the beginning of the line.

```
begin{pgfpicture}
    \pgfsetdash{{0.5cm}{0.1cm}{0.2cm}}{0cm}
    \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
    \pgfpathlineto{\pgfpoint{2cm}{0mm}}
    \pgfusepath{stroke}
    \pgfsetdash{{0.5cm}{0.1cm}{0.2cm}}{0.1cm}
    \pgfpathmoveto{\pgfpoint{0mm}{1mm}}
    \pgfpathlineto{\pgfpoint{2cm}{1mm}}
    \pgfusepath{stroke}
    \pgfsetdash{{0.5cm}{0.1cm}{0.2cm}}{0.2cm}}
    \pgfusepathlineto{\pgfpoint{2cm}{1mm}}
    \pgfusepath{stroke}
    \pgfsetdash{{0.5cm}{0.1cm}{0.2cm}}{0.2cm}}
    \pgfpathlineto{\pgfpoint{0mm}{2mm}}
    \pgfpathlineto{\pgfpoint{2cm}{2mm}}
    \pgfusepath{stroke}
    \pgfusepath{stroke}
    \end{pgfpicture}
```

Use \pgfsetdash{}{0pt} to get a solid dashing.

81.2.4 Graphic Parameter: Stroke Color

$\protect\pro$

Sets the color used for stroking lines to $\langle color \rangle$, where $\langle color \rangle$ is a LATEX color like red or black!20!red. Unlike the \color command, the effect of this command lasts till the end of the current {pgfscope} and not till the end of the current TEX group.

The color used for stroking may be different from the color used for filling. However, a \color command will always "immediately override" any special settings for the stroke and fill colors.

In plain TEX, this command will also work, but the problem of defining a color arises. After all, plain TEX does not provide LATEX colors. For this reason, PGF implements a minimalistic "emulation" of the \definecolor, \colorlet, and \color commands. Only gray-scale and rgb colors are supported. For most cases this turns out to be enough.

$\protection{ \langle color \rangle }$

Sets both the stroke and fill color. The difference to the normal \color command is that the effect lasts till the end of the current {pgfscope}, not only till the end of the current TeX group.

81.2.5 Graphic Parameter: Stroke Opacity

You can set the stroke opacity using \pgfsetstrokeopacity. This command is described in Section 92.

81.2.6 Graphic Parameter: Arrows

After a path has been drawn, PGF can add arrow tips at the ends. It will only add arrows at the beginning of the first subpath and at the end of the last subpath. For closed paths the result is not defined, that is, it may change without notice in the future.

$\protect\pro$

Sets the arrow tip kind used at the start of a (possibly curved) path. When this option is used, the line will often be slightly shortened to ensure that the tip of the arrow will exactly "touch" the "real" start of the line.

To "clear" the start arrow, say \pgfsetarrowsstart{}.

```
\begin{pgfpicture}
   \pgfsetarrowsstart{latex}
   \pgfpathmoveto{\pgfpointorigin}
   \pgfpathlineto{\pgfpoint{1cm}{0cm}}
   \pgfusepath{stroke}
   \pgfsetarrowsstart{to}
   \pgfpathmoveto{\pgfpoint{0cm}{2mm}}
   \pgfpathlineto{\pgfpoint{1cm}{2mm}}
   \pgfusepath{stroke}
   \pgfusepath{stroke}
   \pgfusepath{stroke}
   \end{pgfpicture}
```

The effect of this command persists only till the end of the current TEX scope.

The different possible arrow kinds are explained in Section 82.

$\protect\pro$

Sets the arrow tip kind used at the end of a path.

```
\begin{pgfpicture}
    \pgfsetarrowsstart{latex}
    \pgfsetarrowsend{to}
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathlineto{\pgfpoint{1cm}{0cm}}
    \pgfusepath{stroke}
    \end{pgfpicture}
```

$\protect\pro$

Sets the start arrow kind to $\langle start \ kind \rangle$ and the end kind to $\langle end \ kind \rangle$.

```
\
\begin{pgfpicture}
   \pgfsetarrows{latex-to}
   \pgfpathmoveto{\pgfpointorigin}
   \pgfpathlineto{\pgfpoint{1cm}{0cm}}
   \pgfusepath{stroke}
   \end{pgfpicture}
```

$\protect\pro$

This command will shortened the start of every stroked path by the given dimension. This shortening is done in addition to automatic shortening done by a start arrow, but it can be used even if no start arrow is given.

This command is useful if you wish arrows or lines to "stop shortly before" a given point.

```
\begin{pgfpicture}
  \pgfpathcircle{\pgfpointorigin}{5mm}
  \pgfusepath{stroke}
  \pgfsetarrows{latex-}
  \pgfsetshortenstart{4pt}
  \pgfpathmoveto{\pgfpoint{5mm}{0cm}} % would be on the circle
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
  \end{pgfpicture}
```

$\protect\pro$

Works like \pgfsetshortenstart.

81.2.7 Inner Lines

When a path is stroked, it is possible to request that it is stroked twice, the second time with a different line width and a different color. This is a useful effect for creating "double" lines, for instance by setting the line width to 2pt and stroking a black line and then setting the inner line width to 1pt and stroking a white line on the same path as the original path. This results in what looks like two lines, each of thickness 0.5pt, spaced 1pt apart.

You may wonder why there is direct support for this "double stroking" in the basic layer. After all, this effect is easy to achieve "by hand". The main reason is that arrow tips must be treated in a special manner

when such "double lines" are present. First, the order of actions is important: First, the (thick) main line should be stroked, then the (thin) inner line, and only then should the arrow tip be drawn. Second, the way an arrow tip looks typically depends strongly on the width of the inner line, so the arrow tip code, which is part of the basic layer, needs access to the inner line thickness.

Two commands are used to set the inner line width and color.

$\protect\pro$

This command sets the width of the inner line. Whenever a path is stroked (and only then), it will be stroked normally and, afterward, it is stroked once more with the color set to the inner line color and the line width set to $\langle dimension \rangle$.

In case arrow tips are added to a path, the path is first stroked normally, then the inner line is stroked, and then the arrow tip is added. In case the main path is shortened because of the added arrow tip, this shortened path is double stroked, not the original path (which is exactly what you want).

When the inner line width is set to 0pt, which is the default, no inner line is stroked at all (not even a line of width 0pt). So, in order to "switch off" double stroking, set $\langle dimension \rangle$ to 0pt.

The setting of the inner line width is local to the current TEX group and not to the current PGF scope. Note that inner lines will not be drawn for paths that are also used for clipping. However, this may change in the future, so you should not depend on this.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfsetlinewidth{2pt}
  \pgfsetinnerlinewidth{1pt}
  \pgfusepath{stroke}
  \end{pgfpicture}
```

$\protect\pro$

This command sets the $\langle color \rangle$ that is to be used when the inner line is stroked. The effect of this command is also local to the current T_EX group.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfsetlinewidth{2pt}
  \pgfsetinnerlinewidth{1pt}
  \pgfsetinnerstrokecolor{red!50}
  \pgfusepath{stroke}
  \end{pgfpicture}
```

81.3 Filling a Path

Filling a path means coloring every interior point of the path with the current fill color. It is not always obvious whether a point is "inside" a path when the path is self-intersecting and/or consists or multiple parts. In this case either the nonzero winding number rule or the even-odd crossing number rule is used to decide which points lie "inside." These rules are explained in Section 15.4.

81.3.1 Graphic Parameter: Interior Rule

You can set which rule is used using the following commands:

\pgfseteorule

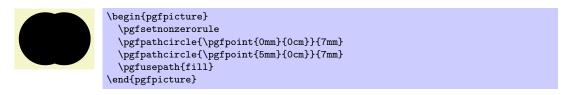
Dictates that the even-odd rule is used in subsequent fillings in the current T_EX scope. Thus, for once, the effect of this command does not persist past the current T_FX scope.



```
\begin{pgfpicture}
  \pgfseteorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

\pgfsetnonzerorule

Dictates that the nonzero winding number rule is used in subsequent fillings in the current TEX scope. This is the default.



81.3.2 Graphic Parameter: Filling Color

$\protect\pro$

Sets the color used for filling paths to $\langle color \rangle$. Like the stroke color, the effect lasts only till the next use of \backslash color.

81.3.3 Graphic Parameter: Fill Opacity

You can set the fill opacity using \pgfsetfillopacity. This command is described in Section 92.

81.4 Clipping a Path

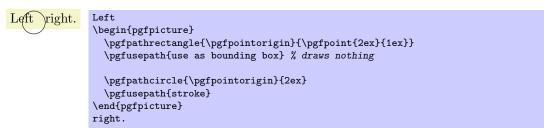
When you add the clip option, the current path is used for clipping subsequent drawings. The same rule as for filling is used to decide whether a point is inside or outside the path, that is, either the even-odd rule or the nonzero rule.

Clipping never enlarges the clipping area. Thus, when you clip against a certain path and then clip again against another path, you clip against the intersection of both.

The only way to enlarge the clipping path is to end the {pgfscope} in which the clipping was done. At the end of a {pgfscope} the clipping path that was in force at the beginning of the scope is reinstalled.

81.5 Using a Path as a Bounding Box

When you add the use as bounding box option, the bounding box of the picture will be enlarged such that the path in encompassed, but any *subsequent* paths of the current TEX scope will not have any effect on the size of the bounding box. Typically, you use this command at the very beginning of a {pgfpicture} environment. Alternatively, you can use \pgfresetboundingbox, followed by \pgfusepath{use as bounding box} to overrule the picture's bounding box completely.



82 Arrow Tips

82.1 Overview

82.1.1 When Does PGF Draw Arrow Tips?

PGF offers an interface for placing arrow tips at the end of lines. The interface works as follows:

1. You (or someone else) assigns a name to a certain kind of arrow tip. For example, the arrow tip latex is the arrow tip used by the standard LaTeX picture environment; the arrow tip to looks like the tip of the arrow in TeX's \to command; and so on.

This is done once at the beginning of the document.

2. Inside some picture, at some point you specify that in the current scope from now on you would like tips of, say, kind to to be added at the end and/or beginning of all paths.

When an arrow kind has been installed and when PGF is about to stroke a path, the following things happen:

- (a) The beginning and/or end of the path is shortened appropriately.
- (b) The path is stroked.
- (c) The arrow tip is drawn at the beginning and/or end of the path, appropriately rotated and appropriately resized.

In the above description, there are a number of occurrences of the word "appropriately". The exact details are not quite trivial and described later on.

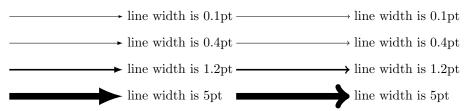
82.1.2 Meta-Arrow Tips

In PGF, arrows are "meta-arrows" in the same way that fonts in TEX are "meta-fonts." When a meta-arrow is resized, it is not simply scaled, but a possibly complicated transformation is applied to the size.

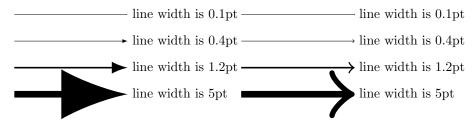
A meta-font is not one particular font at a specific size with a specific stroke width (and with a large number of other parameters being fixed). Rather, it is a "blueprint" (actually, more like a program) for generating such a font at a particular size and width. This allows the designer of a meta-font to make sure that, say, the font is somewhat thicker and wider at very small sizes. To appreciate the difference: Compare the following texts: "Berlin" and "Berlin". The first is a "normal" text, the second is the tiny version scaled by a factor of two. Obviously, the first look better. Now, compare "Berlin" and "Berlin". This time, the normal text was scaled down, while the second text is a "normal" tiny text. The second text is easier to read.

PGF's meta-arrows work in a similar fashion: The shape of an arrow tip can vary according to the line width of the arrow tip being used. Thus, an arrow tip drawn at a line width of 5pt will typically *not* be five times as large as an arrow tip of line width 1pt. Instead, the size of the arrow will get bigger only slowly as the line width increases.

To appreciate the difference, here are the latex and to arrows, as drawn by PGF at four different sizes:



Here, by comparison, are the same arrows when they are simply "resized" (as done by some programs):



As can be seen, simple scaling produces arrow tips that are way too large at larger sizes and way too small at smaller sizes.

In addition to the line width, other options may also influence the appearance of an arrow tip. In particular, the width of the inner line (the line used to create the effect of a double line) influences arrow tips as well as other options that are specific to the arrow tip.

82.2 Declaring an Arrow Tip Kind

To declare an arrow kind "from scratch," the following command is used:

```
\proonup \
```

This command declares a new arrow kind. An arrow kind has two names, which will typically be the same. When the arrow tip needs to be drawn, the $\langle arrow \ tip \ code \rangle$ will be invoked, but the canvas transformation is set up beforehand to a rotation such that when an arrow tip pointing right is specified, the arrow tip that is actually drawn points in the direction of the line.

Naming the arrow kind. The $\langle start\ name \rangle$ is the name used for the arrow tip when it is at the start of a path, the $\langle end\ name \rangle$ is the name used at the end of a path. For example, the arrow kind that looks like a parenthesis has the $\langle start\ name \rangle$ (and the $\langle end\ name \rangle$) so that you can say \pgfsetarrows{(-)} to specify that you want parenthesis arrows and both ends.

The $\langle end \ name \rangle$ and $\langle start \ name \rangle$ can be quite arbitrary and may contain spaces.

Basics of the arrow tip code. Let us next have a look at the $\langle arrow \ tip \ code \rangle$. This code will be used to draw the arrow tip when PGF thinks this is necessary. The code should draw an arrow that "points right," which means that is should draw an arrow at the end of a line coming from the left and ending at the origin.

As an example, suppose we wanted to declare an arrow tip consisting of two arcs, that is, we want the arrow tip to look more or less like the red part of the following picture:

We could use the following as $\langle arrow \ tip \ code \rangle$ for this:

```
\pgfarrowsdeclare{arcs}{arcs}{...}
{
  \pgfsetdash{}{0pt} % do not dash
  \pgfsetroundjoin % fix join
  \pgfsetroundcap % fix cap
  \pgfpathmoveto{\pgfpoint{-10pt}{10pt}}
  \pgfpatharc{180}{270}{10pt}
  \pgfpatharc{90}{180}{10pt}
  \pgfusepathqstroke
}
```

Indeed, when the ... is set appropriately (in a moment), we can write the following:

```
\begin{tikzpicture}
\draw[-arcs,line width=3pt] (-2,0) -- (0,0);
\draw[arcs-arcs,line width=1pt] (-2,-1.5) -- (0,-1);
\useasboundingbox (-2,-2) rectangle (0,0.75);
\end{tikzpicture}
```

As can be seen in the second example, the arrow tip is automatically rotated as needed when the arrow is drawn. This is achieved by a canvas rotation.

Special considerations about the arrow tip code. There are several things you need to be aware of when designing arrow tip code:

- Inside the code, you may not use the \pgfusepath command. The reason is that this command internally calls arrow construction commands, which is something you obviously do not want to happen.
 - Instead of \pgfusepath, use the quick versions. Typically, you will use \pgfusepathqstroke, \pgfusepathqfill, or \pgfusepathqfillstroke.
- The code will be executed only once, namely the first time the arrow tip needs to be drawn. The resulting low-level driver commands are protocolled and stored away. In all subsequent uses of the arrow tip, the protocolled code is directly inserted.
- However, the code will be executed anew for each line width. Thus, an arrow of line width 2pt may result in a different protocol than the same arrow for a line width of 0.4pt.
- If you stroke the path that you construct, you should first set the dashing to solid and set up fixed joins and caps, as needed. This will ensure that the arrow tip will always look the same.
- When the arrow tip code is executed, it is automatically put inside a low-level scope, so nothing will "leak out" from the scope.
- The high-level coordinate transformation matrix will be set to the identity matrix when the code is executed for the first time.

Designing meta-arrows. The $\langle arrow \ tip \ code \rangle$ should adjust the size of the arrow in accordance with the line width. For a small line width, the arrow tip should be small, for a large line width, it should be larger. However, the size of the arrow typically *should not* grow in direct proportion to the line width. On the other hand, the size of the arrow head typically *should* grow "a bit" with the line width.

For these reasons, PGF will not simply execute your arrow code within a scaled scope, where the scaling depends on the line width. Instead, your $\langle arrow \ tip \ code \rangle$ is reexecuted again for each different line width

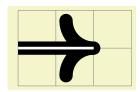
In our example, we could use the following code for the new arrow tip kind arcs' (note the prime):

```
\newdimen\arrowsize
\pgfarrowsdeclare{arcs'}{arcs'}{...}
{
    \arrowsize=0.2pt
    \advance\arrowsize by .5\pgflinewidth
    \pgfsetdash{}{0pt} % do not dash
    \pgfsetroundjoin % fix join
    \pgfsetroundcap % fix cap
    \pgfpathmoveto{\pgfpoint{-4\arrowsize}{4\arrowsize}}
    \pgfpatharc{180}{270}{4\arrowsize}
    \pgfpatharc{90}{180}{4\arrowsize}
    \pgfusepathqstroke
}
```

```
\begin{tikzpicture}
\draw[-arcs',line width=3pt] (-2,0) -- (0,0);
\draw[arcs'-arcs',line width=1pt] (-2,-1.5) -- (0,-1);
\useasboundingbox (-2,-1.75) rectangle (0,0.5);
\end{tikzpicture}
```

However, sometimes, it can also be useful to have arrows that do not resize at all when the line width changes. This can be achieved by giving absolute size coordinates in the code, as done for arc. On the other hand, you can also have the arrow resize linearly with the line width by specifying all coordinates as multiples of \pgflinewidth.

The left and right extend. Let us have another look at the exact left and right "ends" of our arrow tip. Let us draw the arrow tip arc' at a very large size:



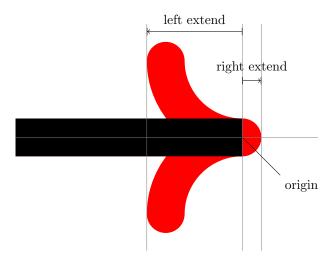
```
\begin{tikzpicture}
  \draw[help lines] (-2,-1) grid (1,1);
  \draw[line width=10pt,-arcs'] (-2,0) -- (0,0);
  \draw[line width=2pt,white] (-2,0) -- (0,0);
  \end{tikzpicture}
```

As one can see, the arrow tip does not "touch" the origin as it should, but protrudes a little over the origin. One remedy to this undesirable effect is to change the code of the arrow tip such that everything is shifted half an \arrowsize to the left. While this will cause the arrow tip to touch the origin, the line itself will then interfere with the arrow: The arrow tip will be partly "hidden" by the line itself.

PGF uses a different approach to solving the problem: The $\langle extend\ code \rangle$ argument can be used to "tell" PGF how much the arrow protrudes over the origin. The argument is also used to tell PGF where the "left" end of the arrow is. However, this number is important only when the arrow is being reversed or composed with other arrow tips.

Once PGF knows the right extend of an arrow kind, it can *shorten* lines by this amount when drawing arrows.

Here is a picture that visualizes the arrow extends. The arrow tip itself is shown in red once more:



The ⟨extend code⟩ is normal TeX code that is executed whenever PGF wants to know how far the arrow tip will protrude to the right and left. The code should call the following two commands: \pgfarrowsrightextend and \pgfarrowsleftextend. Both arguments take one argument that specifies the size. Here is the final code for the arcs' arrow tip:

```
\pgfarrowsdeclare{arcs''}{arcs''}
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfarrowsleftextend{-4\arrowsize-.5\pgflinewidth}
  \pgfarrowsrightextend{.5\pgflinewidth}
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfsetdash{}{Opt} % do not dash
  \pgfsetroundjoin % fix join
                     % fix cap
  \pgfsetroundcap
  \pgfpathmoveto{\pgfpoint{-4\arrowsize}{4\arrowsize}}
  \pgfpatharc{180}{270}{4\arrowsize}
  \pgfusepathqstroke
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{90}{180}{4\arrowsize}
  \pgfusepathqstroke
\begin{tikzpicture}
  \draw[help lines] (-2,-1) grid (1,1);
 \draw[line width=10pt,-arcs',] (-2,0) -- (0,0);
  \draw[line width=2pt,white] (-2,0) -- (0,0);
\end{tikzpicture}
```

Taking inner lines into account. In addition to the line width, there is another parameter that (may) influence the way an arrow looks: The inner line width, which is the line width of the second line that is stroked on top of a normal line in order to create the effect of a "double" line. When this line width changes, the arrow tip code is also reexecuted (and cached), so your code may depend on the current value of the inner line width.

The following example shows how this works. The implies arrow defined below has to set up the line width not for the "main" line width, but for the main line width minus the inner line width, divided by 2.

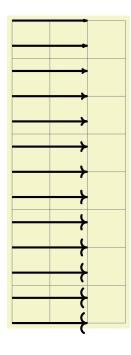
Here is the effect for different combinations of line width and inner line width:

Arrow options. You may wish to have further options to influence the appearance of an arrow tip. For instance, for a "pointed" arrow you may wish to set the opening angle of the tip. Then, whenever this option changes, that arrow tip code also needs to be reexecuted, even though the line width has stayed the same.

You can use the commands \pgfsetarrowoptions and \pgfgetarrowoptions to set such options for an arrow tip. Whenever an arrow tip needs to be rendered, it is checked whether the arrow tip code has already been executed for the current (expanded) value of the options. If so, the cached code is used; otherwise the code is executed once more. Naturally, inside the code the current value of the arrow options should be taken into account.

Arrow options can and must be specified individually for each arrow type.

In the following example, we make the arc angle an option.



```
\pgfarrowsdeclare{var arc}{var arc} % options is an angle
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
 \pgfarrowsleftextend{-4\arrowsize-.5\pgflinewidth}
  \pgfarrowsrightextend{.5\pgflinewidth}
 \arrowsize=0.2pt
 \advance\arrowsize by .5\pgflinewidth
 \pgfsetdash{}{Opt} % do not dash
 \pgfsetroundjoin % fix join
 \pgfsetroundcap
                   % fix cap
  \pgfpathmoveto{\pgfpointorigin}
 \pgfusepathqstroke
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{90}{180-\pgfgetarrowoptions{var arc}}{4\arrowsize}
  \pgfusepathqstroke
\begin{tikzpicture}
 \draw[help lines] (-2,-4) grid (1,4);
  \foreach \option in {-60,-50,...,60}
   \pgfsetarrowoptions{var arc}{\option}
   \draw[ultra thick,-var arc] (-2,\option/15) -- (0,\option/15);
\end{tikzpicture}
```

 $\protect\pro$

Sets the options for the $\langle arrow \ tip \rangle$ to $\langle text \rangle$. The default, before any call to this macro is made, is 0.

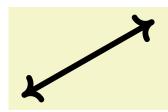
 $\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

This will expand to the current value of the options for the $\langle arrow \ tip \rangle$.

82.3 Declaring a Derived Arrow Tip Kind

It is possible to declare arrow kinds in terms of existing ones. For these command to work correctly, the left and right extends must be set correctly.

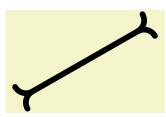
This command can be used to create an alias (another name) for an existing arrow kind.



```
\pgfarrowsdeclarealias{<}{>}{arcs''}{arcs''}%
\begin{tikzpicture}
  \pgfsetarrows{<->}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

 $\protect\pro$

This command creates a new arrow kind that is the "reverse" of an existing arrow kind. The (automatically created) code of the new arrow kind will contain a flip of the canvas and the meanings of the left and right extend will be reversed.



```
\pgfarrowsdeclarereversed{arcs reversed}{arcs';}{arcs';}%
\begin{tikzpicture}
  \pgfsetarrows{arcs reversed-arcs reversed}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

 $\propto \propto \pro$

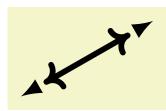
This command creates a new arrow kind that combines two existing arrow kinds. The first arrow kind is the "innermost" arrow kind, the second arrow kind is the "outermost."

The code for the combined arrow kind will install a canvas translation before the innermost arrow kind in drawn. This translation is calculated such that the right tip of the innermost arrow touches the right end of the outermost arrow. The optional $\langle offset \rangle$ can be used to increase (or decrease) the distance between the inner and outermost arrow.

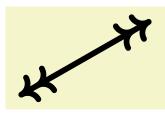


```
\pgfarrowsdeclarecombine[\pgflinewidth]
  {combined}{combined}{arcs''}{arcs''}{latex}{
\begin{tikzpicture}
  \pgfsetarrows{combined-combined}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
  \end{tikzpicture}
```

In the star variant, the end of the line is not in the outermost arrow, but inside the innermost arrow.



```
\pgfarrowsdeclarecombine*[\pgflinewidth]
  {combined'}{combined'}{arcs''}{arcs''}{latex}{
\begin{tikzpicture}
  \pgfsetarrows{combined'-combined'}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```



```
\pgfarrowsdeclaredouble{<<}{>>}{arcs''}{arcs''}%
\begin{tikzpicture}
  \pgfsetarrows{<<->>}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

 $\proonup \proonup \$



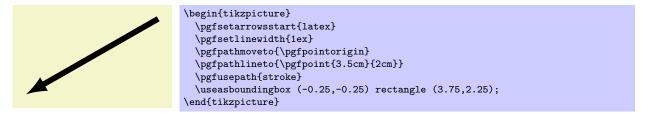
```
\pgfarrowsdeclaretriple{<<<}{>>>>}{arcs''}{arcs''}%
\begin{tikzpicture}
  \pgfsetarrows{<<<->>>}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

82.4 Using an Arrow Tip Kind

The following commands install the arrow kind that will be used when stroking is done.

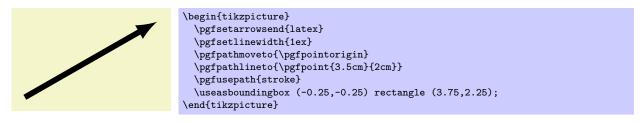
 $\protect\pro$

Installs the given $\langle start \ arrow \ kind \rangle$ for all subsequent strokes in the current TEX-group. If $\langle start \ arrow \ kind \rangle$ is empty, no arrow tips will be drawn at the start of the last segment of paths.



$\protect\pro$

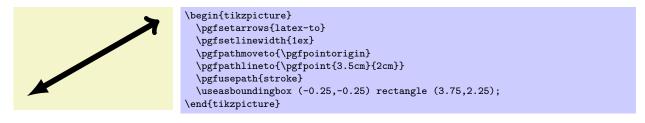
Like \pgfsetarrowsstart, only for the end of the arrow.



Warning: If the compatibility mode is active (which is the default), there also exist old commands called \pgfsetstartarrow and \pgfsetendarrow, which are incompatible with the meta-arrow management.

 $\protect\pro$

Calls \pgfsetarrowsstart for $\langle start \ kind \rangle$ and \pgfsetarrowsend for $\langle end \ kind \rangle$.



82.5 Predefined Arrow Tip Kinds

The following arrow tip kinds are always defined:

```
stealth-stealth stealth reversed yields thick and thin stealth reversed-stealth reversed yields thick and thin stealth reversed yields
```

83 Nodes and Shapes

This section describes the shapes module.

```
\usepgfmodule{shapes} % MEX and plain TeX and pure pgf \usepgfmodule[shapes] % ConTeXt and pure pgf
```

This module defines commands both for creating nodes and for creating shapes. The package is loaded automatically by pgf, but you can load it manually if you have only included pgfcore.

83.1 Overview

PGF comes with a sophisticated set of commands for creating *nodes* and *shapes*. A *node* is a graphical object that consists (typically) of (one or more) text labels and some additional stroked or filled paths. Each node has a certain *shape*, which may be something simple like a rectangle or a circle, but it may also be something complicated like a uml class diagram (this shape is currently not implemented, though). Different nodes that have the same shape may look quite different, however, since shapes (need not) specify whether the shape path is stroked or filled.

83.1.1 Creating and Referencing Nodes

You create a node by calling the macro \pgfnode or the more general \pgfmultipartnode. This macro takes several parameters and draws the requested shape at a certain position. In addition, it will "remember" the node's position within the current {pgfpicture}. You can then, later on, refer to the node's position. Coordinate transformations are "fully supported," which means that if you used coordinate transformations to shift or rotate the shape of a node, the node's position will still be correctly determined by PGF. This is not the case if you use canvas transformations instead.

83.1.2 Anchors

An important property of a node or a shape in general are its *anchors*. Anchors are "important" positions in a shape. For example, the center anchor lies at the center of a shape, the north anchor is usually "at the top, in the middle" of a shape, the text anchor is the lower left corner of the shape's text label (if present), and so on.

Anchors are important both when you create a node and when you reference it. When you create a node, you specify the node's "position" by asking PGF to place the shape in such a way that a certain anchor lies at a certain point. For example, you might ask that the node is placed such that the north anchor is at the origin. This will effectively cause the node to be placed below the origin.

When you reference a node, you always reference an anchor of the node. For example, when you request the "north anchor of the node just placed" you will get the origin. However, you can also request the "south anchor of this node," which will give you a point somewhere below the origin. When a coordinate transformation was in force at the time of creation of a node, all anchors are also transformed accordingly.

83.1.3 Layers of a Shape

The simplest shape, the coordinate, has just one anchor, namely the center, and a label (which is usually empty). More complicated shapes like the rectangle shape also have a *background path*. This is a PGF-path that is defined by the shape. The shape does not prescribe what should happen with the path: When a node is created, this path may be stroked (resulting in a frame around the label), filled (resulting in a background color for the text), or just discarded.

Although most shapes consist just of a background path plus some label text, when a shape is drawn, up to seven different layers are drawn:

- 1. The "behind the background layer." Unlike the background path, which can be used in different ways by different nodes, the graphic commands given for this layer will always stroke or always fill the path they construct. They might also insert some text that is "behind everything."
- 2. The background path layer. How this path is used depends on the arguments of the \pgfnode command.

- 3. The "before the background path layer." This layer works like the first one, only the commands of this layer are executed after the background path has been used (in whatever way the creator of the node chose).
- 4. The label layer. This layer inserts the node's text box(es).
- 5. The "behind the foreground layer." This layer, like the first layer, once more contains graphic commands that are "simply executed."
- 6. The foreground path layer. This path is treated in the same way as the background path, only it is drawn after the label text has been drawn.
- 7. The "before the foreground layer."

Which of these layers are actually used depends on the shape.

83.1.4 Node Parts

A shape typically does not consist only of different background and foreground paths, but it may also have text labels. Indeed, for many shapes the text labels are the more important part of the shape.

Most shapes will have only one text label. In this case, this text label is simply passed as a parameter to the \pgfnode command. When the node is drawn, the text label is shifted around such that its lower left corner is at the text anchor of the node.

More complicated shapes may have more than one text label. Nodes of such shapes are called *multipart* nodes. The different node parts are simply the different text labels. For example, a uml class shape might have a class name part, a method part and an attributes part. Indeed, single part nodes are a special case of multipart nodes: They only have one part named text.

When a shape is declared, you must specify the node parts. There is a simple command called \nodeparts that takes a list of the part names as input. When you create a node of a multipart shape, for each part of the node you must have set up a TEX-box containing the text of the part. For a part named XYZ you must set up the box \pgfnodepartXYZbox. The box will be placed at the anchor XYZ. See the description of \pgfmultipartnode for more details.

83.2 Creating Nodes

83.2.1 Creating Simple Nodes

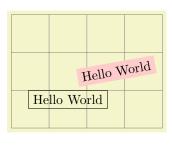
 $\label{label text} $$ \left(\operatorname{shape} \right) = \left(\operatorname{anchor} \right) = \left(\operatorname{label text} \right) = \left(\operatorname{name} \right)$

This command creates a new node. The $\langle shape \rangle$ of the node must have been declared previously using $\protect\prote$

The shape is shifted such that the $\langle anchor \rangle$ is at the origin. In order to place the shape somewhere else, use the coordinate transformation prior to calling this command.

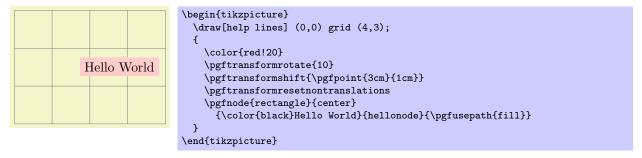
The $\langle name \rangle$ is a name for later reference. If no name is given, nothing will be "saved" for the node, it will just be drawn.

The $\langle path \ usage \ command \rangle$ is executed for the background and the foreground path (if the shape defines them).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \pgftransformshift{\pgfpoint{1.5cm}{1cm}}
    \pgfnode{rectangle}{north}{Hello World}{hellonode}{\pgfusepath{stroke}}
}
  {
    \color{red!20}
    \pgftransformrotate{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgfnode{rectangle}{center}
     {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
}
}end{tikzpicture}
```

As can be seen, all coordinate transformations are also applied to the text of the shape. Sometimes, it is desirable that the transformations are applied to the point where the shape will be anchored, but you do not wish the shape itself to the transformed. In this case, you should call \pgftransformresetnontranslations prior to calling the \pgfnode command.



The $\langle label\ text \rangle$ is typeset inside the TeX-box \pgfnodeparttextbox. This box is shown at the text anchor of the node, if the node has a text part. See the description of \pgfmultipartnode for details.

83.2.2 Creating Multi-Part Nodes

```
\proonup \
```

This command is the more general (and less user-friendly) version of the \pgfnode command. While the \pgfnode command can only be used for shapes that have a single part (which is the case for most shapes), this command can also be used with multi-part nodes.

When this command is called, for each node part of the node you must have set up one TEX-box. Suppose the shape has two parts: The text part and the lower part. Then, prior to calling \pgfmultipartnode, you must have set up the boxes \pgfnodeparttextbox and \pgfnodepartlowerbox. These boxes may contain any TEX-text. The shape code will then compute the positions of the shape's anchors based on the sizes of the these shapes. Finally, when the node is drawn, the boxes are placed at the anchor positions text and lower.

Note: Be careful when using the \setbox command inside a {pgfpicture} command. You will have to use \pgfinterruptpath at the beginning of the box and \endpgfinterruptpath at the end of the box to make sure that the box is typeset correctly. In the above example this problem was sidestepped by moving the box construction outside the environment.

Note: It is not necessary to use \newbox for every node part name. Although you need a different box for each part of a single shape, two different shapes may very well use the same box even when the names of the parts are different. Suppose you have a circle split shape that has a lower part and you have a uml class shape that has a methods part. Then, in order to avoid exhausting TEX's limited number of box registers, you can say

```
\newbox\pgfnodepartlowerbox
\let\pgfnodepartmethodsbox=\pgfnodepartlowerbox
```

Also, when you have a node part name with spaces like class name, it may be useful to create an alias:

```
\newbox\mybox
\expandafter\let\csname pgfnodepartclass namebox\endcsname=\mybox
```

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

This command creates a node of shape **coordinate** at the given $\langle coordinate \rangle$. Exactly the same effect can be achieved using first a shift of the coordinate system to $\langle coordinate \rangle$, followed by creating a node of shape **coordinate** named $\langle name \rangle$. However, this command is easier and more natural to use and, more importantly, it is much faster.

```
\protect\operatorname{pgfnodealias}\{\langle new\ name \rangle\}\{\langle existing\ node \rangle\}
```

This command does not actually create a new node. Rather, it allows you to subsequently access the node $\langle existing\ node \rangle$ using the name $\langle new\ name \rangle$.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\be
```

This command renames an existing node.

There are a number of values that have an influence on the size of a node. These values are stored in the following keys.

```
/pgf/minimum width=\langle dimension \rangle
```

(no default, initially 1pt)

alias /tikz/minimum width

This key stores the *recommended* minimum width of a shape. Thus, when a shape is drawn and when the shape's width would be smaller than $\langle dimension \rangle$, the shape's width is enlarged by adding some empty space.

Note that this value is just a recommendation. A shape may choose to ignore this key.



```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

\pgfset{minimum width=3cm}
  \pgfnode{rectangle}{center}{Hello World}{}\pgfusepath{stroke}}

\end{tikzpicture}
```

 $/pgf/minimum height=\langle dimension \rangle$

(no default, initially 1pt)

alias /tikz/minimum height

Works like /pgf/minimum width.

/pgf/minimum size= $\langle dimension \rangle$

(no default)

alias /tikz/minimum size

This style both /pgf/minimum width and /pgf/minimum height to $\langle dimension \rangle$.

```
/pgf/inner xsep=\langle dimension \rangle
```

(no default, initially 0.3333em)

alias /tikz/inner xsep

This key stores the *recommended* horizontal inner separation between the label text and the background path. As before, this value is just a recommendation and a shape may choose to ignore this key.



```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

\pgfset{inner xsep=1cm}
  \pgfnode{rectangle}{center}{Hello World}{}{\pgfusepath{stroke}}

\end{tikzpicture}
```

 $/pgf/inner ysep=\langle dimension \rangle$

(no default, initially 0.3333em)

 ${
m alias}$ /tikz/inner ysep

Works like /pgf/inner xsep.

/pgf/inner sep= $\langle dimension \rangle$

(no default)

alias /tikz/inner sep

This style sets both /pgf/inner xsep and /pgf/inner ysep to $\langle dimension \rangle$.

```
/pgf/outer xsep=\langle dimension \rangle
```

(no default, initially .5\pgflinewidth)

alias /tikz/outer xsep

This key stores the recommended horizontal separation between the background path and the "outer anchors." For example, if $\langle dimension \rangle$ is 1cm then the east anchor will be 1cm to the right of the right border of the background path. As before, this value is just a recommendation.



```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

\pgfset{outer xsep=.5cm}
  \pgfnode{rectangle}{center}{Hello World}{x}{\pgfusepath{stroke}}

\pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfusepath{fill}
  \end{tikzpicture}
```

```
/pgf/outer ysep=\langle dimension\rangle
alias /tikz/outer ysep
Works like /pgf/outer xsep.
/pgf/outer sep=\langle dimension\rangle
(no default, initially .5\pgflinewidth)
(no default)
```

alias /tikz/outer sep

This style sets both /pgf/outer xsep and /pgf/outer ysep to $\langle dimension \rangle$.

83.2.3 Deferred Node Positioning

Normally, when a node is created using a command like \pgfnode, the node is immediately inserted into the current picture. In particular, you have no chance to change the position of a created node after it has been created. Using \pgfpositionnodelater in concert with \pgfpositionnodenow, you can create a node whose position is determined only at some later time.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

This command is not a replacement for \pgfnode. Rather, when this command is used in a scope, all subsequent node creations in this scope will be affected in the following way: When a node is created, it is not inserted into the current picture. Instead, it is stored in the box \pgfpositionnodelaterbox. Furthermore, the node is not relevant for the picture's bounding box, but a bounding box for the node is computed and stored in the macros \pgfpositionnodelaterminx to \pgfpositionnodelatermaxy. Then, the \(\macro name \) is called with the following macros set up:

\pgfpositionnodelaterbox

A box register number (0 currently) that stores the node's paths and texts. You should move the contents of this box to a box of your choice inside $\langle macro\ name \rangle$.

\pgfpositionnodelatername

The name of the just-created-node. This name will be the originally "desired" name of the box plus the fixed prefix not yet positionedPGFINTERNAL. The idea is to ensure that the original name is not inadvertently used before the node is actually positioned. When \pgfpositionnodenow is called, it will change the name to the original name.

\pgfpositionnodelaterminx

The minimal x-position of a bounding box of the node. This bounding box refers to the node when it is positioned with the anchor at the origin. It is guaranteed, that this macro will contain a dimension in the format $\langle number \rangle pt$.

\pgfpositionnodelaterminy

\pgfpositionnodelatermaxx

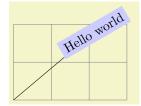
\pgfpositionnodelatermaxy

Once a late node has been created, you can add arbitrary code in the same picture. Then, at some later point, you call \points positionnodenow to finally position the node at a given position. At this point, the above macros must have the exact same values they had when $\mbox{$\langle macro\ name \rangle$}$ was called. Note that the above macros are local to a scope that ends right after the call to $\mbox{$\langle macro\ name \rangle$}$, so it is your job to copy the values to safety inside $\mbox{$\langle macro\ name \rangle$}$.

By setting $\langle macro\ name \rangle$ to $\$ to $\$ which is the default), you can switch off the whole mechanism. When a picture is interrupted, this is done automatically.

$\protect\pro$

This command is used to position a node that has previously been created using the command \pgfpositionnodelater . When \pgfpositionnodenow is called, the macros and boxes mentioned in the description of \pgfpositionnodenow must be set to the value they had when the $\protect\mathbb{macro}$ name was called. Provided this is the case, this command will insert the box into the current picture, shifted by $\protect\mathbb{coordinate}$. Then, the late code (see below) is called. Subsequently, you can refer to the node with its original name as if it had just been created.



```
\newbox\mybox
\def\mvsaver{
  \global\setbox\mybox=\box\pgfpositionnodelaterbox
  \global\let\myname=\pgfpositionnodelatername
  \global\let\myminx=\pgfpositionnodelaterminx
  \global\let\myminy=\pgfpositionnodelaterminy
  \global\let\mymaxx=\pgfpositionnodelatermaxx
  \global\let\mymaxy=\pgfpositionnodelatermaxy
\begin{tikzpicture}
    \pgfpositionnodelater{\mysaver}
    \node [fill=blue!20,below,rotate=30] (hi) {Hello world};
  \draw [help lines] (0,0) grid (3,2);
 \let\pgfpositionnodelatername=\myname
  \let\pgfpositionnodelaterminx=\myminx
 \let\pgfpositionnodelaterminy=\myminy
  \let\pgfpositionnodelatermaxx=\mymaxx
  \let\pgfpositionnodelatermaxy=\mymaxy
 \setbox\pgfpositionnodelaterbox=\box\mybox
  \pgfpositionnodenow{\pgfqpoint{2cm}{2cm}}
 \draw (hi) -- (0,0);
\end{tikzpicture}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

When you call this macro inside a scope for which the $\protect{\protect}{\protect}$ will be stored internally. Later, when the node named \protect is actually positioned using \protect pgfpositionnodenow, the \protect will be executed. When this macro is called multiple times with the same \protect name, the \protect accumulates. However, When \protect pgfpositionnodenow is called, the code stored for the node is cleared.

The main purpose of this mechanism is to allow TikZ to store so-called "late options" with a node that will be positioned only later.

83.3 Using Anchors

Each shape defines a set of anchors. We saw already that the anchors are used when the shape is drawn: the shape is placed in such a way that the given anchor is at the origin (which in turn is typically translated somewhere else).

One has to look up the set of anchors of each shape, there is no "default" set of anchors, except for the center anchor, which should always be present. Also, most shapes will declare anchors like north or east, but this is not guaranteed.

83.3.1 Referencing Anchors of Nodes in the Same Picture

Once a node has been defined, you can refer to its anchors using the following commands:

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

This command is another "point command" like the commands described in Section 78. It returns the coordinate of the given $\langle anchor \rangle$ in the given $\langle node \rangle$. The command can be used in commands like pgfpathmoveto.



```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
  \end{pgfpicture}
```

In the above example, you may have noticed something curious: The rotation transformation is still in force when the anchors are invoked, but it does not seem to have an effect. You might expect that the rotation should apply to the already rotated points once more.

However, \pgfpointanchor returns a point that takes the current transformation matrix into account: The inverse transformation to the current coordinate transformation is applied to an anchor point before returning it.

This behavior may seem a bit strange, but you will find it very natural in most cases. If you really want to apply a transformation to an anchor point (for example, to "shift it away" a little bit), you have to invoke \pgfpointanchor without any transformations in force. Here is an example:



```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

{
    \pgftransformreset
    \pgfpointanchor{x}{east}
    \xdef\mycoordinate{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}
}

\pgfpathcircle{\mycoordinate}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

A special situation arises when the $\langle node \rangle$ lies in a picture different from the current picture. In this case, if you have not told PGF that the picture should be "remembered," the $\langle node \rangle$ will be treated as if it lay in the current picture. For example, if the $\langle node \rangle$ was at position (3, 2) in the original picture, it is treated as if it lay at position (3, 2) in the current picture. However, if you have told PGF to remember the picture position of the node's picture and also of the current picture, then \pgfpointanchor will return a coordinate that corresponds to the position of the node's anchor on the page, transformed into the current coordinate system. For examples and more details see Section 83.3.2.

$\protect\pro$

This command returns the point on the border of the shape that lies on a straight line from the center of the node to $\langle point \rangle$. For complex shapes it is not guaranteed that this point will actually lie on the border, it may be on the border of a "simplified" version of the shape.



```
\begin{pgfpicture}
\begin{pgfscope}
\pgftransformrotate{30}
\pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
\end{pgfscope}
\pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{2cm}{1cm}}}{2pt}
\pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}}{2pt}
\pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}}{2pt}
\pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}}{2pt}
\pgfpathcircle{\pgfpoint{-1cm}}{2pt}
\pgfusepath{fill}
\end{pgfpicture}
```

83.3.2 Referencing Anchors of Nodes in Different Pictures

As a picture is typeset, PGF keeps track of the positions of all nodes inside the picture. What PGF does not remember is the position of the picture *itself* on the page. Thus, if you define a node in one picture and then

try to reference this node while another picture is typeset, PGF will only know the position of the nodes that you try to typeset inside the original picture, but it will not know where this picture lies. What is missing is the relative positioning of the two pictures.

To overcome this problem, you need to tell PGF that it should remember the position of pictures on a page. If these positions are remembered, then PGF can compute the offset between the pictures and make nodes in different pictures accessible.

Determining the positions of pictures on the page is, alas, not-so-easy. Because of this, PGF does not do so automatically. Rather, you have to proceed as follows:

- 1. You have to use a backend driver that supports position tracking. pdfTEX is one such driver, dvips currently is not.
- 2. You have to say \pgfrememberpicturepositiononpagetrue somewhere before or inside every picture
 - in which you wish to reference a node and
 - from which you wish to reference a node in another picture.

The second item is important since PGF does not only need to know the position of the picture in which the node you wish to reference lies, but it also needs to know where the current picture lies.

- 3. You typically have to run TEX twice (depending on the backend driver) since the position information typically gets written into an external file on the first run and is available only on the second run.
- 4. You have to switch off automatic bounding bound computations. The reason is that the node in the other picture should not influence the size of the bounding box of the current picture. You should say \pgfusepath{use as bounding box} before using a coordinate in another picture.

83.4 Special Nodes

There are several special nodes that are always defined and which you should not attempt to redefine.

Predefined node current bounding box

This node is of shape rectangle. Unlike normal nodes, its size changes constantly and always reflects the size of the bounding box of the current picture. This means that, for instance, that

```
\pgfpointanchor{current bounding box}{south east}
```

returns the lower left corner of the bounding box of the current picture.

Predefined node current path bounding box

This node is also of shape rectangle. Its size is the size of the bounding box of the current path.

Predefined node current page

This node is inside a virtual remembered picture. The size of this node is the size of the current page. This means that if you create a remembered picture and inside this picture you reference an anchor of this node, you reference an absolute position on the page. To demonstrate the effect, the following code puts some text in the lower left corner of the current page. Note that this works only if the backend driver supports it, otherwise the text is inserted right here.

```
\pgfrememberpicturepositiononpagetrue
\begin{pgfpicture}
  \pgfusepath{use as bounding box}
  \pgftransformshift{\pgfpointanchor{current page}{south west}}
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \pgftext[left,base]{
    \textcolor{red}{
      Text absolutely positioned in the lower left corner.}
  }
} \end{pgfpicture}
```

There is also an option that allows you to create new special nodes quite similar to the above:

```
/pgf/local bounding box=\langle node \ name \rangle
```

(no default)

alias /tikz/local bounding box

This defines a new node $\langle node\ name \rangle$ whose size is the bounding box around all objects in the current scope starting at the position where this option was given. After the end of the scope, the $\langle node\ name \rangle$ is still available. You can use this option to keep track of the size of a certain area. Note that excessive use of this option (keeping track of dozens of bounding boxes at the same time) will slow things down.

```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  { [local bounding box=outer box]
     \draw (1,1) circle (.5) [local bounding box=inner box] (2,2) circle (.5);
  }
  \draw (outer box.south west) rectangle (outer box.north east);
  \draw[red] (inner box.south west) rectangle (inner box.north east);
  \end{tikzpicture}
```

83.5 Declaring New Shapes

There are only three predefined shapes, see Section 49.2, so there must be some way of defining new shapes. Defining a shape is, unfortunately, a not-quite-trivial process. The reason is that shapes need to be both very flexible (their size will vary greatly according to circumstances) and they need to be constructed reasonably "fast." PGF must be able to handle pictures with several hundreds of nodes and documents with thousands of nodes in total. It would not do if PGF had to compute and store, say, dozens of anchor positions for every node.

83.5.1 What Must Be Defined For a Shape?

In order to define a new shape, you must provide:

- a shape name,
- code for computing the saved anchors and saved dimensions,
- code for computing *anchor* positions in terms of the saved anchors,
- optionally code for the background path and foreground path,
- optionally code for things to be drawn before or behind the background and foreground paths.
- optionally a list of node parts.

83.5.2 Normal Anchors Versus Saved Anchors

Anchors are special places in a shape. For example, the north east anchor, which is a normal anchor, lies at the upper right corner of the rectangle shape, as does \northeast, which is a saved anchor. The difference is the following: saved anchors are computed and stored for each node, anchors are only computed as needed. The user only has access to the normal anchors, but a normal anchor can just "copy" or "pass through" the location of a saved anchor.

The idea behind all this is that a shape can declare a very large number of normal anchors, but when a node of this shape is created, these anchors are not actually computed. However, this causes a problem: When we wish to reference an anchor of a node at some later time, we must still be able to compute the position of the anchor. For this, we may need a lot of information: What was the transformation matrix that was in force when the node was created? What was the size of the text box? What were the values of the different separation dimensions? And so on.

To solve this problem, PGF will always compute the locations of all *saved anchors* and store these positions. Then, when an normal anchor position is requested later on, the anchor position can be given just from knowing where the locations of the saved anchors are.

As an example, consider the rectangle shape. For this shape two anchors are saved: The \northeast corner and the \southwest corner. A normal anchor like north west can now easily be expressed in terms of these coordinates: Take the x-position of the \southwest point and the y-position of the \northeast point. The rectangle shape currently defines 13 normal anchors, but needs only two saved anchors. Adding new anchors like a south south east anchor would not increase the memory and computation requirements of pictures.

All anchors (both saved and normal) are specified in a local *shape coordinate space*. This is also true for the background and foreground paths. The \pgfnode macro will automatically apply appropriate transformations to the coordinates so that the shape is shifted to the right anchor or otherwise transformed.

83.5.3 Command for Declaring New Shapes

The following command declares a new shape:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command declares a new shape named $\langle shape \ name \rangle$. The shape name can later be used in commands like \pgfnode.

The $\langle shape\ specification \rangle$ is some TEX code containing calls to special commands that are only defined inside the $\langle shape\ specification \rangle$ (similarly to commands like \draw that are only available inside the \text{tikzpicture} environment).

Example: Here is the code of the coordinate shape:

```
\pgfdeclareshape{coordinate}
{
  \savedanchor\centerpoint{%
  \pgf@x=.5\wd\pgfnodeparttextbox%
  \pgf@y=.5\ht\pgfnodeparttextbox%
  \advance\pgf@y by -.5\dp\pgfnodeparttextbox%
}
\anchor{center}{\centerpoint}
  \anchorborder{\centerpoint}
}
```

The special commands are explained next. In the examples given for the special commands a new shape will be constructed, which we might call simple rectangle. It should behave like the normal rectangle shape, only without bothering about the fine details like inner and outer separations. The skeleton for the shape is the following.

```
\pgfdeclareshape{simple rectangle}{
   ...
}
```

$\nodeparts{\langle list \ of \ node \ parts\rangle}$

This command declares which parts make up nodes of this shape. A *node part* is a (possibly empty) text label that is drawn when a node of the shape is created.

By default, a shape has just one node part called text. However, there can be several node parts. For example, the circle split shape has two parts: the text part, which shows that the upper text, and a lower part, which shows the lower text. For the circle split shape the \nodeparts command was called with the argument {text,lower}.

When a multipart node is created, the text labels are drawn in the sequences listed in the *\lambda list of node parts*\rangle. For each node part, you must have declared one anchor and the TeX-box of the part is placed at this anchor. For a node part called XYZ the TeX-box \pgfnodepartXYZbox is placed at anchor XYZ.

```
\scalebox{} \sca
```

This command declares a saved anchor. The argument $\langle command \rangle$ should be a TEX macro name like \centerpoint.

The $\langle code \rangle$ will be executed each time \pgfnode (or \pgfmultipartnode) is called to create a node of the shape $\langle shape\ name \rangle$. When the $\langle code \rangle$ is executed, the TeX-boxes of the node parts will contain the text labels of the node. Possibly, these box are void. For example, if there is just a text part, the node \pgfnodeparttextbox will be set up when the $\langle code \rangle$ is executed.

The $\langle code \rangle$ can use the width, height, and depth of the box(es) to compute the location of the saved anchor. In addition, the $\langle code \rangle$ can take into account the values of dimensions like $\protect{\protect\pr$

The net effect of the $\langle code \rangle$ should be to set the two TEX dimensions $\pgf@x$ and $\pgf@y$. One way to achieve this is to say $\pgfpoint{\langle x\ value \rangle}{\langle y\ value \rangle}$ at the end of the $\langle code \rangle$, but you can also just set these variables. The values that $\pgf@x$ and $\pgf@y$ have after the code has been executed, let us call them x and y, will be recorded and stored together with the node that is created by the command \pgfnode .

The macro $\langle command \rangle$ is defined to be $\pgfpoint\{x\}\{y\}$. However, the $\langle command \rangle$ is only locally defined while anchor positions are being computed. Thus, it is possible to use very simple names for $\langle command \rangle$, like $\c enter$ or $\a vert$, without causing a name-clash. (To be precise, very simple $\langle command \rangle$ names will clash with existing names, but only locally inside the computation of anchor positions; and we do not need the normal $\c enter$ command during these computations.)

For our simple rectangle shape, we will need only one saved anchor: The upper right corner. The lower left corner could either be the origin or the "mirrored" upper right corner, depending on whether we want the text label to have its lower left corner at the origin or whether the text label should be centered on the origin. Either will be fine, for the final shape this will make no difference since the shape will be shifted anyway. So, let us assume that the text label is centered on the origin (this will be specified later on using the text anchor). We get the following code for the upper right corner:

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.5\ht\pgfnodeparttextbox % height of the box, ignoring the depth
  \pgf@x=.5\wd\pgfnodeparttextbox % width of the box
}
```

If we wanted to take, say, the \pgfshapeminwidth into account, we could use the following code:

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.\ht\pgfnodeparttextbox % height of the box
  \pgf@x=.\wd\pgfnodeparttextbox % width of the box
  \setlength{\pgf@xa}{\pgfshapeminwidth}
  \ifdim\pgf@x<.5\pgf@xa
  \pgf@x=.5\pgf@xa
  \fi
}</pre>
```

Note that we could not have written .5\pgfshapeminwidth since the minimum width is stored in a "plain text macro," not as a real dimension. So if \pgfshapeminwidth depth were 2cm, writing .5\pgfshapeminwidth would yield the same as .52cm.

In the "real" rectangle shape the code is somewhat more complex, but you get the basic idea.

This command is similar to \savedanchor, only instead of setting $\langle command \rangle$ to \pgfpoint{x}{y}, the $\langle command \rangle$ is set just to (the value of) x.

In the simple rectangle shape we might use a saved dimension to store the depth of the shape box.

```
\saveddimen{\depth}{
  \pgf@x=\dp\pgfnodeparttextbox
}
```

```
\scalebox{$\scalebox{$\scalebox{$\sim$}} (code)$}
```

This command is similar to \saveddimen, only at some point in $\langle code \rangle$, $\langle command \rangle$ should be defined appropriately, (this could be a value, or some text).

In the regular polygon shape, a saved macro is used to store the number of sides of the polygon.

```
\savedmacro{\sides}{\let\sides\pgfpolygonsides}
```

```
\n
```

This command declares an anchor named $\langle name \rangle$. Unlike for saved anchors, the $\langle code \rangle$ will not be executed each time a node is declared. Rather, the $\langle code \rangle$ is only executed when the anchor is specifically requested; either for anchoring the node during its creation or as a position in the shape referenced later on.

The $\langle name \rangle$ is a quite arbitrary string that is not "passed down" to the system level. Thus, names like south or 1 or :: would all be fine.

A saved anchor is not automatically also a normal anchor. If you wish to give the users access to a saved anchor you must declare a normal anchor that just returns the position of the saved anchor.

When the $\langle code \rangle$ is executed, all saved anchor macros will be defined. Thus, you can reference them in your $\langle code \rangle$. The effect of the $\langle code \rangle$ should be to set the values of \pgf@x and \pgf@y to the coordinates of the anchor.

Let us consider some example for the simple rectangle shape. First, we would like to make the upper right corner publicly available, for example as north east:

\anchor{north east}{\upperrightcorner}

The \upperrightcorner macro will set \pgf@x and \pgf@y to the coordinates of the upper right corner. Thus, \pgf@x and \pgf@y will have exactly the right values at the end of the anchor's code. Next, let us define a north west anchor. For this anchor, we can negate the \pgf@x variable:

```
\anchor{north west}{
  \upperrightcorner
  \pgf@x=-\pgf@x
}
```

Finally, it is a good idea to always define a center anchor, which will be the default location for a shape.

\anchor{center}{\pgfpointorigin}

You might wonder whether we should not take into consideration that the node is not placed at the origin, but has been shifted somewhere. However, the anchor positions are always specified in the shape's "private" coordinate system. The "outer" transformation that has been applied to the shape upon its creation is applied automatically to the coordinates returned by the anchor's $\langle code \rangle$.

Our simple rectangle only has one text label (node part) called text. This is the default situation, so we do not need to do anything. For the text node part we must set up a text anchor. This anchor is used upon creation of a node to determine the lower left corner of the text label (within the private coordinate system of the shape). By default, the text anchor is at the origin, but you may change this. For example, we would say

```
\anchor{text}{%
  \upperrightcorner%
  \pgf@x=-\pgf@x%
  \pgf@y=-\pgf@y%
}
```

to center the text label on the origin in the shape coordinate space. Note that we could not have written the following:

Do you see why this is wrong? The problem is that the box \pgfnodeparttextbox will most likely not have the correct size when the anchor is computed. After all, the anchor position might be recomputed at a time when several other nodes have been created.

If a shape has several node parts, we would have to define an anchor for each part.

```
\deferredanchor{\langle name \rangle} {\langle code \rangle}
```

This command declares an anchor named $\langle name \rangle$. It works like \anchor. However, unlike for anchors declared by \anchor, $\langle name \rangle$ will not be expanded during the shape declaration (i.e. not during \pgfdeclareshape). Rather, the $\langle name \rangle$ is expanded when the node is actually used (with \pgfnode or more likely with \node). This may be useful if the anchor name is context dependent (depending, for example, on the value of a key).

```
\makeatletter
\def\foo{foo}
\pgfdeclareshape{simple shape}{%
  \savedanchor{\center}{%
    \pgfpointorigin}
  \anchor{center}{\center}
  \savedanchor{\anchorfoo}{%
    \pgf@x=1cm
    \pgf@y=0cm}
 \deferredanchor{anchor \foo}{\anchorfoo}}
\begin{tikzpicture}
 \node[simple shape] (Test1) at (0,0) {};
 \fill (Test1.anchor foo) circle (2pt) node[below] {anchor foo anchor};
 \def\foo{bar}
 \node[simple shape] (Test2) at (2,2) {};
 \fill (Test2.anchor bar) circle (2pt) node[below] {anchor bar anchor};
\end{tikzpicture}
```


A border anchor is an anchor point on the border of the shape. What exactly is considered as the "border" of the shape depends on the shape.

When the user requests a point on the border of the shape using the \protect

For our simple rectangle we must compute a point on the border of a rectangle whose one corner is the origin (ignoring the depth for simplicity) and whose other corner is \upperrightcorner. The following code might be used:

```
\anchorborder{%
    % Call a function that computes a border point. Since this
    % function will modify dimensions like \pgf@x, we must move them to
    % other dimensions.
    \@tempdima=\pgf@x
    \@tempdimb=\pgf@y
    \pgfpointborderrectangle{\pgfpoint{\@tempdima}{\upperrightcorner}}
}
```

$\begin{tabular}{ll} \begin{tabular}{ll} \beg$

This command specifies the path that "makes up" the background of the shape. Note that the shape cannot prescribe what is going to happen with the path: It might be drawn, shaded, filled, or even thrown away. If you want to specify that something should "always" happen when this shape is drawn (for example, if the shape is a stop-sign, we *always* want it to be filled with a red color), you can use commands like \beforebackgroundpath, explained below.

When the $\langle code \rangle$ is executed, all saved anchors will be in effect. The $\langle code \rangle$ should contain path construction commands.

For our simple rectangle, the following code might be used:

```
\backgroundpath{
  \pgfpathrectanglecorners
    {\upperrightcorner}
    {\pgfpointscale{-1}{\upperrightcorner}}
}
```

As the name suggests, the background path is used "behind" the text labels. Thus, this path is used first, then the text labels are drawn, possibly obscuring part of the path.

$\foreground path \{\langle code \rangle\}$

This command works like \backgroundpath, only it is invoked after the text labels have been drawn. This means that this path can possibly obscure (part of) the text labels.

$\begin{tabular}{ll} \mathbf{d} & \mathbf$

Unlike the previous two commands, $\langle code \rangle$ should not only construct a path, it should also use this path in whatever way is appropriate. For example, the $\langle code \rangle$ might fill some area with a uniform color.

Whatever the $\langle code \rangle$ does, it does it first. This means that any drawing done by $\langle code \rangle$ will be even behind the background path.

Note that the $\langle code \rangle$ is protected with a {pgfscope}.

$\begin{tabular}{ll} \textbf{beforebackgroundpath} \{\langle code \rangle\} \end{array}$

This command works like \behindbackgroundpath, only the $\langle code \rangle$ is executed after the background path has been used, but before the texts label are drawn.

$\begin{tabular}{ll} \mathbf{code} \end{array}$

The $\langle code \rangle$ is executed after the text labels have been drawn, but before the foreground path is used.

$\begin{tabular}{ll} \textbf{beforeforegroundpath} \{\langle code \rangle\} \end{array}$

This $\langle code \rangle$ is executed at the very end.

\interits avedanchors[from={ $\langle another\ shape\ name \rangle$ }]

This command allows you to inherit the code for saved anchors from $\langle another\ shape\ name \rangle$. The idea is that if you wish to create a new shape that is just a small modification of a another shape, you can recycle the code used for $\langle another\ shape\ name \rangle$.

The effect of this command is the same as if you had called \savedanchor and \saveddimen for each saved anchor or saved dimension declared in \(\langle another shape name \rangle \). Thus, it is not possible to "selectively" inherit only some saved anchors, you always have to inherit all saved anchors from another shape. However, you can inherit the saved anchors of more than one shape by calling this command several times.

This command can be used to inherit the code used for the drawings behind the background path from $\langle another\ shape\ name \rangle$.

Inherits the background path code from $\langle another\ shape\ name \rangle$.

Inherits the before background path code from (another shape name).

Inherits the behind foreground path code from $\langle another\ shape\ name \rangle$.

Inherits the foreground path code from $\langle another\ shape\ name \rangle$.

Inherits the before foreground path code from $\langle another\ shape\ name \rangle$.

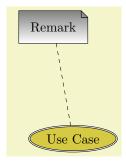
$\interitanchor[from={\langle another shape name \rangle}] {\langle name \rangle}$

Inherits the code of one specific anchor named $\langle name \rangle$ from $\langle another\ shape\ name \rangle$. Thus, unlike saved anchors, which must be inherited collectively, normal anchors can and must be inherited individually.

\inheritanchorborder[from= $\{\langle another\ shape\ name \rangle\}$]

Inherits the border anchor code from $\langle another \ shape \ name \rangle$.

The following example shows how a shape can be defined that relies heavily on inheritance:



```
\pgfdeclareshape{document}{
  \inheritsavedanchors[from=rectangle] % this is nearly a rectangle
  \inheritanchorborder[from=rectangle]
 \inheritanchor[from=rectangle]{center}
 \inheritanchor[from=rectangle]{north}
  \inheritanchor[from=rectangle]{south}
  \inheritanchor[from=rectangle]{west}
  \inheritanchor[from=rectangle]{east}
 % ... and possibly more
  \backgroundpath{% this is new
    % store lower right in xa/ya and upper right in xb/yb
    \southwest \pgf@xa=\pgf@x \pgf@ya=\pgf@y
    \northeast \pgf@xb=\pgf@x \pgf@yb=\pgf@y
    % compute corner of ''flipped page'
    \pgf@xc=\pgf@xb \advance\pgf@xc by-5pt % this should be a parameter
    \pgf@yc=\pgf@yb \advance\pgf@yc by-5pt
    % construct main path
    \pgfpathmoveto{\pgfpoint{\pgf@xa}{\pgf@ya}}
    \pgfpathlineto{\pgfpoint{\pgf@xa}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@ya}}
    \pgfpathclose
    % add little corner
    \pgfpathmoveto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
}\hskip-1.2cm
\begin{tikzpicture}
  \node[shade,draw,shape=document,inner sep=2ex] (x) {Remark};
 \node[fill=examplefill,draw,ellipse,double]
    at ([shift=(-80:3cm)]x) (y) {Use Case};
 \draw[dashed] (x) -- (y);
\end{tikzpicture}
```

84 Matrices

```
\usepgfmodule{matrix} % MEX and plain TEX and pure pgf \usepgfmodule[matrix] % ConTEXt and pure pgf
```

The present section documents the commands of this module.

84.1 Overview

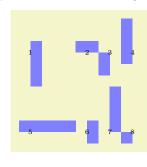
Matrices are a mechanism for aligning several so-called cell pictures horizontally and vertically. The resulting alignment is placed in a normal node and the command for creating matrices, \pgfmatrix, takes options very similar to the \pgfnode command.

In the following, the basic idea behind the alignment mechanism is explained first. Then the command \pgfmatrix is explained. At the end of the section, additional ways of modifying the width of columns and rows are discussed.

84.2 Cell Pictures and Their Alignment

A matrix consists of rows of *cells*. Cells are separated using the special command \pgfmatrixnextcell, rows are ended using the command \pgfmatrixendrow (the command \\\) is set up to mean the same as \pgfmatrixendrow by default). Each cell contains a *cell picture*, although cell pictures are not complete pictures as they lack layers. However, each cell picture has its own bounding box like a normal picture does. These bounding boxes are important for the alignment as explained in the following.

Each cell picture will have an origin somewhere in the picture (or even outside the picture). The position of these origins are important for the alignment: On each row the origins will be on the same horizontal line and for each column the origins will also be on the same vertical line. These two requirements mean that the cell pictures may need to be shifted around so that the origins wind up on the same lines. The top of a row is given by the top of the cell picture whose bounding box's maximum y-position is largest. Similarly, the bottom of a row is given by the bottom of the cell picture whose bounding box's minimum y-position is the most negative. Similarly, the left end of a row is given by the left end of the cell whose bounding box's x-position is the most negative; and similarly for the right end of a row.



```
\begin{tikzpicture}[x=3mm,y=3mm,fill=blue!50]
  \def\atorig#1{\node[black] at (0,0) {\tiny #1};}

\pgfmatrix{rectangle}{center}{mymatrix}
  {\pgfusepath{}}{\pgfpointorigin}{}
  {
    \fill (0,-3) rectangle (1,1);\atorig1 \pgfmatrixnextcell
    \fill (-1,0) rectangle (1,1);\atorig2 \pgfmatrixnextcell
    \fill (-1,-2) rectangle (0,0);\atorig3 \pgfmatrixnextcell
    \fill (-1,-1) rectangle (0,3);\atorig4 \\
    \fill (-1,0) rectangle (4,1);\atorig5 \pgfmatrixnextcell
    \fill (0,-1) rectangle (1,1);\atorig6 \pgfmatrixnextcell
    \fill (0,0) rectangle (1,4);\atorig7 \pgfmatrixnextcell
    \fill (-1,-1) rectangle (0,0);\atorig8 \\
}
\end{tikzpicture}
```

84.3 The Matrix Command

All matrices are typeset using the following command:

This command creates a node that contains a matrix. The name of the node is $\langle name \rangle$, its shape is $\langle shape \rangle$ and the node is anchored at $\langle anchor \rangle$.

The $\langle matrix\ cell \rangle$ parameter contains the cells of the matrix. In each cell drawing commands may be given, which create a so-called cell picture. For each cell picture a bounding box is computed and the cells are aligned according to the rules outlined in the previous section.

The resulting matrix is used as the text box of the node. As for a normal node, the $\langle usage \rangle$ commands are applied, so that the path(s) of the resulting node is (are) stroked or filled or whatever.

Specifying the cells and rows. Even though this command uses **\halign** internally, there are two special rules for indicating cells:

- 1. Cells in the same row must be separated using the macro \pgfmatrixnextcell rather than &. Using & will result in an error message.
 - However, you can make & an active character and have it expand to \pgfmatrixnextcell. This way, it will "look" as if & is used.
- 2. Rows are ended using the command \pgfmatrixendrow, but \\ is set up to mean the same by default. However, some environments like {minipage} redefine \\, so it is good to have \pgfmatrixendrow as a "fallback."
- 3. Every row including the last row must be ended using the command \\ or \pgfmatrixendrow.

Both \pgfmatrixnextcell and \pgfmatrixendrow (and, thus, also \\) take an optional argument as explained in the Section 84.4

Anchoring matrices at nodes inside the matrix. The parameter $\langle shift \rangle$ is an additional negative shift for the node. Normally, such a shift could be given beforehand (that is, the shift could be preapplied to the current transformation matrix). However, when $\langle shift \rangle$ is evaluated, you can refer to temporary positions of nodes inside the matrix. In detail, the following happens: When the matrix has been typeset, all nodes in the matrix temporarily get assigned their positions in the matrix box. The origin of this coordinate system is at the left baseline end of the matrix box, which corresponds to the text anchor. The position $\langle shift \rangle$ is then interpreted inside this coordinate system and then used for shifting. This allows you to use the parameter $\langle shift \rangle$ in the following way: If you use text as the $\langle anchor \rangle$ and specify \pgfpointanchor{inner node}{some anchor} for the parameter $\langle shift \rangle$, where inner node is a node that is created in the matrix, then the whole matrix will be shifted such that inner node.some anchor lies at the origin of the whole picture.

Rotations and scaling. The matrix node is never rotated or shifted, because the current coordinate transformation matrix is reset (except for the translational part) at the beginning of \pgfmatrix. This is intentional and will not change in the future. If you need to rotate the matrix, you must install an appropriate canvas transformation yourself.

However, nodes and stuff inside the cell pictures can be rotated and scaled normally.

Callbacks. At the beginning and at the end of each cell the special macros \pgfmatrixbegincode, \pgfmatrixendcode and possibly \pgfmatrixemptycode are called. The effect is explained in Section 84.5.

Executing extra code. The parameter $\langle pre\text{-}code \rangle$ is executed at the beginning of the outermost T_EX-group enclosing the matrix node. It is inside this T_EX-group, but outside the matrix itself. It can be used for different purposes:

- 1. It can be used to simplify the next cell macro. For example, saying \let\&=\pgfmatrixnextcell allows you to use \& instead of \pgfmatrixnextcell. You can also set the catcode of & to active.
- 2. It can be used to issue an \aftergroup command. This allows you to regain control after the \pgfmatrix command. (If you do not know the \aftergroup command, you are probably blessed with a simple and happy life.)

Special considerations concerning macro expansion. As said before, the matrix is typeset using **halign** internally. This command does a lot of strange and magic things like expanding the first macro of every cell in a most unusual manner. Here are some effects you may wish to be aware of:

- It is not necessary to actually mention \pgfmatrixnextcell or \pgfmatrixendrow inside the \(\matrix cells \). It suffices that the macros inside \(\matrix cells \) expand to these macros sooner or later.
- In particular, you can define clever macros that insert columns and rows as needed for special
 effects.

84.4 Row and Column Spacing

It is possible to control the space between columns and rows rather detailedly. Two commands are important for the row spacing and two commands for the column spacing.

$\protect\pro$

This macro sets the default separation list for columns. The details of the format of this list are explained in the description of the next command.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

This command has two purposes: First, it is used to separate cells. Second, by providing the optional argument $\langle additional\ sep\ list \rangle$ you can modify the spacing between the columns that are separated by this command.

The optional $\langle additional \ sep \ list \rangle$ may only be provided when the $\backslash pgfmatrixnextcell$ command starts a new column. Normally, this will only be the case in the first row, but sometimes a later row has more elements than the first row. In this case, the $\backslash pgfmatrixnextcell$ commands that start the new columns in the later row may also have the optional argument. Once a column has been started, subsequent uses of this optional argument for the column have no effect.

To determine the space between the two columns that are separated by \pgfmatrixnextcell, the following algorithm is executed:

- 1. Both the default separation list (as set up by \pgfsetmatrixcolumnsep) and the \(\lambda additional sep \) list\\\) are processed, in this order. If the \(\lambda additional sep \) list\\\\) argument is missing, only the default separation list is processed.
- 2. Both lists may contain dimensions, separated by commas, as well as occurrences of the keywords between origins and between borders.
- 3. All dimensions occurring in either list are added together to arrive at a dimension d.
- 4. The last occurrence of either of the keywords is located. If neither keyword is present, we proceed as if between borders were present.

At the end of the algorithm, a dimension d has been computed and one of the two modes between borders and between origins has been determined. Depending on which mode has been determined, the following happens:

- For the between borders mode, an additional horizontal space of d is added between the two columns. Note that d may be negative.
- For the between origins mode, the spacing between the two columns is computed differently: Recall that the origins of the cell pictures in both pictures lie on two vertical lines. The spacing between the two columns is set up such that the horizontal distance between these two lines is exactly d.

This mode may only be used between columns already introduced in the first row.

All of the above rules boil down to the following effects:

• A default spacing between columns should be set up using \pgfsetmatrixcolumnsep. For example, you might say \pgfsetmatrixcolumnsep{5pt} to have columns spaced apart by 5pt. You could say

\pgfsetmatrixcolumnsep{1cm,between origins}

to specify that horizontal space between the origins of cell pictures in adjacent columns should be 1cm by default – regardless of the actual size of the cell pictures.

• You can now use the optional argument of \pgfmatrixnextcell to locally overrule the spacing between two columns. By saying \pgfmatrixnextcell[5pt] you add 5pt to the space between of the two columns, regardless of the mode.

You can also (locally) change the spacing mode for these two columns. For example, even if the normal spacing mode is between origins, you can say

\pgfmatrixnextcell[5pt,between borders]

to locally change the mode for these columns to between borders.

```
    8
    1
    6

    3
    5
    7

    4
    9
    2
```

```
\begin{tikzpicture}[every node/.style=draw]
  \pgfsetmatrixcolumnsep{1mm}
  \pgfmatrix{rectangle}{center}{mymatrix}
   \node \{8\}; \&[2mm] \node(a)\{1\}; \&[1cm,between origins] \node(b)\{6\}; \\
   \node {3}; \&
                    \node
                           {5}; \&
                                                     \node
                                                            {7}; \\
   \node {4}; \&
                                                            {2}; \\
                    \node
                           {9}; \&
 \draw [<->,red,thick,every node/.style=] (a.center) -- (b.center)
      node [above,midway] {11mm};
\end{tikzpicture}
```

```
\begin{tikzpicture}[every node/.style=draw]
  \pgfsetmatrixcolumnsep{1cm,between origins}
  \pgfmatrix{rectangle}{center}{mymatrix}
    \node (a) \{8\}; \& \node (b) \{1\}; \&[between borders] \node (c) \{6\}; \\
    \node
              {3}; \& \node
                                 {5}; \&
                                                           \node
                                                                      {7}; \\
    \node
              {4}; \& \node
                                 {9}; \&
                                                            \node
                                                                      {2}; \\
  \begin{scope}[every node/.style=]
    \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm}; \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway]
    {10mm}:
  \end{scope}
\end{tikzpicture}
```

The mechanism for the between-row-spacing is the same, only the commands are called differently.

$\protect\pro$

This macro sets the default separation list for rows.

This command ends a line. The optional $\langle additional\ sep\ list \rangle$ is used to determine the spacing between the row being ended and the next row. The modes and the computation of d is done in the same way as for columns. For the last row the optional argument has no effect.

Inside matrices (and only there) the command \\ is set up to mean the same as this command.

84.5 Callbacks

There are three macros that get called at the beginning and end of cells. By redefining these macros, which are empty by default, you can change the appearance of cells in a very general manner.

\pgfmatrixemptycode

This macro is executed for empty cells. This means that PGF uses some macro magic to determine whether a cell is empty (it immediately ends with \pgfmatrixemptycode or \pgfmatrixendrow) and, if so, put this macro inside the cell.

```
a empty b empty c d empty
```

As can be seen, the macro is not executed for empty cells at the end of row when columns are added only later on.

\pgfmatrixbegincode

This macro is executed at the beginning of non-empty cells. Correspondingly, \pgfmatrixendcode is added at the end of every non-empty cell.

Note that between \pgfmatrixbegincode and \pgfmatrixendcode there will *not* only be the contents of the cell. Rather, PGF will add some (invisible) commands for book-keeping purposes that involve \let and \gdef. In particular, it is not a good idea to have \pgfmatrixbegincode end with \csname and \pgfmatrixendcode start with \endcsname.

\pgfmatrixendcode

See the explanation above.

The following two counters allow you to access the current row and current column in a callback:

\pgfmatrixcurrentrow

This counter stores the current row of the current cell of the matrix. Do not even think about changing this counter.

\pgfmatrixcurrentcolumn

This counter stores the current column of the current cell of the matrix.

85 Coordinate and Canvas Transformations

85.1 Overview

PGF offers two different ways of scaling, shifting, and rotating (these operations are generally known as transformations) graphics: You can apply coordinate transformations to all coordinates and you can apply canvas transformations to the canvas on which you draw. (The names "coordinate" and "canvas" transformations are not standard, I introduce them only for the purposes of this manual.)

The difference is the following:

• As the name "coordinate transformation" suggests, coordinate transformations apply only to coordinates. For example, when you specify a coordinate like \pgfpoint{1cm}{2cm} and you wish to "use" this coordinate—for example as an argument to a \pgfpathmoveto command—then the coordinate transformation matrix is applied to the coordinate, resulting in a new coordinate. Continuing the example, if the current coordinate transformation is "scale by a factor of two," the coordinate \pgfpoint{1cm}{2cm} actually designates the point (2cm, 4cm).

Note that coordinate transformations apply *only* to coordinates. They do not apply to, say, line width or shadings or text.

• The effect of a "canvas transformation" like "scale by a factor of two" can be imagined as follows: You first draw your picture on a "rubber canvas" normally. Then, once you are done, the whole canvas is transformed, in this case stretched by a factor of two. In the resulting image *everything* will be larger: Text, lines, coordinates, and shadings.

In many cases, it is preferable that you use coordinate transformations and not canvas transformations. When canvas transformations are used, PGF looses track of the coordinates of nodes and shapes. Also, canvas transformations often cause undesirable effects like changing text size. For these reasons, PGF makes it easy to setup the coordinate transformation, but a bit harder to change the canvas transformation.

85.2 Coordinate Transformations

85.2.1 How PGF Keeps Track of the Coordinate Transformation Matrix

PGF has an internal coordinate transformation matrix. This matrix is applied to coordinates "in certain situations." This means that the matrix is not always applied to every coordinate "no matter what." Rather, PGF tries to be reasonably smart at when and how this matrix should be applied. The most prominent examples are the path construction commands, which apply the coordinate transformation matrix to their inputs.

The coordinate transformation matrix consists of four numbers a, b, c, and d, and two dimensions s and t. When the coordinate transformation matrix is applied to a coordinate (x,y) the new coordinate (ax + by + s, cx + dy + t) results. For more details on how transformation matrices work in general, please see, for example, the PDF or PostScript reference or a textbook on computer graphics.

The coordinate transformation matrix is equal to the identity matrix at the beginning. More precisely, a = 1, b = 0, c = 0, d = 1, s = 0pt, and t = 0pt.

The different coordinate transformation commands will modify the matrix by concatenating it with another transformation matrix. This way the effect of applying several transformation commands will accumulate.

The coordinate transformation matrix is local to the current TEX group (unlike the canvas transformation matrix, which is local to the current {pgfscope}). Thus, the effect of adding a coordinate transformation to the coordinate transformation matrix will last only till the end of the current TEX group.

85.2.2 Commands for Relative Coordinate Transformations

The following commands add a basic coordinate transformation to the current coordinate transformation matrix. For all commands, the transformation is applied in addition to any previous coordinate transformations.

```
\verb|\pgftransformshift{|\langle point \rangle|}
```

Shifts coordinates by $\langle point \rangle$.



$\protect\pro$

Shifts coordinates by $\langle dimension \rangle$ along the x-axis.



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (2,1) -- (1,0);
\pgftransformxshift{.5cm}
\draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

$\protect\operatorname{\mathtt{pgftransformyshift}} \$

Like $\protect\operatorname{pgftransform}$ for the y-axis.

$\protect\pro$

Scales coordinates by $\langle factor \rangle$.



$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Scales coordinates by $\langle factor \rangle$ in the x-direction.



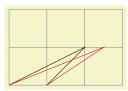
```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (2,1) -- (1,0);
\pgftransformxscale{.75}
\draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

Like $\protect\operatorname{pgftransformxscale}$, only for the y-axis.

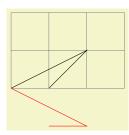
$\protect\operatorname{\begin{tabular}{l} pgftransformxslant{\langle factor\rangle}\end{tabular}}$

Slants coordinates by $\langle factor \rangle$ in the x-direction. Here, a factor of 1 means 45° .



$\verb|\pgftransformyslant{|} \langle factor \rangle \}|$

Slants coordinates by $\langle factor \rangle$ in the y-direction.



$\protect\operatorname{\mathtt{ar{pgftransformrotate}}}$

Rotates coordinates counterclockwise by $\langle angles \rangle$ given in degrees.



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (2,1) -- (1,0);
\pgftransformrotate{30}
\draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

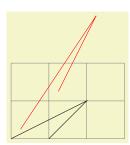
This command transforms the coordinate system in such a way that the triangle given by the points $\langle a \rangle$, $\langle b \rangle$ and $\langle c \rangle$ lies at the coordinates (0,0), (1pt,0pt) and (0pt,1pt).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformtriangle
  {\pgfpoint{1cm}{0cm}}
  {\pgfpoint{0cm}{2cm}}
  {\pgfpoint{3cm}{1cm}}
  \draw (0,0) -- (1pt,0pt) -- (0pt,1pt) -- cycle;
  \end{tikzpicture}
```

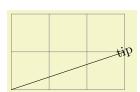
$\verb| pgftransformcm{}(\langle a \rangle){}(\langle b \rangle){}(\langle c \rangle){}(\langle d \rangle){}(\langle point \rangle)$

Applies the transformation matrix given by a, b, c, and d and the shift $\langle point \rangle$ to coordinates (in addition to any previous transformations already in force).



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (2,1) -- (1,0);
\pgftransformcm{1}{1}{0}{1}{\pgfpoint{.25cm}}{.25cm}}
\draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

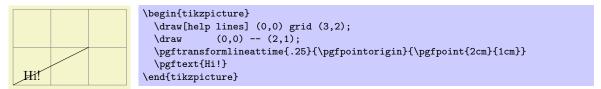

Shifts coordinates to the end of the line going from $\langle start \rangle$ to $\langle end \rangle$ with the correct rotation.

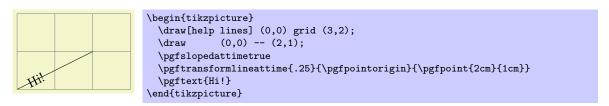


\pgftransformlineattime $\{\langle time \rangle\} \{\langle start \rangle\} \{\langle end \rangle\}$

Shifts coordinates by a specific point on a line at a specific time. The point by which the coordinate is shifted is calculated by calling \pgfpointlineattime, see Section 78.5.2.

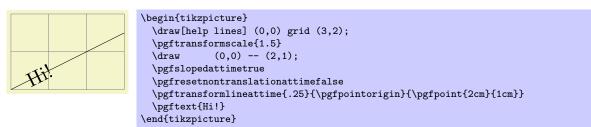
In addition to shifting the coordinate, a rotation may also be applied. Whether this is the case depends on whether the TeX if \ifpgfslopedattime is set to true or not.

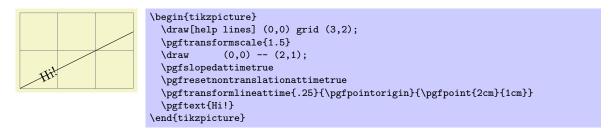




If \ifpgfslopedattime is true, another TEX \if is important: \ifpgfallowupsidedowattime. If this is false, PGF will ensure that the rotation is done in such a way that text is never "upside down."

There is another TEX \if that influences this command. If you set \ifpgfresetnontranslationattime to true, then, between shifting the coordinate and (possibly) rotating/sloping the coordinate, the command \pgftransformresetnontranslations is called. See the description of this command for details.





```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

Shifts coordinates by a specific point on a curve at a specific time, see Section 78.5.2 once more.

As for the line-at-time transformation command, \ifpgfslopedattime decides whether an additional rotation should be applied. Again, the value of \ifpgfallowupsidedowattime is also considered.

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}}{\text{Hi!}}
  \end{tikzpicture}
```

```
begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  draw (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgfslopedattimetrue
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}}{\pgftext{Hi!}}
  \end{tikzpicture}
```

The value of \ifpgfresetnontranslationsattime is also taken into account.

Shifts coordinates by a specific point on an arc at a specific time, see Section 78.5.2 once more.

As for the previous commands, \ifpgfslopedattime decides whether an additional rotation should be applied and \ifpgfallowupsidedowattime is also considered.

The value of \ifpgfresetnontranslationsattime is also taken into account.

\ifpgfslopedattime

Decides whether the "at time" transformation commands also rotate coordinates or not.

\ifpgfallowupsidedowattime

Decides whether the "at time" transformation commands should allow the rotation be done in such a way that "upside-down text" can result.

\ifpgfresetnontranslationsattime

Decides whether the "at time" transformation commands should reset the non-translations between shifting and rotating.

85.2.3 Commands for Absolute Coordinate Transformations

The coordinate transformation commands introduced up to now are always applied in addition to any previous transformations. In contrast, the commands presented in the following can be used to change the transformation matrix "in absolute terms." Note that this is, in general, dangerous and will often produce unexpected effects. You should use these commands only if you really know what you are doing.

\pgftransformreset

Resets the coordinate transformation matrix to the identity matrix. Thus, once this command is given no transformations are applied till the end of the scope.



$\verb|\pgftransformresetnontranslations| \\$

This command sets the a, b, c, and d part of the coordinate transformation matrix to a = 1, b = 0, c = 0, and d = 1. However, the current shifting of the matrix is not modified.

The effect of this command is that any rotation/scaling/slanting is undone in the current TeX group, but the origin is not "moved back."

This command is mostly useful directly before a \pgftext command to ensure that the text is not scaled or rotated.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{2}
  \pgftransformrotate{30}
  \pgftransformxshift{1cm}
  {\color{red}\pgftext{rotated}}
  \pgftransformresetnontranslations
  \pgftext{shifted only}
  \end{tikzpicture}
```

\pgftransforminvert

Replaces the coordinate transformation matrix by a coordinate transformation matrix that "exactly undoes the original transformation." For example, if the original transformation was "scale by 2 and then shift right by 1cm" the new one is "shift left by 1cm and then scale by 1/2."

This command will produce an error if the determinant of the matrix is too small, that is, if the matrix is near-singular.



85.2.4 Saving and Restoring the Coordinate Transformation Matrix

There are two commands for saving and restoring coordinate transformation matrices.

$\protect\operatorname{\begin{tabular}{c} \protect\operatorname{\begin{tabular}{c} \protect\begin{tabular}{c} \protect\operatorname{\begin{tabular}{c} \protect\begin{tabular}{c} \protect\begin{tabu$

This command will (locally) define $\langle macro \rangle$ to a representation of the current coordinate transformation matrix. This matrix can later on be reinstalled using \pgfsettransform.

$\protect\pro$

Reinstalls a coordinate transformation matrix that was previously saved using \pgfgettransform.

```
\pgfgettransformentries{\langle macro for a\}}{\langle macro for b\}}{\langle macro for c\}}{\langle macro for d\}}{\langle macro for d\}}{\langle macro for shift y\}}
```

This command is similar to \pgfgettransform except that it stores the current coordinate transformation matrix in a set of six macros.

The matrix can later on be reinstalled using \pgfsettransformentries. Furthermore, all these macros (or just a few of them) can be used as arguments for \pgftransformcm.

```
\pgfsettransformentries\{\langle a \rangle\}\{\langle b \rangle\}\{\langle c \rangle\}\{\langle d \rangle\}\{\langle shiftx \rangle\}\{\langle shifty \rangle\}
```

Reinstalls a coordinate transformation matrix that was previously saved using the storage command \pgfgettransformentries. This command can also be used to replace any previously existing coordinate transformation matrix (it is thus equivalent to \pgftransformreset followed by \pgftransformcm).

85.3 Canvas Transformations

The canvas transformation matrix is not managed by PGF, but by the output format like PDF or PostScript. All that PGF does is to call appropriate low-level \pgfsys@ commands to change the canvas transformation matrix.

Unlike coordinate transformations, canvas transformations apply to "everything," including images, text, shadings, line thickness, and so on. The idea is that a canvas transformation really stretches and deforms the canvas after the graphic is finished.

Unlike coordinate transformations, canvas transformations are local to the current {pgfscope}, not to the current TeX group. This is due to the fact that they are managed by the backend driver, not by TeX or PGF.

Unlike the coordinate transformation matrix, it is not possible to "reset" the canvas transformation matrix. The only way to change it is to concatenate it with another canvas transformation matrix or to end the current {pgfscope}.

Unlike coordinate transformations, PGF does not "keep track" of canvas transformations. In particular, it will not be able to correctly save the coordinates of shapes or nodes when a canvas transformation is used.

PGF does not offer a whole set of special commands for modifying the canvas transformation matrix. Instead, different commands allow you to concatenate the canvas transformation matrix with a coordinate transformation matrix (and there are numerous commands for specifying a coordinate transformation, see the previous section).

\pgflowlevelsynccm

This command concatenates the canvas transformation matrix with the current coordinate transformation matrix. Afterward, the coordinate transformation matrix is reset.

The effect of this command is to "synchronize" the coordinate transformation matrix and the canvas transformation matrix. All transformations that were previously applied by the coordinate transformations matrix are now applied by the canvas transformation matrix.



$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

This command concatenates the canvas transformation matrix with the coordinate transformation specified by $\langle transformation \ code \rangle$.



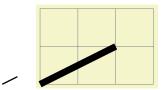
$\protect\pro$

This command creates a local {pgfscope}. Inside this scope, \pgflowlevel is first called with the argument $\langle transformation\ code \rangle$, then the $\langle code \rangle$ is inserted.



```
\begin{pgflowlevelscope} \{\langle transformation\ code \rangle\} \\ \langle environment\ contents \rangle \\ \begin{pgflowlevelscope} \end{pgflowlevelscope} \}
```

This environment first surrounds the $\langle environment\ contents \rangle$ by a {pgfscope}. Then it calls \pgflowlevel with the argument $\langle transformation\ code \rangle$.



```
\draw[help lines] (0,0) grid (3,2);
\pgfsetlinewidth{1pt}
\begin{pgflowlevelscope}{\pgftransformscale{5}}
   \draw (0,0) -- (0.4,.2);
\end{pgflowlevelscope}
\begin{pgflowlevelscope}{\pgftransformxshift{-1cm}}
   \draw (0,0) -- (0.4,.2);
\end{pgflowlevelscope}
\bedin{pgflowlevelscope}{\pgftransformxshift{-1cm}}
   \draw (0,0) -- (0.4,.2);
\end{pgflowlevelscope}
\end{tikzpicture}
```

```
\label{eq:cope} $$ \left( \mbox{$cope} \left( \mbox{$contents$} \right) \right) $$ \end{$contents} $$ \end{$pgflowlevelscope} $$
```

Plain TEX version of the environment.

 $\label{lem:code} $$ \artpgflowlevelscope {$\langle transformation\ code \rangle$} $$ \artpgflowlevelscope $$ \artpgflowlevelscope $$$

 ${
m ConT}_{
m E}{
m Xt}$ version of the environment.

86 Patterns

86.1 Overview

There are many ways of filling a path. First, you can fill it using a solid color and this is also the fastest method. Second, you can also fill it using a shading, which means that the color changes smoothly between two (or more) different colors. Third, you can fill it using a tiling pattern and it is explained in the following how this is done.

A tiling pattern can be imagined as a rectangular tile (hence the name) on which a small picture is painted. There is not a single tile, but (conceptually) an infinite number of tiles, all showing the same picture, and these tiles are arranged horizontally and vertically to fill the plane. When you use a tiling pattern to fill a path, what happens is that the path clips out a "window" through which we see part of this infinite plane.

Patterns come in two versions: *inherently colored patterns* and *form-only patterns*. (These are often called "color patterns" and "uncolored patterns," but these names are misleading since uncolored patterns do have a color and the color changes. As I said, the name is misleading...) An inherently colored pattern is just a colored tile like, say, a red star with a black outline. A form-only pattern can be imagined as a tile that is a kind of rubber stamp. When this pattern is used, the stamp is used to print copies of the stamp picture onto the plane, but we can use a different stamp color each time we use a form-only pattern.

PGF provides a special support for patterns. You can declare a pattern and then use it very much like a fill color. PGF directly maps patterns to the pattern facilities of the underlying graphic languages (PostScript, PDF, and SVG). This means that filling a path using a pattern will be nearly as fast as if you used a uniform color.

There are a number of pitfalls and restrictions when using patterns. First, once a pattern has been declared, you cannot change it anymore. In particular, it is not possible to enlarge it or change the line width. Such flexibility would require that the repeating of the pattern were not done by the graphic language, but on the PGF level. This would make patterns orders of magnitude slower to produce and to render. However, PGF does provide a more-or-less successful emulation of "mutable" patterns, although internally, a new (fixed) instance of a pattern is declared when the parameters of a pattern change.

Second, the phase of patterns is not well-defined, that is, it is not clear where the origin of the "first" tile is. To be more precise, PostScript and PDF on the one hand and SVG on the other hand define the origin differently. PostScript and PDF define a fixed origin that is independent of where the path lies. This has the highly desirable effect that if you use the same pattern to fill multiple paths, the outcome is the same as if you had filled a single path consisting of the union of all these paths. By comparison, SVG uses the upper-left (?) corner of the path to be filled as the origin. However, the SVG specification is a bit vague on this question.

86.2 Declaring a Pattern

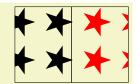
Before a pattern can be used, it must be declared. The following command is used for this:

 $\label{local_property} $$ \operatorname{clarepatternformonly} [\langle variables \rangle] {\langle bottom\ left \rangle} {\langle top\ right \rangle} {\langle tile\ size \rangle} {\langle code \rangle} $$$

This command declares a new form-only pattern. The $\langle name \rangle$ is a name for later reference. The two parameters $\langle lower\ left \rangle$ and $\langle upper\ right \rangle$ must describe a bounding box that is large enough to encompass the complete tile.

The size of a tile is given by $\langle tile \ size \rangle$, that is, a tile is a rectangle whose lower left corner is the origin and whose upper right corner is given by $\langle tile \ size \rangle$. This might make you wonder why the second and third parameters are needed. First, the bounding box might be smaller than the tile size if the tile is larger than the picture on the tile. Second, the bounding box might be bigger, in which case the picture will "bleed" over the tile.

The $\langle code \rangle$ should be PGF code than can be protocolled. It should not contain any color code.



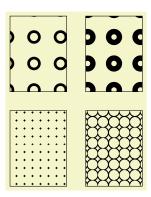
```
\pgfdeclarepatternformonly{stars}
{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
{\pgfpoint{1cm}{1cm}}
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{fill}
\begin{tikzpicture}
  \filldraw[pattern=stars] (0,0) rectangle (1.5,2);
  \filldraw[pattern=stars,pattern color=red]
                           (1.5,0) rectangle (3,2);
\end{tikzpicture}
```

The optional argument $\langle variables \rangle$ consists of a comma separated list of macros, registers or keys, representing the parameters of the pattern that may vary. If a variable is a key, then the full path name must be used (specifically, it must start with /). As an example, a list might look like the following: \mymacro,\mydimen,/pgf/my key. Note that macros and keys should be "simple". They should only store values in themselves.

The effect of $\langle variables \rangle$, is the following: Normally, when this argument is empty, once a pattern has been declared, it becomes "frozen". This means that it is not possible to enlarge the pattern or change the line width later on. By specifying $\langle variables \rangle$, no pattern is actually created. Instead, the arguments are stored away (so the macros, registers or keys do not have to be defined in advance).

When the fill pattern is set, PGF checks if the pattern has already been created with the $\langle variables \rangle$ set to their current values (PGF is usually "smart enough" to distinguish between macros, registers and keys). If so, this already-declared-pattern is used as the fill pattern. If not, a new instance of the pattern (which will have a unique internal name) is declared using the current values of $\langle variables \rangle$. These values are then saved and the fill pattern set accordingly.

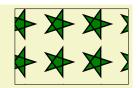
The following shows an example of a pattern which varies according to the values of the macro \size, the key /tikz/radius, and the TeX dimension \thickness.



```
\pgfdeclarepatternformonly[/tikz/radius,\thickness,\size]{rings}
{\left\{\begin{array}{ll} {\left(-0.5*\right)} & {\left(-0.5*\right)} \end{array}\right.}
{\bf \{\pgfpoint\{0.5*\size\}\{0.5*\size\}\}}
{\pgfpoint{\size}{\size}}
  \pgfsetlinewidth{\thickness}
  \pgfpathcircle\pgfpointorigin{\pgfkeysvalueof{/tikz/radius}}
  \pgfusepath{stroke}
\newdimen\thickness
\tikzset{
  radius/.initial=4pt,
  size/.store in=\size, size=20pt,
  thickness/.code={\thickness=#1},
  thickness=0.75pt
\begin{tikzpicture}[rings/.style={pattern=rings}]
  \filldraw [rings, radius=2pt, size=6pt]
                                                   (0,0)
                                                            rectangle +(1.5,2);
  \filldraw [rings, radius=2pt, size=8pt]
                                                    (2,0)
                                                           rectangle +(1.5,2);
  \filldraw [rings, radius=6pt, thickness=2pt] (0,2.5) rectangle +(1.5,2);
  \filldraw [rings, radius=8pt, thickness=4pt] (2,2.5) rectangle +(1.5,2);
\end{tikzpicture}
```

 $\label{eq:code} $$ \begin{array}{l} \cline{1.5mm} \cli$

This command works like \pgfdeclarepatternuncolored, only the pattern will have an inherent color. To set the color, you should use PGF's color commands, not the \color command, since this fill is not protocolled.



```
\pgfdeclarepatterninherentlycolored{green stars}
{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
{
\pgfpoint{1cm}{1cm}}
{
\pgfsetfillcolor{green!50!black}
\pgfsetfillcolor{green!50!black}
\pgfpathmoveto{\pgfpointpolar{0}{4mm}}
\pgfpathlineto{\pgfpointpolar{144}{4mm}}
\pgfpathlineto{\pgfpointpolar{1248}{4mm}}
\pgfpathlineto{\pgfpointpolar{72}{4mm}}
\pgfpathlineto{\pgfpointpolar{72}{4mm}}
\pgfpathlineto{\pgfpointpolar{216}{4mm}}
\pgfpathclose%
\pgfusepath{stroke,fill}
}
\begin{tikzpicture}
\filldraw[pattern=green stars] (0,0) rectangle (3,2);
\end{tikzpicture}
```

86.3 Setting a Pattern

Once a pattern has been declared, it can be used.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command specifies that paths that are filled should be filled with the "color" by the pattern $\langle name \rangle$. For an inherently colored pattern, the $\langle color \rangle$ parameter is ignored. For form-only patterns, the $\langle color \rangle$ parameter specifies the color to be used for the pattern.



```
\begin{tikzpicture}
  \pgfsetfillpattern{stars}{red}
  \filldraw (0,0) rectangle (1.5,2);

  \pgfsetfillpattern{green stars}{red}
  \filldraw (1.5,0) rectangle (3,2);
  \end{tikzpicture}
```

87 Declaring and Using Images

This section describes the commands for creating images.

87.1 Overview

To be quite frank, IATEX's \includegraphics is designed better than PGF's image mechanism. For this reason, I recommend that you use the standard image inclusion mechanism of your format. Thus, IATEX users are encouraged to use \includegraphics to include images.

However, there are reasons why you might need to use the image inclusion facilities of PGF:

- There is no standard image inclusion mechanism in your format. For example, plain TeX does not have one, so PGF's inclusion mechanism is "better than nothing."
 - However, this applies only to the pdftex backend. For all other backends, PGF currently maps its commands back to the graphicx package. Thus, in plain TEX, this does not really help. It might be a good idea to fix this in the future such that PGF becomes independent of LATEX, thereby providing a uniform image abstraction for all formats.
- You wish to use masking. This is a feature that is only supported by PGF, though I hope that someone will implement this also for the graphics package in LATEX in the future.

Whatever your choice, you can still use the usual image inclusion facilities of the graphics package.

The general approach taken by PGF to including an image is the following: First, \pgfdeclareimage declares the image. This must be done prior to the first use of the image. Once you have declared an image, you can insert it into the text using \pgfuseimage. The advantage of this two-phase approach is that, at least for PDF, the image data will only be included once in the file. This can drastically reduce the file size if you use an image repeatedly, for example in an overlay. However, there is also a command called \pgfimage that declares and then immediately uses the image.

To speedup the compilation, you may wish to use the following class option:

\usepackage[draft]{pgf}

In draft mode boxes showing the image name replace the images. It is checked whether the image files exist, but they are not read. If either height or width is not given, 1cm is used instead.

87.2 Declaring an Image

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

Declares an image, but does not paint anything. To draw the image, use \pgfuseimage{\langle image name \rangle}. The \langle filename \rangle may not have an extension. For PDF, the extensions .pdf, .jpg, and .png will automatically tried. For PostScript, the extensions .eps, .epsi, and .ps will be tried.

The following options are possible:

- height=\langle dimension \rangle\$ sets the height of the image. If the width is not specified simultaneously, the aspect ratio of the image is kept.
- width=(dimension) sets the width of the image. If the height is not specified simultaneously, the aspect ratio of the image is kept.
- page=\(page number\)\ selects a given page number from a multipage document. Specifying this option will have the following effect: first, PGF tries to find a file named

```
\langle filename \rangle.page\langle page\ number \rangle.\langle extension \rangle
```

If such a file is found, it will be used instead of the originally specified filename. If not, PGF inserts the image stored in $\langle filename \rangle$. $\langle extension \rangle$ and if a recent version of pdflatex is used, only the selected page is inserted. For older versions of pdflatex and for dvips the complete document is inserted and a warning is printed.

- interpolate=\langle true or false \rangle selects whether the image should be "smoothed" when zoomed. False by default.
- mask=(mask name) selects a transparency mask. The mask must previously be declared using \pgfdeclaremask (see below). This option only has an effect for pdf. Not all viewers support masking.

```
\pgfdeclareimage[interpolate=true,height=1cm]{image1}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]{image2}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{brave-gnu-world-logo}
```

 $\protect\operatorname{\mathtt{New}}\ image\ name \$ $\cite{custing}\ image\ name \$

The $\{\langle existing \ image \ name \rangle\}$ is "cloned" and the $\{\langle new \ image \ name \rangle\}$ can now be used whenever the original image is used. This command is useful for creating aliases for alternate extensions and for accessing the last image inserted using \pgfimage.

Example: \pgfaliasimage{image.!30!white}{image.!25!white}

87.3 Using an Image

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Inserts a previously declared image into the *normal text*. If you wish to use it in a {pgfpicture} environment, you must put a \pgftext around it.

If the macro \pgfalternateextension expands to some nonempty $\langle alternate\ extension \rangle$, PGF will first try to use the image named $\langle image\ name \rangle$. $\langle alternate\ extension \rangle$. If this image is not defined, PGF will next check whether $\langle alternate\ extension \rangle$ contains a! character. If so, everything up to this exclamation mark and including it is deleted from $\langle alternate\ extension \rangle$ and the PGF again tries to use the image $\langle image\ name \rangle$. $\langle alternate\ extension \rangle$. This is repeated until $\langle alternate\ extension \rangle$ no longer contains a!. Then the original image is used.

The xxcolor package sets the alternate extension to the current color mixin.

```
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]
{image1}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,width=1cm]{image2}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{brave-gnu-world-logo}
\begin{pgfpicture}
\pgftext[at=\pgfpoint{1cm}{5cm},left,base]{\pgfuseimage{image1}}
\pgftext[at=\pgfpoint{1cm}{3cm},left,base]{\pgfuseimage{image2}}
\pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image3}}
\pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
\pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
\pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
\pgfusepath{stroke}
\end{pgfpicture}
```

The following example demonstrates the effect of using \pgfuseimage inside a colormixin environment.

```
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]
 {image1.!25!white}{brave-gnu-world-logo.25}
\pgfdeclareimage[interpolate=true,width=1cm]
 {image2.25!white}{brave-gnu-world-logo.25}
\pgfdeclareimage[interpolate=true,height=1cm]
  {image3.white}{brave-gnu-world-logo.25}
\begin{colormixin}{25!white}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{5cm},left,base]{\pgfuseimage{image1}}
 \pgftext[at=\pgfpoint{1cm}{3cm},left,base]{\pgfuseimage{image2}}
 \pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image3}}
 \pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
 \pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}}1cm}}
 \pgfusepath{stroke}
\end{pgfpicture}
\end{colormixin}
```

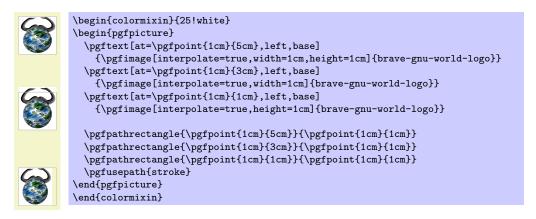
\pgfalternateextension

You should redefine this command to install a different alternate extension.

Example: \def\pgfalternateextension{!25!white}

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Declares the image under the name pgflastimage and immediately uses it. You can "save" the image for later usage by invoking \pgfaliasimage on pgflastimage.



87.4 Masking an Image

 $\protection{pgfdeclaremask[$\langle options \rangle] {\langle mask \ name \rangle} {\langle filename \rangle}}$

Declares a transparency mask named $\langle mask \; name \rangle$ (called a *soft mask* in the PDF specification). This mask is read from the file $\langle filename \rangle$. This file should contain a grayscale image that is as large as the actual image. A white pixel in the mask will correspond to "transparent," a black pixel to "solid," and gray values correspond to intermediate values. The mask must have a single "color channel." This means that the mask must be a "real" grayscale image, not an RGB-image in which all RGB-triples happen to have the same components.

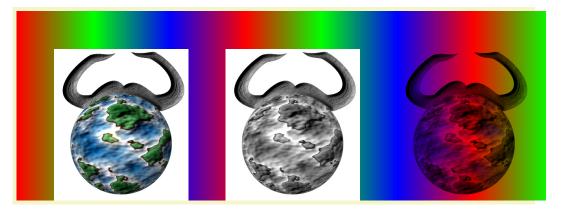
You can only mask images the are in a "pixel format." These are .jpg and .png. You cannot mask .pdf images in this way. Also, again, the mask file and the image file must have the same size.

The following options may be given:

• matte={\langle color components \rangle} sets the so-called matte of the actual image (strangely, this has to be specified together with the mask, not with the image itself). The matte is the color that has been used to preblend the image. For example, if the image has been preblended with a red background, then \langle color components \rangle should be set to \{1 0 0\}. The default is \{1 1 1\}, which is white in the rgb model.

The matte is specified in terms of the parent's image color space. Thus, if the parent is a grayscale image, the matte has to be set to {1}.

Example:



```
%% Draw a large colorful background
\pgfdeclarehorizontalshading{colorful}{5cm}{color(0cm)=(red);
color(2cm)=(green); color(4cm)=(blue); color(6cm)=(red);
color(8cm)=(green); color(10cm)=(blue); color(12cm)=(red);
color(14cm)=(green)}
\hbox{\pgfuseshading{colorful}\hskip-14cm\hskip1cm
\pgfimage[height=4cm]{brave-gnu-world-logo}\hskip1cm
\pgfimage[height=4cm]{brave-gnu-world-logo-mask}\hskip1cm
\pgfdeclaremask{mymask}{brave-gnu-world-logo-mask}\hskip1cm
\pgfimage[mask=mymask,height=4cm,interpolate=true]{brave-gnu-world-logo}}
```

88 Externalizing Graphics

88.1 Overview

There are two fundamentally different ways of inserting graphics into a TEX-document. First, you can create a graphic using some external program like xfig or InDesign and then include this graphic in your text. This is done using commands like \includegraphics or \pgfimage. In this case, the graphic file contains all the low-level graphic commands that describe the picture. When such a file is included, all TEX has to worry about is the size of the picture; the internals of the picture are unknown to TEX and it does not care about them.

The second method of creating graphics is to use a special package that transforms TeX-commands like \draw or \psline into appropriate low-level graphic commands. In this case, TeX has to do all the hard work of "typesetting" the picture and if a picture has a complicated internal structure this may take a lot of time.

While PGF was created to facilitate the second method of creating pictures, there are two main reasons why you may need to employ the first method of image-inclusion, nevertheless:

- 1. Typesetting a picture using TEX can be a very time-consuming process. If TEX needs a minute to typeset a picture, you do not want to wait this minute when you reTEX your document after having changed a single comma.
- 2. Some users, especially journal editors, may not be able to process files that contain PGF commands for the simple reason that the systems of many publishing houses do not have PGF installed.

In both cases, the solution is to "extract" or "externalize" pictures that would normally be typeset every time a document is TEXed. Once the pictures have been extracted into separate graphics files, these graphic files can be reinserted into the text using the first method.

Extracting a graphic from a file is not as easy as it may sound at first since TEX cannot write parts of its output into different files and a bit of trickery is needed. The following macros simplify the workflow:

- 1. You have to tell PGF which files will be used for which pictures. To do so, you enclose each picture that you wish to be "externalized" in a pair of \beginpgfgraphicnamed and \endpgfgraphicnamed macros.
- 2. The next step is to generate the extracted graphics. For this you run TEX with the \jobname set to the graphic file's name. This will cause \pgfname to behave in a very special way: All of your document will simply be thrown away, except for the single graphic having the same name as the current jobname.
- 3. After you have run TeX once for each graphic that your wish to externalize, you can rerun TeX on your document normally. This will have the following effect: Each time a \beginpgfgraphicnamed is encountered, PGF checks whether a graphic file of the given name exists (if you did step 2, it will). If this graphic file exists, it will be input and the text till the corresponding \endpgfgraphicnamed will be ignored.

In the rest of this section, the above workflow is explained in more detail.

88.2 Workflow Step 1: Naming Graphics

In order to put each graphic in an external file, you first need to tell PGF the names of these files.

$\begin{center} \textbf{beginpgfgraphicnamed} \{ \langle \textit{file name prefix} \rangle \} \end{center}$

This command indicates that everything up to the next call of $\ensuremath{\mbox{\mbox{command}}}$ is part of a graphic that should be placed in a file named $\ensuremath{\mbox{\mbox{\mbox{\it prefix}}}\xspace}$, where the $\ensuremath{\mbox{\mbox{\it suffix}}\xspace}$ depends on your backend driver. Typically, $\ensuremath{\mbox{\it suffix}}\xspace$ will be dvi or pdf.

Here is a typical example of how this command is used:

```
% In file main.tex:
...
As we see in Figure~\ref{fig1}, the world is flat.
\begin{figure}
\beginpgfgraphicnamed{graphic-of-flat-world}
\begin{tikzpicture}
\fill (0,0) circle (1cm);
\end{tikzpicture}
\endpgfgraphicnamed
\caption{The flat world.}
\label{fig1}
\end{figure}
```

Each graphic to be externalized should have a unique name. Note that this name will be used as the name of a file in the file system, so it should not contain any funny characters.

This command can have three different effects:

- 1. The easiest situation arises if there does not yet exist a graphic file called \(\lambda file name \text{prefix} \rangle \. \lambda suffix \rangle \), where the \(\lambda suffix \rangle \) is one of the suffixes understood by your current backend driver (so pdf or jpg if you use pdftex, eps if you use dvips, and so on). In this case, both this command and the \endpgfgraphicnamed command simply have no effect.
- 2. A more complex situation arises when a graphic file named \(\frac{file name prefix} \). \(\sum \frac{suffix}{does} \) exist. In this case, this graphic file is included using the \includegraphics command \(\frac{18}{18} \). Furthermore, the text between \(\beginpgfgraphicnamed \) and \(\endpgfgraphicnamed \) is ignored.

 When the text is "ignored," what actually happens is that all text up to the next occurrence of
 - \endpgfgraphicnamed is thrown away without any macro expansion. This means, in particular, that (a) you cannot put \endpgfgraphicnamed inside a macro and (b) the macros used in the graphics need not be defined at all when the graphic file is included.
- 3. The most complex behaviour arises when current the \jobname equals the $\langle file\ name\ prefix \rangle$ and, furthermore, the real job name has been declared. The behaviour for this case is explained later.

Note that the \beginpgfgraphicnamed does not really have any effect until you have generated the graphic files named. Till then, this command is simply ignored. Also, if you delete the graphics file later on, the graphics are typeset normally once more.

\endpgfgraphicnamed

This command just marks the end of the graphic that should be externalized.

88.3 Workflow Step 2: Generating the External Graphics

We have now indicated all the graphics for which we would like graphic files to be generated. In order to generate the files, you now need to modify the \jobname appropriately. This is done in two steps:

1. You use the following command to tell PGF the real name of your .tex file:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

Tells PGF the real name of your job. For instance, if you have a file called survey.tex that contains two graphics that you wish to be called survey-graphic1 and survey-graphic2, then you should write the following.

```
% This is file survey.tex
\documentclass{article}
...
\usepackage{tikz}
\pgfrealjobname{survey}
```

2. You run TEX with the \jobname set to the name of the graphic for which you need an external graphic to be generated. To set the \jobname, you use the --jobname= option of TEX:

```
bash> latex --jobname=survey-graphic1 survey.tex
```

 $^{^{18}}$ Actually, the command key /pgf/images/include external is invoked which calls an appropriate \includegraphics command.

The following things will now happen:

- 1. \pgfrealjobname notices that the \jobname is not the "real" jobname and, thus, must be the name of a graphic that is to be put in an external file.
- 2. At the beginning of the document, PGF changes the definition of TEX's internal \shipout macro. The new shipout macro simply throws away the output. This means that the document is typeset normally, but no output is produced.
- 3. When the \beginpgfgraphicnamed{ $\langle name \rangle$ } command is encountered where the $\langle name \rangle$ is the same as the current \jobname, then a TeX-box is started and $\langle everything \rangle$ up to the following \endpgfgraphicnamed command is stored inside this box.
 - Note that, typically, $\langle everything \rangle$ will contain just a single {tikzpicture} or {pgfpicture} environment. However, this need not be the case, you can use, say, a {pspicture} environment as $\langle everything \rangle$ or even just some normal TEX-text.
- 4. At the \endpgfgraphicnamed, the box is shipped out using the original \shipout command. Thus, unlike everything else, the contents of the graphic is made part of the output.
- 5. When the box containing the graphic is shipped out, the paper size is modified such that it is exactly equal to the height and width of the box.

The net effect of everything described above is that the two commands

```
bash> latex --jobname=survey-graphic1 survey.tex
bash> dvips survey-graphic1
```

produce a file called survey-graphic1.ps that consists of a single page that contains exactly the graphic produced by the code between \beginpgfgraphicnamed{survey-graphic1} and \endpgfgraphicnamed. Furthermore, the size of this single page is exactly the size of the graphic.

If you use pdfTfX, producing the graphic is even simpler:

```
bash> pdflatex --jobname=survey-graphic1 survey.tex
```

produces the single-page pdf-file survey-graphic1.pdf.

88.4 Workflow Step 3: Including the External Graphics

Once you have produced all the pictures in the text, including them into the main document is easy: Simply run TeX again without any modification of the \jobname. In this case the \pgfrealjobname command will notice that the main file is, indeed, the main file. The main file will then be typeset normally and the \beginpgfgraphicnamed commands also behave normally, which means that they will try to include the generated graphic files – which is exactly what you want.

Suppose that you wish to send your survey to a journal that does not have PGF installed. In this case, you now have all the necessary external graphics, but you still need PGF to automatically include them instead of the executing the picture code! One way to solve this problem is to simply delete all of the PGF or TikZ code from your survey.tex and instead insert appropriate \includegraphics commands "by hand." However, there is a better way: You input the file pgfexternal.tex.

File pgfexternal.tex

This file defines the command \beginpgfgraphicnamed and causes it to have the following effect: It includes the graphic file given as a parameter to it and then gobbles everything up to \endpgfgraphicnamed.

Since \beginpgfgraphicnamed does not do macro expansion as it searches for \endpgfgraphicnamed, it is not necessary to actually include the packages necessary for *creating* the graphics. So the idea is that you comment out things like \usepackage{tikz} and instead say \input pgfexternal.tex.

Indeed, the contents of this file is simply the following line:

```
\long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{\includegraphics{#1}}
```

Instead of \input pgfexternal.tex you could also include this line in your main file.

As a final remark, note that the baseline option does not work directly with pictures written to an external graphic file. The simple reason is that there is no way to store this baseline information in an external graphic file. To allow the baseline option (or any T_EX construction with non-zero depth), the baseline information is stored into a separate file. This file is named $\{\langle image\ file \rangle\}$. dpth and contains something like 5pt.

So, if you need baseline information, you will have to keep the external graphic file together with its .dpth file. Furthermore, the short command in \input pgfexternal.tex is no longer enough because it ignores any baseline information. You will need to use \input pgfexternalwithdepth.tex instead (it is shown below). It is slightly longer, but it can be used in the same way as pgfexternal.tex.

```
/pgf/images/include external
```

(initially \pgfimage{#1})

This key constitutes the public interface to exchange the \includegraphics command used for the image inclusion.

Redefining this key allows to provide bounding box or viewport options:

```
\pgfkeys{/pgf/images/include external/.code={\includegraphics[viewport=0 0 211.28 175.686]{#1}}}
```

Do not forget the .code here which redefines the command.

One application could be image externalization and bounding box restrictions: As far as I know, a .pdf graphics with restricted bounding box is always cropped (which is not always desired). One solution could be to use latex and dvips which doesn't have this restriction. Another is to manually provide the viewport option as shown above.

A possible value for viewport can be found in the .pdf image, search for /MediaBox = [...].

88.5 A Complete Example

Let us now have a look at a simple, but complete example. We start out with a normal file called survey.tex that has the following contents:

```
% This is the file survey.tex
\documentclass{article}

\usepackage{graphics}
\usepackage{tikz}

\begin{document}
In the following figure, we see a circle:
\begin{tikzpicture}
 \fill (0,0) circle (10pt);
\end{tikzpicture}

By comparison, in this figure we see a rectangle:
\begin{tikzpicture}
 \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\end{document}
```

Now our editor tells us that the publisher will need all figures to be provided in separate PostScript or .pdf-files. For this, we enclose all figures in ...graphicnamed-pairs and we add a call to the \pgfrealjobname macro:

```
% This is the file survey.tex
\documentclass{article}
\usepackage{graphics}
\usepackage{tikz}
\pgfrealjobname{survey}
\begin{document}
In the following figure, we see a circle:
\beginpgfgraphicnamed{survey-f1}
\begin{tikzpicture}
  \fill (0,0) circle (10pt);
\end{tikzpicture}
\endpgfgraphicnamed
By comparison, in this figure we see a rectangle:
\beginpgfgraphicnamed{survey-f2}
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\endpgfgraphicnamed
\end{document}
```

After these changes, typesetting the file will still yield the same output as it did before – after all, we have not yet created any external graphics.

To create the external graphics, we run pdflatex twice, once for each graphic:

```
bash> pdflatex --jobname=survey-f1 survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
...
) [1] (./survey-f1.aux) )
Output written on survey-f1.pdf (1 page, 1016 bytes).
Transcript written on survey-f1.log.
```

```
bash> pdflatex --jobname=survey-f2 survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
...
(./survey-f2.aux) )
Output written on survey-f2.pdf (1 page, 1002 bytes).
Transcript written on survey-f2.log.
```

We can now send the two generated graphics (survey-f1.pdf and survey-f2.pdf) to the editor. However, the publisher cannot use our survey.tex file, yet. The reason is that it contains the command \usepackage{tikz} and they do not have PGF installed.

Thus, we modify the main file survey.tex as follows:

```
% This is the file survey.tex
\documentclass{article}
\usepackage{graphics}
\input pgfexternal.tex
% \usepackage{tikz}
% \pgfrealjobname{survey}
\begin{document}
In the following figure, we see a circle:
\beginpgfgraphicnamed{survey-f1}
\begin{tikzpicture}
  \fill (0,0) circle (10pt);
\end{tikzpicture}
\endpgfgraphicnamed
By comparison, in this figure we see a rectangle:
\beginpgfgraphicnamed{survey-f2}
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\endpgfgraphicnamed
\end{document}
```

If we now run pdfIATEX, then, indeed, PGF is no longer needed:

```
bash> pdflatex survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
Babel <v3.8h> and hyphenation patterns for english, ..., loaded.
(/usr/local/gwTeX/texmf.texlive/tex/latex/base/article.cls
Document Class: article 2005/09/16 v1.4f Standard LaTeX document class
(/usr/local/gwTeX/texmf.texlive/tex/latex/base/size10.clo))
(/usr/local/gwTeX/texmf.texlive/tex/latex/graphics/graphics.sty
(/usr/local/gwTeX/texmf.texlive/tex/latex/graphics/trig.sty)
(/usr/local/gwTeX/texmf.texlive/tex/latex/config/graphics.cfg)
(/usr/local/gwTeX/texmf.texlive/tex/latex/pdftex-def/pdftex.def))
(/Users/tantau/Library/texmf/tex/generic/pgf/generic/pgf/utilities/pgfexternal.
tex) (./survey.aux)
(/usr/local/gwTeX/texmf.texlive/tex/context/base/supp-pdf.tex
[Loading MPS to PDF converter (version 2006.09.02).]
> <survey-f1.pdf, id=1, 23.33318pt x 19.99973pt> <use survey-f1.pdf>
<survey-f2.pdf, id=2, 13.33382pt x 10.00037pt> <use survey-f2.pdf> [1{/Users/ta
ntau/Library/texmf/fonts/map/pdftex/updmap/pdftex.map} <./survey-f1.pdf> <./sur
vey-f2.pdf>] (./survey.aux) )</usr/local/gwTeX/texmf.texlive/fonts/type1/bluesk
y/cm/cmr10.pfb>
Output written on survey.pdf (1 page, 10006 bytes).
Transcript written on survey.log.
```

To our editor, we send the following files:

- The last survey.tex shown above.
- The graphic file survey-f1.pdf.
- The graphic file survey-f2.pdf.
- The file pgfexternal.tex, whose contents is simply

```
\long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{\includegraphics{#1}}
```

(Alternatively, we can also directly add this line to our survey.tex file).

In case we have used the baseline option, we also need to include any .dpth files and we need to use the file pgfexternalwithdepth.tex instead of pgfexternal.tex. This file also checks for the existence of .dpth files containing baseline information, its contents is

```
\long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{%
   \begingroup
  \setbox1=\hbox{\includegraphics{#1}}%
  \openin1=#1.dpth
  \ifeof1 \box1
  \else
      \read1 to\pgfincludeexternalgraphicsdp\closein1
      \dimen0=\pgfincludeexternalgraphicsdp\relax
      \hbox{\lower\dimen0 \box1 }%
  \fi
  \endgroup
}
```

Again, we could simply copy these lines to our survey.tex file.

89 Creating Plots

This section describes the plot module.

```
\usepgfmodule{plot} % MEX and plain TeX and pure pgf
\usepgfmodule[plot] % ConTeXt and pure pgf
```

This module provides a set of commands that are intended to make it reasonably easy to plot functions using PGF. It is loaded automatically by pgf, but you can load it manually if you have only included pgfcore.

89.1 Overview

There are different reasons for using PGF for creating plots rather than some more powerful program such as GNUPLOT or MATHEMATICA, as discussed in Section 20.1. So, let us assume that – for whatever reason – you wish to use PGF for generating a plot.

PGF (conceptually) uses a two-stage process for generating plots. First, a plot stream must be produced. This stream consists (more or less) of a large number of coordinates. Second a plot handler is applied to the stream. A plot handler "does something" with the stream. The standard handler will issue line-to operations to the coordinates in the stream. However, a handler might also try to issue appropriate curve-to operations in order to smooth the curve. A handler may even do something else entirely, like writing each coordinate to another stream, thereby duplicating the original stream.

Both for the creation of streams and the handling of streams different sets of commands exist. The commands for creating streams start with \pgfplotstream, the commands for setting the handler start with \pgfplothandler.

89.2 Generating Plot Streams

89.2.1 Basic Building Blocks of Plot Streams

A plot stream is a (long) sequence of the following three commands:

- 1. \pgfplotstreamstart,
- 2. \pgfplotstreampoint, and
- 3. \pgfplotstreamend.

Between calls of these commands arbitrary other code may be called. Obviously, the stream should start with the first command and end with the last command. Here is an example of a plot stream:

```
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\newdimen\mydim
\mydim=2cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\advance \mydim by 3cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\pgfplotstreamend
```

\pgfplotstreamstart

This command signals that a plot stream starts. The effect of this command is to call the internal command \pgf@plotstreamstart, which is set by the current plot handler to do whatever needs to be done at the beginning of the plot.

$\protect\operatorname{pgfplotstreampoint}\{\langle point \rangle\}$

This command adds a $\langle point \rangle$ to the current plot stream. The effect of this command is to call the internal command pgf@plotstreampoint, which is also set by the current plot handler. This command should now "handle" the point in some sensible way. For example, a line-to command might be issued for the point.

\pgfplotstreamend

This command signals that a plot stream ends. It calls \pgf@plotstreamend, which should now do any necessary "cleanup."

Note that plot streams are not buffered, that is, the different points are handled immediately. However, using the recording handler, it is possible to record a stream.

89.2.2 Commands That Generate Plot Streams

Plot streams can be created "by hand" as in the earlier example. However, most of the time the coordinates will be produced internally by some command. For example, the \pgfplotxyfile reads a file and converts it into a plot stream.

$\protect\operatorname{pgfplotxyfile} \langle filename \rangle$

This command will try to open the file $\langle filename \rangle$. If this succeeds, it will convert the file contents into a plot stream as follows: A \pgfplotstreamstart is issued. Then, each nonempty line of the file should start with two numbers separated by a space, such as 0.1 1 or 100 -.3. Anything following the numbers is ignored.

Each pair $\langle x \rangle$ and $\langle y \rangle$ of numbers is converted into one plot stream point in the xy-coordinate system. Thus, a line like

```
2 -5 some text
```

is turned into

```
\pgfplotstreampoint{\pgfpointxy{2}{-5}}
```

The two characters % and # are also allowed in a file and they are both treated as comment characters. Thus, a line starting with either of them is empty and, hence, ignored.

When the file has been read completely, \pgfplotstreamend is called.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabula
```

This command works like \pgfplotxyfile, only three numbers are expected on each non-empty line. They are converted into points in the xyz-coordinate system. Consider, the following file:

```
% Some comments
# more comments
2 -5 1 first entry
2 -.2 2 second entry
2 -5 2 third entry
```

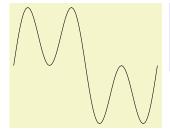
It is turned into the following stream:

```
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{1}}
\pgfplotstreampoint{\pgfpointxyz{2}{-.2}{2}}
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{2}}
\pgfplotstreamend
```

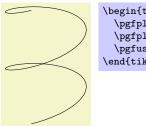
Currently, there is no command that can decide automatically whether the xy-coordinate system should be used or whether the xyz-system should be used. However, it would not be terribly difficult to write a "smart file reader" that parses coordinate files a bit more intelligently.

```
\protection{\langle variable \rangle} {\langle sample \ list \rangle} {\langle point \rangle}
```

This command will produce coordinates by iterating the $\langle variable \rangle$ over all values in $\langle sample\ list \rangle$, which should be a list in the \foreach syntax. For each value of $\langle variable \rangle$, the $\langle point \rangle$ is evaluated and the resulting coordinate is inserted into the plot stream.



```
\begin{tikzpicture}[x=3.8cm/360]
  \pgfplothandlerlineto
  \pgfplotfunction{\x}{0,5,...,360}{\pgfpointxy{\x}{\sin(\x)+\sin(3*\x)}}
  \pgfusepath{\stroke}
\end{tikzpicture}
```



```
\begin{tikzpicture}[y=3cm/360]
  \pgfplothandlerlineto
  \pgfplotfunction{\y}{0,5,...,360}{\pgfpointxyz{sin(2*\y)}}{\y}{cos(2*\y)}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

Be warned that if the expressions that need to evaluated for each point are complex, then this command can be very slow.

```
\protect\operatorname{\mathtt{Npgfplotgnuplot}}[\langle \mathit{prefix} \rangle] \{\langle \mathit{function} \rangle\}
```

This command will "try" to call the GNUPLOT program to generate the coordinates of the $\langle function \rangle$. In detail, the following happens:

This command works with two files: $\langle prefix \rangle$.gnuplot and $\langle prefix \rangle$.table. If the optional argument $\langle prefix \rangle$ is not given, it is set to \jobname.

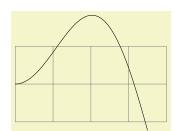
Let us start with the situation where none of these files exists. Then PGF will first generate the file $\langle prefix \rangle$.gnuplot. In this file it writes

```
set terminal table; set output "#1.table"; set format "%.5f"
```

where #1 is replaced by $\langle prefix \rangle$. Then, in a second line, it writes the text $\langle function \rangle$.

Next, PGF will try to invoke the program gnuplot with the argument $\langle prefix \rangle$.gnuplot. This call may or may not succeed, depending on whether the \write18 mechanism (also known as shell escape) is switched on and whether the gnuplot program is available.

Assuming that the call succeeded, the next step is to invoke pgfplotxyfile on the file prefix. table; which is exactly the file that has just been created by gnuplot.



```
\begin{tikzpicture}
  \draw[help lines] (0,-1) grid (4,1);
  \pgfplothandlerlineto
  \pgfplotgnuplot[plots/pgfplotgnuplot-example]{plot [x=0:3.5] x*sin(x)}
  \pgfusepath{stroke}
\end{tikzpicture}
```

The more difficult situation arises when the .gnuplot file exists, which will be the case on the second run of TeX on the TeX file. In this case PGF will read this file and check whether it contains exactly what PGF "would have written" into this file. If this is not the case, the file contents is overwritten with what "should be there" and, as above, gnuplot is invoked to generate a new .table file. However, if the file contents is "as expected," the external gnuplot program is not called. Instead, the \(\prefix \rangle \).table file is immediately read.

As explained in Section 20.6, the net effect of the above mechanism is that gnuplot is called as seldom as possible and that when you pass along the .gnuplot and .table files with your .tex file to someone else, that person can TEX the .tex file without having gnuplot installed and without having the \write18 mechanism switched on.

```
/pgf/plot/gnuplot call=\(gnuplot invocation\) (no default, initially gnuplot)
```

This key can be used to change the way gnuplot is called.

Some portable MikT_FX distribution needs something like the following.

```
\pgfkeys{/pgf/plot/gnuplot call="/Programs/gnuplot/binary/gnuplot"}
```

89.3 Plot Handlers

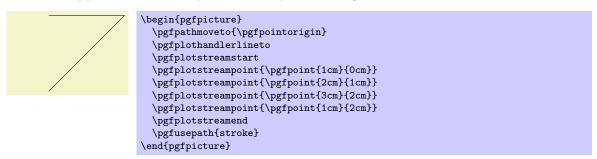
A plot handler determines what "should be done" with a plot stream. You must set the plot handler before the stream starts. The following commands install the most basic plot handlers; more plot handlers are

defined in the file pgflibraryplothandlers, which is documented in Section 44.

All plot handlers work by setting or redefining the following three macros: \pgf@plotstreamstart, \pgf@plotstreampoint, and \pgf@plotstreamend.

\pgfplothandlerlineto

This handler will issue a \pgfpathlineto command for each point of the plot, except possibly for the first. What happens with the first point can be specified using the two commands described below.

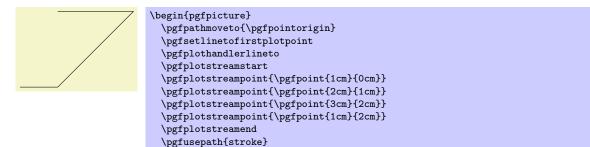


\pgfsetmovetofirstplotpoint

Specifies that the line-to plot handler (and also some other plot handlers) should issue a move-to command for the first point of the plot instead of a line-to. This will start a new part of the current path, which is not always, but often, desirable. This is the default.

\pgfsetlinetofirstplotpoint

Specifies that plot handlers should issue a line-to command for the first point of the plot.



\pgfplothandlerpolygon

This handler works like the line-to plot handler, only the line is closed at the end using \pgfpathclose, resulting in a polygon.

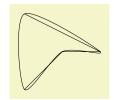
\pgfplothandlerdiscard

This handler will simply throw away the stream.

\end{pgfpicture}

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

When this handler is installed, each time a plot stream command is called, this command will be appended to $\langle macro \rangle$. Thus, at the end of the stream, $\langle macro \rangle$ will contain all the commands that were issued on the stream. You can then install another handler and invoke $\langle macro \rangle$ to "replay" the stream (possibly many times).



```
\begin{pgfpicture}
  \pgfplothandlerrecord{\mystream}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfplotstreamend
  \pgfplothandlerlineto
  \mystream
  \pgfplothandlerclosedcurve
  \mystream
  \pgfusepath{stroke}
  \end{pgfpicture}
```

90 Layered Graphics

90.1 Overview

PGF provides a layering mechanism for composing graphics from multiple layers. (This mechanism is not to be confused with the conceptual "software layers" the PGF system is composed of.) Layers are often used in graphic programs. The idea is that you can draw on the different layers in any order. So you might start drawing something on the "background" layer, then something on the "foreground" layer, then something on the "middle" layer, and then something on the background layer once more, and so on. At the end, no matter in which ordering you drew on the different layers, the layers are "stacked on top of each other" in a fixed ordering to produce the final picture. Thus, anything drawn on the middle layer would come on top of everything of the background layer.

Normally, you do not need to use different layers since you will have little trouble "ordering" your graphic commands in such a way that layers are superfluous. However, in certain situations you only "know" what you should draw behind something else after the "something else" has been drawn.

For example, suppose you wish to draw a yellow background behind your picture. The background should be as large as the bounding box of the picture, plus a little border. If you know the size of the bounding box of the picture at its beginning, this is easy to accomplish. However, in general this is not the case and you need to create a "background" layer in addition to the standard "main" layer. Then, at the end of the picture, when the bounding box has been established, you can add a rectangle of the appropriate size to the picture.

90.2 Declaring Layers

In PGF layers are referenced using names. The standard layer, which is a bit special in certain ways, is called main. If nothing else is specified, all graphic commands are added to the main layer. You can declare a new layer using the following command:

```
\protect\
```

This command declares a layer named $\langle name \rangle$ for later use. Mainly, this will set up some internal bookkeeping.

The next step toward using a layer is to tell PGF which layers will be part of the actual picture and which will be their ordering. Thus, it is possible to have more layers declared than are actually used.

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command tells PGF which layers will be used in pictures. They are stacked on top of each other in the order given. The layer main should always be part of the list. Here is an example:

```
\pgfdeclarelayer{foreground}
\pgfsetlayers{background, main, foreground}
```

This command should be given either outside of any picture or "directly inside" of a picture. Here, the "directly inside" means that there should be no further level of TEX grouping between \pgfsetlayers and the matching \end{pgfpicture} (no closing braces, no \end{...}). It will also work if \pgfsetlayers is provided before \end{tikzpicture} (with similar restrictions).

90.3 Using Layers

Once the layers of your picture have been declared, you can start to "fill" them. As said before, all graphics commands are normally added to the main layer. Using the {pgfonlayer} environment, you can tell PGF that certain commands should, instead, be added to the given layer.

```
\begin{pgfonlayer}{\langle layer\ name \rangle} \\ \langle environment\ contents \rangle \\ \begin{pgfonlayer}
```

The whole $\langle environment\ contents \rangle$ is added to the layer with the name $\langle layer\ name \rangle$. This environment can be used anywhere inside a picture. Thus, even if it is used inside a {pgfscope} or a TeX group, the contents will still be added to the "whole" picture. Using this environment multiple times inside the same picture will cause the $\langle environment\ contents \rangle$ to accumulate.

Note: You can *not* add anything to the main layer using this environment. The only way to add anything to the main layer is to give graphic commands outside all {pgfonlayer} environments.



```
\pgfdeclarelayer{background layer}
\pgfdeclarelayer{foreground layer}
\pgfsetlayers{background layer,main,foreground layer}
\begin{tikzpicture}
  % On main layer:
  \fill[blue] (0,0) circle (1cm);
  \verb|\begin{pgfonlayer}{background layer}|
    fill[yellow] (-1,-1) rectangle (1,1);
  \end{pgfonlayer}
 \begin{pgfonlayer}{foreground layer}
    \node[white] {foreground};
  \end{pgfonlayer}
  \begin{pgfonlayer}{background layer}
    \fill[black] (-.8,-.8) rectangle (.8,.8);
  \end{pgfonlayer}
  % On main layer again:
  \fill[blue!50] (-.5,-1) rectangle (.5,1);
\end{tikzpicture}
```

```
\begin{tabular}{ll} $$ \protect\end{tabular} $$ \protect\end{tabular} $$ \end{tabular} $$ \end{tabular} $$ \end{tabular} $$ \end{tabular} $$ \end{tabular} $$ \end{tabular} $$ \protect\end{tabular} $$ \protect\end{tabula
```

This is the plain T_EX version of the environment.

```
\startpgfonlayer{\langle layer\ name \rangle}
\langle environment\ contents \rangle
\stoppgfonlayer
```

This is the ConTEXt version of the environment.

91 Shadings

91.1 Overview

A shading is an area in which the color changes smoothly between different colors. Similarly to an image, a shading must first be declared before it can be used. Also similarly to an image, a shading is put into a TeX-box. Hence, in order to include a shading in a {pgfpicture}, you have to use \pgftext around it.

There are different kinds of shadings: horizontal, vertical, radial, and functional shadings. However, you can rotate and clip shadings like any other graphics object, which allows you to create more complicated shadings. Horizontal shadings could be created by rotating a vertical shading by 90 degrees, but explicit commands for creating both horizontal and vertical shadings are included for convenience.

Once you have declared a shading, you can insert it into the text using the command \pgfuseshading. This command cannot be used directly in a {pgfpicture}, you have to put a \pgftext around it. The second command for using shadings, \pgfshadepath, on the other hand, can only be used inside {pgfpicture} environments. It will "fill" the current path with the shading.

A horizontal shading is a horizontal bar of a certain height whose color changes smoothly. You must at least specify the colors at the left and at the right end of the bar, but you can also add color specifications for points in between. For example, suppose you wish to create a bar that is red at the left end, green in the middle, and blue at the end, and you would like the bar to be 4cm long. This could be specified as follows:

```
rgb(0cm)=(1,0,0); rgb(2cm)=(0,1,0); rgb(4cm)=(0,0,1)
```

This line means that at 0cm (the left end) of the bar, the color should be red, which has red-green-blue (rgb) components (1,0,0). At 2cm, the bar should be green, and at 4cm it should be blue. Instead of rgb, you can currently also specify gray as color model, in which case only one value is needed, or color, in which case you must provide the name of a color in parentheses. In a color specification the individual specifications must be separated using a semicolon, which may be followed by a whitespace (like a space or a newline). Individual specifications must be given in increasing order.

91.2 Declaring Shadings

91.2.1 Horizontal and Vertical Shadings

 $\verb|\pgfdeclarehorizontalshading[|\langle color| list\rangle]| \{\langle shading| name\rangle\} \{\langle shading| height\rangle\} \{\langle color| specification\rangle\} \}$

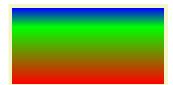
Declares a horizontal shading named $\langle shading\ name \rangle$ of the specified $\langle height \rangle$ with the specified colors. The width of the bar is deduced automatically from the maximum dimension in the specification.

```
\pgfdeclarehorizontalshading{myshadingA}
      {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(green); color(4cm)=(blue)}
      \pgfuseshading{myshadingA}
```

The effect of the $\langle color \ list \rangle$, which is a comma-separated list of colors, is the following: Normally, when this list is empty, once a shading has been declared, it becomes "frozen." This means that even if you change a color that was used in the declaration of the shading later on, the shading will not change. By specifying a $\langle color \ list \rangle$ you can specify that the shading should be recalculated whenever one of the colors listed in the list changes (this includes effects like color mixins). Thus, when you specify a $\langle color \ list \rangle$, whenever the shading is used, PGF first converts the colors in the list to RGB triples using the current values of the colors and taking any mixins and blends into account. If the resulting RGB triples have not yet been used, a new shading is internally created and used. Note that if the option $\langle color \ list \rangle$ is used, then no shading is created until the first use of \pgfuseshading. In particular, the colors mentioned in the shading need not be defined when the declaration is given.

When a shading is recalculated because of a change in the colors mentioned in $\langle color \ list \rangle$, the complete shading is recalculated. Thus even colors not mentioned in the list will be used with their current values, not with the values they had upon declaration.

Declares a vertical shading named $\langle shading\ name \rangle$ of the specified $\langle width \rangle$. The height of the bar is deduced automatically. The effect of $\langle color\ list \rangle$ is the same as for horizontal shadings.



```
\pgfdeclareverticalshading{myshadingC}
  {4cm}{rgb(0cm)=(1,0,0); rgb(1.5cm)=(0,1,0); rgb(2cm)=(0,0,1)}
\pgfuseshading{myshadingC}
```

91.2.2 Radial Shadings

 $\protect\pro$

Declares a radial shading. A radial shading is a circle whose inner color changes as specified by the color specification. Assuming that the center of the shading is at the origin, the color of the center will be the color specified for 0cm and the color of the border of the circle will be the color for the maximum dimension given in the $\langle color\ specified \rangle$. This maximum will also be the radius of the circle. If the $\langle center\ point \rangle$ is not at the origin, the whole shading inside the circle (whose size remains exactly the same) will be distorted such that the given center now has the color specified for 0cm. The effect of $\langle color\ list \rangle$ is the same as for horizontal shadings.



```
\pgfdeclareradialshading{sphere}{\pgfpoint{0.5cm}{0.5cm}}%
   {rgb(0cm)=(0.9,0,0);
   rgb(0.7cm)=(0.7,0,0);
   rgb(1cm)=(0.5,0,0);
   rgb(1.05cm)=(1,1,1)}
\pgfuseshading{sphere}
```

91.2.3 General (Functional) Shadings

 $\label{lower left corner} $$ \left(\operatorname{color list} \right) = \left(\operatorname{corner} \right) = \left(\operatorname{corner}$

Warning: These shadings are the least portable of all and they put the heaviest burden of the renderer. They are slow and, possibly, will not print correctly!

This command creates a functional shading. For such a shading, the color of each point is calculated by calling a function that takes the coordinates of the point as input and yields the color as an output. Note that the function is evaluated by the renderer, not by PGF or TEX or someone else at compile-time. This means that the evaluation of this function has to be done extremely quickly and the function should be very simple. For this reason, only a very restricted set of operations are possible in the function and functions should be kept small. Any errors in the function will only be noticed by the renderer.

The syntax for specifying functions is the following: You use a simplified form of a subset of the PostScript language. This subset will be understood by the PDF-renderer (yes, PDF-renderers do have a basic understanding of PostScript) and also by PostScript renders. This subset is detailed in Section 3.9.4 of the PDF-specification (version 1.7). In essence, the specification states that these functions may contain "expressions involving integers, real numbers, and boolean values only. There are no composite data structures such as strings or arrays, no procedures, and no variables or names." The allowed operators are (exactly) the following: abs, add, atan, ceiling, cos, cvi, cvr, div, exp, floor, idiv, ln, log, mod, mul, neg, round, sin, sqrt, sub, truncate, and, bitshift, eq, false, ge, gt, le, lt, ne, not, or, true, xor, if, ifelse, copy, dup, exch, index, pop.

When the function is evaluated, the top two stack elements are the coordinates of the point for which the color should be computed. The coordinates are dimensionless and given in big points, so for the coordinate (50bp, 72.27pt) the top two stack elements would be 50.0 and 72.0. Otherwise, the (virtual) stack is empty (or should be treated as if it were empty). The function should then replace these two values by three values, representing the red, green, and blue color of the point. The numbers should be real values, not integers since, Apple's PDF renderer is broken in this regard (use cvr at the end if necessary).

Conceptually, the function will be evaluated once for each point of the rectangle \(\lambda \) lower \(\left \) corner \(\rangle \) to \(\lambda \) upper \(right \) corner \(\rangle \), which should be a PGF-point expression like \(\rangle \) pgfpoint{100bp}{100bp}. A renderer may choose to evaluate the function at less points, but, in principle, the function will be evaluated for each pixel independently.

Because of the rather difficult PostScript syntax, use this macro only if you know what you are doing (or if you are adventurous, of course).

As for other shadings, the optional $\langle color \ list \rangle$ is used to determine whether a shading needs to be recalculated when a color has changed.

The $\langle init\ code \rangle$ is executed each time a shading is (re)calculated. Typically, it will contain code to extract coordinates from colors.



```
\pgfdeclarefunctionalshading{twospots}
   {\pgfpointorigin}{\pgfpoint{4cm}{4cm}}{}
  % Save coordinates for later
 2 сору
 % Compute distance from (40bp,45bp), with x doubled
 45 sub dup mul exch
 40 sub dup mul 0.5 mul add sqrt
 % exponential decay
 dup mul neg 1.0005 exch exp 1.0 exch sub
 % Compute distance from (70bp,70bp) from stored coordinate, scaled
 3 1 roll
 70 sub dup mul .5 mul exch
 70 sub dup mul add sgrt
 % Decay
 dup mul neg 1.002 exch exp 1.0 exch sub
 % red component
 1.0 3 1 roll
\pgfuseshading{twospots}
```

Inside the PostScript function $\langle type \ 4 \ function \rangle$ you cannot use colors directly. Rather, you must push the color components on the stack. For this, it is useful to call \pgfshadecolortorgb in the $\langle init \ code \rangle$:

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{
```

This command takes $\langle color \ name \rangle$ as input and stores the color's red/green/blue components real numbers between 0.0 and 1.0 separated by spaces (which is exactly what you need if you want to push it on a stack) in $\langle macro \rangle$. This macro can then be used inside the $\langle type\ 4\ function \rangle$ argument for \pgfdeclarefunctionalshading.

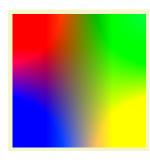


```
\pgfdeclarefunctionalshading[mycol]{sweep}{\pgfpoint{-1cm}{-1cm}}
{\pgfpoint{1cm}{1cm}}{\pgfshadecolortorgb{mycol}{\myrgb}}{
 2 сору
                % whirl
  % Calculate "safe" atan of position
 2 copy abs exch abs add 0.0001 ge { atan } { pop } ifelse
 3 1 roll
 dup mul exch
 dup mul add sqrt
 30 mul
 add
 sin
 1 add 2 div
 dup
 \myrgb
                % push mycol
 5 4 roll
               % multiply all components by calculated value
 mııl
 3 1 roll
 3 index
 mul
 3 1 roll
 4 3 roll
 mul
 3 1 roll
\colorlet{mycol}{white}%
\pgfuseshading{sweep}%
\colorlet{mycol}{red}%
\pgfuseshading{sweep}
```

In addition, three macros suffixed with red, green and blue are defined, which store the individual components of $\langle color \ name \rangle$. These can also be used in the $\langle type \ 4 \ function \rangle$ argument.

 $\mbox{\t mycol}=1.0 \ 0.5 \ 0.0 \ \mbox{\t mycol} =1.0 \ \mbox{\t mycol} =0.5 \ \mbox{\t mycol} =0.0$

```
\pgfshadecolortorgb{orange}{\mycol} |\mycol|=\mycolpreen|=\mycolgreen |\mycolblue|=\mycolblue
```



```
\pgfdeclarefunctionalshading[col1,col2,col3,col4]{bilinear interpolation}
{\pgfpointorigin}{\pgfpoint{100bp}{100bp}}
\pgfshadecolortorgb{col1}{\first}\pgfshadecolortorgb{col2}{\second}
\pgfshadecolortorgb{col3}{\third}\pgfshadecolortorgb{col4}{\fourth}
 100 div exch 100 div 2 copy
                                                 % Calculate y/100 x/100.
                                                % Calculate 1-y/100 1-x/100.
 neg 1 add exch neg 1 add
 3 1 roll 2 copy exch 5 2 roll 6 copy 6 copy
                                                % Set up stack.
 \firstred mul exch \secondred mul add mul
                                                % Process red component.
 4 1 roll
 \thirdred mul exch \fourthred mul add mul
 add
 13 1 roll
 \firstgreen mul exch \secondgreen mul add mul % Process green component.
 4 1 roll
 \thirdgreen mul exch \fourthgreen mul add mul
 add
 7 1 roll
 \firstblue mul exch \secondblue mul add mul
                                                % Process blue component.
 4 1 roll
 \thirdblue mul exch \fourthblue mul add mul
 add
\colorlet{col1}{blue}
\colorlet{col2}{vellow}
\colorlet{col3}{red}
\colorlet{col4}{green}
\pgfuseshading{bilinear interpolation}
```

91.3 Using Shadings

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Inserts a previously declared shading into the text. If you wish to use it in a pgfpicture environment, you should put a \pgfbox around it.



```
\begin{pgfpicture}
  \pgfdeclareverticalshading{myshadingD}
  {20pt}{color(0pt)=(red); color(20pt)=(blue)}
  \pgftext[at=\pgfpoint{1cm}{0cm}] {\pgfuseshading{myshadingD}}
  \pgftext[at=\pgfpoint{2cm}{0.5cm}]{\pgfuseshading{myshadingD}}
\end{pgfpicture}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

This command must be used inside a {pgfpicture} environment. The effect is a bit complex, so let us go over it step by step.

First, PGF will set up a local scope.

Second, it uses the current path to clip everything inside this scope. However, the current path is once more available after the scope, so it can be used, for example, to stroke it.

Now, the $\langle shading\ name \rangle$ should be a shading whose width and height are 100 bp, that is, 100 big points. PGF has a look at the bounding box of the current path. This bounding box is computed automatically when a path is computed; however, it can sometimes be (quite a bit) too large, especially when complicated curves are involved.

Inside the scope, the low-level transformation matrix is modified. The center of the shading is translated (moved) such that it lies on the center of the bounding box of the path. The low-level coordinate system is

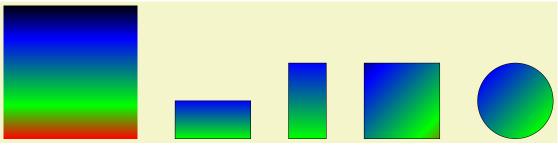
also scaled such that the shading "covers" the path (the details are a bit more complex, see below). Then, the coordinate system is rotated by $\langle angle \rangle$. Finally, if the macro \pgfsetadditionalshadetransform has been used, an additional transformation is applied.

After everything has been set up, the shading is inserted. Due to the transformations and clippings, the effect will be that the shading seems to "fill" the path.

If both the path and the shadings were always rectangles and if rotations were never involved, it would be easy to scale shadings such they always cover the path. However, when a vertical shading is rotated, it must obviously be "magnified" so that it still covers the path. Things get worse when the path is not a rectangle itself.

For these reasons, things work slightly differently "in reality." The shading is scaled and translated such that the point (50bp, 50bp), which is the middle of the shading, is at the middle of the path and such that the point (25bp, 25bp) is at the lower left corner of the path and that (75bp, 75bp) is at upper right corner.

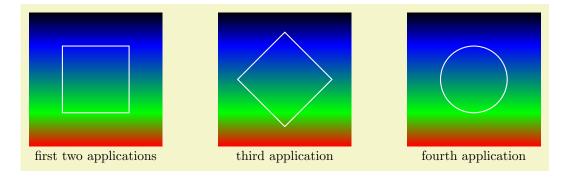
In other words, only the center quarter of the shading will actually "survive the clipping" if the path is a rectangle. If the path is not a rectangle, but, say, a circle, even less is seen of the shading. Here is an example that demonstrates this effect:



```
\pgfdeclareverticalshading{myshadingE}{100bp}
{color(0bp)=(red); color(25bp)=(green); color(75bp)=(blue); color(100bp)=(black)}
\pgfuseshading{myshadingE}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingE}{0}
 \pgfusepath{stroke}
 \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{1cm}{2cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
 \pgfpathrectangle{\pgfpoint{5cm}{0cm}}{\pgfpoint{2cm}{2cm}}
  \pgfshadepath{myshadingE}{45}
 \pgfusepath{stroke}
 \pgfpathcircle{\pgfpoint{9cm}{1cm}}{1cm}
 \pgfshadepath{myshadingE}{45}
 \pgfusepath{stroke}
\end{pgfpicture}
```

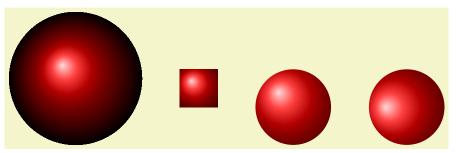
As can be seen above in the last case, the "hidden" part of the shading actually *can* become visible if the shading is rotated. The reason is that it is scaled as if no rotation took place, then the rotation is done.

The following graphics show which part of the shading are actually shown:



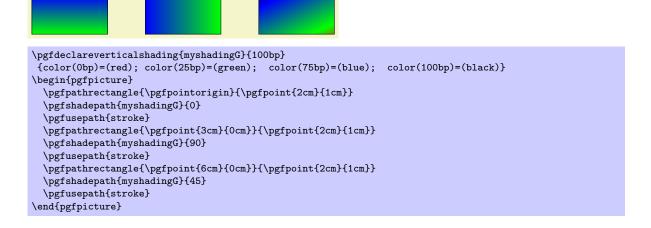
```
\pgfdeclareverticalshading{myshadingF}{100bp}
{color(0bp)=(red); color(25bp)=(green); color(75bp)=(blue); color(100bp)=(black)}
\begin{tikzpicture}
  \draw (50bp,50bp) node {\pgfuseshading{myshadingF}};
 \draw[white,thick] (25bp,25bp) rectangle (75bp,75bp);
 \draw (50bp,0bp) node[below] {first two applications};
 \begin{scope}[xshift=5cm]
   \draw (50bp,50bp) node{\pgfuseshading{myshadingF}};
   \draw[rotate around={45:(50bp,50bp)},white,thick] (25bp,25bp) rectangle (75bp,75bp);
   \draw (50bp,0bp) node[below] {third application};
 \end{scope}
 \begin{scope}[xshift=10cm]
   \draw (50bp,50bp) node{\pgfuseshading{myshadingF}};
   \draw[white,thick] (50bp,50bp) circle (25bp);
   \draw (50bp,0bp) node[below] {fourth application};
 \end{scope}
\end{tikzpicture}
```

An advantage of this approach is that when you rotate a radial shading, no distortion is introduced:

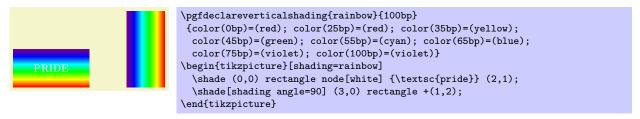


```
\pgfdeclareradialshading{ballshading}{\pgfpoint{-10bp}{10bp}}
{color(0bp)=(red!15!white); color(9bp)=(red!75!white);
color(18bp)=(red!70!black); color(25bp)=(red!50!black); color(50bp)=(black)}
\pgfuseshading{ballshading}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
 \pgfshadepath{ballshading}{0}
 \pgfusepath{}
 \pgfpathcircle{\pgfpoint{3cm}{0cm}}{1cm}
 \pgfshadepath{ballshading}{0}
 \pgfusepath{}
 \pgfpathcircle{\pgfpoint{6cm}{0cm}}{1cm}
 \pgfshadepath{ballshading}{45}
 \pgfusepath{}
\end{pgfpicture}
```

If you specify a rotation of 90° and if the path is not a square, but an elongated rectangle, the "desired" effect results: The shading will exactly vary between the colors at the 25bp and 75bp boundaries. Here is an example:



As a final example, let us define a "rainbow spectrum" shading for use with TikZ.



Note that rainbow shadings are way to colorful in almost all applications.

$\verb|\pgfsetadditionalshadetransform{|\langle transformation \rangle|}|$

This command allows you to specify an additional transformation that should be applied to shadings when the $\protect\operatorname{pgfshadepath}$ command is used. The $\langle transformation \rangle$ should be transformation code like $\protect\operatorname{pgftransformatie}\{20\}$.

92 Transparency

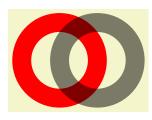
For an introduction to the notion of transparency, fadings, and transparency groups, please consult Section 21.

92.1 Specifying a Uniform Opacity

Specifying a stroke and/or fill opacity is quite easy.

$\protect\pro$

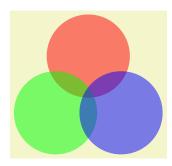
Sets the opacity of stroking operations. The $\langle value \rangle$ should be a number between 0 and 1, where 1 means "fully opaque" and 0 means "fully transparent." A value like 0.5 will cause paths to be stroked in a semitransparent way.



```
\begin{pgfpicture}
  \pgfsetlinewidth{5mm}
  \color{red}
  \pgfpathcircle{\pgfpoint{0cm}{10mm} \pgfusepath{stroke}
  \color{black}
  \pgfsetstrokeopacity{0.5}
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{10mm} \pgfusepath{stroke}
  \end{pgfpicture}
```

$\protect\pro$

Sets the opacity of filling operations. As for stroking, the $\langle value \rangle$ should be a number between 0 and 1. The "filling transparency" will also be used for text and images.



```
\begin{tikzpicture}
\pgfsetfillopacity{0.5}
\fill[red] (90:1cm) circle (11mm);
\fill[green] (210:1cm) circle (11mm);
\fill[blue] (-30:1cm) circle (11mm);
\end{tikzpicture}
```

Note the following effect: If you set up a certain opacity for stroking or filling and you stroke or fill the same area twice, the effect accumulates:

```
\begin{tikzpicture}
  \pgfsetfillopacity{0.5}
  \fill[red] (0,0) circle (1);
  \fill[red] (1,0) circle (1);
  \end{tikzpicture}
```

Often, this is exactly what you intend, but not always. You can use transparency groups, see the end of this section, to change this.

92.2 Specifying a Fading

The method used by PGF for specifying fadings is quite general: You "paint" the fading using any of the standard graphics commands. In more detail: You create a normal picture, which may even contain text, image, and shadings. Then, you create a fading based on this picture. For this, the *luminosity* of each pixel of the picture is analyzed (the brighter the pixel, the higher the luminosity – a black pixel has luminosity 0, a white pixel has luminosity 1, a gray pixel has some intermediate value as does a red pixel). Then, when the fading is used, the luminosity of the pixel determines the opacity of the fading at that position. Positions in the fading where the picture was black will be completely transparent, positions where the picture was white will be completely opaque. Positions that have not been painted at all in the picture are always completely transparent.

$\protect\pro$

This command declares a fading named $\langle name \rangle$ for later use. The "picture" on which the fading is based is given by the $\langle contents \rangle$. The $\langle contents \rangle$ are normally typeset in a TEX box. The resulting box is then used as the "picture." In particular, inside the $\langle contents \rangle$ you must explicitly open a {pgfpicture} environment if you wish to use PGF commands.

Let's start with an easy example. Our first fading picture is just some text:



```
\pgfdeclarefading{fading1}{\color{white}Ti\emph{k}Z}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading1}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
  \end{tikzpicture}
```

What's happening here? The "fading picture" is mostly transparent, except for the pixels that are part of the word TikZ. Now, these pixels are *white* and, thus, have a high luminosity. This in turn means that these pixels of the fading will be highly opaque. For this reason, only those pixels of the big red rectangle "shine through" that are at the positions of these opaque pixels.

It is somewhat counter-intuitive that the white pixels in a fading picture are opaque in a fading. For this reason, the color pgftransparent is defined to be the same as black. This allows one to write pgftransparent for completely transparent parts of a fading picture and pgftransparent!0 for the opaque parts and things like pgftransparent!20 for parts that are 20% transparent.

Furthermore, the color pgftransparent!0 (which is the same as white and which corresponds to completely opaque) is installed at the beginning of a fading picture. Thus, in the above example the \color{white} was not really necessary.

Next, let us create a fading that gets more and more transparent as we go from left to right. For this, we put a shading inside the fading picture that has the color pgftransparent!0 at the left-hand side and the color pgftransparent!100 at the right-hand side.



In our final example, we create a fading that is based on a radial shading.



```
\pgfdeclareradialshading{myshading}{\pgfpointorigin}
{
  color(0mm)=(pgftransparent!0);
  color(5mm)=(pgftransparent!100);
  color(15mm)=(pgftransparent!100)
}
\pgfdeclarefading{fading3}{\pgfuseshading{myshading}}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading3}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
  \fill [red] (0,0) rectangle (2,2);
  \end{tikzpicture}
```

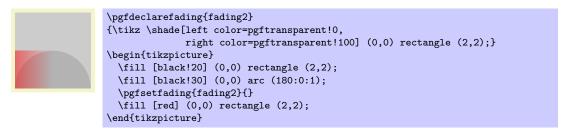
After having declared a fading, we can use it. As for shadings, there are two different commands for using fadings:

```
\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\protect\pro
```

This command sets the graphic state parameter "fading" to a previously defined fading $\langle name \rangle$. This graphic state works like other graphic states, that is, is persists till the end of the current scope or until a different transparency setting is chosen.

When the fading is installed, it will be centered on the origin with its natural size. Anything outside the fading picture's original bounding box will be transparent and, thus, the fading effectively clips against this bounding box.

The $\langle transformations \rangle$ are applied to the fading before it is used. They contain normal PGF transformation commands like \pgftransformshift. You can also scale the fading using this command. Note, however, that the transformation needs to be inverted internally, which may result in inaccuracies and the following graphics may be slightly distorted if you use a strong $\langle transformation \rangle$.



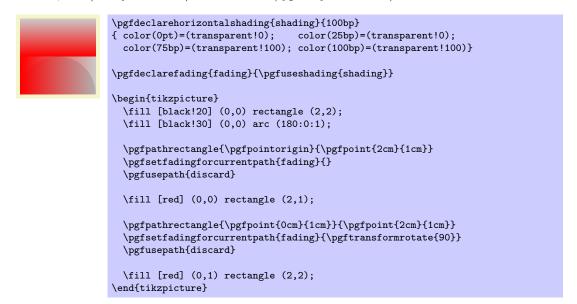


$\protect\pro$

This command works like \pgfsetfading, but the fading is scaled and transformed according to the following rules:

- 1. If the current path is empty, the command has the same effect as \pgfsetfading.
- 2. Otherwise it is assumed that the fading has a size of 100bp times 100bp.
- 3. The fading is resized and shifted (using appropriate transformations) such that the position (25bp, 25bp) lies at the lower-left corner of the current path and the position (75bp, 75bp) lies at the upper-right corner of the current path.

Note that these rules are the same as the ones used in $\protect\operatorname{pgfshadepath}$ for shadings. After these transformations, the $\langle transformations \rangle$ are executed (typically a rotation).



92.3 Transparency Groups

Transparency groups are declared using the following commands.

```
\begin{pgftransparencygroup}
     ⟨environment contents⟩
\end{pgftransparencygroup}
```

This environment should only be used inside a {pgfpicture}. It has the following effect:

- 1. The \(\langle environment contents \rangle \) are stroked/filled "ignoring any outside transparency." This means, all previous transparency settings are ignored (you can still set transparency inside the group, but never mind). This means that if in the \(\langle environment contents \rangle \) you stroke a pixel three times in black, it is just black. Stroking it white afterwards yields a white pixel, and so on.
- 2. When the group is finished, it is painted as a whole. The *fill* transparency settings are now applied to the resulting picture. For instance, the pixel that has been painted three times in black and once in white is just white at the end, so this white color will be blended with whatever is "behind" the group on the page.

Note that, depending on the driver, PGF may have to guess the size of the contents of the transparency group (because such a group is put in an XForm in PDF and a bounding box must be supplied). PGF will use normally use the size of the picture's bounding box at the end of the transparency group plus a safety margin of 1cm. Under normal circumstances, this will work nicely since the picture's bounding box contains everything anyway. However, if you have switched off the picture size tracking or if you are using canvas transformations, you may have to make sure that the bounding box is big enough. The trick is to locally create a picture that is "large enough" and then insert this picture into the main picture while ignoring the size. The following example shows how this is done:



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);

% Stuff outside the picture, but still in a transparency group.
  \node [left,overlay] at (0,1) {
   \begin{tikzpicture}
     \pgfsetfillopacity{0.5}
     \pgftransparencygroup
   \node at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
     {Smoking};
   \endpgftransparencygroup
  \end{tikzpicture}
};
\end{tikzpicture}
```

$\langle environment\ contents \rangle$

\endpgftransparencygroup

Plain TEX version of the {pgftransparencygroup} environment.

```
\startpgftransparencygroup
\langle environment contents \rangle
\stoppgftransparencygroup
```

This is the ConT_EXt version of the environment.

93 Adding libraries to pgf: temporary registers

This section is intended for those who like to write libraries to extend PGF. Of course, this requires a good deal of knowledge about TEX-programming and the structure of the PGF basic layer. Besides, one will encounter the need of temporary variables and, especially, temporary TEX registers. This section describes how to use a set of pre-allocated temporary registers of the basic layer without needing to allocate more of them.

A part of these internals are already mentioned in section 78.7, but the basic layer provides more temporaries than \pgf@x and \pgf@y.

```
Internal dimen register \pgf@x
Internal dimen register \pgf@y
```

These registers are used to process point coordinates in the basic layer of PGF, see section 78.7. After a $\protect\$

The values of \pgf@x and \pgf@y are set globally in contrast to other available PGF registers. You should never assume anything about their value unless the context defines them explicitly.

Please prefer the \pgf@xa, \pgf@xb, ... registers for temporary dimen registers unless you are writing point coordinate commands.

```
Internal dimen register \pgf@xa
Internal dimen register \pgf@xb
Internal dimen register \pgf@xc
Internal dimen register \pgf@ya
Internal dimen register \pgf@yb
Internal dimen register \pgf@yc
```

Temporary registers for T_EX dimensions which can be modified freely. Just make sure changes occur only within T_EX groups.

Attention: PGF uses these registers to perform path operations. For reasons of efficiency, path commands do not always guard them. As a consequence, the code

```
\pgfpointadd{\pgfpoint{\pgf@xa}{\pgfpoint{\pgf@xb}{\pgfgvb}}
```

may fail: Inside \pgfpointadd, the \pgf@xa and friend registers might be modified. In particular, it might happen that \pgf@xb is changed before \pgfpoint{\pgf@xb}{\pgf@yb} is evaluated. The right thing to do would be to first expand everything using \edef and process the values afterwards, resulting in unnecessary expensive operations. Of course, one can avoid this by simply looking into the source code of \pgfpointadd to see which registers are used.

```
Internal dimen register \pgfutil@tempdima
Internal dimen register \pgfutil@tempdimb
```

Further multi-purpose temporary dimen registers. For LATEX, these registers are already allocated as \Otempdima and \Otempdimb and are simply \let to the \pgfutilO... names.

```
Internal count register \c@pgf@counta
Internal count register \c@pgf@countb
Internal count register \c@pgf@countc
Internal count register \c@pgf@countd
```

These multiple-purpose count registers are used throughout PGF to perform integer computations. Feel free to use them as well, just make sure changes are scoped by local T_FX groups.

Internal openout handle \w@pgf@writea

An \openout handle which is used to generate complete output files within locally scoped parts of PGF (for example, to interact with gnuplot). You should always use \immediate in front of output operations involving \w@pgf@writea and you should always close the file before returning from your code.

```
\immediate\openout\w@pgf@writea=myfile.dat
\immediate\write\w@pgf@writea{...}%
\immediate\write\w@pgf@writea{...}%
\immediate\closeout\w@pgf@writea%
```

Internal openin handle \r@pgf@reada

An \openin handle which is used to read files within locally scoped parts of PGF, for example to check if a file exists or to read data files. You should always use \immediate in front of output operations involving \r@pf@writea and you should always close the file before returning from your code.

Internal box \pgfutil@tempboxa

A box for temporary use inside of local TeX scopes. For LaTeX, this box is the same as the already pre-allocated \Otempboxa.

94 Quick Commands

This section explains the "quick" commands of PGF. These commands are executed more quickly than the normal commands of PGF, but offer less functionality. You should use these commands only if you either have a very large number of commands that need to be processed or if you expect your commands to be executed very often.

94.1 Quick Coordinate Commands

```
\protect\operatorname{\mathsf{pgfqpoint}}\{\langle x\rangle\}\{\langle y\rangle\}
```

This command does the same as pgfpoint, but $\langle x \rangle$ and $\langle y \rangle$ must be simple dimensions like 1pt or 1cm. Things like 2ex or 2cm+1pt are not allowed.

```
\pgfqpointxy\{\langle s_x \rangle\}\{\langle s_y \rangle\}
```

This command does the same as $\protect\operatorname{pgfpointxy}$, but $\langle s_x \rangle$ and $\langle s_y \rangle$ must be simple numbers without unit, like 1.234 or 5.0. Mathematical expressions or units are not allowed.

```
\protect\operatorname{pgfqpointxyz}\{\langle s_x \rangle\}\{\langle s_y \rangle\}\{\langle s_z \rangle\}
```

As \pgfqpointxy, but for three-dimensional coordinates. Any argument needs to be a number without unit.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

As $\protect\operatorname{\mathtt{Ngfpointscale}}$, but $\{\langle factor \rangle\}$ must be a simple number without unit, as for the other "quick" commands.

94.2 Quick Path Construction Commands

The difference between the quick and the normal path commands is that the quick path commands

- do not keep track of the bounding boxes,
- do not allow you to arc corners,
- do not apply coordinate transformations.

However, they do use the soft-path subsystem (see Section 97 for details), which allows you to mix quick and normal path commands arbitrarily.

All quick path construction commands start with \pgfpathq.

```
\protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta
```

Either starts a path or starts a new part of a path at the coordinate ($\langle x \ dimension \rangle$, $\langle y \ dimension \rangle$). The coordinate is *not* transformed by the current coordinate transformation matrix. However, any low-level transformations apply.

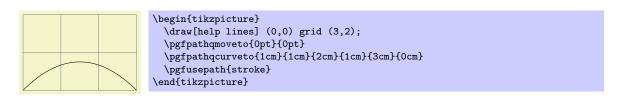


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformxshift{1cm}
  \pgfpathqmoveto{0pt}{0pt} % not transformed
  \pgfpathqlineto{1cm}{1cm} % not transformed
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
  \end{tikzpicture}
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

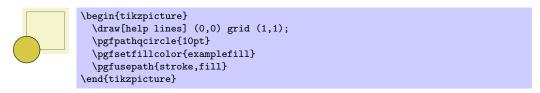
The quick version of the line-to operation.

The quick version of the curve-to operation. The first support point is (s_x^1, s_y^1) , the second support point is (s_x^2, s_y^2) , and the target is (t_x, t_y) .



$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Adds a radius around the origin of the given $\langle radius \rangle$. This command is orders of magnitude faster than $\protect\operatorname{pgfpointorigin}\{\langle radius \rangle\}$.



94.3 Quick Path Usage Commands

The quick path usage commands perform similar tasks as \pgfusepath, but they

- do not add arrows,
- do not modify the path in any way, in particular,
- ends are not shortened.
- corners are not replaced by arcs.

Note that you have to use the quick versions in the code of arrow tip definitions since, inside these definition, you obviously do not want arrows to be drawn.

\pgfusepathqstroke

Strokes the path without further ado. No arrows are drawn, no corners are arced.

```
\begin{pgfpicture}
\pgfpathqcircle{5pt}
\pgfusepathqstroke
\end{pgfpicture}
```

\pgfusepathqfill

Fills the path without further ado.

\pgfusepathqfillstroke

Fills and then strokes the path without further ado.

\pgfusepathqclip

Clips all subsequent drawings against the current path. The path is not processed.

94.4 Quick Text Box Commands

$\protect\pro$

This command inserts a T_EX box into a {pgfpicture} by "escaping" to T_EX, inserting the box number \(box number \) at the origin, and then returning to the typesetting the picture.

$\protect\operatorname{\mathsf{pgfqboxsynced}}\{\langle box\ number\rangle\}$

This command works similarly to the $\protect\operatorname{pgfqbox}$ command. However, before inserting the text in $\langle box number \rangle$, the current coordinate transformation matrix is applied to the current canvas transformation matrix (is it "synced" with this matrix, hence the name).

Thus, this command basically has the same effect as if you first called \pgflowlevelsynccm followed by \pgfqbox . However, this command will use \hskip and \raise commands for the "translational part" of the coordinate transformation matrix, instead of adding the translational part to the current canvas transformation matrix directly. Both methods have the same effect (box $\langle box\ number \rangle$ is translated to where it should be), but the method used by \pgfqboxsynced ensures that hyperlinks are placed correctly. Note that scaling and rotation will not (cannot, even) apply to hyperlinks.

Part X

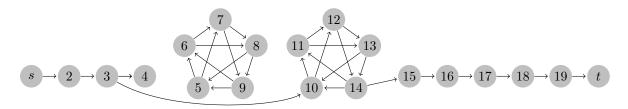
The System Layer

by Till Tantau

This part describes the low-level interface of PGF, called the *system layer*. This interface provides a complete abstraction of the internals of the underlying drivers.

Unless you intend to port PGF to another driver or unless you intend to write your own optimized frontend, you need not read this part.

In the following it is assumed that you are familiar with the basic workings of the graphics package and that you know what TFX-drivers are and how they work.



```
\begin{tikzpicture}
 [shorten >=1pt,->,
  vertex/.style={circle,fill=black!25,minimum size=17pt,inner sep=0pt}]
 \frac{\n}{16} in \frac{\n}{16}, \frac{\n}{16}, \frac{\n}{16}, \frac{\n}{16}, \frac{\n}{16}, \frac{\n}{16}, \frac{\n}{16}
   \node[vertex] (G-\nome) at (\x,0) {{\rm ame}};
 \foreach \name/\angle/\text in {P-1/234/5, P-2/162/6, P-3/90/7, P-4/18/8, P-5/-54/9}
   \node[vertex,xshift=6cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};
 \frac{q-1/234/10, q-2/162/11, q-3/90/12, q-4/18/13, q-5/-54/14}{1}
   \node[vertex,xshift=9cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};
 \draw (G-\from) -- (G-\to);
 foreach from/to in {1/2,2/3,3/4,4/5,5/1,1/3,2/4,3/5,4/1,5/2}
   { \draw (P-\from) -- (P-\to); \draw (Q-\from) -- (Q-\to); }
 \draw (G-3) ... controls +(-30:2cm) and +(-150:1cm) ... (Q-1);
 \draw (Q-5) -- (G-15);
\end{tikzpicture}
```

95 Design of the System Layer

95.1 Driver Files

The PGF system layer mainly consists of a large number of commands starting with \pgfsys0. These commands will be called *system commands* in the following. The higher layers "interface" with the system layer by calling these commands. The higher layers should never use \special commands directly or even check whether \pdfoutput is defined. Instead, all drawing requests should be "channeled" through the system commands.

The system layer is loaded and set up by the following package:

```
\usepackage{pgfsys} % MEX
\input pgfsys.tex % plain TEX
\usemodule[pgfsys] % ConTEXt
```

This file provides "default implementations" of all system commands, but most simply produce a warning that they are not implemented. The actual implementations of the system commands for a particular driver like, say, pdftex reside in files called pgfsys-xxxx.sty, where xxxx is the driver name. These will be called driver files in the following.

When pgfsys.sty is loaded, it will try to determine which driver is used by loading pgf.cfg. This file should set up the macro \pgfsysdriver appropriately. The pgfsys.sty will input the appropriate pgfsys-\drivername\.sty.

\pgfsysdriver

This macro should expand to the name of the driver to be used by pgfsys. The default from pgf.cfg is pgfsys-\Gin@driver. This is very likely to be correct if you are using IATEX. For plain TEX, the macro will be set to pgfsys-pdftex.def if pdftex is used and to pgfsys-dvips.def otherwise.

File pgf.cfg

This file should set up the command \pgfsysdriver correctly. If \pgfsysdriver is already set to some value, the driver normally should not change it. Otherwise, it should make a "good guess" at which driver will be appropriate.

The currently supported backend drivers are discussed in Section 10.2.

95.2 Common Definition Files

Some drivers share many \pgfsys@ commands. For the reason, files defining these "common" commands are available. These files are not usable alone.

File pgfsys-common-postscript

This file defines some \pgfsys@ commands so that they produce appropriate PostScript code.

File pgfsys-common-pdf

This file defines some \pgfsys@ commands so that they produce appropriate PDF code.

96 Commands of the System Layer

96.1 Beginning and Ending a Stream of System Commands

A "user" of the PGF system layer (like the basic layer or a frontend) will interface with the system layer by calling a stream of commands starting with \pgfsys@. From the system layer's point of view, these commands form a long stream. Between calls to the system layer, control goes back to the user.

The driver files implement system layer commands by inserting \special commands that implement the desired operation. For example, \pgfsys@stroke will be mapped to \special{pdf: S} by the driver file for pdftex.

For many drivers, when such a stream of specials starts, it is necessary to install an appropriate transformation and perhaps perform some more bureaucratic tasks. For this reason, every stream will start with a \pgfsys@beginpicture and will end with a corresponding ending command.

\pgfsys@beginpicture

Called at the beginning of a {pgfpicture}. This command should "set up things."

Most drivers will need to implement this command.

\pgfsys@endpicture

Called at the end of a {pgfpicture}.

Most drivers will need to implement this command.

$\protect\operatorname{\footnotemap}{\operatorname{\footnotemap}}$

Called after a {pgfpicture} has been typeset. The picture will have been put in box $\langle box \rangle$. This command should insert the box into the normal text. The box $\langle box \rangle$ will still be a "raw" box that contains only the \special's that make up the description of the picture. The job of this command is to resize and shift $\langle box \rangle$ according to the baseline shift and the size of the box.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@beginpurepicture

This version of the \pgfsys@beginpicture picture command can be used for pictures that are guaranteed not to contain any escaped boxes (see below). In this case, a driver might provide a more compact version of the command.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@endpurepicture

Called at the end of a "pure" {pgfpicture}.

This command has a default implementation and need not be implemented by a driver file.

Inside a stream it is sometimes necessary to "escape" back into normal typesetting mode; for example to insert some normal text, but with all of the current transformations and clippings being in force. For this escaping, the following command is used:

$\protect\pro$

Called to insert a (horizontal) TeX box inside a {pgfpicture}.

Most drivers will need to (re-)implement this command.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

Called to insert a (horizontal) TeX box inside a {pgfpicture}, but with the current coordinate transformation matrix synced with the canvas transformation matrix.

This command should do the same as if you used \pgflowlevelsynccm followed by \pgfsys@hbox. However, the default implementation of this command will use a "TeX-translation" for the translation part of the transformation matrix. This will ensure that hyperlinks "survive" at least translations. On the other hand, a driver may choose to revert to a simpler implementation. This is done, for example, for the SVG implementation, where a TeX-translation makes no sense.

96.2 Path Construction System Commands

$\protect\$ \pgfsys@moveto{ $\langle x \rangle$ }{ $\langle y \rangle$ }

This command is used to start a path at a specific point (x, y) or to move the current point of the current path to (x, y) without drawing anything upon stroking (the current path is "interrupted").

Both $\langle x \rangle$ and $\langle y \rangle$ are given as TEX dimensions. It is the driver's job to transform these to the coordinate system of the backend. Typically, this means converting the TEX dimension into a dimensionless multiple of $\frac{1}{72}$ in. The function \pgf@sys@bp helps with this conversion.

Example: Draw a line from (10pt, 10pt) to the origin of the picture.

```
\pgfsys@moveto{10pt}{10pt}
\pgfsys@lineto{0pt}{0pt}
\pgfsys@stroke
```

This command is protocolled, see Section 98.

\pgfsys@lineto $\{\langle x \rangle\}\{\langle y \rangle\}$

Continue the current path to (x, y) with a straight line.

This command is protocolled, see Section 98.

```
\mathbf{y_1} = \mathbf{x_2} \{\langle x_1 \rangle \} \{\langle y_1 \rangle \} \{\langle x_2 \rangle \} \{\langle x_2 \rangle \} \{\langle x_3 \rangle \} \{\langle y_3 \rangle \}
```

Continue the current path to (x_3, y_3) with a Bézier curve that has the two control points (x_1, y_1) and (x_2, y_2) .

Example: Draw a good approximation of a quarter circle:

```
\pgfsys@moveto{10pt}{0pt}
\pgfsys@curveto{10pt}{5.55pt}{5.55pt}{10pt}{0pt}{10pt}
\pgfsys@stroke
```

This command is protocolled, see Section 98.

```
\prootemark \pgfsys@rect{\langle x\rangle}{\langle y\rangle}{\langle width\rangle}{\langle height\rangle}
```

Append a rectangle to the current path whose lower left corner is at (x, y) and whose width and height in big points are given by $\langle width \rangle$ and $\langle height \rangle$.

This command can be "mapped back" to \pgfsys@moveto and \pgfsys@lineto commands, but it is included since PDF has a special, quick version of this command.

This command is protocolled, see Section 98.

\pgfsys@closepath

Close the current path. This results in joining the current point of the path with the point specified by the last \pgfsys@moveto operation. Typically, this is preferable over using \pgfsys@lineto to the last point specified by a \pgfsys@moveto, since the line starting at this point and the line ending at this point will be smoothly joined by \pgfsys@closepath.

Example: Consider

```
\pgfsys@moveto{0pt}{0pt}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@closepath
\pgfsys@stroke
```

and

```
\pgfsys@moveto{0bp}{0bp}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@lineto{0bp}{0bp}
\pgfsys@stroke
```

The difference between the above will be that in the second triangle the corner at the origin will be wrong; it will just be the overlay of two lines going in different directions, not a sharp pointed corner.

96.3 Canvas Transformation System Commands

$\pgfsys@transformcm{\langle a \rangle}{\langle b \rangle}{\langle c \rangle}{\langle d \rangle}{\langle d \rangle}{\langle e \rangle}{\langle f \rangle}$

Perform a concatenation of the canvas transformation matrix with the matrix given by the values $\langle a \rangle$ to $\langle f \rangle$, see the PDF or PostScript manual for details. The values $\langle a \rangle$ to $\langle d \rangle$ are dimensionless factors, $\langle e \rangle$ and $\langle f \rangle$ are TEX dimensions

 $\label{lem:example:pgfsys@transformcm{1}{0}{0}{1}{1cm}{1cm}.$

This command is protocolled, see Section 98.

$\protect\operatorname{\footnotemath{\it pgfsys@transformshift{\langle x\ displacement\rangle}}{\langle y\ displacement\rangle}}$

This command will change the origin of the canvas to (x, y).

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 98.

$\protect\operatorname{\form} \protect\operatorname{\form} \pro$

This command will scale the canvas (and everything that is drawn) by a factor of $\langle x \, scale \rangle$ in the x-direction and $\langle y \, scale \rangle$ in the y-direction. Note that this applies to everything, including lines. So a scaled line will have a different width and may even have a different width when going along the x-axis and when going along the y-axis, if the scaling is different in these directions. Usually, you do not want this.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 98.

96.4 Stroking, Filling, and Clipping System Commands

\pgfsys@stroke

Stroke the current path (as if it were drawn with a pen). A number of graphic state parameters influence this, which can be set using appropriate system commands described later.

Line width The "thickness" of the line. A width of 0 is the thinnest width renderable on the device. On a high-resolution printer this may become invisible and should be avoided. A good choice is 0.4pt, which is the default.

Stroke color This special color is used for stroking. If it is not set, the current color is used.

Cap The cap describes how the endings of lines are drawn. A round cap adds a little half circle to these endings. A butt cap ends the lines exactly at the end (or start) point without anything added. A rectangular cap ends the lines like the butt cap, but the lines protrude over the endpoint by the line thickness. (See also the PDF manual.) If the path has been closed, no cap is drawn.

Join This describes how a bend (a join) in a path is rendered. A round join draws bends using small arcs. A bevel join just draws the two lines and then fills the join minimally so that it becomes convex. A miter join extends the lines so that they form a single sharp corner, but only up to a certain miter limit. (See the PDF manual once more.)

Dash The line may be dashed according to a dashing pattern.

Clipping area If a clipping area is established, only those parts of the path that are inside the clipping area will be drawn.

In addition to stroking a path, the path may also be used for clipping after it has been stroked. This will happen if the \pgfsys@clipnext is used prior to this command, see there for details.

This command is protocolled, see Section 98.

\pgfsys@closestroke

This command should have the same effect as first closing the path and then stroking it.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@fill

This command fills the area surrounded by the current path. If the path has not yet been closed, it is closed prior to filling. The path itself is not stroked. For self-intersecting paths or paths consisting of multiple parts, the nonzero winding number rule is used to determine whether a point is inside or outside the path, except if \ifpgfsys@eorule holds – in which case the even-odd rule should be used. (See the PDF or PostScript manual for details.)

The following graphic state parameters influence the filling:

Interior rule If \ifpgfsys@eorule is set, the even-odd rule is used, otherwise the non-zero winding number rule.

Fill color If the fill color is not especially set, the current color is used.

Clipping area If a clipping area is established, only those parts of the filling area that are inside the clipping area will be drawn.

In addition to filling the path, the path will also be used for clipping if \pgfsys@clipnext is used prior to this command.

This command is protocolled, see Section 98.

\pgfsys@fillstroke

First, the path is filled, then the path is stroked. If the fill and stroke colors are the same (or if they are not specified and the current color is used), this yields almost the same as a \pgfsys@fill. However, due to the line thickness of the stroked path, the fill-stroked area will be slightly larger.

In addition to stroking and filling the path, the path will also be used for clipping if \pgfsys@clipnext is used prior to this command.

This command is protocolled, see Section 98.

\pgfsys@discardpath

Normally, this command should "throw away" the current path. However, after \pgfsys@clipnext has been called, the current path should subsequently be used for clipping. See \pgfsys@clipnext for details.

This command is protocolled, see Section 98.

\pgfsys@clipnext

This command should be issued after a path has been constructed, but before it has been stroked and/or filled or discarded. When the command is used, the next stroking/filling/discarding command will first be executed normally. Then, afterwards, the just-used path will be used for subsequent clipping. If there has already been a clipping region, this region is intersected with the new clipping path (the clipping cannot get bigger). The nonzero winding number rule is used to determine whether a point is inside or outside the clipping area or the even-odd rule, depending on whether \ippgfsys@eorule holds.

96.5 Graphic State Option System Commands

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Sets the width of lines, when stroked, to $\langle width \rangle$, which must be a TFX dimension.

This command is protocolled, see Section 98.

\pgfsys@buttcap

Sets the cap to a butt cap. See \pgfsys@stroke.

This command is protocolled, see Section 98.

\pgfsys@roundcap

Sets the cap to a round cap. See \pgfsys@stroke.

This command is protocolled, see Section 98.

\pgfsys@rectcap

Sets the cap to a rectangular cap. See \pgfsys@stroke.

\pgfsys@miterjoin

Sets the join to a miter join. See \pgfsys@stroke.

This command is protocolled, see Section 98.

$\verb|\pgfsys@setmiterlimit{| \langle factor \rangle|}$

Sets the miter limit of lines to $\langle factor \rangle$. See the PDF or PostScript for details on what the miter limit is. This command is protocolled, see Section 98.

\pgfsys@roundjoin

Sets the join to a round join. See \pgfsys@stroke.

This command is protocolled, see Section 98.

\pgfsys@beveljoin

Sets the join to a bevel join. See \pgfsys@stroke.

This command is protocolled, see Section 98.

$\protect\pro$

Sets the dashing patter. $\langle pattern \rangle$ should be a list of TeX dimensions separated by commas. $\langle phase \rangle$ should be a single dimension.

Example: \pgfsys@setdash{3pt,3pt}{0pt}

The list of values in $\langle pattern \rangle$ is used to determine the lengths of the "on" and "off" phases of the dashing. For example, if $\langle pattern \rangle$ is 3bp,4bp, then the dashing pattern is "3bp on followed by 4bp off, followed by 3bp on, followed by 4bp off, and so on." A pattern of .5pt,4pt,3pt,1.5pt means ".5pt on, 4pt off, 3pt on, 1.5pt off, .5pt on, ..." If the number of entries is odd, the last one is used twice, so 3pt means "3pt on, 3pt off, 3pt on, 3pt off, ..." An empty list means "always on."

The second argument determines the "phase" of the pattern. For example, for a pattern of 3bp,4bp and a phase of 1bp, the pattern would start: "2bp on, 4bp off, 3bp on, 4bp off, 3bp on, 4bp off, ..."

This command is protocolled, see Section 98.

\ifpgfsys@eorule

Determines whether the even odd rule is used for filling and clipping or not.

96.6 Color System Commands

The PGF system layer provides a number of system commands for setting colors. These command coexist with commands from the color and xcolor package, which perform similar functions. However, the color package does not support having two different colors for stroking and filling, which is a useful feature that is supported by PGF. For this reason, the PGF system layer offers commands for setting these colors separately. Also, plain TEX profits from the fact that PGF can set colors.

For PDF, implementing these color commands is easy since PDF supports different stroking and filling colors directly. For PostScript, a more complicated approach is needed in which the colors need to be stored in special PostScript variables that are set whenever a stroking or a filling operation is done.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Sets the color used for stroking and filling operations to the given red/green/blue tuple (numbers between 0 and 1).

This command is protocolled, see Section 98.

$\verb|\pgfsys@color@rgb@stroke|| \langle red \rangle| \{\langle green \rangle\} \{\langle blue \rangle\}|$

Sets the color used for stroking operations to the given red/green/blue tuple (numbers between 0 and 1).

Example: Make stroked text dark red: \pgfsys@color@rgb@stroke{0.5}{0}{0}

The special stroking color is only used if the stroking color has been set since the last \color or \pgfsys@color@xxx command. Thus, each \color command will reset both the stroking and filling colors by calling \pgfsys@color@reset.

$\verb|\pgfsys@color@rgb@fill{<|red>|}{<|green>|}{<|blue>|}$

Sets the color used for filling operations to the given red/green/blue tuple (numbers between 0 and 1). This color may be different from the stroking color.

This command is protocolled, see Section 98.

$\verb|\pgfsys@color@cmyk{| \langle cyan \rangle} {\langle magenta \rangle} {\langle yellow \rangle} {\langle black \rangle}$

Sets the color used for stroking and filling operations to the given cmyk tuple (numbers between 0 and 1).

This command is protocolled, see Section 98.

$\verb|\pgfsys@color@cmyk@stroke|| \langle cyan \rangle| \{\langle magenta \rangle\} \{\langle yellow \rangle\} \{\langle black \rangle\}|$

Sets the color used for stroking operations to the given cmyk tuple (numbers between 0 and 1).

This command is protocolled, see Section 98.

$\proonup \proonup \$

Sets the color used for filling operations to the given cmyk tuple (numbers between 0 and 1).

This command is protocolled, see Section 98.

Sets the color used for stroking and filling operations to the given cmy tuple (numbers between 0 and 1).

This command is protocolled, see Section 98.

$\protect\operatorname{\colorectorectory} \protect\operatorname{\colorectorectorectory} \qroke \{\langle cyan \rangle\} \{\langle magenta \rangle\} \{\langle yellow \rangle\}$

Sets the color used for stroking operations to the given cmy tuple (numbers between 0 and 1).

This command is protocolled, see Section 98.

Sets the color used for filling operations to the given cmy tuple (numbers between 0 and 1).

This command is protocolled, see Section 98.

$\protect\pro$

Sets the color used for stroking and filling operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 98.

$\verb|\pgfsys@color@gray@stroke|| \langle black \rangle| |$

Sets the color used for stroking operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 98.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Sets the color used for filling operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 98.

\pgfsys@color@reset

This command will be called when the \color command is used. It should purge any internal settings of stroking and filling color. After this call, till the next use of a command like \pgfsys@color@rgb@fill, the current color installed by the \color command should be used.

If the TEX-if \pgfsys@color@reset@inorder is set to true, this command may "assume" that any call to a color command that sets the fill or stroke color came "before" the call to this command and may try to optimize the output accordingly.

An example of an incorrect "out of order" call would be using \pgfsys@color@reset at the beginning of a box that is constructed using \setbox. Then, when the box is constructed, no special fill or stroke color might be in force. However, when the box is later on inserted at some point, a special fill color might already have been set. In this case, this command is not guaranteed to reset the color correctly.

\pgfsys@color@reset@inordertrue

Sets the optimized "in order" version of the color resetting. This is the default.

\pgfsys@color@reset@inorderfalse

Switches off the optimized color resetting.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protec$

This slightly obscure command causes the color stack to be tricked. When called, this command should set the current color to $\langle E^{\dagger}T_{F}X \; color \rangle$ without causing any change in the color stack.

Example: \pgfsys@color@unstacked{red}

96.7 Pattern System Commands

```
\label{localize} $$ \operatorname{\code}(x_1) = (x_1) = (x_2) =
```

This command declares a new colored or uncolored pattern, depending on whether $\langle flag \rangle$ is 0, which means uncolored, or 1, which means colored. Uncolored patterns have no inherent color, the color is provided when they are set. Colored patters have an inherent color.

The $\langle name \rangle$ is a name for later use when the pattern is to be shown. The pairs (x_1, y_1) and (x_2, y_2) must describe a bounding box of the pattern $\langle code \rangle$.

The tiling step of the pattern is given by $\langle x \ step \rangle$ and $\langle y \ step \rangle$.

Example:

```
\pgfsys@declarepattern{hori}{-.5pt}{0pt}{3pt}{3pt}{3pt}{3pt}{0pt}\pgfsys@moveto{0pt}{0pt}\pgfsys@lineto{0pt}{3pt}\pgfsys@stroke}
{0}
```

```
\verb|\pgfsys@setpatternuncolored|{\langle name\rangle}|{\langle red\rangle}|{\langle green\rangle}|{\langle blue\rangle}|
```

Sets the fill color to the pattern named $\langle name \rangle$. This pattern must previously have been declared with $\langle flag \rangle$ set to 0. The color of the pattern is given in the parameters $\langle red \rangle$, $\langle green \rangle$, and $\langle blue \rangle$ in the usual way.

The fill color "pattern" will persist till the next color command that modifies the fill color.

$\protect\operatorname{\footnotemath{\it pgfsys0setpatterncolored}} \protect\operatorname{\footnotemath{\it colored}} \protect\operatorname{\$

Sets the fill color to the pattern named $\langle name \rangle$. This pattern must have been declared with the 1 flag.

96.8 Scoping System Commands

The scoping commands are used to keep changes of the graphics state local.

\pgfsys@beginscope

Saves the current graphic state on a graphic state stack. All changes to the graphic state parameters mentioned for \pgfsys@stroke and \pgfsys@fill will be local to the current graphic state and the old values will be restored after \pgfsys@endscope is used.

Warning: PDF and PostScript differ with respect to the question of whether the current path is part of the graphic state or not. For this reason, you should never use this command unless the path is currently empty. For example, it might be a good idea to use \pgfsys@discardpath prior to calling this command.

This command is protocolled, see Section 98.

\pgfsys@endscope

Restores the last saved graphic state.

96.9 Image System Commands

The system layer provides some commands for image inclusion.

\pgfsys@imagesuffixlist

This macro should expand to a list of suffixes, separated by ':', that will be tried when searching for an image.

Example: \def\pgfsys@imagesuffixlist{eps:epsi:ps}

\pgfsys@defineimage

Called, when an image should be defined.

This command does not take any parameters. Instead, certain macros will be preinstalled with appropriate values when this command is invoked. These are:

- \pgf@filename File name of the image to be defined.
- \pgf@imagewidth Will be set to the desired (scaled) width of the image.
- \pgf@imageheight Will be set to the desired (scaled) height of the image.

If this macro and also the height macro are empty, the image should have its "natural" size.

If only one of them is specified, the undefined value the image is scaled so that the aspect ratio is kept.

If both are set, the image is scaled in both directions independently, possibly changing the aspect ratio.

The following macros presumable mostly make sense for drivers that can handle PDF:

- \pgf@imagepage The desired page number to be extracted from a multi-page "image."
- \pgf@imagemask If set, it will be set to /SMask x 0 R where x is the PDF object number of a soft mask to be applied to the image.
- \pgf@imageinterpolate If set, it will be set to /Interpolate true or /Interpolate false, indicating whether the image should be interpolated in PDF.

The command should now set up the macro \pgf@image such that calling this macro will result in typesetting the image. Thus, \pgf@image is the "return value" of the command.

This command has a default implementation and need not be implemented by a driver file.

96.10 Shading System Commands

$\protect\operatorname{\colored} \protect\operatorname{\colored} \pro$

Declares a horizontal shading for later use. The effect of this command should be the definition of a macro called $\protect\$

 $\langle name \rangle$ is the name of the shading, which is also used in the output macro name. $\langle height \rangle$ is the height of the shading and must be given as a TeX dimension like 2cm or 10pt. $\langle specification \rangle$ is a shading color specification as specified in Section 91. The shading specification implicitly fixes the width of the shading.

When $\protect\ abox$ of height $\langle height \rangle$ and the width implicit in the shading declaration.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Like the horizontal version, only for vertical shadings. This time, the height of the shading is implicit in $\langle specification \rangle$ and the width is given as $\langle width \rangle$.

$\protect{\protection} \protection \protection \protection} \protection \prot$

Declares a radial shading. Like the previous macros, this command should set up the macro $\protect\operatorname{\texttt{Qpgfshading}}\langle name \rangle!$, which upon invocation should insert a radial shading whose size is implicit in $\langle specification \rangle$.

The parameter $\langle starting \ point \rangle$ is a PGF point specifying the inner starting point of the shading.

$\protect\operatorname{\footnotemark} \ \operatorname{\footnotemark} \ \operatorname{\$

Declares a shading using a PostScript-like function that provides a color for each point. Like the previous macros, this command should set up the macro \P shading name! so that it will produce a box containing the desired shading.

Parameter $\langle name \rangle$ is the name of the shading. Parameter $\langle type \ 4 \ function \rangle$ is a Postscript-like function (type 4 function of the PDF specification) as described in Section 3.9.4 of the PDF specification version 1.7. Parameters $\langle lower \ left \ corner \rangle$ and $\langle upper \ right \ corner \rangle$ are PGF points that specifies the lower left and upper right corners of the shading, respectively.

When $\langle type\ 4\ function \rangle$ is evaluated, the coordinate of the current point will be on the (virtual) PostScript stack in bp units. After the function has been evaluated, the stack should consist of three numbers (not integers! – the Apple PDF renderer is broken in this regard, so add cvrs at the end if needed) that represent the red, green, and blue components of the color.

A buggy function will result is totally unpredictable chaos during rendering.

96.11 Transparency System Commands

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Sets the opacity of stroking operations.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

Sets the opacity of filling operations.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabu$

This takes a TeX box and converts it into a transparency group. This means that any transparency settings apply to the box as a whole. For instance, if a box contains two overlapping black circles and you draw the box and, thus, the two circles normally with 50% transparency, then the overlap will be darker than the rest. By comparison, if the circles are part of a transparency group, the overlap will get the same color as the rest.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Declares the fading $\langle name \rangle$. The $\langle box \rangle$ is a TEX-box. Its content's luminosity determines the opacity of the resulting fading. This means that the lighter a pixel inside the box, the more opaque the fading will be at this position.

$\pgfsys@usefading \langle name \rangle \{\langle a \rangle\} \{\langle b \rangle\} \{\langle c \rangle\} \{\langle d \rangle\} \{\langle e \rangle\} \{\langle f \rangle\} \}$

Installs a previously declared fading $\langle name \rangle$ in the current graphics state. Afterwards, all drawings will be masked by the fading. The fading should be centered on the origin and have its original size, except that the parameters $\langle a \rangle$ to $\langle f \rangle$ specify a transformation matrix that should be applied additionally to the fading before it is installed. The transformation should not apply to the following graphics, however.

\pgfsys@definemask

This command declares a fading (known as a soft mask in this context) based on an image and for usage with images. It works similar to \pgfsys@defineimage: Certain macros are set when the command is called. The result should be to set the macro \pgf@mask to a pdf object count that can subsequently be used as a transparency mask. The following macros will be set when this command is invoked:

- \pgf@filename File name of the mask to be defined.
- \pgf@maskmatte The so-called matte of the mask (see the PDF documentation for details). The matte is a color specification consisting of 1, 3 or 4 numbers between 0 and 1. The number of numbers depends on the number of color channels in the image (not in the mask!). It will be assumed that the image has been preblended with this color.

96.12 Reusable Objects System Commands

$\protect\operatorname{pgfsys@invoke}\{\langle literals \rangle\}$

This command gets protocolled literals and should insert them into the .pdf or .dvi file using an appropriate \special.

$\protect{\langle name \rangle} {\langle lower\ left \rangle} {\langle upper\ right \rangle} {\langle code \rangle}$

Declares an object for later use. The idea is that the object can be precached in some way and then be rendered more quickly when used several times. For example, an arrow head might be defined and prerendered in this way.

The parameter $\langle name \rangle$ is the name for later use. $\langle lower \ left \rangle$ and $\langle upper \ right \rangle$ are PGF points specifying a bounding box for the object. $\langle code \rangle$ is the code for the object. The code should not be too fancy.

This command has a default implementation and need not be implemented by a driver file.

$\protect{\langle name \rangle} {\langle extra\ code \rangle}$

Renders a previously declared object. The first parameter is the name of the object. The second parameter is extra code that should be executed right *before* the object is rendered. Typically, this will be some transformation code.

This command has a default implementation and need not be implemented by a driver file.

96.13 Invisibility System Commands

All drawing or stroking or text rendering between calls of the following commands should be suppressed. A similar effect can be achieved by clipping against an empty region, but the following commands do not open a graphics scope and can be opened and closed "orthogonally" to other scopes.

\pgfsys@begininvisible

Between this command and the closing \pgfsys@endinvisible all output should be suppressed. Nothing should be drawn at all, which includes all paths, images and shadings. However, no groups (neither TEX groups nor graphic state groups) should be opened by this command.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 98.

\pgfsys@endinvisible

Ends the invisibility section, unless invisibility blocks have been nested. In this case, only the "last" one restores visibility.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 98.

96.14 Position Tracking Commands

The following commands are used to determine the position of text on a page. This is a rather complicated process in general since at the moment when the text is read by TEX, the final position cannot be determined, yet. For example, the text might be put in a box which is later put in the headline or perhaps in the footline or perhaps even on a different page.

For these reasons, position tracking is typically a two-stage process. In a first stage you indicate that a certain position is of interest by *marking* it. This will (depending on the details of the backend driver) cause page coordinates or this position to be written to an .aux file when the page is shipped. Possibly, the position might also be determined at an even later stage. Then, on a second run of TEX, the position is read from the .aux file and can be used.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

Marks a position on the page. This command should be given while normal typesetting is done such as in

The value of \$x\$ is \pgfsys@markposition{here}important.

It causes the position here to be saved when the page is shipped out.

$\verb|\pgfsys@getposition{|} \langle name \rangle \} \{ \langle macro \rangle \}$

This command retrieves a position that has been marked on an earlier run of T_EX on the current file. The $\langle macro \rangle$ must be a macro name such as \mathbb{mymacro}. It will be redefined such that it is

• either just \relax or

• a \pgfpoint... command.

The first case will happen when the position has not been marked at all or when the file is typeset for the first time, when the coordinates are not yet available.

In the second case, executing $\langle macro \rangle$ yields the position on the page that is to be interpreted as follows: A coordinate like $\protect\operatorname{pgfpoint}{2cm}$ means "2cm to the right and 3cm up from the origin of the page." The position of the origin of the page is not guaranteed to be at the lower left corner, it is only guaranteed that all pictures on a page use the same origin.

To determine the lower left corner of a page, you can call $\protect\operatorname{\sc pgfsys\sc position}$ with $\langle name \rangle$ set to the special name pgfpageorigin. By shifting all positions by the amount returned by this call you can position things absolutely on a page.

Example: Referencing a point of the page:

```
The value of $x$ is \pgfsys@markposition{here}important.

Lots of text.

\hbox{\pgfsys@markposition{myorigin}%
\begin{pgfpicture}
% Switch of size protocol
\pgfpathmoveto{\pgfpointorigin}
\pgfusepath{use as bounding box}

\pgfsys@getposition{here}{\hereposition}
\pgfsys@getposition{myorigin}{\thispictureposition}

\pgftransformshift{\pgfpointscale{-1}{\thispictureposition}}

\pgftransformshift{\hereposition}

\pgfpathcircle{\pgfpointorigin}{1cm}
\pgfusepath{draw}
\end{pgfpicture}}
```

96.15 Internal Conversion Commands

The system commands take T_EX dimensions as input, but the dimensions that have to be inserted into PDF and PostScript files need to be dimensionless values that are interpreted as multiples of $\frac{1}{72}$ in. For example, the T_EX dimension 2bp should be inserted as 2 into a PDF file and the T_EX dimension 10pt as 9.9626401. To make this conversion easier, the following command may be useful:

```
\protect\ \pgf@sys@bp{\langle dimension \rangle}
```

Inserts how many multiples of $\frac{1}{72}$ in the $\langle dimension \rangle$ is into the current protocol stream (buffered).

Example: \pgf@sys@bp{\pgf@x} or \pgf@sys@bp{1cm}.

Note that this command is not a system command that can/needs to be overwritten by a driver.

97 The Soft Path Subsystem

This section describes a set of commands for creating *soft paths* as opposed to the commands of the previous section, which created *hard paths*. A soft path is a path that can still be "changed" or "molded." Once you (or the PGF system) is satisfied with a soft path, it is turned into a hard path, which can be inserted into the resulting .pdf or .ps file.

Note that the commands described in this section are "high-level" in the sense that they are not implemented in driver files, but rather directly by the PGF-system layer. For this reason, the commands for creating soft paths do not start with \pgfsys@, but rather with \pgfsyssoftpath@. On the other hand, as a user you will never use these commands directly, they are described as part of the low-level interface.

97.1 Path Creation Process

When the user writes a command like \draw (0bp,0bp) -- (10bp,0bp); quite a lot happens behind the scenes:

1. The frontend command is translated by TikZ into commands of the basic layer. In essence, the command is translated to something like

```
\pgfpathmoveto{\pgfpoint{0bp}{0bp}}
\pgfpathlineto{\pgfpoint{10bp}{0bp}}
\pgfusepath{stroke}
```

2. The \pgfpathxxxx commands do not directly call "hard" commands like \pgfsys@xxxx. Instead, the command \pgfpathmoveto invokes a special command called \pgfsyssoftpath@moveto and \pgfpathlineto invokes \pgfsyssoftpath@lineto.

The \pgfsyssoftpath@xxxx commands, which are described below, construct a soft path. Each time such a command is used, special tokens are added to the end of an internal macro that stores the soft path currently being constructed.

- 3. When the \pgfusepath is encountered, the soft path stored in the internal macro is "invoked." Only now does a special macro iterate over the soft path. For each line-to or move-to operation on this path it calls an appropriate \pgfsys@moveto or \pgfsys@lineto in order to, finally, create the desired hard path, namely, the string of literals in the .pdf or .ps file.
- 4. After the path has been invoked, \pgfsys@stroke is called to insert the literal for stroking the path.

Why such a complicated process? Why not have \pgfpathlineto directly call \pgfsys@lineto and be done with it? There are two reasons:

1. The PDF specification requires that a path is not interrupted by any non-path-construction commands. Thus, the following code will result in a corrupted .pdf:

```
\pgfsys@moveto{0}{0}
\pgfsys@setlinewidth{1}
\pgfsys@lineto{10}{0}
\pgfsys@stroke
```

Such corrupt code is *tolerated* by most viewers, but not always. It is much better to create only (reasonably) legal code.

2. A soft path can still be changed, while a hard path is fixed. For example, one can still change the starting and end points of a soft path or do optimizations on it. Such transformations are not possible on hard paths.

97.2 Starting and Ending a Soft Path

No special action must be taken in order to start the creation of a soft path. Rather, each time a command like \pgfsyssoftpath@lineto is called, a special token is added to the (global) current soft path being constructed.

However, you can access and change the current soft path. In this way, it is possible to store a soft path, to manipulate it, or to invoke it.

$\protect\operatorname{\footnotemath{\it Qgetcurrentpath{\it \{}\langle macro\ name\rangle\it\}}}$

This command will store the current soft path in $\langle macro\ name \rangle$.

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{$

This command will set the current soft path to be the path stored in $\langle macro\ name \rangle$. This macro should store a path that has previously been extracted using the \poline{log} stored and has possibly been modified subsequently.

\pgfsyssoftpath@invokecurrentpath

This command will turn the current soft path in a "hard" path. To do so, it iterates over the soft path and calls an appropriate \pgfsys@xxxx command for each element of the path. Note that the current soft path is not changed by this command. Thus, in order to start a new soft path after the old one has been invoked and is no longer needed, you need to set the current soft path to be empty. This may seem strange, but it is often useful to immediately use the last soft path again.

\pgfsyssoftpath@flushcurrentpath

This command will invoke the current soft path and then set it to be empty.

97.3 Soft Path Creation Commands

$\protect\operatorname{Monoto}\{\langle x\rangle\}\{\langle y\rangle\}$

This command appends a "move-to" segment to the current soft path. The coordinates $\langle x \rangle$ and $\langle y \rangle$ are given as normal TeX dimensions.

Example: One way to draw a line:

```
\pgfsyssoftpath@moveto{0pt}{0pt}
\pgfsyssoftpath@lineto{10pt}{10pt}
\pgfsyssoftpath@flushcurrentpath
\pgfsys@stroke
```

$\protect\operatorname{\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\operatorname{\begin{tabular}{l} \protect\begin{tabular}{l} \protect\begin{ta$

Appends a "line-to" segment to the current soft path.

```
\verb|\pgfsyssoftpath@curveto|{\langle a \rangle} {\langle b \rangle} {\langle c \rangle} {\langle d \rangle} {\langle x \rangle} {\langle x \rangle} {\langle y \rangle}
```

Appends a "curve-to" segment to the current soft path with controls (a, b) and (c, d).

```
\verb|\pgfsyssoftpath@rect{| (lower left x)}{| (lower left y)}{| (width)}{| (height)}|
```

Appends a rectangle segment to the current soft path.

\pgfsyssoftpath@closepath

Appends a "close-path" segment to the current soft path.

97.4 The Soft Path Data Structure

A soft path is stored in a standardized way, which makes it possible to modify it before it becomes "hard." Basically, a soft path is a long sequence of triples. Each triple starts with a *token* that identifies what is going on. This token is followed by two dimensions in braces. For example, the following is a soft path that means "the path starts at (0bp, 0bp) and then continues in a straight line to (10bp, 0bp)."

\pgfsyssoftpath@movetotoken{Obp}{Obp}\pgfsyssoftpath@linetotoken{10bp}{Obp}

A curve-to is hard to express in this way since we need six numbers to express it, not two. For this reasons, a curve-to is expressed using three triples as follows: The command

\pgfsyssoftpath@curveto{1bp}{2bp}{3bp}{4bp}{5bp}{6bp}

results in the following three triples:

```
\pgfsyssoftpath@curvetosupportatoken{1bp}{2bp}
\pgfsyssoftpath@curvetosupportbtoken{3bp}{4bp}
\pgfsyssoftpath@curvetotoken{5bp}{6bp}
```

These three triples must always "remain together." Thus, a lonely **supportbtoken** is forbidden. In details, the following tokens exist:

- \pgfsyssoftpath@movetotoken indicates a move-to operation. The two following numbers indicate the position to which the current point should be moved.
- \pgfsyssoftpath@linetotoken indicates a line-to operation.
- \pgfsyssoftpath@curvetosupportatoken indicates the first control point of a curve-to operation.

 The triple must be followed by a \pgfsyssoftpath@curvetosupportbtoken.
- \pgfsyssoftpath@curvetosupportbtoken indicates the second control point of a curve-to operation.

 The triple must be followed by a \pgfsyssoftpath@curvetotoken.
- \pgfsyssoftpath@curvetotoken indicates the target of a curve-to operation.
- \pgfsyssoftpath@rectcornertoken indicates the corner of a rectangle on the soft path. The triple must be followed by a \pgfsyssoftpath@rectsizetoken.
- \pgfsyssoftpath@rectsizetoken indicates the size of a rectangle on the soft path.
- \pgfsyssoftpath@closepath indicates that the subpath begun with the last move-to operation should be closed. The parameter numbers are currently not important, but if set to anything different from {Opt}{Opt}, they should be set to the coordinate of the original move-to operation to which the path "returns" now.

98 The Protocol Subsystem

This section describes commands for *protocolling* literal text created by PGF. The idea is that some literal text, like the string of commands used to draw an arrow head, will be used over and over again in a picture. It is then much more efficient to compute the necessary literal text just once and to quickly insert it "in a single sweep."

When protocolling is "switched on," there is a "current protocol" to which literal text gets appended. Once all commands that needed to be protocolled have been issued, the protocol can be obtained and stored using \pgfsysprotocol@getcurrentprotocol. At any point, the current protocol can be changed using a corresponding setting command. Finally, \pgfsysprotocol@invokecurrentprotocol is used to insert the protocolled commands into the .pdf or .dvi file.

Only those \pgfsys@ commands can be protocolled that use the command \pgfsysprotocol@literal internally. For example, the definition of \pgfsys@moveto in pgfsys-common-pdf.def is

\def\pgfsys@moveto#1#2{\pgfsysprotocol@literal{#1 #2 m}}

All "normal" system-level commands can be protocolled. However, commands for creating or invoking shadings, images, or whole pictures require special \special's and cannot be protocolled.

$\verb|\pgfsysprotocol@literalbuffered{|\langle literal\ text\rangle|}|$

Adds the *(literal text)* to the current protocol, after it has been "\edefed." This command will always be protocolled.

$\protect\pro$

First calls $\protect{\protect} pgfsysprotocol@literalbuffered on $\langle literal\ text \rangle$.}$ Then, if protocolling is currently switched off, the $\langle literal\ text \rangle$ is passed on to $\protect{\protect} pgfsys@invoke$.

\pgfsysprotocol@bufferedtrue

Turns on protocolling. All subsequent calls of \pgfsysprotocol@literal will append their argument to the current protocol.

\pgfsysprotocol@bufferedfalse

Turns off protocolling. Subsequent calls of \pgfsysprotocol@literal directly insert their argument into the current .pdf or .ps.

Note that if the current protocol is not empty when protocolling is switched off, the next call to \pgfsysprotocol@literal will first flush the current protocol, that is, insert it into the file.

$\verb|\pgfsysprotocol@getcurrentprotocol{|\langle macro\ name \rangle|}|$

Stores the current protocol in $\langle macro\ name \rangle$ for later use.

$\verb|\pgfsysprotocol@setcurrentprotocol{|} {\macro\ name|}|$

Sets the current protocol to $\langle macro\ name \rangle$.

\pgfsysprotocol@invokecurrentprotocol

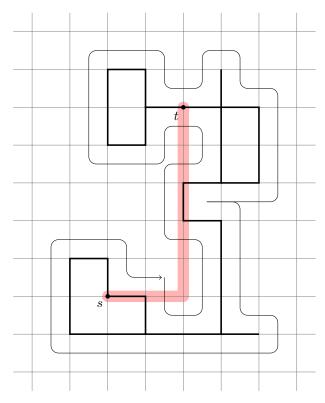
Inserts the text stored in the current protocol into the .pdf or .dvi file. This does not change the current protocol.

\pgfsysprotocol@flushcurrentprotocol

First inserts the current protocol, then sets the current protocol to the empty string.

Part XI

References and Index



```
\begin{tikzpicture}
      \draw[line width=0.3cm,color=red!30,line cap=round,line join=round] (0,0)--(2,0)--(2,5);
      \draw[help lines] (-2.5,-2.5) grid (5.5,7.5);
      \draw[very thick] (1,-1)--(-1,-1)--(-1,1)--(0,1)--(0,0)--
             (1,0)-(1,-1)-(3,-1)-(3,2)-(2,2)-(2,3)-(3,3)--
             (3,5)--(1,5)--(1,4)--(0,4)--(0,6)--(1,6)--(1,5)
             (3,3)--(4,3)--(4,5)--(3,5)--(3,6)
             (3,-1)--(4,-1);
      \draw[below left] (0,0) node(s){$s$};
\draw[below left] (2,5) node(t){$t$};
      \fill (0,0) circle (0.06cm) (2,5) circle (0.06cm);
      \draw[->,rounded corners=0.2cm,shorten >=2pt]
            (1.5,0.5)-- ++(0,-1)-- ++(1,0)-- ++(0,2)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++(0,1)-- ++
            ++(1,0)-- ++(0,1)-- ++(1,0)-- ++(0,-1)-- ++(1,0)-- ++(0,-3)-- ++(-2,0)--
            ++(1,0)-- ++(0,-3)-- ++(1,0)-- ++(0,-1)-- ++(-6,0)-- ++(0,3)-- ++(2,0)--
             ++(0,-1)-- ++(1,0);
\end{tikzpicture}
```

Index

This index only contains automatically generated entries. A good index should also contain carefully selected keywords. This index is not a good index.

```
' key, 177
                                                           butterfly key, 236
                                                           butterfly/from key, 236
                                                           butterfly/level key, 236
       - key, 215
                                                           \verb|butterfly/to| key, 236|
-- kev. 215
                                                           by key, 117
-- path operation, 125
                                                           calc library, 119
-| path operation, 125
                                                           canvas coordinate system, 108
|- path operation, 125
                                                           canvas polar coordinate system, 110
--cycle path operation, 126
                                                           Cartesian placement key, 220
--plot path operation, 259
                                                           cells key, 241
-> key, 215
                                                           chain polar shift key, 225
.. path operation, 125
                                                           chain shift key, 221
<- key, 215
                                                           child path operation, 248
<-> key, 215
                                                           child anchor key, 257
> key, 144
(color class name) key, 213
                                                           circle path operation, 127
(shape name) option, 159
                                                           circular placement key, 225
                                                           class key, 96
above key, 169, 170
                                                           clear < key, 212
                                                           clear > key, 212
above left key, 169, 172
above right key, 170, 173
                                                           \clip, 139
absolute key, 180
                                                           clip key, 154
alias key, 158
                                                           clique key, 233
align kev, 165
                                                           clockwise key, 227
allow upside down key, 178
                                                           color key, 140
                                                           color class key, 213
ampersand replacement key, 244
anchor key, 112, 169, 243
                                                           color option option, 140
angle key, 110, 113
                                                           column (number) key, 241
append after command key, 124
                                                           column sep key, 239
arc path operation, 129
                                                           complete bipartite key, 235
arrows key, 144
                                                           \coordinate, 160
arrows option, 144
                                                           coordinate path operation, 160
as key, 203
                                                           Coordinate systems
at key, 128, 159
                                                                barycentric, 111
at end key, 179
                                                                canvas, 108
at start key, 179
                                                                canvas polar, 110
auto key, 176
                                                                node, 112
                                                                perpendicular, 115
barycentric coordinate system, 111
                                                                tangent, 114
base left key, 174
                                                                xy polar, 111
base right key, 174
                                                                xyz, 109
baseline key, 102
                                                                xyz polar, 110
below key, 169, 172
                                                           cos path operation, 132
below left key, 170, 173
                                                           counterclockwise key, 227
below right key, 170, 173
                                                           css kev, 96
                                                           current point is local key, 119
bend key, 131
bend at end key, 132
                                                           cycle key, 234
bend at start key, 132
bend pos key, 131
                                                           dash pattern key, 142
branch down key, 222
                                                           dash phase kev, 142
branch down sep key, 225
                                                           dashdotdotted kev. 143
branch left key, 222
                                                           dashdotted kev, 143
branch left sep key, 225
                                                           dashed key, 143
branch right key, 222
                                                           Data formats, see Formats
branch right sep key, 224
                                                           declare key, 218
branch up key, 222
                                                           default edge kind key, 215
branch up sep key, 225
                                                           default edge operator key, 218
```

```
delta angle key, 129
                                                           execute at begin cell key, 242
densely dashdotdotted key, 144
                                                           execute at begin picture key, 102
densely dashdotted key, 143
                                                           execute at begin scope key, 104
densely dashed key, 143
                                                           execute at begin to key, 136
densely dotted key, 143
                                                           execute at empty cell key, 242
domain key, 261
                                                           execute at end cell key, 242
                                                           execute at end picture \mathrm{key},\,102
dotted kev, 143
double key, 145
                                                           execute at end scope key, 104
double distance key, 146
                                                           execute at end to key, 136
double distance between line centers key, 146
                                                           File, see Packages and files
double equal sign distance key, 146
                                                           \fill, 139
\draw, 139
                                                           fill key, 146
draw key, 140
                                                           \filldraw, 139
                                                           font key, 164
edge key, 197
                                                           fresh nodes key, 202
edge path operation, 186
edge from parent key, 256
                                                           \graph, 196
edge from parent path operation, 256
                                                           graph path operation, 196
edge from parent macro key, 257
                                                           graph.standard library, 231
edge from parent path key, 256
                                                           Graphic options and styles
edge label key, 134, 197
edge label' key, 134, 197
                                                                \langle shape name \rangle, 159
                                                                arrows, 144
edge node key, 134, 197
edge quotes key, 209
                                                                color option, 140
                                                           Graphs
edge quotes center key, 209
                                                                subgraph C_n, 233
edge quotes mid key, 209
                                                                subgraph Grid_n, 233
edges key, 197
                                                                subgraph I_n, 231
ellipse path operation, 128
                                                                subgraph I_nm, 231
\langle empty \rangle path operation, 124
                                                                \verb|subgraph| K_n, 232|
empty nodes key, 204
                                                                subgraph K_nm, 232
end angle key, 129
                                                                subgraph P_n, 232
Environments
                                                           graphs library, 192
    scope, 104
    \mathtt{tikzpicture},\,101,\,103
                                                           grid path operation, 129
escape key, 95
                                                           grid placement key, 223
even odd rule key, 148
                                                           group polar shift key, 226
every (part name) node part key, 164
                                                           group shift key, 221
every (shape) node key, 160
                                                           grow key, 253
                                                           {\tt grow}\ {\tt down}\ {\rm key},\ 222
every cell key, 241
                                                           grow down sep key, 224
every child key, 251
                                                           grow left key, 222
every child node key, 251
                                                           grow left sep key, 224
every circle key, 128
                                                           grow right key, 222
every edge key, 187
                                                           grow right sep key, 223
every edge quotes key, 188
                                                           grow up key, 221
every even column key, 241
                                                           grow up sep key, 224
every even row key, 242
                                                           grow' key, 254
every graph key, 196
every label key, 181
                                                           growth function key, 255
every label quotes key, 184
                                                           growth parent anchor key, 255
every matrix key, 237
                                                           Handlers for keys, see Key handlers
every new -- key, 200
                                                           help lines key, 131
every new -> key, 200
                                                           horizontal line through key, 115
every new <- key, 201
every new <-> key, 201
                                                           id key, 96
every node key, 159
                                                           induced complete bipartite key, 235
every odd column key, 241
                                                           induced cycle key, 234
every odd row key, 242
                                                           induced independent set key, 234
every path key, 124
                                                           induced path key, 234
every picture key, 103
                                                           inner sep key, 160
every pin key, 182
                                                           inner xsep key, 161
every pin edge key, 182
                                                           inner ysep key, 161
every pin quotes key, 185
                                                           insert path key, 124
every scope key, 104
                                                           intersections library, 116
every to key, 134, 136
```

label key, 180	no placement key, 220
label distance key, 181	Node, see Predefined node
label position key, 180	\node, 160
late options key, 190	node coordinate system, 112
Layout, see Page layout	node key, 114
left key, 169, 172	node path operation, 158
left anchor key, 200	node also path operation, 190
let path operation, 136	node distance key, 172
level key, 227, 251	node halign header key, 167
level (level) key, 227	node quotes mean key, 185
level (number) key, 251	\nodepart, 163
level distance key, 252	nodes key, 197, 241
Libraries	nonzero rule key, 148
calc, 119	not (color class name) key, 213
graph.standard, 231	not (cotor couss name / kcy, 219
graphs, 192	of key, 117
intersections, 116	on grid key, 171
positioning, 170	operator key, 214
quotes, 183	Options for graphics, see Graphic options and styles
scopes, 104	Options for packages, see Package options
line cap key, 142	outer sep key, 161
line join key, 142	outer xsep key, 161
line width key, 141	outer ysep key, 161
loosely dashdotdotted key, 144	overlay key, 189
loosely dashdotted key, 144	overlay key, 100
loosely dashed key, 143	\p, 136
loosely dotted key, 143	Packages and files
100sely dotted key, 149	pgfsys-dvi.def, 96
m key, 232	pgfsys-dvipdfm.def, 94
matching key, 235	pgfsys-dvips.def, 94
matching and star key, 235	pgfsys-pdftex.def, 93
math nodes key, 204	pgfsys-tex4ht.def, 95
\matrix, 237	pgfsys-textures.def, 94
matrix key, 237	pgfsys-vtex.def, 94
matrix anchor key, 243	pgfsys-xetex.def, 94
mid left key, 174	tikz, 101
mid right key, 174	parabola path operation, 131
midway key, 178	parabola height key, 132
minimum height key, 161	parent anchor key, 257
minimum size key, 162	parse key, 199
minimum width key, 161	\path, 123
missing key, 255	path key, 234
miter limit key, 142	Path operations
multi key, 207	, 125
U /	-I, 125
\n, 136	I-, 125
n key, 231	cycle, 126
name key, 112, 117, 158, 202	plot, 259
name intersections key, 116	, 125
name path key, 116	arc, 129
name path global key, 116	child, 248
name separator key, 203	circle, 127
name shore V key, 232	${\tt coordinate},160$
name shore W key, 232	\cos , 132
near end key, 178	edge, 186
near start key, 178	edge from parent, 256
new -	ellipse, 128
new -	$\langle \mathit{empty} \rangle$, 124
- key, 201	graph, 196
\mathtt{new} \mathtt{key} , 200	grid, 129
new -> key, 200	let, 136
new <- key, 201	node, 158
new <-> key, 200	$\verb"node also", 190$
${\tt new}$ set ${\tt key},205$	parabola, 131

${ t plot}, 259$	recolor (color class name) by key, 213
rectangle, 126	rectangle path operation, 126
sin, 132	remember picture key, 189
svg, 133	right key, 169, 172
to, 133	right anchor key, 200
path picture key, 149	rounded corners key, 127
path picture bounding box node, 149	row $\langle number \rangle$ key, 242
\pattern, 139	row (row number) column (column number) key, 242
pattern key, 147	row sep key, 240
pattern color key, 147	•
perpendicular coordinate system, 115	samples key, 261
/pgf/	samples at key, 261
inner sep, 160	scope environment, 104
inner xsep, 161	\scoped , 105
inner ysep, 161	scopes library, 104
minimum height, 161	semithick key, 141
minimum size, 162	set key, 205
${\tt minimum\ width},161$	\shade, 139
outer sep, 161	shade key, 150
outer xsep, 161	\shadedraw, 139
outer ysep, 161	shading key, 150
shape aspect, 162	shading angle key , 151
shape border rotate, 163	shape key, 159
shape border uses incircle, 163	shape aspect key, 162
${\tt trim\ lowlevel},153$	shape border rotate key , 163
\pgfextra, 138	shape border uses incircle key, 163
pgfsys-dvi.def file, 96	sharp corners key, 127
pgfsys-dvipdfm.def file, 94	shorten < key, 145
pgfsys-dvips.def file, 94	shorten > key, 145
pgfsys-pdftex.def file, 93	sibling distance key , 253
pgfsys-tex4ht.def file, 95	simple key, 207
pgfsys-textures.def file, 94	sin path operation, 132
pgfsys-vtex.def file, 94	sloped key, 177
pgfsys-xetex.def file, 94	solid key, 143
phase key, 226	solution key, 114
pin key, 181	sort by key, 117
pin distance key, 182	source edge clear key, 211
pin edge key, 182	source edge node key, 211
pin position key, 182	source edge style key, 211
placement/compute position key, 230	start angle key, 129
placement/depth key, 230	step key, 130
placement/element count key, 228, 230	Styles for graphics, see Graphic options and styles
placement/level key, 227	subgraph C_n graph, 233
placement/logical node depth key, 230	subgraph Grid_n graph, 233
placement/logical node width key, 229	subgraph I_n graph, 231
placement/place key, 231	subgraph I_nm graph, 231
placement/width key, 228	subgraph K_n graph, 232
plot path operation, 259	subgraph K_nm graph, 232
point key, 114	subgraph P_n graph, 232
pos key, 175	svg path operation, 133
positioning library, 170	swap key, 177
postaction key, 156	114
preactions key, 155	tangent coordinate system, 114
Predefined node	target edge clear key, 211
path picture bounding box, 149	target edge node key, 211
prefix after command key, 124	target edge style key, 210
put node text on incoming edges key, 212	text key, 164
put node text on outgoing edges key, 212	text depth key, 168
. 11 100	text height key, 168
quotes library, 183	text width key, 165
quotes mean label key, 183	thick key, 141
quotes mean pin key, 185	thin key, 141
radius key 110 127 226	\tikz, 103

', 177	double equal sign distance, 146
>, 144	draw, 140
above, 169, 170	edge from parent, 256
above left, 169, 172	edge from parent macro, 257
above right, 170, 173	edge from parent path, 256
absolute, 180	edge label, 134
alias, 158	edge label', 134
align, 165	edge node, 134
allow upside down, 178	end angle, 129
ampersand replacement, 244	even odd rule, 148
anchor, 112, 169, 243	every $\langle part \; name \rangle$ node part, 164
append after command, 124	every (shape) node, 160
· ·	
arrows, 144	every cell, 241
at, 128, 159	every child, 251
at end, 179	every child node, 251
at start, 179	every circle, 128
auto, 176	every edge, 187
base left, 174	every edge quotes, 188
base right, 174	every even column, 241
baseline, 102	every even row, 242
below, 169, 172	every label, 181
below left, 170, 173	every label quotes, 184
below right, 170, 173	every matrix, 237
bend, 131	every new, 200
bend at end, 132	every new ->, 200
bend at start, 132	every new <-, 201
bend pos, 131	every new <->, 201
cells, 241	every node, 159
child anchor, 257	every odd column, 241
clip, 154	every odd row, 242
color, 140	every path, 124
column (number), 241	every picture, 103
column sep, 239	every pin, 182
cs/	every pin edge, 182
angle, 110, 113	every pin quotes, 185
horizontal line through, 115	every scope, 104
name, 112	every to, $134, 136$
node, 114	execute at begin cell, 242
point, 114	execute at begin picture, 102
radius, 110	execute at begin scope, 104
solution, 114	execute at begin to, 136
vertical line through, 116	execute at empty cell, 242
x, 109	execute at end cell, 242
x radius, 110, 111	execute at end picture, 102
y, 109	execute at end scope, 104
y radius, 110, 111	execute at end to, 136
z, 109	fill, 146
current point is local, 119	font, 164
dash pattern, 142	graph/
dash phase, 142	level, 227
dashdotdotted, 143	level $\langle level \rangle, 227$
dashdotted, 143	parse, 199
dashed, 143	graphs/
delta angle, 129	 , 215
densely dashdotdotted, 144	-> , 215
densely dashdotted, 143	<-, 215
densely dashed, 143	<->, 215
densely dotted, 143	$\langle {\it color class name} \rangle, 213$
domain, 261	as, 203
dotted, 143	branch down, 222
double, 145	branch down sep, 225
double distance, 146	branch left, 222
double distance between line centers, 146	branch left sep, 225

branch right, 222	new <->, 200
branch right sep, 224	no placement, 220
branch up, 222	nodes, 197
branch up sep, 225	not $\langle color\ class\ name \rangle,213$
butterfly, 236	operator, 214
butterfly/from, 236	path, 234
butterfly/level, 236	phase, 226
butterfly/to, 236	placement/compute position, 230
Cartesian placement, 220	placement/depth, 230
chain polar shift, 225	placement/level 227
chain shift, 221	placement/legical node depth 230
circular placement, 225 clear <, 212	placement/logical node depth, 230 placement/logical node width, 229
clear >, 212	placement/place, 231
clique, 233	placement/place, 231 placement/width, 228
clockwise, 227	put node text on incoming edges, 212
color class, 213	put node text on outgoing edges, 212
complete bipartite, 235	radius, 226
counterclockwise, 227	recolor (color class name) by, 213
cycle, 234	right anchor, 200
declare, 218	simple, 207
default edge kind, 215	source edge clear, 211
default edge operator, 218	source edge node, 211
edge, 197	source edge style, 211
edge label, 197	target edge clear, 211
edge label', 197	target edge node, 211
edge node, 197	target edge style, 210
edge quotes, 209	trie, 206
edge quotes center, 209	typeset, 203
edge quotes mid, 209	use existing node, 202
edges, 197	V, 231
empty nodes, 204	W, 232
every graph, 196	wrap after, 233
fresh nodes, 202	x, 220
grid placement, 223	y, 220
group polar shift, 226	grow, 253
group shift, 221	grow', 254
grow down, 222	growth function, 255
grow down sep, 224	growth parent anchor, 255
grow left, 222	help lines, 131
grow left sep, 224	insert path, 124
grow right, 222	intersection/
grow right sep, 223	by, 117
grow up, 221	name, 117
grow up sep, 224	of, 117
induced complete bipartite, 235	sort by, 117
induced cycle, 234	total, 117
induced independent set, 234	label, 180
induced path, 234	label distance, 181
left anchor, 200	label position, 180
m, 232	late options, 190
matching, 235	left, 169, 172
matching and star, 235	level, 251
math nodes, 204	level $\langle \textit{number} \rangle$, 251
multi, 207	level distance, 252
n, 231	line cap, 142
name, 202	line join, 142
name separator, 203	line width, 141
name shore $V, 232$	loosely dashdotdotted, 144
name shore W , 232	loosely dashdotted, 143
new, 200	loosely dashed, 143
new ->, 200	loosely dotted, 143
new <-, 201	$\mathtt{matrix},237$

matrix ancher 242	oganna 05
matrix anchor, 243	escape, 95
mid left, 174	$\mathtt{id},96$
mid right, 174	text,164
midway, 178	text depth, 168
missing, 255	text height, 168
miter limit, 142	text width, 165
name, 158	thick, 141
	thin, 141
name intersections, 116	
name path, 116	to path, 135
name path global, 116	${\tt transform\ shape},175$
near end, 178	$ exttt{trim left}, 152$
near start, 178	$ exttt{trim right}, 153$
$\mathtt{new}\ \mathtt{set},205$	ultra thick, 141
node distance, 172	ultra thin, 141
node halign header, 167	use as bounding box, 151
node quotes mean, 185	variable, 261
-	
nodes, 241	very near end, 179
nonzero rule, 148	very near start, 179
on grid, 171	very thick, 141
overlay, 189	very thin,141
parabola height, 132	\mathtt{x} radius, 127
parent anchor, 257	xstep, 130
path picture, 149	y radius, 127
pattern, 147	ystep, 130
pattern color, 147	tikz package, 101
pin, 181	\tikzaliascoordinatesystem, 115
pin distance, 182	\tikzdeclarecoordinatesystem, 115
pin edge, 182	$\verb+\tikzgraphforeachcolorednode+, 215$
pin position, 182	$\texttt{\tikzgraphnodefullname}, 204$
pos, 175	\tikzgraphnodename, 203
postaction, 156	\tikzgraphnodepath, 203
preactions, 155	\tikzgraphnodetext, 203
prefix after command, 124	\tikzgraphpreparecolor, 216
-	
quotes mean label, 183	\tikzgraphsset, 196
quotes mean pin, 185	\tikzlastnode, 190
radius, 127	tikzpicture environment, 101, 103
remember picture, 189	$\texttt{\tikzset}, 106$
right, 169, 172	to path operation, 133
rounded corners, 127	to path key, 135
row $\langle number \rangle$, 242	total key, 117
row (row number) column (column number),	transform shape key, 175
242	trie key, 206
row sep, 240	trim left key, 152
samples, 261	trim lowlevel key, 153
samples at, 261	trim right key, 153
semithick, 141	typeset key , 203
set,205	
shade, 150	ultra thick key, 141
shading, 150	ultra thin key, 141
shading angle, 151	use as bounding box key, 151
	use existing node key, 202
shape, 159	
sharp corners, 127	\useasboundingbox, 139
shorten <, 145	\usetikzlibrary, 101
shorten >, 145	
sibling distance, 253	V key, 231
sloped, 177	variable $key, 261$
solid, 143	vertical line through key, 116
start angle, 129	very near end key, 179
	very near start key, 179
step, 130	very thick key, 141
swap, 177	
tex4ht node/	very thin key, 141
class, 96	tt l 929
css, 96	W key, 232
	wrap after kev. 233

\x, 137
x key, 109, 220
x radius key, 110, 111, 127
xstep key, 130
xy polar coordinate system, 111
xyz coordinate system, 109
xyz polar coordinate system, 110

z key, 109