Lecture 30
Euler Methods

Numerical Solution of an IVP

Suppose we wish to numerically solve the initial value problem
\[
\dot{y} = f(t, y), \quad y(a) = y_0, \quad (30.1)
\]
on an interval of time \([a, b]\).

By a numerical solution, we must mean an approximation of the solution at a finite number of points, i.e.
\[
(t_0, y_0), (t_1, y_1), (t_2, y_2), \ldots, (t_n, y_n),
\]
where \(t_0 = a\) and \(t_n = b\). The first of these points is exactly the initial value. If we take \(n\) steps as above, and the steps are evenly spaced, then the time change in each step is
\[
h = \frac{b - a}{n}, \quad (30.2)
\]
and the times \(t_i\) are given simply by \(t_i = a + ih\). This leaves the most important part of finding a numerical solution: determining \(y_1, y_2, \ldots, y_n\) in a way that is as consistent as possible with (30.1). To do this, first write the differential equation in the indexed notation
\[
\dot{y}_i \approx f(t_i, y_i), \quad (30.3)
\]
and will then replace the derivative \(\dot{y}\) by a difference. There are many ways we might carry this out and in the next section we study the simplest.

The Euler Method

The most straightforward approach is to replace \(\dot{y}_i\) in (30.3) by its forward difference approximation. This gives
\[
\frac{y_{i+1} - y_i}{h} = f(t_i, y_i).
\]
Rearranging this gives us a way to obtain \(y_{i+1}\) from \(y_i\) known as Euler’s method:
\[
y_{i+1} = y_i + hf(t_i, y_i). \quad (30.4)
\]
With this formula, we can start from \((t_0, y_0)\) and compute all the subsequent approximations \((t_i, y_i)\). This is very easy to implement, as you can see from the following program (which can be downloaded from the class web site).

```matlab
function [T, Y] = myeuler(f, tspan, y0, n)
% function [T, Y] = myeuler(f, tspan, y0, n)
% Solves dy/dt = f(t,y) with initial condition y(a) = y0
```
% on the interval [a,b] using n steps of Euler's method.
% Inputs: f -- a function f(t,y) that returns a column vector of the same
% length as y
% tspan -- a vector [a,b] with the start and end times
% y0 -- a column vector of the initial values, y(a) = y0
% n -- number of steps to use
% Outputs: T -- a n+1 column vector containing the times
% Y -- a (n+1) by d matrix where d is the length of y
% Y(j,i) gives the ith component of y at time T(j)

a = tspan(1); b = tspan(2); % parse starting and ending points
h = (b-a)/n; % step size
t = a; T = a; % t is the current time and T will record all times
y = y0; % y is the current variable values, as a column vector
Y = y0'; % Y will record the values at all steps, each in a row
for i = 1:n
    y = y + h*f(t,y); % Euler update of y to next time
    t = a + i*h; % The next time.
    T = [T; t]; % Record t and y into T and Y.
    Y = [Y; y'];
end

To use this program we need a function, such as the vector function for the pendulum:

```matlab
f = inline('[y(2);-.1*y(2)-sin(y(1)) + sin(t)]','t','y')
```

Then type

```matlab
[T Y] = myeuler(f,[0 20],[1;-1.5],5);
```

Here [0 20] is the time span you want to consider, [1;-1.5] is the initial value of the vector y and 5 is the number of steps. The output T contains times and Y contains values of the vector as the times. Try

```matlab
size(T)
size(Y)
```

Since the first coordinate of the vector is the angle, we only plot its values:

```matlab
theta = Y(:,1);
plot(T,theta)
```

In this plot it is clear that n = 5 is not adequate to represent the function. Type

```matlab
hold on
```

then redo the above with 5 replaced by 10. Next try 20, 40, 80, and 200. As you can see the graph becomes increasingly better as n increases. We can compare these calculations with MATLAB's built-in function with the commands

```matlab
[T Y] = ode45(f,[0 20],[1;-1.5]);
theta = Y(:,1);
plot(T,theta,'r')
```

The problem with the Euler method

You can think of the Euler method as finding a linear approximate solution to the initial value problem on each time interval. An obvious shortcoming of the method is that it makes the approximation based on information at the beginning of the time interval only. This problem is illustrated well by the following IVP:

\[
\ddot{x} + x = 0 \quad \text{with} \quad x(0) = 1 \quad \text{and} \quad \dot{x}(0) = 0.
\]  

(30.5)
You can easily check that the exact solution of this IVP is 

\[ x(t) = \cos(t). \]

If we make the standard change of variables 

\[ y_1 = x \quad \text{and} \quad y_2 = \dot{x}, \]

then we get 

\[ \dot{y}_1 = y_2 \quad \text{and} \quad \dot{y}_2 = -y_1. \]

Then the solution should be \( y_1(t) = \cos(t) \) and \( y_2(t) = \sin(t) \). If we then plot the solution in the \((y_1, y_2)\) plane, we should get exactly a unit circle. We can solve this IVP with Euler’s method:

```matlab
> g = inline('[y(2);-y(1)]','t','y')
> [T Y] = myeuler(g,[0 4*pi],[1;0],20)
> y1 = Y(:,1);
> y2 = Y(:,2);
> plot(y1,y2)
```

As you can see the approximate solution goes far from the true solution. Even if you increase the number of steps, the Euler solution will eventually drift outward away from the circle because it does not take into account the curvature of the solution.

**The Modified Euler Method**

An idea which is similar to the idea behind the trapezoid method would be to consider \( f \) at both the beginning and end of the time step and take the average of the two. Doing this produces the Modified (or Improved) Euler method represented by the following equations:

\[
\begin{align*}
    k_1 &= hf(t_i, y_i) \\
    k_2 &= hf(t_i + h, y_i + k_1) \\
    y_{i+1} &= y_i + \frac{1}{2} (k_1 + k_2). 
\end{align*}
\]

(30.6)

Here \( k_1 \) captures the information at the beginning of the time step (same as Euler), while \( k_2 \) is the information at the end of the time step. The following program implements the Modified method. It may be downloaded from the class web site.

```matlab
function [T , Y] = mymodeuler(f,tspan,y0,n)
% Solves dy/dt = f(t,y) with initial condition y(a) = y0
% on the interval [a,b] using n steps of the modified Euler’s method.
% Inputs: f -- a function f(t,y) that returns a column vector of the same
%         length as y
%         tspan -- a vector [a,b] with the start and end times
%         y0 -- a column vector of the initial values, y(a) = y0
%         n -- number of steps to use
% Outputs: T -- a n+1 column vector containing the times
%          Y -- a (n+1) by d matrix where d is the length of y
%          Y(j,i) gives the ith component of y at time T(j)

a = tspan(1); b = tspan(2); % parse starting and ending points
h = (b-a)/n; % step size
```
t = a;  T = a;  \% t is the current time and T will record all times
y = y0;  \% y is the current variable values, as a column vector
Y = y0';  \% Y will record the values at all steps, each in a row
for i = 1:n
    k1 = h*f(t,y);  \% y increment based on current time
    k2 = h*f(t+h,y+k1);  \% y increment based on next time
    y = y + .5*(k1+k2);  \% update y using the average
    t = a + i*h;  \% The next time.
    T = [T; t];  \% Record t and y into T and Y.
    Y = [Y; y'];
end

We can test this program on the IVP above:
> [T Ym] = mymodeuler(g,[0 4*pi],[1;0],20)
> ym1 = Ym(:,1);
> ym2 = Ym(:,2);
> plot(ym1,ym2)

Exercises

30.1 Download the files myeuler.m and mymodeuler.m from the class web site.

1. Type the following commands:
   > hold on
   > f = inline('sin(t)*cos(x)','t','x')
   > [T Y]=myeuler(f,[0,12],.1,10);
   > plot(T,Y)
   Position the plot window so that it can always be seen and type
   > [T Y]=myeuler(f,[0,12],.1,20);
   > plot(T,Y)
   Continue to increase the last number in the above until the graph stops changing (as far as you can see). Record this number and print the final graph. Type hold off and kill the plot window.

2. Follow the same procedure using mymodeuler.m.

3. Describe what you observed. In particular compare how fast the two methods converge as n is increased (h is decreased).