ODE Lab: Creating your own ODE solver in MATLAB

Table of Contents

In this lab, you will write your own ODE solver for the Improved Euler method (also known as the Heun method), and compare its results to those of `ode45`.

You will also learn how to write a function in a separate m-file and execute it.

Opening the m-file lab3.m in the MATLAB editor, step through each part using cell mode to see the results. Compare the output with the PDF, which was generated from this m-file.

There are six (6) exercises in this lab that are to be handed in on the due date. Write your solutions in the template, including appropriate descriptions in each step. Save the .m files and submit them online on Quercus.

Student Information

Student Name: Hshmat Sahak

Student Number: 1005903710

Creating new functions using m-files.

Create a new function in a separate m-file:

Specifics: Create a text file with the file name f.m with the following lines of code (text):

```
function y = f(a,b,c)
y = a+b+c;
```

Now MATLAB can call the new function f (which simply accepts 3 numbers and adds them together). To see how this works, type the following in the matlab command window: sum = f(1,2,3)

Exercise 1

Objective: Write your own ODE solver (using the Heun/Improved Euler Method).

Details: This m-file should be a function which accepts as variables (t0,tN,y0,h), where t0 and tN are the start and end points of the interval on which to solve the ODE, y0 is the initial condition of the ODE, and h is the stepsize. You may also want to pass the function into the ODE the way `ode45` does (check lab 2).

Note: you will need to use a loop to do this exercise. You will also need to recall the Heun/Improved Euler algorithm learned in lectures.

Exercise 2

Objective: Compare Heun with `ode45`.

Specifics: For the following initial-value problems (from lab 2, exercises 1, 4-6), approximate the solutions with your function from exercise 1 (Improved Euler Method). Plot the graphs of your Improved Euler Approximation with the `ode45` approximation.

(a) `y' = y tan t + sin t, y(0) = -1/2` from `t = 0` to `t = pi`

(b) `y' = 1 / y^2 , y(1) = 1` from `t=1` to `t=10`

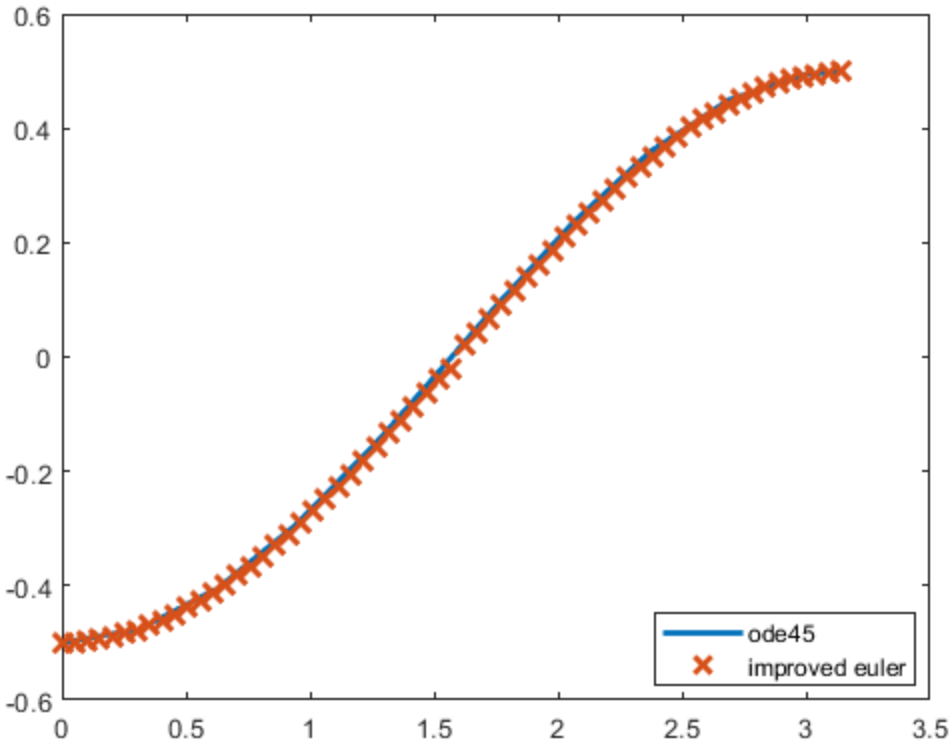
(c) `y' = 1 - t y / 2, y(0) = -1` from `t=0` to `t=10`

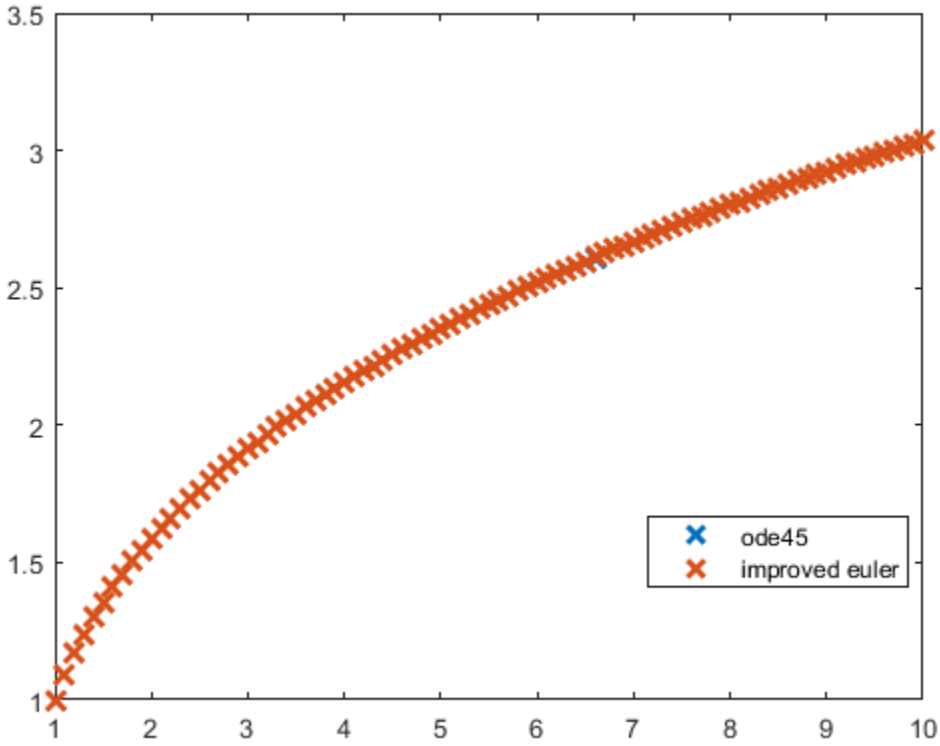
(d) `y' = y^3 - t^2, y(0) = 1` from `t=0` to `t=1`

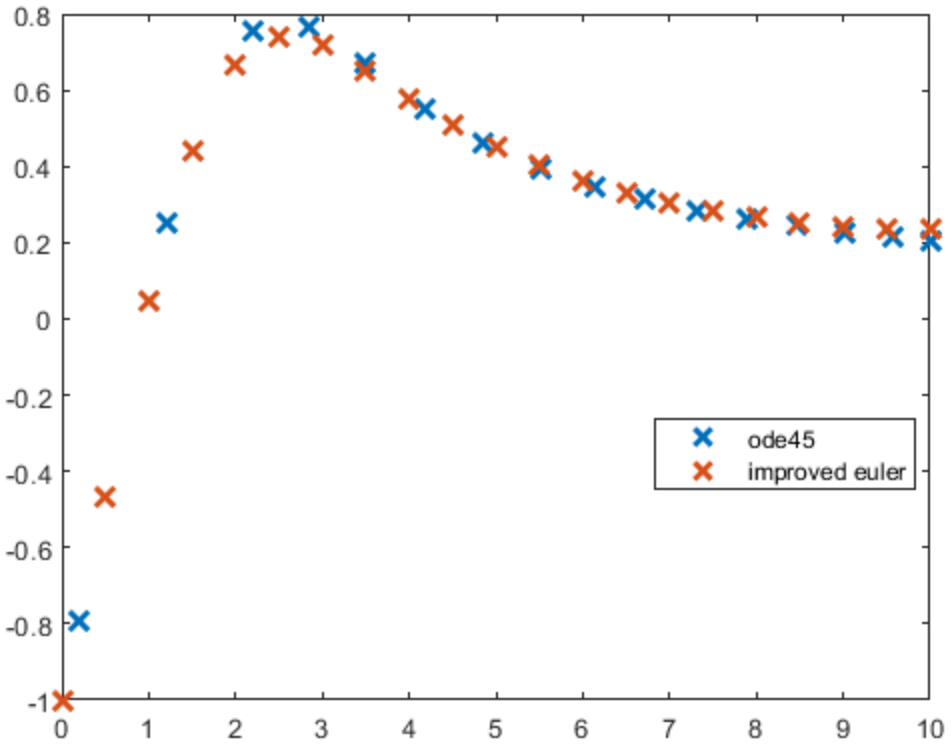
Comment on any major differences, or the lack thereof. You do not need to reproduce all the code here. Simply make note of any differences for each of the four IVPs.

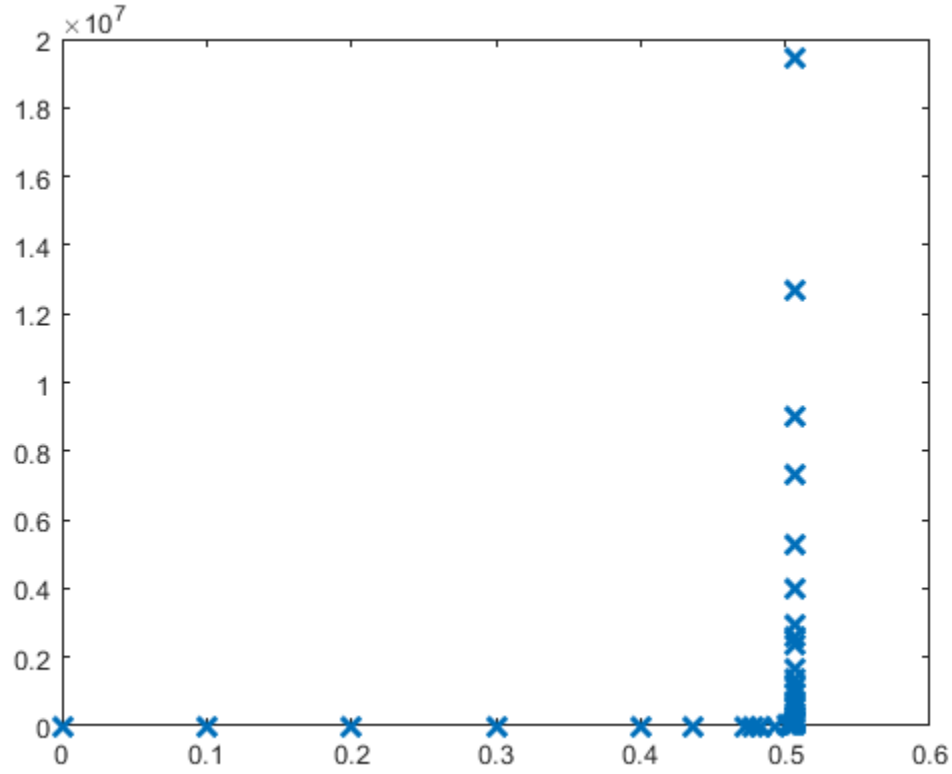
```
f1 = @(t,y) y*tan(t) + sin(t);
soln1_45 = ode45(f1, [0, pi], -1/2);
soln1_ie = improved_euler(f1, 0, pi, -1/2, 0.05);
f2 = @(t,y) 1/y^2;
soln2_45 = ode45(f2, [1,10], 1);
soln2_ie = improved_euler(f2, 1, 10, 1, 0.1);
f3 = @(t,y) 1-t*y/2;
soln3_45 = ode45(f3, [0,10], -1);
soln3_ie = improved_euler(f3, 0, 10, -1, 0.5);
f4 = @(t,y) y^3-t^2;
soln4_45 = ode45(f4, [0,1], 1);
soln4_ie = improved_euler(f4, 0, 1, 1, 0.1);

% For the first one, the two solutions are similar. The estimated
 solution
% seems to have a "gap", a sort of jump, near the middle(pi/2). This
 could
% be credited to the tangent term in the expression for y' going to
% infinity as t->pi/2. The exact solution only has a cos term so it is
% continuous everywhere but here, y->0 and tant->inifinity(plus or
 minus),
% so we get unexpected behaviour
plot(soln1_45.x, soln1_45.y, 'LineWidth', 2);
hold on;
plot(soln1_ie.x, soln1_ie.y, 'x', 'MarkerSize', 10, 'LineWidth', 2);
legend('ode45', 'improved euler', 'Location', 'Best');
snapnow;
clf;

% The exact and analytic appear nearly identical
plot(soln2_45.x, soln2_45.y, 'x', 'MarkerSize', 10, 'LineWidth', 2)
hold on;
plot(soln2_ie.x, soln2_ie.y, 'x', 'MarkerSize', 10, 'LineWidth', 2);
```

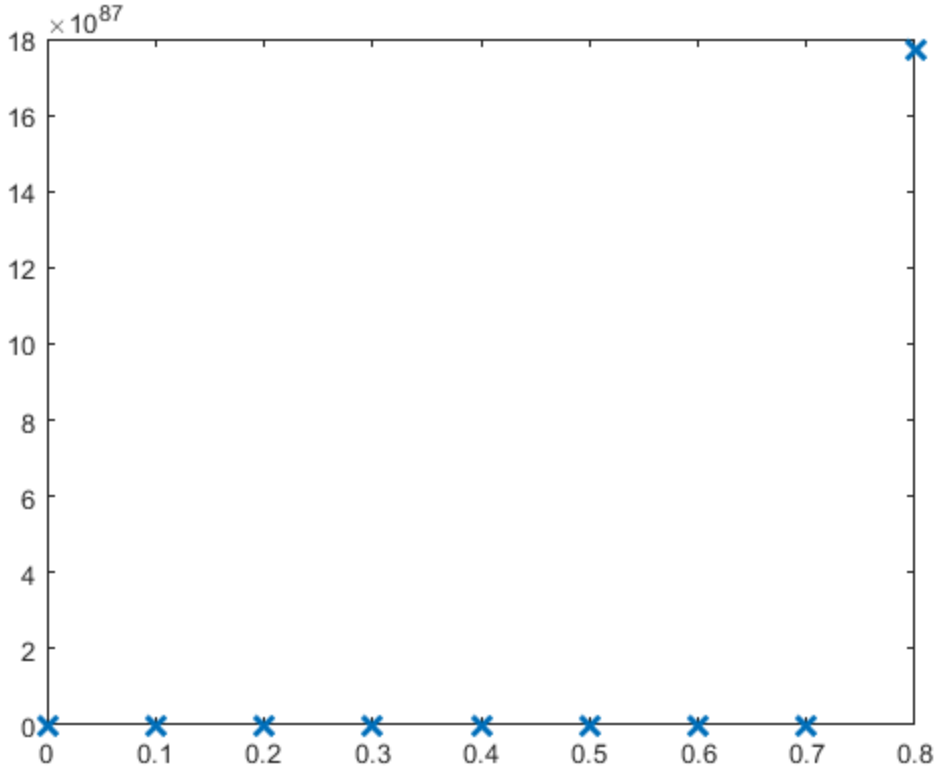
```
legend('ode45', 'improved euler', 'Location', 'Best');
snapnow;
clf;

% Again, the solutions seem to agree. No obvious differences
plot(soln3_45.x, soln3_45.y, 'x', 'MarkerSize', 10, 'LineWidth', 2)
hold on;
plot(soln3_ie.x, soln3_ie.y, 'x', 'MarkerSize', 10, 'LineWidth', 2);
legend('ode45', 'improved euler', 'Location', 'Best');
snapnow;
clf;

% Both the improved euler and ode45 solutions shoot off to inifnity as
 we
% approach our upper limit of integration. Given the nature of how
 these
% solvers work, this is expected. However, the improved euler goes to
% infinity much faster(it switches from about 0 to a large number in
 one
% step) while the ode45 solution progresses to infinity with shorter
 and
% shorter time steps. This is because we have not implemented an
 adaptive
% time step
plot(soln4_45.x, soln4_45.y, 'x', 'MarkerSize', 10, 'LineWidth', 2)
snapnow;
clf;
plot(soln4_ie.x, soln4_ie.y, 'x', 'MarkerSize', 10, 'LineWidth', 2);
snapnow;
clf;
```


0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
0.5
1
1.5
2
2.5
3
3.5
ode45
improved euler


3.5
3
2.5
2
1.5
1
1
2
3
4
5
6
7
8
9
10
ode45
improved euler


0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6
7
8
9
10
ode45
improved euler


×10^7
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
0.1
0.2
0.3
0.4
0.5
0.6


×10^87
18
16
14
12
10
8
6
4
2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

Exercise 3

Objective: Use Euler's method and verify an estimate for the global error.

Details:

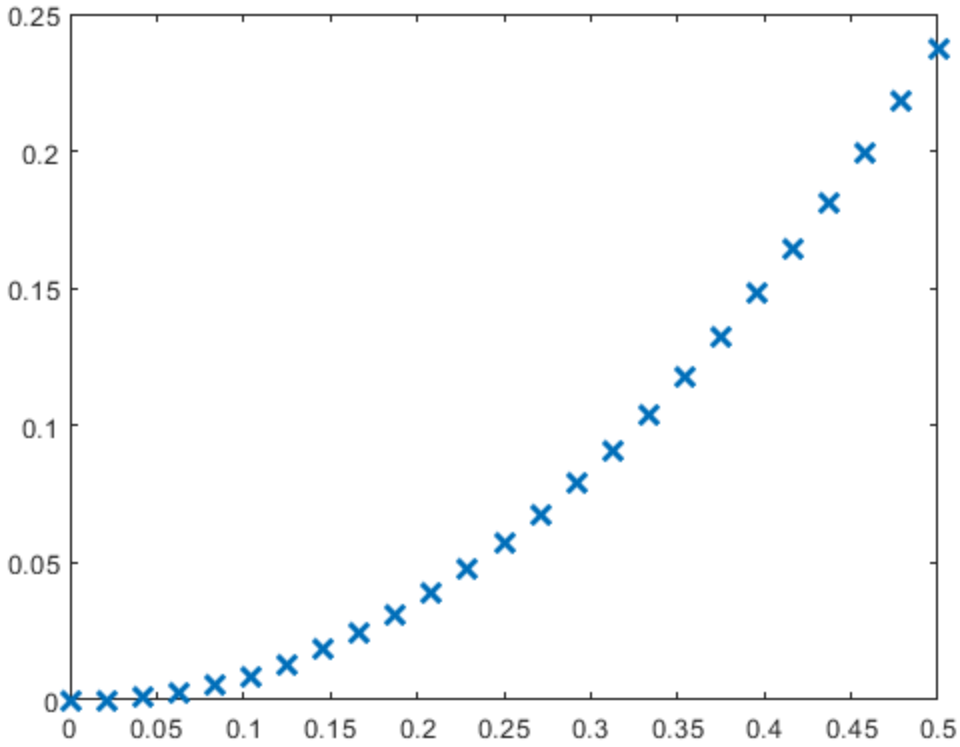
(a) Use Euler's method (you can use euler.m from iode) to solve the IVP `y' = 2 t sqrt( 1 - y^2 ) , y(0) = 0`

from `t=0` to `t=0.5`.

```
   y0=0;
   f = @(t, y) 2*t*sqrt(1-y^2);
   xvals = linspace(0,0.5,25);
   y = euler(f, y0, xvals);
   plot(xvals, y, 'x', 'MarkerSize', 10, 'LineWidth', 2);

% (b) Calculate the solution of the IVP and evaluate it at |t=0.5|.
%   General solution: y = sin(t^2) + sin(C)
%   Particular solution: y = sin(t^2)
%   y(0.5) = sin(0.25) ~= 0.247403959
    y_particular = sin(0.5^2);
    fprintf('y(0.5) = %g\n', y_particular);

% (c) Read the attached derivation of an estimate of the global error
 for
%     Euler's method. Type out the resulting bound for En here in
%     a comment. Define each variable.
%   E_n <= ((1+M)dt/2)(e^(M*dt*n)-1), where:
%   E_n: error at nth time step
%   M = max (f, delf/delx, delf/dely) over the interval (0,0.5)
%   dt = time step
%   n = how many steps you have gone(each with time step dt)

%   maximum f = max value of 2t*sqrt(1-y^2) <= 2*0.5*=1
%   maximum f <= 1
%   delf/delx = 2*sqrt(1-y^2)
%   maximum delf/delt <= 2*1 = 2
%   delf/dely = -2*t*y/(sqrt(1-y^2))
%   maximum delf/dely <= 0
%   M = 2

%   So, E_n <= 1.5*dt*(e^(3*dt*n)-1)

% (d) Compute the error estimate for |t=0.5| and compare with the
 actual
% error.
    M = 2;
    dt = 0.01;

    t0 = 0;
    tf = 0.5;
    t = t0:dt:tf;
```

```
    f_exact = @(t) sin(t^2);
    actual_val = f_exact(0.5);
    euler_val = y(length(y));
    act_error = abs(actual_val - euler_val);
    fprintf('Actual Error: %g\n', act_error);

    n = 0.5/dt;
    err_func = @(M, n, dt)((1 + M) * dt * (exp(M * dt * n) - 1)) / 2;
    computed_err = err_func(M, n, dt);
    fprintf('Computed error: %g\n', computed_err);
    % Actual error is smaller

% (e) Change the time step and compare the new error estimate with the
% actual error. Comment on how it confirms the order of Euler's
 method.
%   We will take half the time step used above.
    M = 2;
    dt = 0.001;

    t0 = 0;
    tf = 0.5;
    t = t0:dt:tf;
    f_exact = @(t) sin(t^2);
    actual_val = f_exact(0.5);
    euler_val = y(length(y));
    act_error = abs(actual_val - euler_val);
    fprintf('Actual Error: %g\n', act_error);

    n = 0.5/dt;
    err_func = @(M, n, dt)((1 + M) * dt * (exp(M * dt * n) - 1)) / 2;
    computed_err = err_func(M, n, dt);
    fprintf('Computed error: %g\n', computed_err);
% we see that when dt is reduced by a factor of 10(0.0257742 vs
 0.00257742), the global error is
% reduced also by a factor of 10. This is characteristic of first
 order
% methods; we conclude that euler's method is first order
```

```
y(0.5) = 0.247404
Actual Error: 0.00987983
Computed error: 0.0257742
Actual Error: 0.00987983
Computed error: 0.00257742
```



Adaptive Step Size

As mentioned in lab 2, the step size in `ode45` is adapted to a specific error tolerance.

The idea of adaptive step size is to change the step size `h` to a smaller number whenever the derivative of the solution changes quickly. This is done by evaluating f(t,y) and checking how it changes from one iteration to the next.

Exercise 4

Objective: Create an Adaptive Euler method, with an adaptive step size `h`.

Details: Create an m-file which accepts the variables `(t0,tN,y0,h)`, as in exercise 1, where `h` is an initial step size. You may also want to pass the function into the ODE the way `ode45` does.

Create an implementation of Euler's method by modifying your solution to exercise 1. Change it to include the following:

(a) On each timestep, make two estimates of the value of the solution at the end of the timestep: `Y` from one Euler step of size `h` and `Z` from two successive Euler steps of size `h/2`. The difference in these two values is an estimate for the error.

(b) Let `tol=1e-8` and `D=Z-Y`. If `abs(D)<tol`, declare the step to be successful and set the new solution value to be `Z+D`. This value has local error `O(h^3)`. If `abs(D)>=tol`, reject this step and repeat it with a new step size, from (c).

(c) Update the step size as `h = 0.9*h*min(max(tol/abs(D),0.3),2)`.

Comment on what the formula for updating the step size is attempting to achieve. tol/abs(D)>2: Doing good-> increase step size, is optimistic If 0.3 < tol/abs(D) < 2: We are not doing too good otherwise the ratio would be larger. But we are also not doing too bad. So decrease, but not by much. Specifically become 0.9*0.3= 0.27h If tol/abs(D) < 0.3: Doing bad with current time step, decrease time step proportional to how bad we are doing(ie the worse we do, the more we decrease)

Exercise 5

Objective: Compare Euler to your Adaptive Euler method.

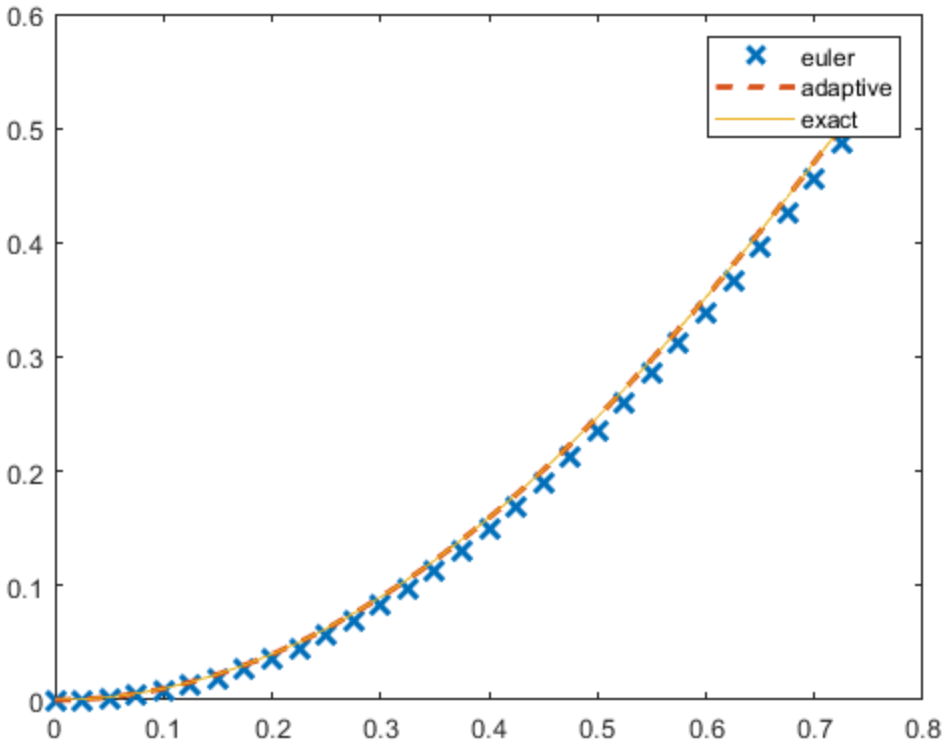
Details: Consider the IVP from exercise 3.

(a) Use Euler method to approximate the solution from `t=0` to `t=0.75` with `h=0.025`.

(b) Use your Adaptive Euler method to approximate the solution from `t=0` to `t=0.75` with initial `h=0.025`.

(c) Plot both approximations together with the exact solution.

```
% Let's first recall the function:
% y' = 2 t sqrt( 1 - y^2 )  ,  y(0) = 0
f = @(t,y) 2*t*sqrt(1-y^2);

x_euler = 0:0.025:0.75;
y_euler = euler(f, 0, 0:0.025:0.75);

x_exact = 0:0.025:0.75;
y_exact = sin(x_exact.^2);

adaptive_soln = adaptive_euler(f, 0, 0.75, 0, 0.025);
plot(x_euler, y_euler, 'x', 'MarkerSize', 10, 'LineWidth', 2)
hold on;
plot(adaptive_soln.x, adaptive_soln.y, '--', 'MarkerSize',
 10, 'LineWidth', 2);
hold on;
plot(x_exact, y_exact, 'LineWidth', 0.1);
legend('euler','adaptive','exact');
```



Exercise 6

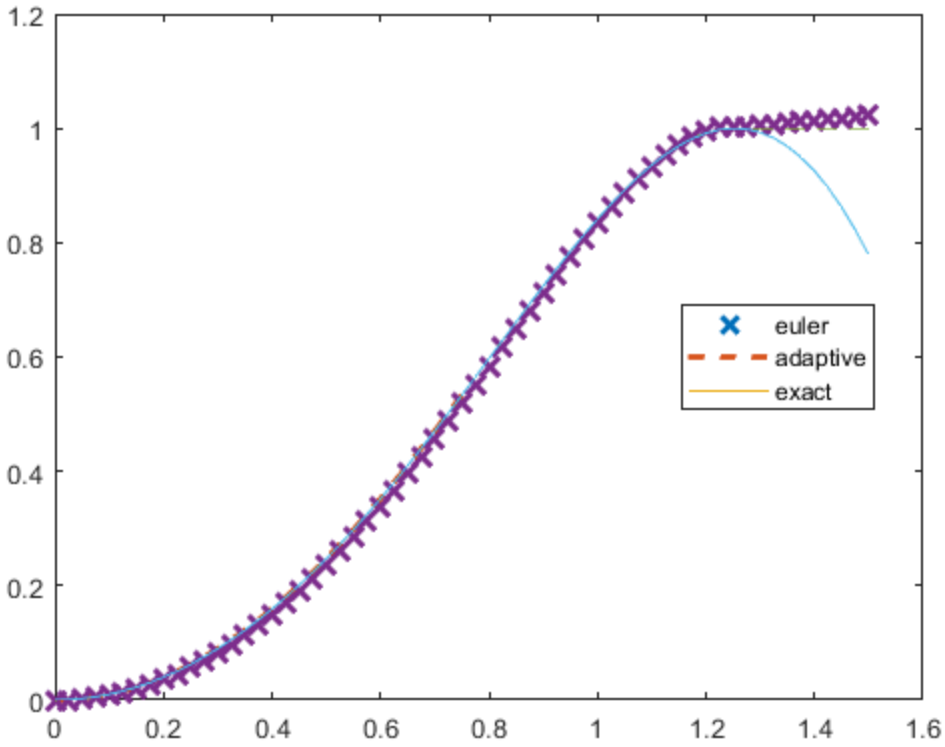
Objective: Problems with Numerical Methods.

Details: Consider the IVP from exercise 3 (and 5).

(a) From the two approximations calculated in exercise 5, which one is closer to the actual solution (done in 3.b)? Explain why. The adaptive euler method is the more accurate method. This is because the time step is chosen such that we minimize the expected error in each step. If the function increases rapidly starting from the beginning of any time step, the normal eulers method won't adjust accordingly as it uses only the slope at the beginning of the time step. This error would then propagate. However, the adaptive euler would detect a higher slope at midpoint, calculate a larger value, and give a large difference. This gives the extra feature that whenever the function varies considerably between the time step, we move through that interval much more slowly, in more time steps, so that we minimize overall error.

```
% (b) Plot the exact solution (from exercise 3.b), the Euler's
% approximation (from exercise 3.a) and the adaptive Euler's
 approximation
% (from exercise 5) from |t=0| to |t=1.5|.
f = @(t,y) 2*t*sqrt(1-y^2);
x_euler = 0:0.025:1.5;
y_euler = euler(f, 0, 0:0.025:1.5);

x_exact = 0:0.025:1.5;
y_exact = sin(x_exact.^2);
```

```
adaptive_soln = adaptive_euler(f, 0, 1.5, 0, 0.025);

plot(x_euler, y_euler, 'x', 'MarkerSize', 10, 'LineWidth', 2)
hold on;
plot(adaptive_soln.x, adaptive_soln.y, '--', 'MarkerSize',
 10, 'LineWidth', 0.1);
hold on;
plot(x_exact, y_exact, 'LineWidth', 0.1);
legend('euler','adaptive','exact', 'Location', 'Best');
% (c) Notice how the exact solution and the approximations become very
% different. Why is that? Write your answer as a comment.
% First note that if y=1, then y'=0, so if in our approximation we
 ever
% approximate that y obtains a value of 1, it will stay there
 forever(straight line). This
% doesnt do everything to explain why the approximations look like
% horizontal lines at y=1.

% Important is to notice that where the approximation and exact
 solutions
% deviate, the function has a local maximum, it is concave down
 everywhere
% in some interval containing the point (t0,1), where sin(t0^2) = 1.
 So, in
% our approximation; as y->1-, our slope approches 0+ (decreases to
 0), and the
% tangent line lies above the curve. Thus, whether we use our normal
 euler or
% adaptive euler, at some point, y>1. Note that at the time step that
 gives
% this result, the slope at the initial time step is so small that is
 it
% essentially the same as the slope at the middle of the time step, so
 the
% adaptive solution allows it(as y->1, y'->0).

% Once we reach a point where y>1, we get a complex number value for
 the
% slope. This causes unexpected behaviour seen in  figures for
 approximate
% solutions
```



Published with MATLAB® R2020a