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: Emma Seabrook

Student Number: 1005834563

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`

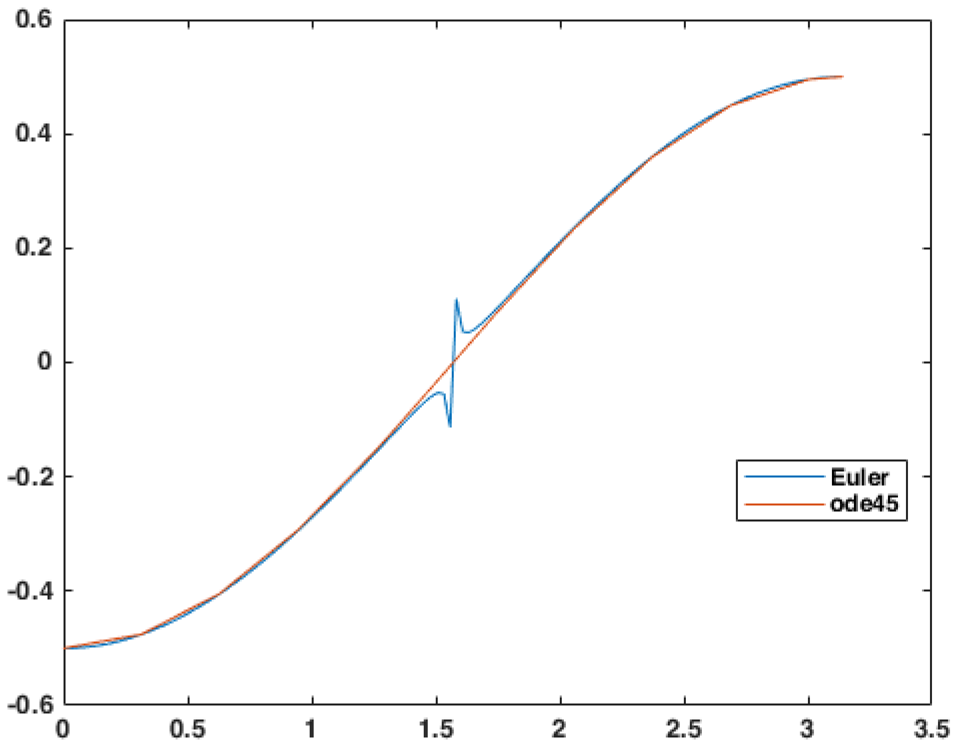
(d) `y' = y^3 - t^2, y(0) = 1` from `t=0` to `t=1`

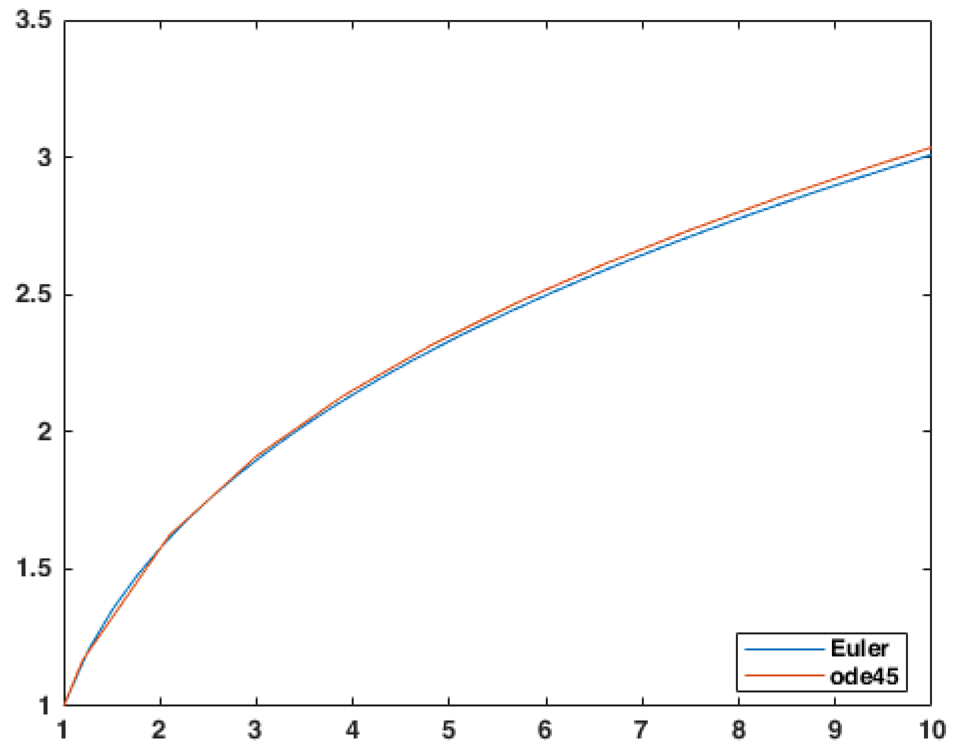
Comment on any major differences, or the lack thereof. You do not need to reproduce all the code here. Simply make note of any differences for each of the four IVPs.

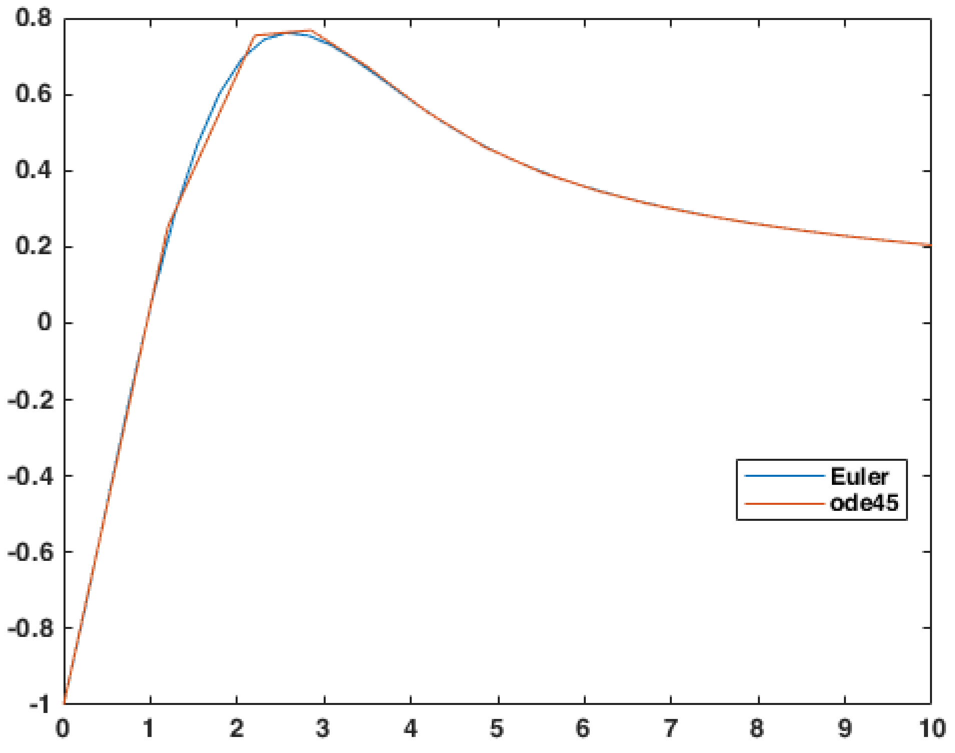
```
y1 = @(t,y) y*tan(t) + sin(t);
y2 = @(t,y)  1 / y^2;
y3 = @(t,y) 1 - t*y / 2;
y4 = @(t,y) y.^3 - t.^2;

y1_0 = -1/2;
t1_0 = 0;
t1_x = pi;

y2_0 = 1;
t2_0 = 1;
t2_x = 10;

y3_0 = -1;
t3_0 = 0;
t3_x = 10;

y4_0 = 1;
t4_0 = 0;
t4_x = 1;

figure(1);
[x_e_1,y_e_1] = imEuler(y1,t1_0,t1_x,y1_0,0.025);
soln1 = ode45(y1,[t1_0,t1_x], y1_0);
plot(x_e_1, y_e_1, soln1.x, soln1.y);
legend('Euler', 'ode45', 'Location','Best');
```

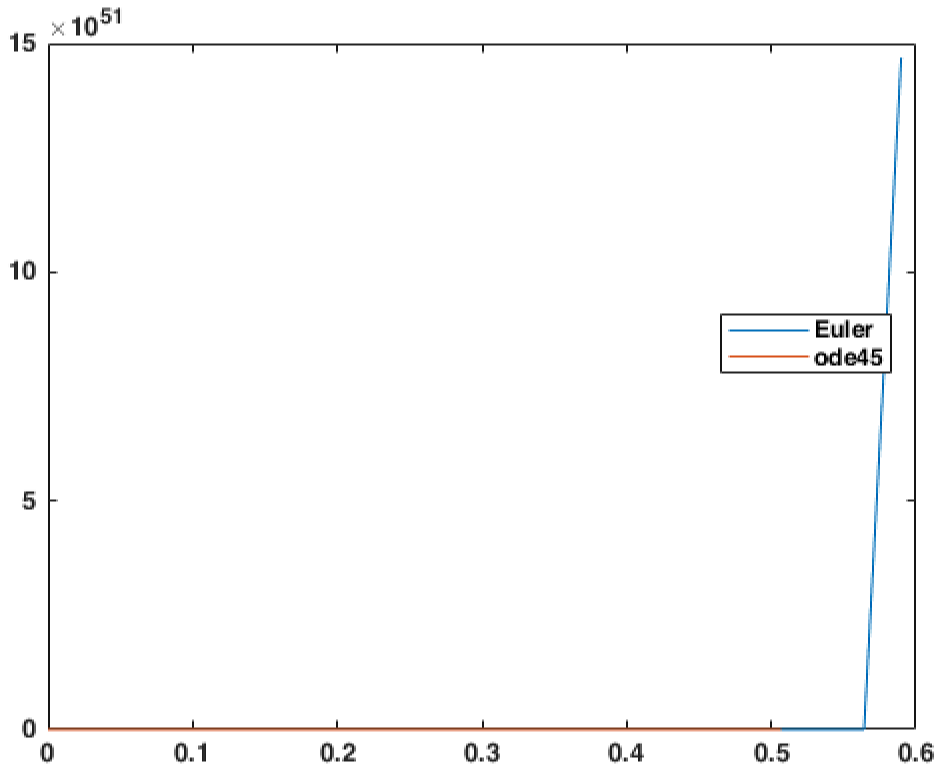
```
figure(2);
[x_e_2,y_e_2] = imEuler(y2,t2_0,t2_x,y2_0,0.25);
soln2 = ode45(y2,[t2_0,t2_x], y2_0);
plot(x_e_2, y_e_2, soln2.x, soln2.y);
legend('Euler', 'ode45', 'Location','Best');

figure(3);
[x_e_3,y_e_3] = imEuler(y3,t3_0,t3_x,y3_0,0.25);
soln3 = ode45(y3,[t3_0,t3_x], y3_0);
plot(x_e_3, y_e_3, soln3.x, soln3.y);
legend('Euler', 'ode45', 'Location','Best');

figure(4)
[x_e_4,y_e_4] = imEuler(y4,t4_0,t4_x,y4_0,0.025);
soln4 = ode45(y4,[t4_0,t4_x], y4_0);
plot(x_e_4, y_e_4, soln4.x, soln4.y);
legend('Euler', 'ode45', 'Location','Best');
```

```
Warning: Failure at t=5.066046e-01.  Unable to
meet integration tolerances without reducing the
step size below the smallest value allowed
(1.776357e-15) at time t.
```




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


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
Euler
ode45



Comments:

The Euler method works well for approximating the second and third IVPs. For the first IVP, however, the Euler method of approximation creates a sudden spike in the middle of the graph. In the fourth IVP, the ode45 approximation stops around t=0.5066 whereas the improved euler approximation displays the approximation from t=0 to t=1.

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`.

(b) Calculate the solution of the IVP and evaluate it at `t=0.5`.

(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.

(d) Compute the error estimate for `t=0.5` and compare with the actual error.

(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.

```
f = @(t,y) 2*t*sqrt(1-y^2);
y0 = 0;
x_e = euler(f,y0,0:0.1:0.5);
disp(x_e(6));
% (b) The solution to the IVP is y = sin(t^2) = 0.247 (at t=
% (c) En = [(e^(M*del_t*n)-1)*del_t*(1+M)]/2
%  M = 2 (the maximum value), del_t is 0.1 (stepsize), and n is 5.
% (d) En = 0.2577. The actual error at this point is 0.0477.
x_e = euler(f,y0,0:0.001:0.5);
disp(x_e(501));
% (e) En with a stepsize of 0.001 (M=2,n=500) is 0.00257. The actual
 error
% at this point is 0.0001
```

```
    0.1993

    0.2469
```

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.

Comments:

The updated step size checks whether the [tolerated error/the error calculated (D)] is within the bounds of 0.3 and 2. If it is less than the bounds, the step size will be 0.27*h and if it is greater than the upperbound, the step size will be 1.8*h. This ensures that the step size lies within a reasonable range and that it will gradually approximate closer and closer values of y. If there is a large error, the next step size will be larger and will bring the approximation closer to the true solution faster whilst a smaller error will keep the step size small as it is already accurate enough.

Exercise 5

Objective: Compare Euler to your Adaptive Euler method.

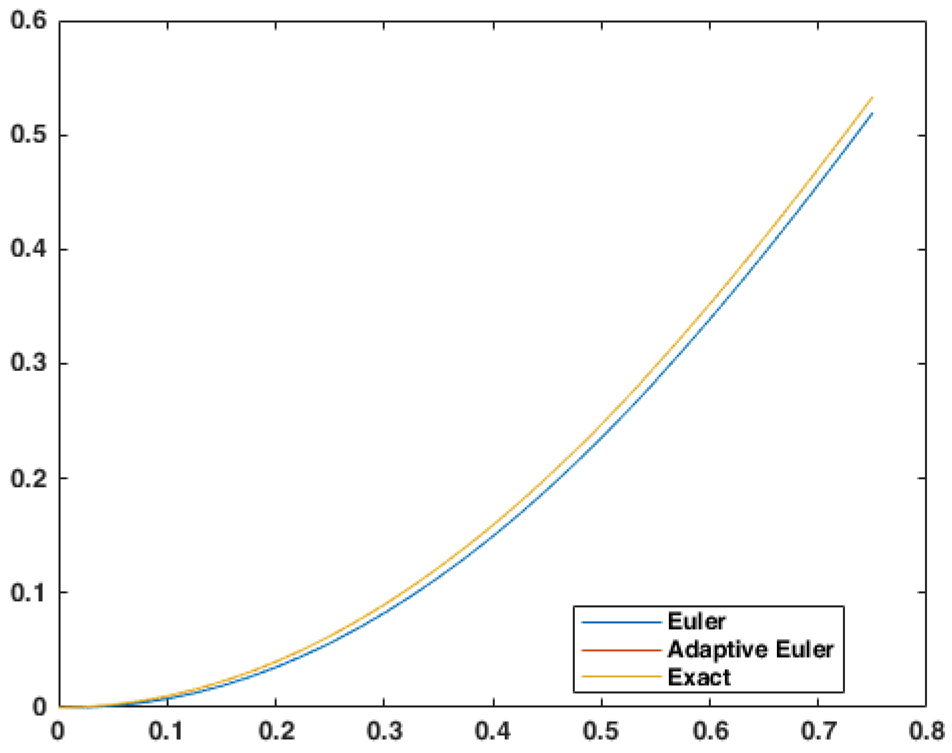
Details: Consider the IVP from exercise 3.

(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.

```
f = @(t,y) 2*t*sqrt(1-y^2);
y0 = 0;
y_e = euler(f,y0,0:0.025:0.75);

[x,y] = adaptive(f,0,0.75,y0,0.025);
x_e = linspace(0,0.75,0.75/0.025 +1);

plot(x_e, y_e, x,y, x_e, sin(x_e.^2));
legend('Euler', 'Adaptive Euler','Exact', 'Location','Best');
```



Exercise 6

Objective: Problems with Numerical Methods.

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

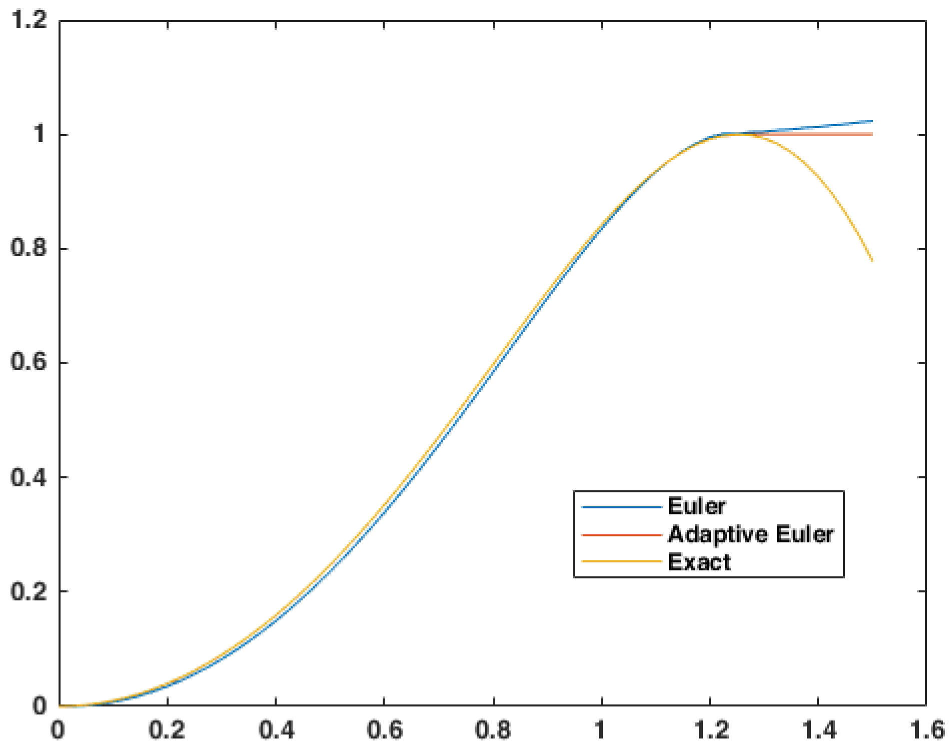
(a) From the two approximations calculated in exercise 5, which one is closer to the actual solution (done in 3.b)? Explain why.

(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`.

(c) Notice how the exact solution and the approximations become very different. Why is that? Write your answer as a comment.

```
f = @(t,y) 2*t*sqrt(1-y^2);
y0 = 0;

y_e = euler(f,y0,0:0.025:1.5);

[x,y] = adaptive(f,0,1.5,y0,0.025);
x_e = linspace(0,1.5,1.5/0.025 +1);

plot(x_e, y_e, x,y, x_e, sin(x_e.^2));
legend('Euler', 'Adaptive Euler','Exact', 'Location','Best');
```

```
Warning: Imaginary parts of complex X and/or Y
arguments ignored
```



Comments:

My adaptive Euler method is closer to the actual solution. This can not only be seen by zooming in on the graph but also logically, the adapted Euler method should be more accurate as it adapts the timestep depending on the error of each step. From this plot, we can see that the exact and the approximations become very different because the slope changes very drastically. The two methods were developped with an increasing function in mind while the function sin(x^2) varies in slope rapidly and the approximations cannot keep up with the rapidly changing slope.

Published with MATLAB® R2019b