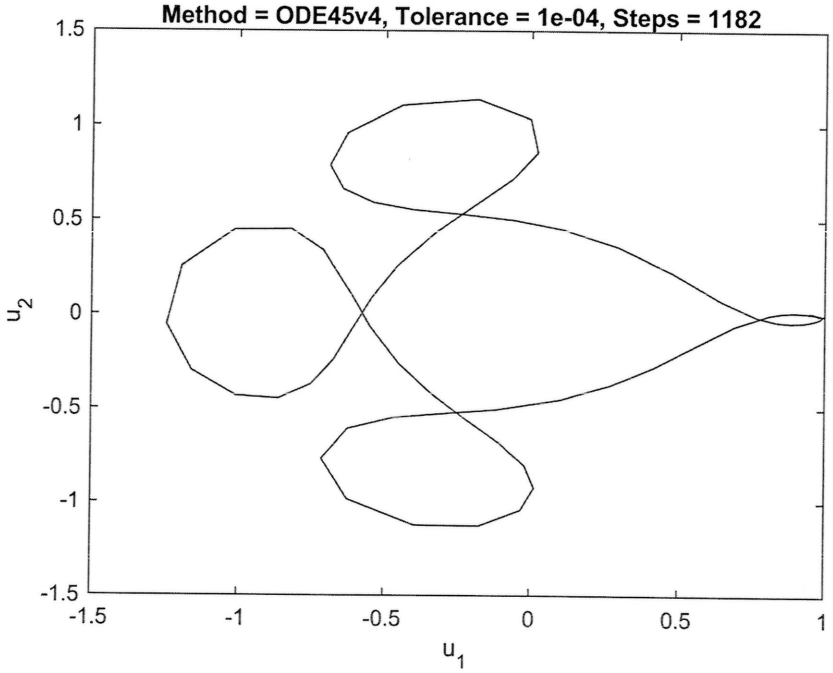
Andrew Alferman
MTH 552 Winter '17
Homework #5
Due Wed, February 22, 2017

1.a) The attached code has been modified as directed. A tolerance of approximately $1x10^{-4}$ was needed for the graph to look reasonably correct. Per the figure below, the ODE function for the orbit was called 1182 times in order to create the plot, as counted by the global variable "steps".



b) The butcher array of the method can be directly found in the ode45v4 code and is repeated below. In this array, the first row of c represents the values for the fifth order accurate formula, and the second row represents the values for the fourth order accurate formula. The method uses the fifth order accurate formula to carry forward the solution, as seen on line 84 of the ode45v4.m file, which reads: `y = y + h*f*gamma(:,1);` where the 1 is for the first row.

1/4	1/4	0	0	0	0	0
3/8	3/32	9/32	0	0	0	0
12/13	1932/2197	-7200/2197	7296/2107	0	0	0
1	8341/4104	-32832/4104	29440/4104	-845/4104	0	0
1/2	-6080/20520	41040/20520	-28352/20520	9295/20520	-5643/20520	0
	902880/7618050	0	3953664/7618050	3855735/7618050	-1371249/7618050	277020/7618050
	-2090/752400	0	22528/752400	21970/752400	-15048/752400	-27360/752400

Simple Python code for Problem 2a-2b)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 22 20:01:28 2017

@author: alfermaa
"""

import numpy as np

a0 = -3/4
a1 = -1/2
a2 = 1/4
a3 = 1.
b0 = 5/8
b1 = 0.
b2 = 19/8
b3 = 0.

c0 = a0 + a1 + a2 + a3
c1 = -b0 + a1 - b1 + 2*a2 - b2 + 3*a3 - b3
c2 = a1 - 2*b1 + 4*a2 - 4*b2 + 9*a3 - 6*b3
c3 = a1 - 3*b1 + 8*a2 - 12*b2 + 27*a3 - 27*b3
c4 = a1 - 4*b1 + 16*a2 - 32*b2 + 81*a3 - 108*b3

print('c0 = {}'.format(c0))
print('c1 = {}'.format(c1))
print('c2 = {}'.format(c2))
print('c3 = {}'.format(c3))
print('c4 = {}'.format(c4))

coeff = [a3,a2,a1,a0]
roots = np.roots(coeff)
print('Roots: {}'.format(roots))
```

Example output:

```
c0 = 0.0
c1 = 0.0
c2 = 0.0
c3 = 0.0
c4 = 8.5
Roots: [ 1.000+0.j          -0.625+0.59947894j -0.625-0.59947894j]
```

2.a) Consider the following numerical methods for solving an IVP $y'(x) = f(x, y(x))$ $y(x_0) = y_0$. Note $\Delta x = h$.

Method 1:

$$y_{n+1} + y_n - 2y_{n-1} = \frac{h}{4}\left(f(x_{n+1}, y_{n+1}) + 8f(x_n, y_n) + 3(x_{n-1}, y_{n-1})\right)$$

Theorem: A linear multistep method is defined by

$$\sum_{j=0}^{r} \alpha_j U^{n+j} = \Delta t \sum_{j=0}^{r} \beta_j f(t_{n+j}, U^{n+j})$$

The method is consistent if $c_0 = c_1 = 0$, and consistent of order at least $q \geq 1$ if and only if $c_0 = c_1 = \dots = c_q = 0$ with

$$c_0 = \sum_{j=0}^{r} \alpha_j \qquad c_k = \sum_{j=0}^{r} (j^k \alpha_j - k j^{k-1} \beta_j), \quad k = 1, \dots, q$$

For method 1:

$\alpha_0 = -2$, $\alpha_1 = 1$, $\alpha_2 = 1$ $\quad \beta_0 = \frac{3}{4}$, $\beta_1 = 2$, $\beta_2 = \frac{1}{4}$ $\quad r = 2$

$c_0 = \alpha_0 + \alpha_1 + \alpha_2 = -2 + 1 + 1 = 0$ ✓

$c_1 = (0)^1\alpha_0 - 1(0)^{1-1}\beta_0) + (1^1\alpha_1 - 1(1)^0\beta_1) + (2^1\alpha_2 - 1(2)^0\beta_2)$

$= -\beta_0 + \alpha_1 - \beta_1 + 2\alpha_2 - \beta_2 = -\frac{3}{4} + 1 - 2 + 2(1) - \frac{1}{4} = 0$ ✓

$c_2 = (0^2\alpha_0 - 2(0)^1\beta_0) + (1^2\alpha_1 - 2(1)^1\beta_1) + (2^2\alpha_2 - 2(2)^1\beta_2)$

$= -2(0)\beta_0 + \alpha_1 - 2\beta_1 + 4\alpha_2 - 4\beta_2 = \alpha_1 - 2\beta_1 + 4\alpha_2 - 4\beta_2$

$= 1 - 2(2) + 4(1) - 4\left(\frac{1}{4}\right) = 0$ ✓

$c_3 = (0^3\alpha_0 - 3(0)^2\beta_0) + (1^3\alpha_1 - 3(1)^2\beta_1) + (2^3\alpha_2 - 3(2)^2\beta_2)$

$= \alpha_1 - 3\beta_1 + 8\alpha_2 - 12\beta_2 = 1 - 6 + 8 - 3 = 0$ ✓

$c_4 = 0^4\alpha_0 - 4(0)^3\beta_0 + 1^4\alpha_1 - 4(1)^3\beta_1 + 2^4\alpha_2 - 4(2)^3\beta_2$

$= \alpha_1 - 4\beta_1 + 16\alpha_2 - 32\beta_2 = 1 - 8 + 16 - 8 = 1$ ×

∴ Consistent to order 3.

Theorem: A numerical method is ~~conv~~ convergent if and only if it is consistent and if it satisfies a root condition.

Root condition: $\rho(t)$ satisfies the root condition if all roots of ρ satisfy $|R| \leq 1$. If $|R| = 1$, the root must be simple (multiplicity $m = 1$).

2a.) Additionally:

Theorem: The method is convergent if and only if it is zero-stable

$$\rho(t) = \sum_{j=0}^{r} \alpha_j t^j$$

$$r=2 \rightarrow \rho(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2$$

$$-2 + t + t^2 = 0$$

Roots are 1 and -2, with $|-2| > 1$. Therefore the root condition is not satisfied for method 1 and the method is not convergent.

Method 2:

$$y_{n+1} - y_n = \frac{h}{3}\left(3f(x_n, y_n) - 2f(x_{n-1}, y_{n-1})\right)$$

$$\downarrow$$

$$y_{n+2} - y_{n+1} = \frac{h}{3}\left(3f(x_{n+1}, y_{n+1}) - 2f(x_{n+1}, y_{n+1})\right)$$

$$\alpha_0 = 0,\ \alpha_1 = -1,\ \alpha_2 = 1,\ \beta_0 = -\frac{2}{3},\ \beta_1 = 1,\ \beta_2 = 0$$

$$c_0 = 0 - 1 + 1 = 0 \checkmark$$

$$c_1 = 0\alpha_1 - \beta_0 + 2\alpha_2 - \beta_2 + \alpha_1 - \beta_1 = \frac{2}{3} + 2 - 1 - 1 = \frac{2}{3}$$

$\therefore$ Method is not consistent. This also means it is not convergent.

Method 3:

$$y_{n+1} + \frac{1}{4}y_n - \frac{1}{2}y_{n-1} - \frac{3}{4}y_{n-2} = \frac{h}{8}\left(19f(x_n, y_n) + 5f(x_{n-2}, y_{n-2})\right)$$

$$\downarrow$$

$$y_{n+3} + \frac{1}{4}y_{n+2} - \frac{1}{2}y_{n+1} - \frac{3}{4}y_n = \frac{h}{8}\left(19f(x_{n+2}, y_{n+2}) + 5f(x_n, y_n)\right)$$

$$\alpha_0 = -\frac{3}{4},\ \alpha_1 = -\frac{1}{2},\ \alpha_2 = \frac{1}{4},\ \alpha_3 = 1,\ \beta_0 = \frac{5}{8},\ \beta_1 = 0,\ \beta_2 = \frac{19}{8},\ \beta_3 = 0$$

$$c_0 = -\frac{3}{4} - \frac{1}{2} + \frac{1}{4} + 1 = 0 \checkmark$$

$$c_1 = -\beta_0 + \alpha_1 - \beta_1 + 2\alpha_2 - \beta_2 + 3\alpha_3 - \beta_3 = -\frac{5}{8} - \frac{1}{2} - 0 + \frac{1}{2} - \frac{19}{8} + 3 - 0 = 0 \checkmark$$

$$c_2 = \alpha_1 - 2\beta_1 + 4\alpha_2 - 4\beta_2 + 9\alpha_3 - 6\beta_3 = 0 \checkmark$$ ← (Computed using Python Script)

$$c_3 = \alpha_1 - 3\beta_1 + 8\alpha_2 - 12\beta_2 + 27\alpha_3 - 27\beta_3 = 0 \checkmark$$ ← (Computed using Python Script)

-See back of page 1

$$c_4 = \alpha_1 - 4\beta_1 + 16\alpha_2 - 32\beta_2 + 81\alpha_3 - 108\beta_3 = 8.5$$

2.a) $c_0 = c_1 = c_2 = c_3 = 0$

$c_4 \neq 0$

Therefore the method is consistent to order 3.

$$\rho(t) = \sum_{j=0}^{r} \alpha_j t^j$$

$$r=3 \Rightarrow \rho(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \alpha_3 t^3$$

$$-\frac{3}{4} - \frac{1}{2}t + \frac{1}{4}t^2 + t^3 = 0$$

(Found using very basic Python code) ← See back of page 1

Roots are $1, -\frac{5}{8} + \frac{\sqrt{23}}{8}i, -\frac{5}{8} - \frac{\sqrt{23}}{8}i$.

These roots satisfy the root condition $|R| \leq 1$, therefore the method is convergent because it is also consistent to order 3.

Method 4:

$$y_{n+1} - y_{n-1} = h\left(f(x_{n+1}, y^*_{n+1}) + f(x_{n-1}, y_{n-1})\right)$$

$$y^*_{n+1} - 3y_n + 2y_{n-1} = \frac{h}{2}\left(f(x_n, y_n) - 3f(x_{n-1}, y_{n-1})\right)$$

$\alpha_0 = -1, \alpha_1 = 0, \alpha_2 = 1 \quad r = 3$

$\sum_{j=0}^{r} \alpha_j = 0$ ✓

$\rho(t) = -1 + t^2$, roots are 1 and -1

Root condition is satisfied and $c_0 = 0$, however we can't say that it's convergent based on this alone because we have not proven that the method is consistent.

$$\phi_t(u(t), \ldots, u(t), t, 0) = \left(f(x_{n+1}, y^*_{n+1}) + f(x_{n-1}, y_{n-1})\right)$$

Consistent if and only if:

$$\phi_t = f(t, u(t)) \sum_{j=0}^{r} j\alpha_j \qquad \sum_{j=0}^{r} j\alpha_j = 0\alpha_0 + 1\alpha_1 + 2\alpha_2 = 2$$

It is not apparent that $\phi_t = 2f(t, u(t))$, so the method is likely <u>not</u> consistent, unless there are mathemagics that prove me wrong. This would mean that the method <u>is not</u> convergent.

2.b.) All work for part b) was completed in part a). In summary:
Method 1 is order of consistency 3.
Method 2 is order of consistency 0 (not consistent).
Method 3 is order of consistency 3.
Method 4 is not a multistep method because multistep methods only consist of evaluations of f at (x_{n+j}, y_{n+j}) where j is an integer, and method 4 evaluates it at (x_{n+1}, y^k_{n+1}).

c) The remainder of this homework was completed in Matlab, and the remaining writeup is attached.

2.c) The output of the attached Matlab code can be found below. The value of y at each of the positions of x was computed but is not shown. The error column denotes the L_2-error E_n. The results are consistent with the findings in part a). Method 3 had the smallest error at the smallest h value because it was the only convergent method of the three. Method 1 had the largest error as h became smaller because it was not convergent and the results can be interpreted as growing unbounded. Method 2 generally improved in accuracy slightly as h decreased, however the error values were much more consistent across step sizes because it was zeroith order consistent (inconsistent). In most cases with only a few exceptions, the error grew as the computation proceeded because the errors at each step accumulated.

Method 1

h = 0.2			h = 0.1			h = 0.01		
x	error	evaluations of f(x,y)	x	error	evaluations of f(x,y)	x	error	evaluations of f(x,y)
0.2	0	18	0.2	0.0265	20	0.2	1.0221	102
0.4	0.2268	33	0.4	0.135	39	0.4	1.94E+006	202
0.6	0.6347	48	0.6	0.9024	59	0.6	3.69E+012	302
0.8	2.0429	63	0.8	6.1563	79	0.8	7.02E+018	402
1	6.4894	78	1	42.0364	99	1	1.33E+025	502

Method 2

h = 0.2			h = 0.1			h = 0.01		
x	error	evaluations of f(x,y)	x	error	evaluations of f(x,y)	x	error	evaluations of f(x,y)
0.2	0	3	0.2	1.2737	4	0.2	1.1104	22
0.4	2.7354	4	0.4	1.1019	6	0.4	0.9078	42
0.6	1.089	5	0.6	0.795	8	0.6	0.5929	62
0.8	3.6827	6	0.8	0.5538	10	0.8	0.3652	82
1	3.3487	7	1	0.3842	12	1	0.2208	102

Method 3

h = 0.2			h = 0.1			h = 0.01		
x	error	evaluations of f(x,y)	x	error	evaluations of f(x,y)	x	error	evaluations of f(x,y)
0.2	2.24E-016	4	0.2	0	5	0.2	5.63E-005	23
0.4	0	5	0.4	0.2578	7	0.4	2.34E-005	43
0.6	0.9746	6	0.6	1.4975	9	0.6	7.17E-006	63
0.8	3.7505	7	0.8	8.0876	11	0.8	1.94E-006	83
1	15.7165	8	1	43.5066	13	1	4.90E-003	103

d?) See answer to part c).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This code is applicable to problem 1 of Homework 5
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all
close all

% Main routine (driver) for orbital ODE problem
% Specify name of user supplied function M-file with rhs of ode
odefun = 'orbitODE';
% Specify the method to be used
% Options are ExplicitEuler, RK1, RK4, HW2, and ODE45v4
method = 'ODE45v4';
% Specify if you want automatic time steps, and if so, the tolerance
% autostep is only applicable as long as ODE45v4 isn't selected
autostep = true;
TOL = 1.0*(10^-4);
global steps;
steps = 0;
% If automatic time steps are not to be used, specify the number of
 steps
NSTEP=5*(10^5);
% If ODE45v4 is to be used, state whether or not you want every step
 output
trace = 0;
% Specify the initial conditions
% Specify initial and final times
t0 = 0; tfinal = 17.1;
TSPAN = [t0,tfinal];
% Specify column vector of initial values
U0 = [0.994;0.0;0.0;-2.00158510637908252240537862224];
mu = 0.012277471;

% Build the Butcher array based on the method selected
% Only applicable as long as ODE45v4 isn't selected
% This is all probably bad programming practice, but it works and was
 easy.
if (strcmp(method, 'ODE45v4') ~= 1)
    if strcmp(method, 'RK4')
        A = [0 0 0 0; 0.5 0 0 0; 0 0.5 0 0; 0 0 1 0];
        b = [1/6;1/3;1/3;1/6];
        c = [0;0.5;0.5;1];
    elseif (strcmp(method,'ExplicitEuler') || strcmp(method,'RK1'))
        A = 0;
        b = 1;
        c = 0;
    elseif strcmp(method,'HW2')
        A = [0 0; 1 0];
        b = [0.5;0.5];
        c = [0;1];
    end
```

```
    % Series of if statements that will return the order of the RK
method used
    if round(sum(b),4) == 1.0000
        qorder = 1;
        if round(b.'*c,4) == 0.5000
            qorder = 2;
            if round(b.'*(c.^2),4) == round(1/3,4) && ...
               round(sum(b'.*sum(A'.*c)),4) == round(1/6,4)
                qorder = 3;
                if round(b.'*(c.^3),4) == 1/4 && ...
                   round(sum(b'.*c'.*sum(A'.*c)),4) == 1/8 && ...
                   round(sum(b'.*sum(A'.*c.^2)),4) == round(1/12,4)
&& ...
                   round(sum(A.*b)*sum(A'.*c)',4) == round(1/24,4)
                    qorder = 4;
                end
            end
        end
    end
    % Call the solver, based on if you want automatic time step or not
    if autostep == false
        [t,U] = eulerw17d(odefun,TSPAN,U0,NSTEP,method,A,b,c,mu);
    else
        [t,U] = RKw17sc(odefun,TSPAN,U0,TOL,A,b,c,qorder,mu);
    end
else
    [t,U] = ode45v4(odefun,t0,tfinal,U0,TOL,trace);
end

% plot numerical solution;
figure;
hold off;
% U in ODE45v4 and the other methods annoyingly are transposed
if strcmp(method, 'ODE45v4')
    plot(U(:,1),U(:,2),'b');
else
    plot(U(1,:),U(2,:),'b');
end
xlabel('u_1')
ylabel('u_2')
titlestr = sprintf(['Method = ' method ...
    ', Tolerance = %2.0e, Steps = %i'],TOL,steps);
title(titlestr)
```

Published with MATLAB® R2016b

```
function [tout, yout] = ode45v4(ypfun, t0, tfinal, y0, tol, trace)
%ODE45 Solve differential equations, higher order method.
% ODE45 integrates a system of ordinary differential equations using
% 4th and 5th order Runge-Kutta formulas.
% [T,Y] = ODE45V4('yprime', T0, Tfinal, Y0) integrates the system of
% ordinary differential equations described by the M-file YPRIME.M,
% over the interval T0 to Tfinal, with initial conditions Y0.
% [T, Y] = ODE45V4(F, T0, Tfinal, Y0, TOL, 1) uses tolerance TOL
% and displays status while the integration proceeds.
%
% INPUT:
% ypfun  - String containing name of user-supplied problem
 description.
%         Call: yprime = fun(t,y) where F = 'fun'.
%         t      - Time (scalar).
%         y      - Solution column-vector.
%         yprime - Returned derivative column-vector; yprime(i) =
 dy(i)/dt.
% t0    - Initial value of t.
% tfinal- Final value of t.
% y0    - Initial value column-vector.
% tol   - The desired accuracy. (Default: tol = 1.e-6).
% trace - If nonzero, each step is printed. (Default: trace = 0).
%
% OUTPUT:
% tout  - Returned integration time points (column-vector).
% yout  - Returned solution, one solution column-vector per tout-
value.
%
% The result can be displayed by: plot(tout, yout).
%
% See also ODE23, ODEDEMO.

% C.B. Moler, 3-25-87, 8-26-91, 9-08-92.
% Copyright (c) 1984-94 by The MathWorks, Inc.

% The Fehlberg coefficients:
alpha = [1/4  3/8  12/13  1  1/2]';
beta  = [ [    1      0      0     0      0    0]/4
          [    3      9      0     0      0    0]/32
          [ 1932  -7200   7296     0      0    0]/2197
          [ 8341 -32832  29440  -845      0    0]/4104
          [-6080  41040 -28352  9295  -5643    0]/20520 ]';
gamma = [ [902880  0  3953664  3855735  -1371249  277020]/7618050
          [ -2090  0    22528    21970    -15048  -27360]/752400 ]';
pow = 1/5;
if nargin < 5, tol = 1.e-6; end
if nargin < 6, trace = 0; end

% Initialization
t = t0;
hmax = (tfinal - t)/16;
```

```
h = hmax/8;
y = y0(:);
f = zeros(length(y),6);
chunk = 128;
tout = zeros(chunk,1);
yout = zeros(chunk,length(y));
k = 1;
tout(k) = t;
yout(k,:) = y.';

if trace
   clc, t, h, y
end

% The main loop

while (t < tfinal) & (t + h > t)
   if t + h > tfinal, h = tfinal - t; end

   % Compute the slopes
   temp = feval(ypfun,t,y);
   f(:,1) = temp(:);
   for j = 1:5
      temp = feval(ypfun, t+alpha(j)*h, y+h*f*beta(:,j));
      f(:,j+1) = temp(:);
   end

   % Estimate the error and the acceptable error
   delta = norm(h*f*gamma(:,2),'inf');
   tau = tol*max(norm(y,'inf'),1.0);

   % Update the solution only if the error is acceptable
   if delta <= tau
      t = t + h;
      y = y + h*f*gamma(:,1);
      k = k+1;
      if k > length(tout)
         tout = [tout; zeros(chunk,1)];
         yout = [yout; zeros(chunk,length(y))];
      end
      tout(k) = t;
      yout(k,:) = y.';
   end
   if trace
      home, t, h, y
   end

   % Update the step size
   if delta ~= 0.0
      h = min(hmax, 0.8*h*(tau/delta)^pow);
   end
end

if (t < tfinal)
```

```
    disp('Singularity likely.')
    t
end

tout = tout(1:k);
yout = yout(1:k,:);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function ypfun = orbitODE(t,y)
% Implements the ODE from Homework 2 Problem 3 as a first order system
global steps;
steps = steps + 1;
mu = 0.012277471;
muhat = 1.0 - mu;
D1 = ((y(1) + mu)^2 + y(2)^2)^1.5;
D2 = ((y(1) - muhat)^2 + y(2)^2)^1.5;
ypfun = zeros(size(y));
ypfun(1) = y(3);
ypfun(2) = y(4);
ypfun(3) = y(1) + 2*y(4) - muhat*((y(1)+mu)/D1) - mu*((y(1)-muhat)/
D2);
ypfun(4) = y(2) - 2*y(3) - (muhat*(y(2)/D1)) - (mu*(y(2)/D2));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Published with MATLAB® R2016b

```
close all
clear all

% driver for numerical experiments in homework
%Need to specify h before running it.
%Change only parameter p when using different methods.
% Input parameters:
a = 0; % left side of integration interval
eta = [.5; -3]; % initial value (column vector)
p = 2; % parameter p of method. p=0: single step, p>0: multistep
h = 0.2
% end of input

global count ;%counter for evaluations of function f(x,y)
count = 0;

m = max(size(eta)); % number of equations
y = zeros(m,p+1); f=y;
for j=1:p                          %initialize matrices y and f.
  y(:,j+1) = exsolhw3(a+j*h); %The (j+1)-th column of y contains y(a
+j*h)
  f(:,j+1) = fun(a+j*h,y(:,j+1)); % compute f(a+j*h,y(a+j*h))
end
y(:,1) = eta; f(:,1) = fun(a,eta);

nstep = .2/h;
xn = a+p*h;
for k=1:5 %solve ode. Stop after nstep steps  to compute error
  for n=1:nstep
    [y,f] = method(xn,h,y,f); %compute solution at x+h
    xn = xn+h;
  end
  x(k) = xn-p*h; %x-value where error is computed.
  err(k) = norm(y(:,1) - exsolhw3(x(k))) %compute error
  c(k) = count; %function evaluations needed so far
end
z=[x' err' c'];
disp('      x       error   evaluations of f(x,y)')
disp(z)
```

Published with MATLAB® R2016b

```
function [y,f]=method(x,h,y,f)
% [y,f]=method(x,h,y,f) computes one step of a general linear
 multistep method.
% y = matrix whose (j+1)st column is y_{n-p+j}, j=0,...,p
% f = matrix whose (j+1)st column is f(x_{n-p+j},y_{n-p+j}), j=0,...,p
% On output, the (J+1)st columns of y and f are y_{n+1-p+j} and
% f(x{n+1-p+j},y_{n+1-p+j}), respectively.
% Here x_{n-p+j} = x + (j-p)*h.
% To use a different method, change the column vectors alpha and beta.

%Specify the parameters alpha and beta in column vectors.
%Note that since MATLAB does not allow for 0 indices, you must set
%alpha(j+1) = alpha_j, beta(j+1) = beta_j, j=0,...,p+1,

alpha=[-3/4; -1/2; 1/4; 1];      % Example: y_{n+1}-y_{n} =
beta=(1/8)*[5; 0; 19; 0]; % (h/3)*[3*f(x_{n},y_{n}) -
 2*f(x_{n-1},y_{n-1})]


p = max(size(alpha)) - 2;
a1 = -alpha(1:p+1)/alpha(p+2);
b1 = h*beta(1:p+1)/alpha(p+2);

tmp = y*a1+ f*b1; %Computes sum_{j=0}^p  [-alpha_j y_{n-p+j) +
                  %              +h*beta_j*f(x_{n-p+j},y_{n-p+j})]/
alpha(p+2)

if (beta(p+2) == 0) %method is explicit.
  y1 = tmp;

else  % method implicit. Use fixed point iteration to solve the
 equation
     % y1 = tmp + h*beta(p+2)*f(x+h,y1)/alpha(p+2), with tmp as
 above.

 tol = 1.e-5; itmax = 100; %specify tolerance and maximum # of
 iterations
 bp2 = h*beta(p+2)/alpha(p+2);xh = x+h;%auxiliary variables
 y0 = y(:,p+1); %starting vector for iteration
 t1 = 2*tol; t2=0;iter = 0;%initialize parameters for stopping
 criterion.

 while ((t1 > tol*t2) & (iter < itmax)) %iteration loop
   y1 = tmp + bp2*fun(xh,y0);
   t1 = norm(y1-y0); t2 = norm(y1) + norm(y0); %evaluate stopping
 criterion
   iter = iter+1;
   y0 = y1;
 end

 if (iter == itmax) %print warning if iteration did not converge.
   disp('  ');
```

```
    disp('Slow or no convergence in fixed point iteration.')
    disp('  x    rel. err.    tolerance     iterations ')
    disp([x    t1/t2     tol    iter ])
  end
end

y(:,1:p) = y(:,2:p+1);y(:,p+1)=y1; %update y
f(:,1:p) = f(:,2:p+1);f(:,p+1)=fun(x+h,y1); %update f
```

Published with MATLAB® R2016b