Extending the Numerical Solution of ODEs to more complicated systems

ChE/MSE 505
David Keffer
Dept. of Chemical and Biomolecular Engineering
The University of Tennessee, Knoxville
notes begun: September 19, 2007
last updated: October 16, 2007
dkeffer@utk.edu

In our notes thus far, we have shown how it is a very straight forward process to use a numerical method, such as the Euler method or the Classical Fourth Order Runge-Kutta method, to integrate systems of first order ODEs of the form:

$$\frac{d\underline{y}}{dt} = \underline{f}\left(t, \underline{y}\right) \tag{1}$$

subject to the initial conditions

$$\underline{y}\left(t_o\right) = \underline{y}_o \tag{2}$$

The procedure was outlined in the notes located at
http://clausius.engr.utk.edu/che301/pdf/odes.pdf.

This procedure works for one ODE or for systems of ODEs. This procedure works for linear or nonlinear ODEs. This procedure requires that the problem is an Initial Value Problem (IVP), in which all the conditions are known at the same value of the independent variable, t_o.

A code for performing Euler and Runge-Kutta methods is given on the course website in the ODE505.zip file. The code is in two files. sysode.m contains the "guts" of the routine. sysodeinput.m is where the ODEs, equation (1), are provided.

This procedure and the associated code has three limitations:

- The code only works for first order ODEs
- The code only works if you can isolate the derivatives on the LHS of the ODE, as is shown in equation (1)
- The code only works if you have an IVP, but won't work for Boundary Value Problems (BVPs).

We now proceed to eliminate each of these three limitations.

I. Solving Higher Order ODEs

Just as is the case with the analytical solution of higher order ODEs, the trick to using numerical methods to solve higher order ODEs is to transform them into a set of first order ODEs. This can always be done, regardless of the linearity of the original ODE.

This is a simple three step process. You begin with a problem statement,

$$\frac{d^n y}{dx^n} = f\left(x, y, \frac{dy}{dx}, \frac{d^2 y}{dx^2}, \frac{d^3 y}{dx^3} \cdots \frac{d^{n-2} y}{dx^{n-2}}, \frac{d^{n-1} y}{dx^{n-1}} \right) \tag{I.1}$$

In this example, the problem is an IVP with initial conditions given by

$$y(x_o) = y_o$$

$$\left. \frac{d^m y}{dx^m} \right|_{x_o} = y_o^{(m)} \qquad \text{for } m = 1 \text{ to } n\text{-}1. \tag{I.2}$$

The first step is to define the new variables, y_1 to y_n. This step is always the same regardless of problem.

$$\begin{aligned} y_1 &\equiv y \\ y_2 &\equiv \frac{dy}{dx} \\ y_3 &\equiv \frac{d^2 y}{dx^2} \\ &\vdots \\ y_n &\equiv \frac{d^{n-1} y}{dx^n} \end{aligned} \tag{I.3}$$

The second step is to write ODEs in terms of the new variables. The first n-1 ODEs always have the same form.

$$\begin{aligned} \frac{dy_1}{dx} &= y_2 \\ \frac{dy_2}{dx} &= y_3 \\ &\vdots \\ \frac{dy_{n-1}}{dx} &= y_n \end{aligned} \tag{I.4}$$

The last ODE is simply the original higher-order ODE with the new variables substituted in.

$$\frac{dy_n}{dx} = f\left(x, y_1, y_2, y_3 \ldots y_{n-1}, y_n\right) \tag{I.5}$$

At this point you have a set of *n* first-order ODEs.

The third and final step is to convert the ICs to the new variables.

$$y_1(x_o) = y_o$$

$$y_m(x_o) = y_o^{(m-1)} \quad \text{for } m = 2 \text{ to } n. \tag{I.6}$$

You can now use the Runge-Kutta method or any other ODE solver of your choice to solve this set of *n* first-order ODEs with *n* initial conditions.

An example. We have a third order ODE.

$$\frac{d^3y}{dx^3} = f\left(x, y, \frac{dy}{dx}, \frac{d^2y}{dx^2}\right) = \frac{d^2y}{dx^2} - 2\frac{dy}{dx} + y^2 \exp\left(-\frac{x}{2}\right) \tag{I.7}$$

In this example, the problem is an IVP with initial conditions given by

$$y(x_o = 0) = y_o = 1$$

$$\left.\frac{dy}{dx}\right|_{x_o} = y_o^{(1)} = 0$$

$$\left.\frac{d^2y}{dx^2}\right|_{x_o} = y_o^{(2)} = -1 \tag{I.8}$$

The first step is to define the new variables, y_1 to y_n. This step is always the same regardless of problem.

$$y_1 \equiv y$$
$$y_2 \equiv \frac{dy}{dx}$$
$$y_3 \equiv \frac{d^2y}{dx^2} \tag{I.9}$$

The second step is to write ODEs in terms of the new variables. The first *n*-1 ODEs always have the same form.

$$\frac{dy_1}{dx} = y_2$$
$$\frac{dy_2}{dx} = y_3 \tag{I.10}$$

The last ODE is simply the original higher-order ODE with the new variables substituted in.

$$\frac{dy_3}{dx} = y_3 - 2y_2 + y_1^2 \exp\left(-\frac{x}{2}\right) \tag{I.11}$$

At this point you have a set of 3 first-order ODEs.

The third and final step is to convert the ICs to the new variables.

$$y_1(x_o) = y_o$$
$$y_2(x_o) = y_o^{(1)}$$
$$y_3(x_o) = y_o^{(2)} \tag{I.12}$$

We can use sysode.m to solve this problem. The sysodeinput.m file contains

```
function dydt = sysodeinput(x,y,nvec);
dydt(1) = y(2);
dydt(2) = y(3);
dydt(3) = y(3) - 2*y(2) + y(1)*exp(-x/2);
```

At the command line prompt, I call sysode.m

```
>> [x,y]=sysode(2,100,0,5,[1,0,-1]);
```

The plot of the solution is given in Figure 1.

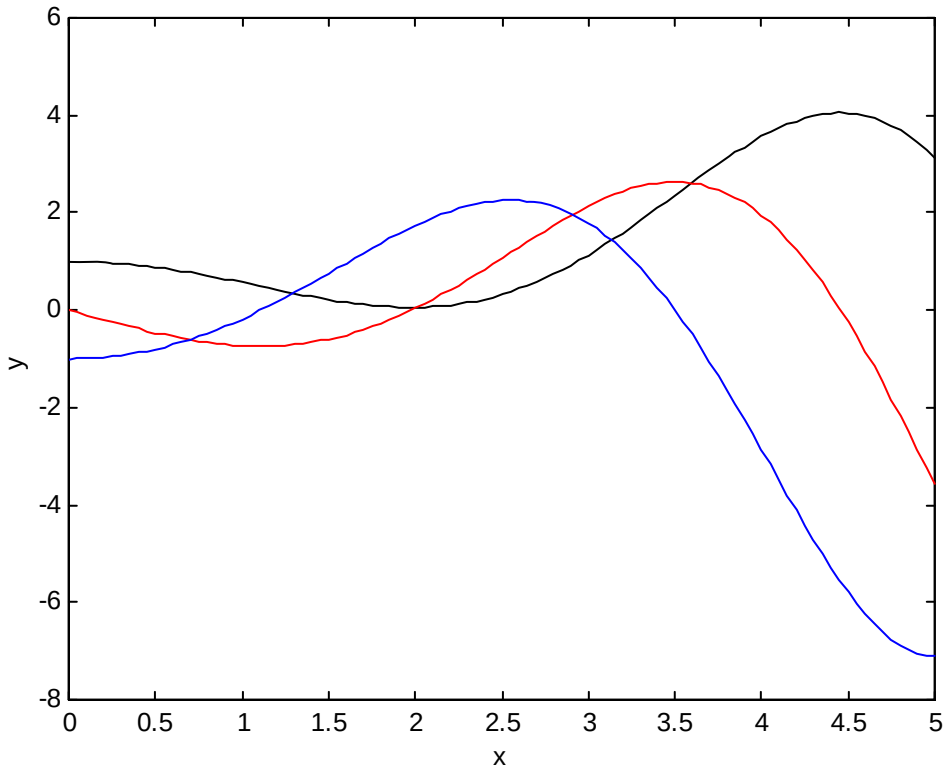

Figure 1. Plot of solution to ODE given in (I.7) with ICs given in (I.8). The black line is the solution. The red line is the first derivative and the blue line is the second derivative.

II. Solving ODEs where you cannot isolate the derivatives on the LHS

If we have an ODE of the form,

$$g\left(z, y, \frac{dy}{dz}\right) = 0$$

we can use our standard numerical procedure so long as we can rearrange it as,

$$\frac{dy}{dz} = f(z, y,)$$

However, some ODEs cannot be written in such a way that the derivative is isolated on the LHS of the equation. For example, the derivative in the ODE

$$\frac{dy}{dz}\sin\left(\frac{dy}{dz}\right) - 2y = 0$$

cannot be isolated on the LHS. In order to solve this type of ODE, we must embed the Newton-Raphson method (or analogous AE numerical routine) within the Runge-Kutta method (or analogous ODE numerical routine). In essence, we will treat the ODE at each instant in time as an Algebraic Equation (AE), using an iterative solver (like Newton-Raphson) to find the instantaneous value of the derivative.

Consider coupling the Euler method with the Newton Raphson Method. At step one, we have

$$y(z_1) = y(z_o) + \Delta z\left(\frac{dy}{dz}\right)\Big|_{y(z_o)}$$

we need to know $\left(\frac{dy}{dz}\right)\Big|_{y(z_o)}$. Therefore, we can call this unknown x, $x \equiv \left(\frac{dy}{dz}\right)\Big|_{y(z_o)}$. Our example ODE above becomes,

$$x\sin(x) - 2y(z_o) = 0.$$

The only unknown in this equation is x. We use the Newton-Raphson method to iteratively solve for x. Once we have converged, we know the derivative and we can plug it into the Euler formula.

This procedure must be done at every time step. In practice we use the converged value of the slope at the previous step as the initial guess of the slope at the next step. If the steps, Δz, are small enough, then this guess will be good enough to lead to convergence. If the NR method does not converge, then we typically choose a smaller Δz until it does. In this way, we need find only a good initial guess for the derivative at the initial condition.

A complete example of the numerical solution of an ODE where you cannot isolate the derivative on the LHS, can be found on the web at http://clausius.engr.utk.edu/che310/index.html. This is the "Efflux from the Tank Experiment". There is a derivation of the ODE in the experimental description. The Matlab codes are provided on the site as well.

III. Solving ODEs with Boundary Conditions (Boundary Value Problems)

Consider two fluids separated by a membrane in which chemical reaction takes place,

$2A \rightarrow B$

If only diffusion and reaction are present, at steady state we can describe the concentration of the reactant

$$0 = D\frac{d^2C_A}{dz^2} - kC_A^2$$

where D is the diffusivity, k is the rate constant, C_A is the concentration of A and z is the length dimension. At steady state, the boundary conditions are

$$C_A\left(z_o = 0\right) = C_{A,o} = 2.0$$

$$C_A\left(z_f = 1\right) = C_{A,f} = 1.5$$

The diffusivity is $D = 2.4$ and $k = 1.2$.

We first convert the second order ODE to a set of two first order ODEs. The ODE can first be rearranged as

$$\frac{d^2C_A}{dz^2} = \frac{k}{D}C_A^2$$

We define new variables.

$$y_1 \equiv C_A$$

$$y_2 \equiv \frac{dC_A}{dz}$$

We write the ODEs in terms of the new variables.

$$\frac{dy_1}{dz} = y_2$$

$$\frac{dy_2}{dz} = \frac{k}{D}y_1^2$$

We write the BCs in terms of the new variables.

$$y_1(z_o = 0) = C_{A,o}$$

$$y_1(z_f = 1) = C_{A,f}$$

We cannot solve this problem using the traditional numerical method of solving ODE IVP, because we do not have initial conditions for both y_1 and y_2. Therefore, we will guess the initial value of y_2 and iteratively solve the set of ODEs until we match the final boundary condition. To do this, we will use the Newton-Raphson (NR) method. The NR method solves an algebraic equation of the form

$$f(x) = 0$$

In this case, our unknown is $y_{2,o} \equiv y_2(z = 0)$. Our algebraic equation is just a rearrangement of the final boundary condition that we want to satisfy.

$$f(y_{2,o}) = y_1(z_f) - C_{A,f} = 0$$

The value of $y_1(z_f)$ that one obtains by solving the set of ODEs is implicitly a function of $y_{2,o}$.

The solution procedure amounts to embedding the Runge-Kutta method (or analogous ODE numerical routine) within the Newton-Raphson method (or analogous AE numerical routine). The solution procedure is as follows.

1. Make an initial guess for the unknown, $y_{2,o}$.
2. Use the Newton-Raphson method with Numerical Derivatives (NRwND) to evaluate the function, $f(y_{2,o})$. This evaluation requires three solutions of the set of ODEs. The first uses the guess, $y_{2,o}$. The second and third use the guesses, $y_{2,o} + h$ and $y_{2,o} - h$, where h is some small number and provide a numerical estimate of the derivative, $\frac{df}{dy_{2,o}}$.
3. Get a new estimate of $y_{2,o}$ via NR.

$$y_{2,o}{}^{new} = y_{2,o}{}^{old} - \frac{f(y_{2,o}{}^{old})}{\left(\frac{df}{dy_{2,o}}\right)\Big|_{y_{2,o}{}^{old}}} = y_{2,o}{}^{old} - \frac{f(y_{2,o}{}^{old})}{\frac{f(y_{2,o}{}^{old} + h) - f(y_{2,o}{}^{old} - h)}{2h}}$$

4. Iterate until converged.

The Matlab code used to solve this problem is given in Appendix III.

The command line input and output used to generate the solution are

```
[x0,err] = newraph_nd(0.1);
```

```
icount = 1 xold = 1.000000e-001 f = 1.843234e+000 df = 1.447914e+000 xnew = -
1.173027e+000 err = 1.000000e+002
icount = 2 xold = -1.173027e+000 f = 9.525421e-002 df = 1.300685e+000 xnew =
-1.246261e+000 err = 5.876289e-002
icount = 3 xold = -1.246261e+000 f = 2.945927e-004 df = 1.292648e+000 xnew =
-1.246489e+000 err = 1.828326e-004
icount = 4 xold = -1.246489e+000 f = 2.891606e-009 df = 1.292623e+000 xnew =
-1.246489e+000 err = 1.794647e-009
```

The problem converges in four iterations. We plot the solution to this problem in Figure 3.

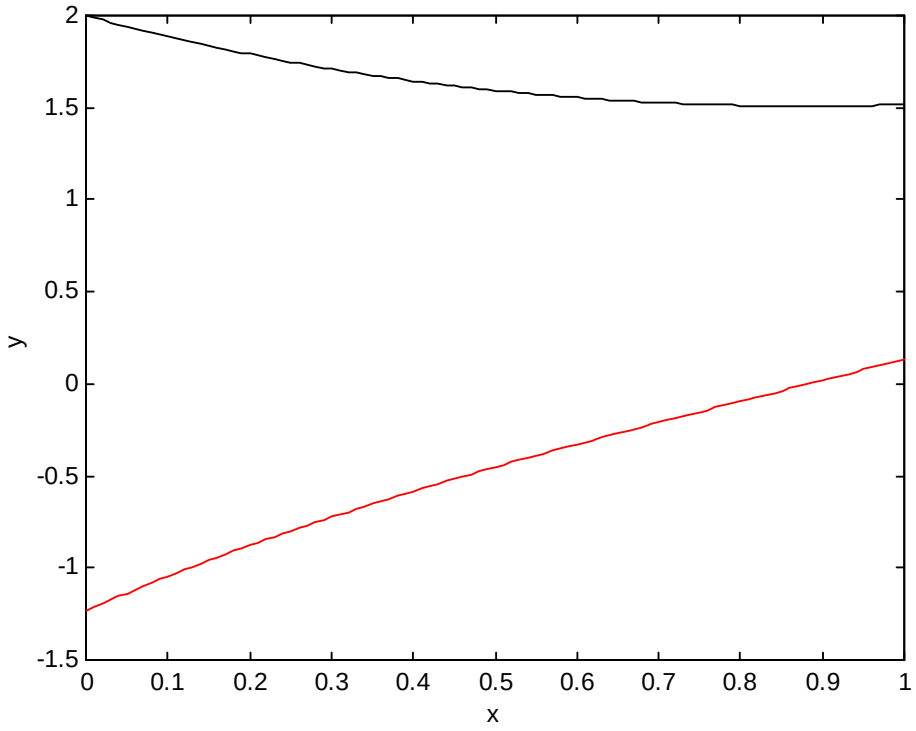

Figure 3. Converged profiles of the concentration of A (top curve, black) and the derivative of the concentration of A (bottom curve, red) as a function of position.

IV. Solving ODEs coupled with Algebraic Equations (constraints)

Describe Zhao Wang's sintering example. Notes Forthcoming.

Appendix III. Matlab Code for ODE BVPs

```
%
% Newton-Raphson method with numerical approximations to the derivative.
%
function [x0,err] = newraph_nd(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold =x0;
while (err > tol & icount <= maxit)
icount = icount + 1;
f = funkeval(xold);
h = min(0.01*xold,0.01);
df = dfunkeval(xold,h);
xnew = xold - f/df;
if (icount > 1)
err = abs((xnew - xold)/xnew);
end
fprintf(1,'icount = %i xold = %e f = %e df = %e xnew = %e err = %e \n',icount, xold, f, …
        df, xnew, err);
xold = xnew;
end
%
x0 = xnew;
if (icount >= maxit)
% you ran out of iterations
fprintf(1,'Sorry. You did not converge in %i iterations.\n',maxit);
fprintf(1,'The final value of x was %e \n', x0);
end

function f = funkeval(x)
yf = 1.5;
method = 2;
n_interval = 100;
zo = 0.0;
zf = 1.0;
y1o = 2.0;
y2o = x;
[tvec,ymat] = sysode(method,n_interval,zo,zf,[y1o,y2o]);
y1f = ymat(n_interval+1,1);
f = y1f - yf;


function df = dfunkeval(x,h)
fp = funkeval(x+h);
fn = funkeval(x-h);
df = (fp - fn)/(2*h);

function dydt = sysodeinput(t,y,nvec);
%
%  ODE input file
%
k = 1.2;
D = 2.4;
dydt(1) = y(2);
dydt(2) = k/D*y(1)^2;



function [x,y]=sysode(m,n,xo,xf,yo);
%  sysode(m,n,xo,xf,yo)
%  This routine solves one non-linear first-order ordinary differential
%  equation initial value problem.
%
%  m = 1 for Euler's method
%  m = 2 for Classical Runge-Kutta 4rth order method
%  n = number of steps
```

```
%  xo = starting value of x
%  xf = ending value of x
%  o = number of first order ordinary differential equations
%  yo = initial condition at xo
%
%  The differential equation must appear in the file 'sysodeinput.m'
%  This program creates an output data file 'sysode.out'
%
%  Author:  David Keffer   Date:  October 23, 1998
%

%
%  STEP ONE.  input parameters
%
if (nargin ~= 5)
   error('sysode requires 5 input arguments');
end
nsize=size(yo);
o=max(nsize(1),nsize(2));
%
%  STEP TWO DEFINE ODE and I
%
%  yo is the initial condition
%  the ode is defined in a file called odeivpn.m
%yo = zeros(1,o);
%
%  the vector nvec contains some info that can be passed
%  to sysodeinput (in case you need it)
%  nvec(1) = i, the iteration number
%  nvec(2) = n, the total number of iterations
%  nvec(3) = kk, the intra-iteration access number
%  nvec(4) = o, the number of first order ODEs
%  nvec(5) = m, the method used to solve
%
%  STEP THREE.  SOLVE THE ODE
%
h = (xf-xo)/n;
x = xo : h : xf;
if (m == 1)
   y = eulerevaln(x,h,n,o,yo);
else
   y = rk4evaln(x,h,n,o,yo);
end
%
%  STEP FOUR.  PLOT THE RESULT
%
figure(1)
for i = 1:o
  if (i==1)
     plot (x,y(:,i),'k-'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==2)
     plot (x,y(:,i),'r-'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==3)
     plot (x,y(:,i),'b-'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==4)
     plot (x,y(:,i),'g-'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==5)
     plot (x,y(:,i),'m-'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==6)
     plot (x,y(:,i),'k:'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==7)
     plot (x,y(:,i),'r:'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==8)
     plot (x,y(:,i),'b:'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==9)
     plot (x,y(:,i),'g:'), xlabel( 'x' ), ylabel ( 'y' )
  elseif (i==10)
     plot (x,y(:,i),'m:'), xlabel( 'x' ), ylabel ( 'y' )
 else
     plot (x,y(:,i),'k-'), xlabel( 'x' ), ylabel ( 'y' )
  end
```

```
  hold on
end
hold off
%
%  STEP FIVE.  WRITE THE RESULT TO sysode.out
%
op1=o+1;
fid = fopen('sysode.out','w');
if (op1 == 2)
  fprintf(fid,'x              y(1) \n');
  fprintf(fid,'%13.7e %13.7e   \n', [x;y']);
elseif (op1 == 3)
  fprintf(fid,'x              y(1)           y(2) \n');
  fprintf(fid,'%13.7e %13.7e %13.7e  \n', [x;y']);
elseif (op1 == 4)
  fprintf(fid,'x              y(1)           y(2)           y(3) \n');
  fprintf(fid,'%13.7e %13.7e %13.7e %13.7e \n', [x;y']);
elseif (op1 == 5)
  fprintf(fid,'x              y(1)           y(2)           y(3)           y(4) \n');
  fprintf(fid,'%13.7e %13.7e %13.7e %13.7e %13.7e \n', [x;y']);
elseif (op1 == 6)
   fprintf(fid,'x              y(1)           y(2)           y(3)           y(4)',...
      '           y(5) \n');
  fprintf(fid,'%13.7e %13.7e %13.7e  %13.7e %13.7e %13.7e \n', [x;y']);
elseif (op1 == 7)
   fprintf(fid,'x              y(1)           y(2)           y(3)           y(4)',...
      '           y(5)           y(6) \n');
  fprintf(fid,'%13.7e %13.7e %13.7e %13.7e %13.7e %13.7e %13.7e  \n', [x;y']);
elseif (op1 == 8)
   fprintf(fid,'x              y(1)           y(2)           y(3)           y(4)',...
      '           y(5)           y(6)           y(7) \n');
  fprintf(fid,'%13.7e %13.7e %13.7e %13.7e %13.7e %13.7e %13.7e %13.7e  \n', [x;y']);
end
fclose(fid);

%
%  multivariate classical fourth order Runge-Kutta
%
function y = rk4evaln(x,h,n,o,yo)
%
%  specify initial values of y
%
y = zeros(n+1,o);
dydt = zeros(1,o);
k1 = zeros(1,o);
k2 = zeros(1,o);
k3 = zeros(1,o);
k4 = zeros(1,o);
yt = zeros(o,1);
ytt = zeros(1,o);
%
for j = 1:o
   y(1,j) = yo(j);
end
%  solve new values of dydt and y
nvec =[0;n;1;o;2];
for i =  1:n
   nvec(1)=i;
   xt = x(i);
   ytt = y(i,:);
   yt = ytt;
   nvec(3)=1;
   k1 = sysodeinput(xt,yt,nvec);
   xt = x(i)+h/2;
   for j = 1:o
      yt(j) = ytt(j)+h/2*k1(j);
   end
   nvec(3)=2;
   k2 = sysodeinput(xt,yt,nvec);
   for j = 1:o
      yt(j) = ytt(j)+h/2*k2(j);
```

```
   end
   nvec(3)=3;
   k3 = sysodeinput(xt,yt,nvec);
   xt = x(i)+h;
   for j = 1:o
      yt(j) = ytt(j)+h*k3(j);
   end
   nvec(3)=4;
   k4 = sysodeinput(xt,yt,nvec);
   for j = 1:o
      y(i+1,j) = y(i,j)+h/6*(k1(j)+2*k2(j)+2*k3(j)+k4(j));
   end
end

%
%  multivariate Euler method
%
function y = eulerevaln(x,h,n,o,yo)
%  specify initial values of y
y = zeros(n+1,o);
dydt = zeros(1,o);
for j = 1:o
   y(1,j) = yo(j);
end
nvec =[0;n;1;o;1];
%  solve new values of dydt and y
for i =  1:n
   nvec(1) = i;
   dydt = sysodeinput(x(i),y(i,:),nvec);
   for j = 1:o
      y(i+1,j) = y(i,j)+h*dydt(j);
   end
end
```