Notes on 8.1-8.2, Boyce and Diprima text.

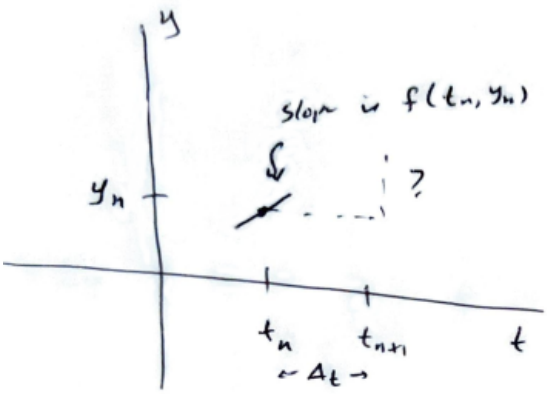
Numerically Solving an IVP

The setup to the problem we consider in Chapter 8 is the following: Given a first order differential equation:

$$\frac{dy}{dt} = f(t, y), \qquad y(t_0) = y_0,$$

we want to provide a numerical approximation to the solution, $\phi(t)$, on some interval of time, where the end of the time interval is known (given) as t_f.

We proceed by subdividing time, so the numerical routine outputs a vector of time values and a vector of solution values at those points in time.
Before considering the global problem, let us consider a single time step from time t_n to t_{n+1}, as shown below.



A First Approximation

The simplest thing we might do is to assume that the slope, $f(t_n, y_n)$, stays constant in the interval from t_n to t_{n+1}. This gives a **tangent line approximation** to the solution, and we find that, by defining $h = \Delta t$ as the time step (presumed constant),

$$y_{n+1} = y_n + hf(t_n, y_n)$$

From here, we can define the algorithm:

Euler's Method, Matlab/Octave Code

```
function [t,y]=euler_forward(f,tinit,yinit,tfin,h)
% INPUT:  Function f(t,y) for y', initial time, initial position, final time
%         and stepsize.
% OUTPUT: Vectors containing the times and approximate solution to y'=f(t,y)
t=tinit:h:tfin;
n=length(t);
y=zeros(1,n);  %Good practice to pre-define the vector.
y(1)=yinit;
```

```
for i=1:n-1
   y(i+1)=y(i)+h*f(t(i),y(i));
end
```

Example Using Euler's Method

In this example, we'll try solving $y' = (t^2 - y^2)\sin(y)$, $y(0) = -1$ on the interval $0 \leq t \leq 1$ using a step size of $h = 0.1$.

First, we need to understand how to pass the function into the algorithm. There are two primary methods we'll use. One is by using an in-line function, the other is to use an $m-$file.

- **Method 1: In-line for short functions**.

  ```
  f=@(t,y) (t.^2-y.^2).*sin(y)
  t0=0; y0=-1; tf=1; h=0.1;
  [t,y]=euler_forward(f,t0,y0,tf,h);
  ```

- **Method 2: M-file for more complicated derivatives**.

  In Matlab, type and save the $m-$file as `myfunc.m` (note: always save a function using its function name!)

  ```
  function dy=myfunc(t,y)
  dy=(t.^2-y.^2).*sin(y)
  ```

  Now in Matlab, we would type:

  ```
  t0=0; y0=-1; tf=1; h=0.1;
  [t,y]=euler_forward(@myfunc,t0,y0,tf,h);
  ```

  Side Remark: It is possible to put functions below a script file in Matlab, but may not be possible in Octave (it wasn't possible in older versions of Matlab).

An alternative: Backwards Euler

In the backwards Euler method, the slope we use for the linear approximation is taken from the end of the interval rather than from the beginning. This yields:

$$y_{n+1} = y_n + h(f(t_{n+1}, y_{n+1})$$

This is an example of an **implicit** method, because we note that y_{n+1} appears on both sides of the equation. The success of this approach depends on whether or not we have a good way of solving this equation for y_{n+1}. Here we'll look at three approaches:

- **Method 1:** We can solve the equation algebraically.

  As an example, supppose that $y' = t + y$. Then backwards Euler becomes:

$$y_{n+1} = y_n + h(t_{n+1} + y_{n+1}) \quad \Rightarrow \quad y_{n+1} = \frac{y_n + ht_{n+1}}{1-h}$$

- **Method 2:** Approximate y_{n+1} using Euler.

  In this case, we'll have:

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \approx y_n + hf(t_{n+1}, y_n + hf(t_n, y_n))$$

- **Method 3:** Use a nonlinear (built-in) solver.

  To use a nonlinear solver, we will typically have to translate our problem to a single function, like $G(z) = 0$, and then the solver will solve for z. In our case, the equation we have is given by:

$$y_{n+1} = y_n + h\, f(t_{n+1}, y_{n+1})$$

  where everything is a known quantity except for y_{n+1}, which we'll call z. Then we write function G as:

$$G(z) = z - y_n - h\, f(t_{n+1}, z)$$

  and "plug it in" to the solver. This takes some effort, and will probably be very slow to compute, especially in comparison to our other methods. The hope is that it produces a much more accurate algorithm.

More Accuracy? Improved Euler (Section 8.2)

Suppose we assume that $\phi(t)$ is a solution to our differential equation, $y' = f(t, y)$. Then we can write:

$$\phi'(t) = f(t, \phi(t))$$

and by integrating from t_n to t_{n+1}, we have:

$$\int_{t_n}^{t_{n+1}} \phi'(t)\, dt = \int_{t_n}^{t_{n+1}} f(t, \phi(t))\, dt$$

So that

$$\phi(t_{n+1}) - \phi(t_n) = \int_{t_n}^{t_{n+1}} f(t, \phi(t))\, dt$$

Or, more precisely

$$\phi(t_{n+1}) = \phi(t_n) + \int_{t_n}^{t_{n+1}} f(t, \phi(t))\, dt$$

Now we see that more accuracy for our algorithm would depend on how accurately we can approximate the integral. If we denote our approximate solution by y, we might approximate the integral using the following:

$$y_{n+1} = y_n + h \cdot \frac{f(t_n, y_n) + f(t_{n+1}, y_{n+1})}{2} \approx y_n + h \cdot \frac{f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n))}{2}$$

This is the **Improved Euler** method. It should be more accurate; the idea is that we're using information about the slope field at t_n as well as t_{n+1} (while Euler's method only uses information about the slope field at the point t_n).

Improved Euler in Matlab/Octave

To see what we'll need to compute, let's abbreviate $f(t_n, y_n)$ as f_n. In fact, we can keep f_n as a vector in our algorithm if we wanted to, although looking at the formula, we'll need two function evaluations at each step.

```
for i=1:n-1
   k1=f(t(n),y(n));
   k2=f(t(n)+h, y(n)+h*k1);  % If we have the time vector, we could use t(n+1)
                             % in place of t(n)+h

   y(n+1)=y(n)+(h/2)*(k1+k2);

   t(n+1)=t(n)+h;            % This line can be deleted if time was computed
                             % already
end
```