Continuing Notes on Ch 8 (Selected topics from 8.3, 8.5, 8.6)

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.

So far, we have Euler's method (Euler's forward method actually), and then we have two modifications of that: Backwards Euler and Improved Euler (8.2).

These are given by the following rules. Recall $y_n = y(t_n)$.

- Forward Euler:

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

- Backward Euler:

  $k_1 = f(t_n, y_n)$

  $y_{n+1} = y_n + h \cdot f(t_{n+1}, y_n + h \cdot k_1)$

- Improved Euler:

  $k_1 = f(t_n, y_n)$

  $k_2 = f(t_{n+1}, y_n + h \cdot k_1)$

  $y_{n+1} = y_n + h \cdot \dfrac{k_1 + k_2}{2}$

Numerical Error

- Round off error:

  Caused by using a discrete number of significant digits to represent real numbers on a computer. Since computers can retain a large number of digits in a computation, round-off error comes into play when we do things that can result in a big loss of significant digits- A classic example of that is when we subtract two nearly equal numbers. Notice that this can happen in DEs if the step size is too small.

  Therefore, even though we said that error decreases with step size, you can take this too far!

  Typically, we don't study the round off error given in an algorithm- rather, we focus on the truncation error.

- Local Truncation Error: The error caused by one step of the algorithm.

- Global Truncation Error: The cumulative error caused by many iterations. The numerical method is **convergent** if global truncation error goes to zero as the step size goes to zero.

Error in Euler's Method

Let $y'(t) = f(t, y)$, $y(t_0) = y_0$ be a given IVP. Define y_1 as the computed output of Euler's method after one time step:

$$y_1 = y(t_0) + hf(t_0, y_0)$$

If we expand the exact solution $y(t)$ using Taylor series based at t_0, then

$$y(t_0+h) = y(t_0) + hy'(t_0) + \frac{h^2}{2}y''(t_0) + O(h^3)$$

The first two terms of that expansion are actually what we called y_1, so we write:

$$y(t_0+h) = y_1 + \frac{h^2}{2}y''(t_0) + O(h^3)$$

So we see that the local truncation error, if the third derivative is bounded, is given by

$$y(t_0+h) - y_1 = \frac{h^2}{2}y''(t_0) + O(h^3)$$

We can be more precise using the Lagrange form for the remainder. That is, if y has a continuous second derivative, there exists $\xi \in [t_0, t_0+h]$ such that

$$y(t_0+h) - y_1 = \frac{1}{2}h^2 y''(\xi).$$

From this, we say that the local truncation error is approximately proportional to h^2.

It can be shown that the global truncation error for Euler's method is proportional to h.

The Runge-Kutta Method

In the Runge-Kutta method, we will be taking sample function evaluations of our derivative, $f(t, y)$ at *four* sample points in the interval $[t_n, t_{n+1}]$ rather than our last modification of Euler's method, which took two samples at the endpoints.

Therefore, given $y' = f(t, y)$ with $y(t_0) = y_0$ and step size h, we'll define

$$k_1 = f(t_n, y_n), \qquad k_2 = f\left(t_n + \frac{1}{2}h, y_n + \frac{1}{2}h \cdot k_1\right)$$

$$k_3 = f\left(t_n + \frac{1}{2}h, y_n + \frac{1}{2}h \cdot k_2\right), \qquad k_4 = f(t_n + h, y_n + h \cdot k_3)$$

Then the step we take is given by:

$$y_{n+1} = y_n + h \cdot \left(\frac{k_1 + 2k_2 + 2k_3 + k_4}{6}\right)$$

It can be shown that the local truncation error is proportional to h^5, and the global truncation error is at most a constant times h^4. We call this method a *fourth order* DE solver.

Getting ready to use the solvers

You might have noticed that all the solvers so far have been designed to solve the first order differential equation:

$$y' = f(t, y), y(t_0) = y_0$$

In fact, we don't have higher order DE solvers, because as it turns out, we can convert all other DEs to this form.

First Order Systems

We'll need to change our notation slightly for a first order **system** of differential equations:

$$\mathbf{y}' = \mathbf{F}(t, y)$$

Then all of our methods work exactly the same as before- The one-dimensional versions we have already defined are employed on each dimension independently.

For example, suppose

$$\begin{array}{rl} y_1' = y_1 + y_2 + t & \qquad y_1(0) = 1 \\ y_2' = y_1 y_2 + \cos(t) & \qquad y_2(0) = 2 \end{array}$$

Then apply Euler's Method with $h = 0.1$.

SOLUTION: We start at the vector $(1, 2)$, and computing the derivatives:

$$\begin{array}{rl} y_1'(0) = y_1(0) + y_2(0) + 0 & = 1 + 2 = 3 \\ y_2'(0) = y_1(0)y_2(0) + \cos(0) & = 2 + 1 = 3 \end{array}$$

Now,

$$\begin{array}{r} y_1(0.1) = 1 + (0.1)3 = 1.3 \\ y_2(0.1) = 2 + (0.1)3 = 2.3 \end{array}$$

And so on. In fact, since we should write the derivative functions as if they are taking in vectors, the Matlab/Octave code will remain unchanged whether we have the scalar equation or a system of equations.

Convert n^{th} order DE to first order system

To solve an n^{th} order differential equation, we have to first convert it to a system of first order. We do that by introducing "dummy" variables. For example, convert the following third order DE to a system of first order:

$$y''' - yy' = 3y'' + \cos(t)$$

SOLUTION: Let $x_1 = y, x_2 = y', x_3 = y''$. Then we want to find expressions for x_1', x_2', x_3' (be sure to go in order).

$$\begin{array}{ll} x_1' & = x_2 \\ x_2' & = x_3 \\ x_3' & = x_1 x_2 + 3x_3 + \cos(t) \end{array}$$

Then remember that we don't need x_2, x_3 for the solution- The original solution y was just x_1.

Homework

The homework will consist of two parts- One is to get all of your code together for the four different algorithms (Forward Euler, Backward Euler, Improved Euler, Runge-Kutta), and then we'll be numerically solving some differential equations and some systems of differential equations.

I will have the code for the three Euler algorithms available for you to use in a "bucket" in Octave-online, so feel free to use that as a starting point.

1. Using the Euler algorithms as a template, write the code for Runge-Kutta (and name the function `runge_kutta.m`).

2. Solve the differential equations in #1, 5, 8 in 8.6 (pg 483), except first use $h = 0.01$, then use $h = 0.005$ to estimate the solution at $t = 1$. Use Euler's method, then Runge-Kutta. Comment on what you see- Can you estimate how accurate your solutions are from these numbers? (I'm not looking for a single answer here- just a short discussion).

3. Read over the help file for the built-in Octave/Matlab function `ode23`, and use it to solve the three differential equations from the previous problem (we're only comparing the solutions at time $t = 1$). Comment on the accuracy. To help you get started, here is a sample piece of code that will solve $y' = 2t$, $y(0) = 1$, on the interval $[0, 2]$. As a side note, we plot the points and you might note that the step size is not constant.

```
f=@(t,y) 2*t;
y0=1; t0=0; tf=2;
[t,y]=ode23(f, [t0, tf], y0);
plot(t,y,'o-')
```

Appendix I: Code

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
```

Code for Backwards Euler

```
function [t,y]=euler_backward(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
   k1=f(t(i),y(i));
   y(i+1)=y(i)+h*f(t(i),y(i)+h*k1);
end
```

Code for Improved Euler

```
function [t,y]=euler_improved(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
   k1=f(t(i),y(i));
   k2=f(t(i+1),y(i)+h*k1);
   y(i+1)=y(i)+(h/2)*(k1+k2);
end
```

Appendix II: Examples

Example 1:

Solve $y' = (t^2 - y^2)\sin(y)$, $y(0) = -1$ on the interval $0 \le t \le 1$ 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).

Example 2:

In this example, we'll take a look at the differences between a numerical approximation to a solution, and the solution itself. We'll also look at a plot of the two functions.

Solve $y' = 2y$ with $y(0) = \frac{1}{2}$, and consider $0 \le t \le 2$ with step size 0.1. Compare the numerical solution using Backwards Euler with the actual solution, $y(t) = \frac{1}{2}e^{2t}$, graphically.

SOLUTION: We'll need two functions this time- We need the function for the derivative and the function for the actual solution. We'll then call the Backwards Euler routine we defined previously and compute the plots.

```
f1=@(t,y) 2*y;
f2=@(t,y) 0.5*exp(2*t);

t0=0; y0=0.5; tf=2; h=0.1;
t=t0:h:tf;
N=length(t);
y=zeros(1,N);
phi=f2(t);

[t,y]=euler_backward(f1,t0,y0,tf,h);

plot(t,y,'ro-',t,phi,'k-');

%Plot the error and find a "line of best fit"
ErrLog1=log(abs(y-phi));
p1=polyfit(t(2:end),ErrLog1(2:end),1);
L1=p1(1)*t+p1(2);
plot(t,ErrLog1,'ro-',t,L1,'k-');
```