Numerical Solution of Ordinary Differential Equations

F. R. Toffoletto

February 1, 2007

1 Finite Differences

To solve differential equations, we need a numerical method to evaluate derivatives. The formal definition of a derivative is then

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{1}$$

where Δx is, for example, the time increment or time step. Note that using an extremely small timestep does not necessarily improve the error, since roundoff errors can become significant. Given that, we need to estimate the difference between $f'(x)$ and its finite difference approximation. To do this we will use Taylor's expansion

$$f(x + \Delta x) \approx f(x) + \Delta x f'(x) + \frac{\Delta x^2}{2} f''(x) + \frac{\Delta x^3}{3!} f'''(x) + \frac{\Delta x^4}{4!} f^{(iv)}(x) + O(\Delta x^5) \tag{2}$$

The approximation to $f'(x)$ can be written as to first order as the forward derivative

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x} + O(\Delta x) \tag{3}$$

However, if you write equation (2) with $-\Delta x$ you get

$$f(x - \Delta x) \approx f(x) - \Delta x f'(x) + \frac{\Delta x^2}{2} f''(x) - \frac{\Delta x^3}{3!} f'''(x) + \frac{\Delta x^4}{4!} f^{(iv)}(x) + O(\Delta x^5) \tag{4}$$

If we subtract equation (3) from equation (2) we get the central difference approximation to the derivative

$$f'(x) \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} + O(\Delta x^2) \tag{5}$$

which is second order. Using the Taylor expansion for $f(x + \Delta x)$ and $f(x - \Delta x)$ we can also get the centered version of the second derivative

$$f''(x) \approx \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{\Delta x^2} + O(\Delta x^2) \tag{6}$$

These results will come in useful later.

2 Euler's Method

Suppose we have a differential equation defined by

$$\frac{dy}{dx} = f(x, y) \tag{7}$$

that we want to solve. Equation 7 gives you the derivative $\frac{dy}{dx}$ everywhere as a function of x,and y and you want to solve for $y(x)$. One way to look at this is to plot an arrow plot for a set of 2-d points, where each arrow is the vector that represents at $\frac{dy}{dx}$ at that point.

Now lets try something a tiny bit more interesting. Newton's law of cooling says that the rate of change of temperature (y) of an object is proportional to the difference between its temperature and some reservoir at temperature c. Mathematically, the above statement can be written as

$$\frac{dy}{dx} = k(c - y(x)) \tag{8}$$

where k is a constant dependent on the heat conduction abilities of the 'wall' that separates the reservoir and the object. For example, 8 can be plotted in such a way in matlab using the following program:

```
function vec(f,x0,x1,n,y0,y1,m)
% function vec(f,x0,x1,n,y0,y1,m)
% plots a vector field for a function
% dy/dx=f(x,y)
% meshgrid sets up, in this case, 2 2-D arrays
% the x-array varies in columns from x0,x1 with n values
% the y-array varies in rows from y0,y1 with m values
[x,y]=meshgrid(x0:(x1-x0)/n:x1,y0:(y1-y0)/m:y1);

dy=feval(f,x,y); %computes f at the points t and y

l=size(dy); % returns the dimension of l
dx=ones(l(1),l(2)); % makes the t-array all 1

% quiver is an arrow plot routine in matlab
quiver(x,y,dx,dy)
xlabel('x');ylabel('y');
grid on
```

An example is shown in Figure 1; at each (x, y) point the program plots a vector showing the slope $\frac{dy}{dx}$. You can picture the solution to equation 7 as computing the streamlines associated with the vectors. Figure 2 overlays figure 1 with the plots of the solution to equation (8). You can see that the solution lines follow the direction of the arrows.

Now getting back to Euler's method, imagine you have an equation of the form of equation 7 and an initial condition at time x_0 of $y(x_0) = y_0$. To get the answer at time x_1, where $x_1 = x_0 + h$,

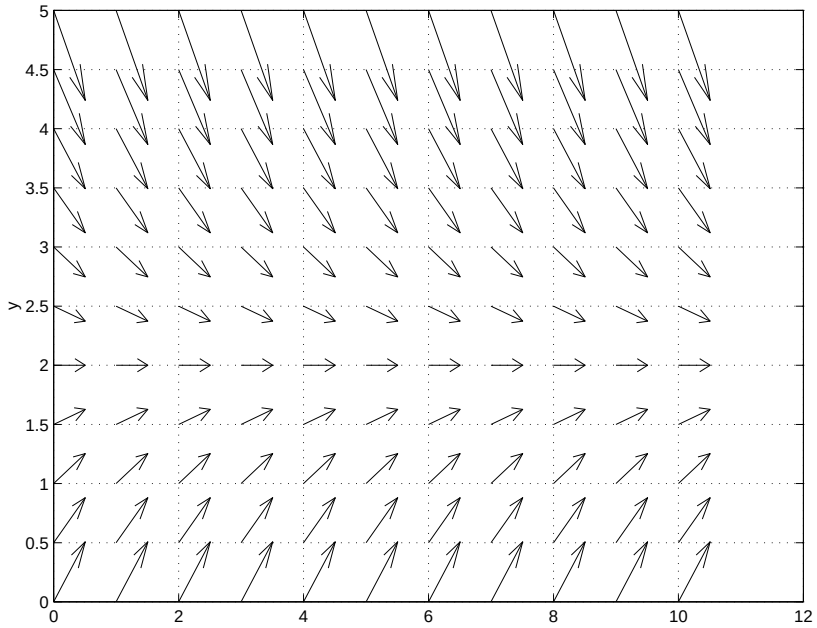

Figure 1:

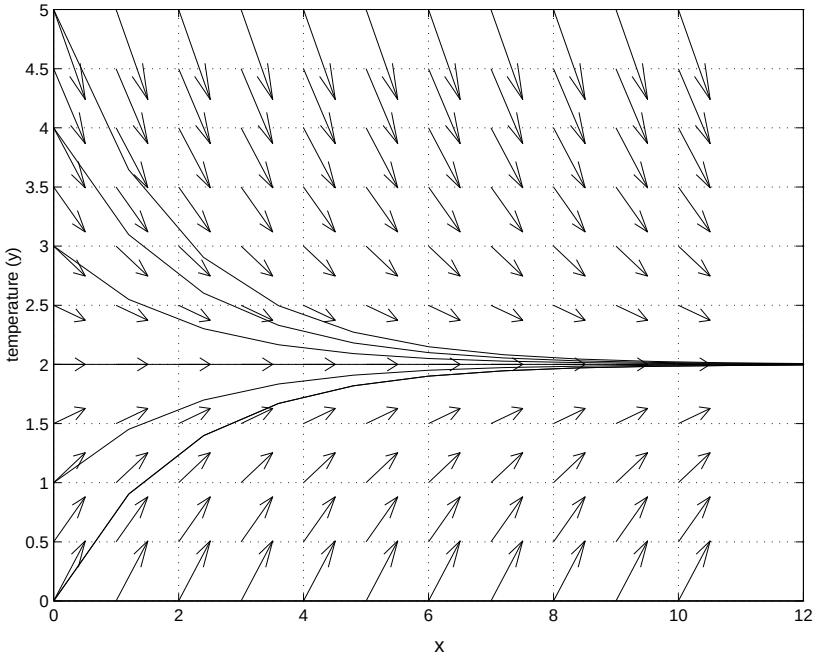

Figure 2:

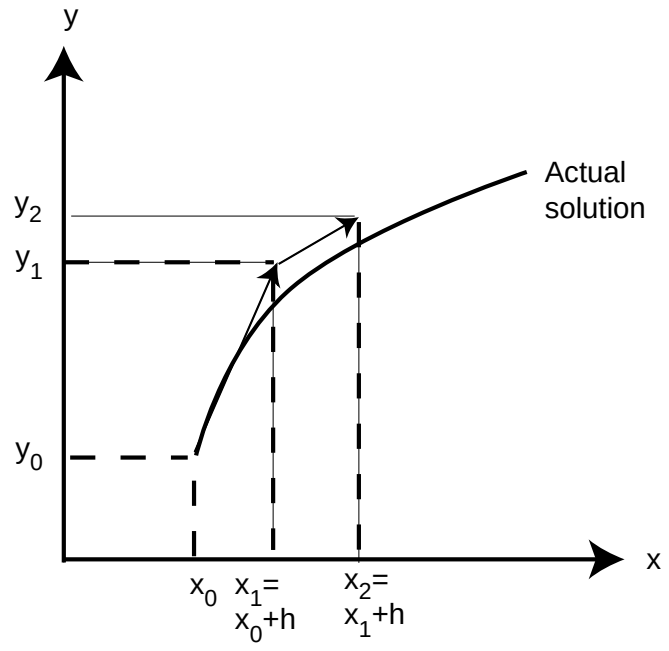

Figure 3: Euler's method

($h = \Delta x$) you would use a Taylor series expansion

$$y_1 = y(x_1) = y(x_0 + h) = y(x_0) + h\frac{dy(x_0, y_0)}{dx} + O(h^2) \tag{9}$$

But from equation 7 $\frac{dy(x_0,y_0)}{dx} = f(x_0, y_0)$, so that

$$y_1 = y(x_1) = y(x_0 + h) = y(x_0) + hf(x_0, y_0) + O(h^2) \tag{10}$$

Euler's method ignores terms of $O(h^2)$ and estimates the value y_1 from equation 10. Geometrically, it can be seen in Figure 3, where y_1 is computed by 10, and in turn where y_2 is computed from

$$y_2 = y(x_2) = y(x_1 + h) = y(x_1) + hf(x_1, y_1) + O(h^2)$$

and so on.

3 Adjustable stepsize Euler method

The thing to note about the Euler method is that it is the simplest method and that you have to take a lot of small steps to get reasonable accuracy. You will note that if the equation $f(x, y) = ax$ then

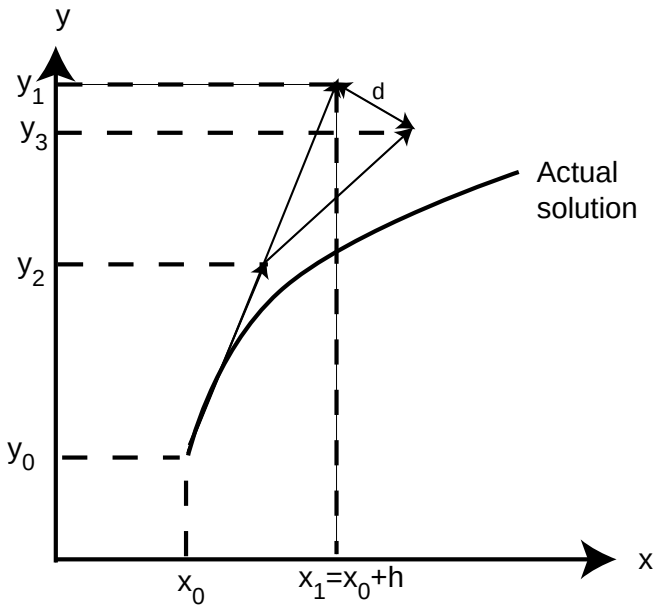

Figure 4: Variable step Euler method

the Euler method would be exact, but the more 'curved' the solution the smaller the steps the Euler method would have to take to get a reasonable answer. One way to improve the method is to have the solver take variable steps: where the solution is linear then take big steps, when the solution has 'bends' and 'kinks', take smaller steps. The basic idea is sketched in Figure 4

The basic procedure is as follows (where *error* is some prespecified error tolerance:

1. Compute an Euler step of length h as

$$y_1 = y(x_0) + hf(x_0, y_0)$$

2. Compute 2 Euler steps of length $h/2$

$$y_2 = y(x_0) + \frac{h}{2} f(x_0, y_0)$$

$$y_3 = y(x_2) + \frac{h}{2} f(x_2, y_2)$$

3. Compute the norm

$$d = |y_1 - y_3|$$

4. If $d > error$ then reduce the step $h \to \frac{h}{2}$ and repeat from (1).

5. If $d < error$ then accuracy is ok, step forward and increase the step $h \to 2h$

4 A note on Local Error, Global Error, and Choosing a stepsize

To judge the accuracy of these methods, we need to distinguish between local truncation error and global truncation error. The error we have discussed so far is local error, that is, the error made from a single timestep. In a typical problem, we want to evaluate a trajectory from $x = 0$ to $x = X$ in steps of Δx. The number of timesteps is $N = X/\Delta x$; if we reduce Δx we must take more steps. If the local error is of order $O\Delta x^2$, then the estimate for the global error is

$$\begin{aligned} globalerror &\propto N \times (localerror) && (11)\\ &= NO(\Delta x^n) = TO(\Delta x^{n-1}) && (12)\end{aligned}$$

So that the Euler method has a local truncation error of $O(\Delta x^2)$ but a global truncation error of $O(\Delta x)$. Of course the analysis is only an estimate since we do not know if the errors accumulate or cancel. The actual global error for a numerical scheme is highly dependent on the problem we are studying.

5 Euler-Cromer and Midpoint Methods

Consider the simple motion of a particle. The equations of motion can be written as

$$\frac{d\vec{r}}{dx} = \vec{v}(x) \quad (13)$$

$$\frac{d\vec{v}}{dx} = \vec{a}(\vec{r}(x), \vec{v}(x)) \quad (14)$$

where $\vec{a}$ is the acceleration. Euler's method gives

$$\vec{r}(t + \Delta x) = \vec{r}(x) + \Delta t\vec{v}(x) + O(\Delta x^2) \quad (15)$$

$$\vec{v}(t + \Delta x) = \vec{v}(x) + \Delta t\vec{a}(\vec{r}(x), \vec{v}(x)) + O(\Delta x^2) \quad (16)$$

Another approach is to slightly rewrite equation (14) as

$$\vec{v}(x + \Delta x) = \vec{v}(x) + \Delta x\vec{a}(\vec{r}(x), \vec{v}(x)), \vec{v}(x)) + O(\Delta x^2) \quad (17)$$

$$\vec{r}(t + \Delta t) = \vec{r}(x) + \Delta t\vec{v}(x + \Delta x) + O(\Delta x^2) \quad (18)$$

Notice here we are using the velocity $\vec{v}$ at the new time. This is the Euler-Cromer method. An alternative approach is to update the position with the following method

$$\vec{r}(x + \Delta x) = \vec{r}(x) + \Delta t\frac{1}{2}(\vec{v}(x + \Delta x) + \vec{v}(x)) + O(\Delta x^2) \quad (19)$$

which is the *mid-point* method.

6 Leap Frog and Verlet Methods

The leap-frog method tries to used time-centered variables

$$\vec{v}(x+\Delta x) = \vec{v}(x-\Delta x) + 2\Delta x \vec{a}(\vec{r}(x), \vec{v}(x)) + O(\Delta x^3) \tag{20}$$

$$\vec{r}(x+2\Delta x) = \vec{r}(x) + 2\Delta x \vec{v}(x+\Delta x) + O(\Delta x^3) \tag{21}$$

so that the variables literally 'leap' around each other. Feynman used this method to calculate the oscillations of a spring and the orbit of a planet.

Another approach is the rewrite the equations of motion as

$$\frac{d^2\vec{r}}{dx^2} = \vec{a}(\vec{r}(x), \vec{v}(x)) \tag{22}$$

$$\frac{d\vec{r}}{dx} = \vec{v}(x) \tag{23}$$

Using central differences for the first and second derivatives we get

$$\frac{\vec{r}(x+\Delta x) - 2\vec{r}(x) + \vec{r}(x-\Delta x)}{\Delta x^2} = \vec{a}(\vec{r}(x), \vec{v}(x)) \tag{24}$$

This is know as the Verlet method.

7 Runge-Kutta Schemes

One of the most popular schemes in use is the Runge-Kutta scheme, named after its inventors. It basically uses the Euler scheme but instead of truncating at second order, it truncates at higher order. The one I will introduce here, but not attempt to derive, is the 4th order Runge-Kutta Scheme (RK4). The basic idea is that if you have a solution at step k defined by y_k, you move to the next step y_{k+1} by the following algorithm

$$y_{k+1} = y_k + w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4 \tag{25}$$

where

$$\begin{aligned} k_1 &= hf(x_k, y_k) \\ k_2 &= hf(x_k + a_1 h, y_k + b_1 k_1) \\ k_3 &= hf(x_k + a_2 h, y_k + b_2 k_1 + b_3 k_2) \\ k_4 &= hf(x_k + a_3 h, y_k + b_4 k_1 + b_5 k_2 + b_6 k_3) \end{aligned}$$

The trick is then to compute the values for a_n, b_n and w_n. One does this by using Taylor series to 4th order and using equation 7. One gets, after some work:

$$\begin{aligned} b_1 &= a_1 \\ b_2 + b_3 &= a_2 \\ b_4 + b_5 + b_6 &= a_3 \end{aligned}$$

$$
\begin{aligned}
w_1 + w_2 + w_3 + w_4 &= 1 \\
w_2 a_1 + w_3 a_2 + w_4 a_3 &= \frac{1}{2} \\
w_2 a_1^2 + w_3 a_2^2 + w_4 a_3^2 &= \frac{1}{3} \\
w_2 a_1^3 + w_3 a_2^3 + w_4 a_3^3 &= \frac{1}{4} \\
w_3 a_1 b_1 + w_4 (a_1 b_5 + a_2 b_6) &= \frac{1}{6} \\
w_3 a_1 a_2 b_3 + w_4 a_3 (a_1 b_5 + a_2 b_6) &= \frac{1}{8} \\
w_3 a_1^2 b_1 + w_4 (a_1^2 b_5 + a_2^2 b_6) &= \frac{1}{12} \\
w_4 a_1 b_3 b_6 &= \frac{1}{8}
\end{aligned}
$$

There are 11 equations and 13 unknowns. So two more conditions are needed to solve the above mess, this is done by convenience, the choice changes the method. One choice is to set

$$
a_1 = \frac{1}{2} \quad b_2 = 0
$$

The solution is then

$$
\begin{aligned}
a_2 = \frac{1}{2} \quad & a_3 = 1 \\
b_1 = \frac{1}{2} \quad & b_3 = \tfrac{1}{2} \\
b_4 = 0 \quad & b_5 = 0 \quad b_6 = 1 \\
w_1 = \frac{1}{6} \quad & w_2 = \tfrac{1}{3} \\
w_3 = \frac{1}{3} \quad & w_4 = \tfrac{1}{6}
\end{aligned}
$$

so that

$$
y_{k+1} = y_k + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4) \tag{26}
$$

which is the RK4 scheme. where

$$
\begin{aligned}
f_1 &= f(x_k, y_k) \\
f_2 &= f(x_k + \frac{h}{2}, y_k + \frac{h}{2} f_1) \\
f_3 &= f(x_k + \frac{h}{2}, y_k + \frac{h}{2} f_2) \\
f_4 &= f(x_k + h, y_k + h f_3)
\end{aligned}
$$

Other choices of a_1, b_2 lead to other similar schemes.

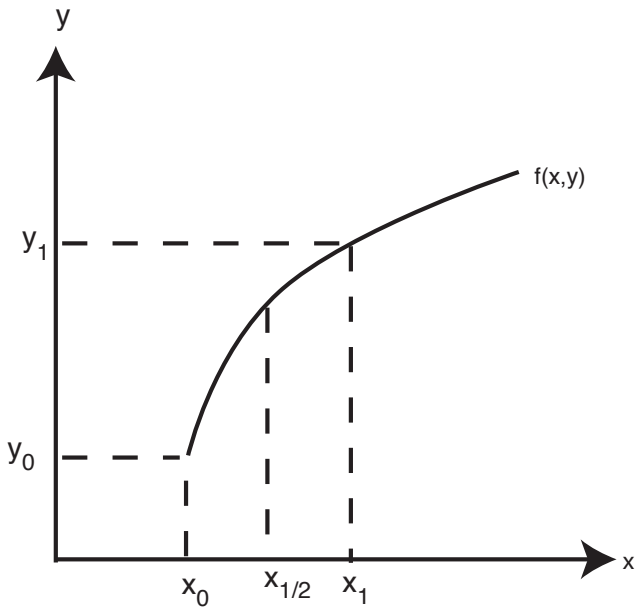

Figure 5: Simpson's rule

7.1 Relationship of Runge-Kutta to Simpson's rule

As you may recall simpson's rule is a numerical integration scheme and can be used to get an idea as to where 26 comes from. Figure 5 shows the setup for computing Simpson's rule. Basically, you fit a parabola through the 3 points and then integrate the parabola. If you do this you end up with

$$\int_{x_0}^{x_1} f(x, y(x))dt \approx \frac{h}{6}(f(x_0, y(x_0)) + 4f(x_{1/2}, y(x_{1/2})) + f(x_1, y(x_1))$$

where $x_{1/2}$ is the midpoint of the interval. This is Simpson's rule. Since

$$y(x_1) - y(x_0) = \int_{x_0}^{x_1} f(x, y(x))dx$$

and if we assume that

$$f(x_{1/2}, y(x_{1/2})) \approx \frac{f_2 + f_3}{2}$$

then we get equation 26.

8 Adaptive Runge-Kutta Schemes

An obvious next improvement for the RK4 scheme is to keep the accuracy of the 4th order scheme with the intelligence of the adaptive stepsize. One such scheme is the RKF45 scheme. RKF45 uses

2 different approximations for the solution, one 4th order and the other 5th (!) order. The answers are compared just like the adaptive scheme. If the solution agrees to some specified error, then the solution proceeds, if not the stepsize is reduced and the iteration is repeated. The 6 values computed (don't try this at home) are then

$$\begin{aligned}
k_1 &= hf(x_k, y_k) \\
k_2 &= hf(x_k + \frac{1}{4}h, y_k + \frac{1}{4}k_1) \\
k_3 &= hf(x_k + \frac{3}{8}h, y_k + \frac{3}{32}k_1 + \frac{9}{32}k_2) \\
k_4 &= hf(x_k + \frac{12}{13}h, y_k + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3) \\
k_5 &= hf(t_k + h, y_k + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4) \\
k_6 &= hf(x_k + \frac{1}{2}h, y_k - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2656}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5)
\end{aligned}$$

The 4th order step is then

$$y_{k+1} = y_k + \frac{25}{216}k_1 + \frac{1408}{2565}f_3 + \frac{2197}{4101}f_4 + \frac{1}{5}f_5 \tag{27}$$

and the 5th order step (notice that 5 values of the function are used, instead of 4 for the 4th order)

$$z_{k+1} = y_k + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 \tag{28}$$

The optimal step size is then estimated by comparing the values of y_{k+1} and z_{k+1}. Notice that the value k_2 is not used. A MATLAB version of this method is available on request.

An alternative approach to doing adaptive step sizing is the one used in the book by Garcia. This implementation looks at the differences in the results when you take 2 small steps and you end up at location $\vec{x}_{small}$ and 1 large step where you end up at location $\vec{x}_{big}$. Since the truncation error for the Runge-Kutta method is of order Δx^5 then the new step can be estimated from the error $\Delta_c = |\vec{x}_{big} - \vec{x}_{small}|$ given by

$$\Delta x_{est} = \Delta x \left| \frac{\Delta_i}{\Delta_c} \right| \tag{29}$$

where Δ_i is the user-specified error. This is only and estimate, so the new step size is given by $\Delta x_{new} = S_1 \Delta x_{est}$ where S_1 is a safety factor (< 1). A second safety factor S_2 is used to make sure that the program does not increase the stepsize too enthusiastically. The new timestep is then

$$\Delta x_{new} = \begin{cases} S_2 \Delta x_{old} & \text{if} S_1 \Delta x_{est} > S_2 \Delta x_{old} \\ \Delta x_{old}/S_2 & \text{if} S_1 \Delta x_{est} > \Delta x_{old}/S_2 \\ S_1 \Delta x_{est} & \text{otherwise} \end{cases} \tag{30}$$

9 Predictor-corrector Methods

Let's go back to equation 7 and rewrite it in integral form

$$y(x+h) = y(x) + \int_x^{x+h} f(x, y(x))dx \tag{31}$$

If we knew $y(x)$ in the interval $[x, x+h]$ we could integrate 31 to get $y(x+h)$. The problem is that we need to know the solution to get the solution. The basic idea of predictor-corrector methods is to guess $y(x)$ by some form of extrapolation (predictor) and then use it to get a more accurate estimate of $y(x)$ (the corrector part). The basic approach for extrapolation is to assume that the function $f(x)$ can be expressed as a polynomial

$$f(x, y(x)) = a + bx + cx^2 \tag{32}$$

Integrating 31 using 32 we get

$$\int_x^{x+h} f(x, y(x))dx \approx ah + bxh + \frac{1}{2}bh^2 + cx^2h + cxh^2 + \frac{1}{3}ch^3 \tag{33}$$

If we assume that the solution to 33 can be expressed as some weighted sum of the values of the function f(x,y(x)) in the previous intervals (i.e. $f(x, y(x))$, $f(x-h, y(x-h))$, and $f(x-2h, y(x-2h)))$ in the form

$$\int_x^{x+h} f(x, y(x))dx \approx h[\alpha f(x, y(x)) + \beta f(x-h, y(x-h)) + \gamma f(x-2h, y(x-2h)) \tag{34}$$

Using 32 in equation 34 and then equating the result to 33 we get 3 equations for α, β, and γ in the form

$$\begin{aligned} \alpha + \beta + \gamma &= 1 \\ -\beta + 2\gamma &= \frac{1}{2} \\ \beta + 4\gamma &= \frac{1}{3} \end{aligned}$$

which leads to the solution

$$\begin{aligned} \alpha &= \frac{23}{12} \\ \beta &= -\frac{16}{12} \\ \gamma &= \frac{5}{12} \end{aligned}$$

The predictor step $y_p(x+h)$ we then get from 31 as

$$y_p(x+h) = \frac{h}{12}[23f(x, y(x)) - 16f(x-h, y(x-h) + 5f(x-2h, y(x-2h)] \tag{35}$$

For the corrector step we then assume that

$$\int_x^{x+h} f(x, y(x))dx \approx h[\alpha' f(x+h, y(x+h)) + \beta' f(x, y(x)) + \gamma' f(x-h, y(x-h)) \tag{36}$$

and repeat the above procedure to get the coefficients α', β', and γ'. The corrector step then gives

$$y_c(x+h) = y(x) - \frac{h}{12}[9f(x+h, y_p(x+h)) + 8f(x, y(x)) - f(x-h, y(x-h)) + f(x-2h, y(x-2h))] \tag{37}$$

This approach is called the 3rd-order Adams-Bashforth-Moulton method. A more commonly used version of the 4-th order Adams-Bashforth-Moulton method which goes as

$$y_p(x+h) = y(x) + \frac{h}{24}[55f(x+h, y_p(x-h)) + 37f(x-2h, y(x-2h)) - 9f(x-3h, y(x-3h))] \quad (38)$$

for the predictor step

$$y_c(x+h) = y(x) + \frac{h}{24}[9f(x+h, y_p(x+h)) + 19f(x, y(x)) - 5f(x-h, y(x-h)) + f(x-2h, y(x-2h))] \quad (39)$$

for the corrector.

10 Other methods

Other methods such as the *Burlisch-Stoer* method use extrapolation methods to estimate what the solution would be if the stepsize approached zero. The coding of these methods can be quite involved, an example of which can be found in *Numerical Recipes*. This particular method is very precise and is often used in tracing particles through electric and magnetic fields where conservation of energy is important.

11 Programs

11.1 Program Euler's method

My MATLAB version of the Euler's method is

```
function [t,y]=euler(f,a,b,y0,n,param)
% function e=euler(f,a,b,y0,n)
% input:
% f = function entered as a string 'f'
% a and b are the left and right limits
% ya = inital condition
% n = number of steps
% param = parameters passed to the function
% output:
% e=[t y]
% t = time
% y = solution

t=linspace(a,b,n+1);
h=t(2)-t(1);
y(:,1)=y0;

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

The usage for the program would then be `[x,y]=euler('fun2',0,5,0,10,param)` which passes 'fun2' as $\frac{dy}{dx}$, and returns a 2-d vector [x y] for the solution over an interval $x = 0$ to $x = 5$ with an initial condition $y_0 = 0$ with 10 steps. To check accuracy, you could compare the result to the known solution, this is done in the following program for 10 steps

```
y0=0;
n=10
param=[];
[x,y]=euler('fun2',0,5,y0,n,param);
plot(t,fun2i(x,y,y0),x,y,'+-');
xlabel('x'); ylabel('temperature (y)');

legend('exact','euler')
grid
error1=norm(y-fun2i(x,y,y0))/n;
disp([' euler fixed step error =',num2str(error1),...
' for ',num2str(n),' steps'])
```

11.2 Variable step Euler's method

To implement this in matlab is a little trickier than standard Euler since you do not know that how many steps the solver will need to from from a starting point $x = a$ to an ending point $x = b$. For the regular Euler method $h = (b-a)/n$ so the solver will return 2 $n-$dimensional vectors for the solution. The trick is to use the feature in MATLAB where you have, for example, a vector $x = [12]$ and you want to add an additional value (say 3) to make it of length 3 instead of 2. In matlab you simply issue the command $x = [x3]$. Try it. Another thing one has to be careful of is that sometimes the computed values of h may get very small or very large, so one has to set maximum and minimum values of h. The following m-file is my version of a variable-step Euler solver.

```
 function [t,y]=eulera(f,a,b,y0,eps,param)
% function e=eulea2(f,a,b,y0,eps)
% input:
% f = function entered as a string 'f'
% a and b are the left and right intervals
% ya = inital condition
% eps = error tolerance
% output:
% e=[t' y']
% t = time
% y = solution
% adjustable stepsize version

h0 = (b-a)/10; % initial step
hmin = 1.e-4;
hmax = (b-a)/10;
```

```
hmin = (b-a)/100;
h = h0;

y = [y0];
t = [a];

while t(end) <= b

   h2 = h/2;
   [tt,yt]=euler(f,t(end),t(end)+h,y(:,end),1,param);
   y1=yt(:,end);
   [tt,yt]=euler(f,t(end),t(end)+h2,y(:,end),1,param);
   y2=yt(:,end);
   [tt,yt]=euler(f,t(end)+h2,t(end)+h,y2,1,param);
   y3=yt(:,end);
   error = abs(y1-(y3)) ;

  if error > eps
   h = min(h/2,hmin);
  else
   y = [y,y1];
   t = [t, t(end)+h];
   h = max(2*h,hmax);
  end

end
```

11.3 RK4 Method

My implementation in matlab of the RK4 scheme is shown in the m-file rk4.m, shown below.

```
function [t,y]=rk4(f,a,b,y0,n,param)
% function e=euler(f,a,b,y0,n)
% input:
% f = function entered as a string 'f'
% a and B are the left and right intervals
% ya = inital condition
% n = number of steps
% param = parameters passed to function
% output:
% e=[t' y']
% t = time
% y = solution
% fourth order scheme
```

```
t=linspace(a,b,n+1);
h=t(2)-t(1);
y(:,1) = y0;

for i=1:n
   k1=h*feval(f,t(i),y(:,i),param);
   k2=h*feval(f,t(i)+h/2,y(:,i)+k1/2,param);
   k3=h*feval(f,t(i)+h/2,y(:,i)+k2/2,param);
   k4=h*feval(f,t(i)+h,y(:,i)+k3,param);
   y(:,i+1)=y(:,i)  +(k1+2*k2+2*k3+k4)/6.;
end
```

11.4 MATLAB ODE45 program

The algorithm ODE45 that is used in matlab is roughly of the type discussed in section 8. (I couldn't find the exact reference since the Rice library doesn't seem to carry the journal that has the article describing the method.) Implementation of ODE45 is done in the following way

```
options = odeset('RelTol',1e-3,'AbsTol',1e-4);
[x4,y4]=ode45('fun2',[a,b],y0,options)
```

odeset This sets the options that are used by ODE45.

RelTol The is the relative tolerance the the solver should strive for (Sort of like a percentage error).

AbsTol This sets the absolute tolerance.

11.5 A comparison of the 4 algorithms

A program that compares all the solvers mentioned in this section would looks like

```
y0=0;
a=0;
b=50;
param=[];

n=20
tic
[t,y]=euler('fun2',a,b,y0,n,param);
timee=toc
toc

% euler variable step
error=1.0e-4
tic
```

```
[t2,y2]=eulera('fun2',a,b,y0,error,param);
times=toc
toc
m=size(t2)

tic
% rk4 scheme
[t3,y3]=rk4('fun2',a,b,y0,n,param);
timek=toc
toc

% ode45 scheme
tic
options = odeset('RelTol',1e-3,'AbsTol',1e-4);
[t4,y4]=ode45('fun2',[a,b],y0,options);
time45=toc
toc

plot(t,fun2i(t,y,y0),t,y,'r-+',t2,y2,'g^-',t3,y3,'y*',t4,y4,'mp')

legend('exact','euler','euler variable','rk4','ode45')
grid

% compare the errors

error1=norm(y-fun2i(t,y,y0));

disp(['    euler fixed step error = ',num2str(error1),...
' for ',num2str(n),' steps in time ',num2str(timee),' seconds'])

m=size(t2);
error2=norm(y2-fun2i(t2,y,y0))/m(1);

disp([' euler variable step error = ',num2str(error2),...
' for ', num2str(m(2)),' steps in time ',num2str(times),' seconds'])

o=size(t3);
error3=norm(y3-fun2i(t3,y,y0))/o(1);

disp(['          rk4 method error =   ',num2str(error3),...
' for ', num2str(o(2)),' steps in time ',num2str(timek),' seconds'])

p=size(t4);
error4=norm(y4-fun2i(t4,y,y0))/p(1);
```

```
disp([' ode45 variable step error =   ',num2str(error4),...
' for ', num2str(p(1)),' steps in time ',num2str(time45),...
' seconds'])
```

Running the program for the Newton's law of cooling example for 50 timesteps produces the following output

```
   euler fixed step error = 1.0766 for 20 steps in time 0.024936 seconds
 euler variable step error = 0.054254 for 221 steps in time 5.3437 seconds
          rk4 method error =   0.050202 for 21 steps in time 0.041293 seconds
 ode45 variable step error =   8.1678e-06 for 69 steps in time 0.41626 seconds
```

The Matlab routine ODE45 is by far the most accurate, but also the slowest in this case (although this is not necessarily always true). The output is shown in figure 6. As you can see the RK4 scheme is as accurate as the variable step Euler but somewhat faster. It did only take 20 steps, but recall each step required 4 function evaluations, so if the function you are trying to integrate is computationally expensive, that RK4 may not necessarily be the best for the job. In addition, if the function $f(x, y)$ is noisy, say when it is obtained from a numerical grid, RK4 schemes may have trouble since it assumes that the derivatives of $f(x, y)$ are smooth.

11.6 Footnote: Matlab Commands used

disp Displays the array, without printing the array name. Note that for multiple character output you have to define an array.

num2str This converts numbers to strings.

norm Computes matrix or vector norm.

12 References and resources

1. Much of this article was based on the book *Numerical Methods Using Matlab*, by J. H. Matthews and K. D. Fink, Prentice Hall, 1987.

2. A well know and popular book called *Numerical Recipes* which is now avail-able free of charge on the internet at
   `http://www.ulib.org/webRoot/Books/Numerical_Recipes/` there are fortran (77 and 90) and C version of the book.

3. *The Nature of Mathematical Modeling* by Neil Gershenfeld has some nice (but brief) discussions of numerical methods.

4. *Numerical Methods for Physics* by Alejandro Garcia.

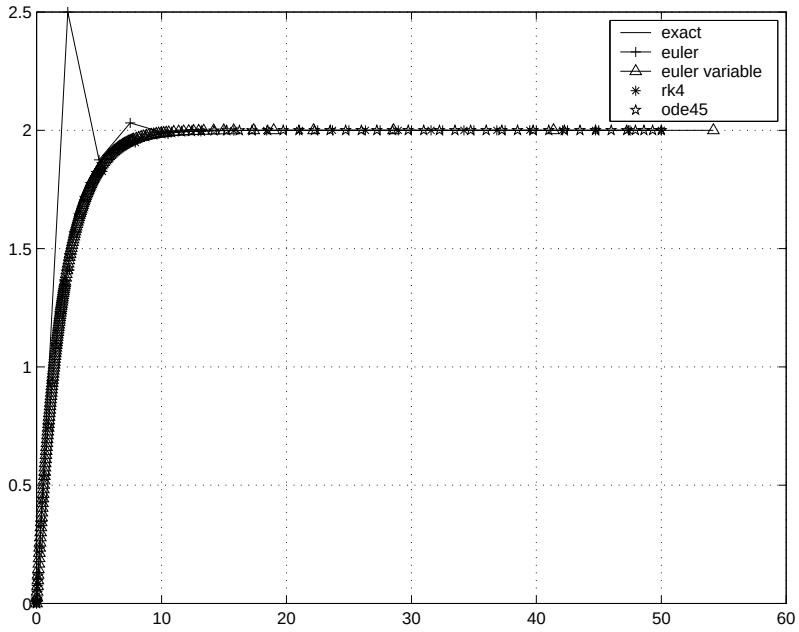

Figure 6: A comparison of methods for n=20.