Final Project

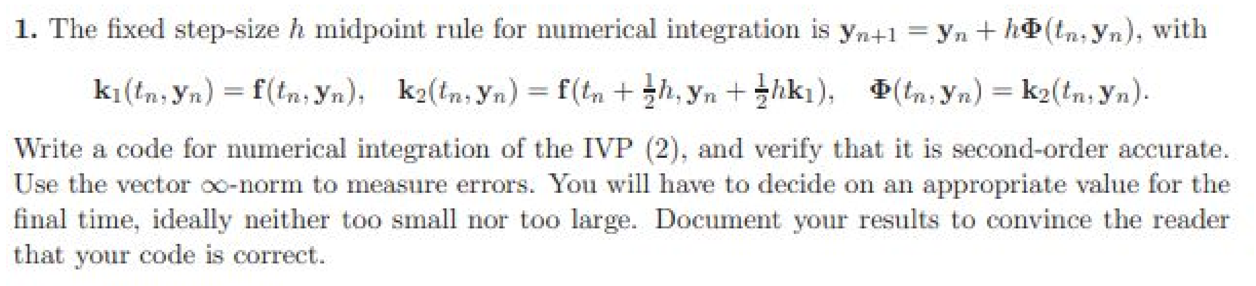
Numerical Computing

15th May 2020

Numerical Computing

Table of Contents

1 First Problem

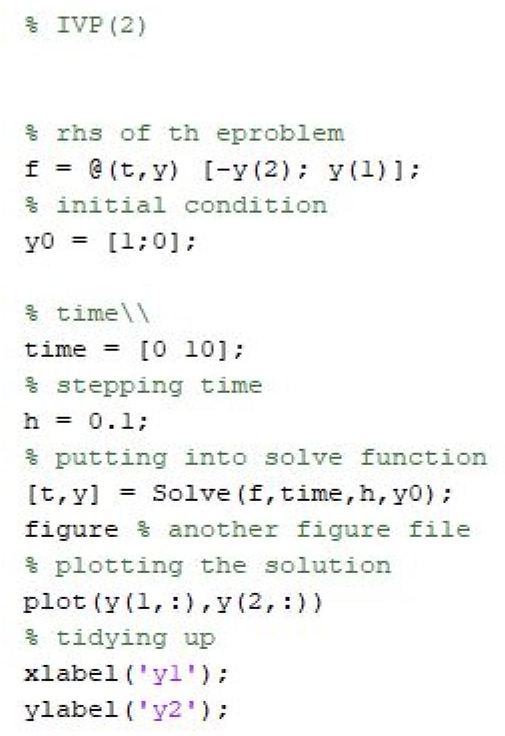


1. The fixed step-size h midpoint rule for numerical integration is $\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{\Phi}(t_n, \mathbf{y}_n)$, with

$$\mathbf{k}_1(t_n, \mathbf{y}_n) = \mathbf{f}(t_n, \mathbf{y}_n), \quad \mathbf{k}_2(t_n, \mathbf{y}_n) = \mathbf{f}(t_n + \tfrac{1}{2}h, \mathbf{y}_n + \tfrac{1}{2}h\mathbf{k}_1), \quad \mathbf{\Phi}(t_n, \mathbf{y}_n) = \mathbf{k}_2(t_n, \mathbf{y}_n).$$

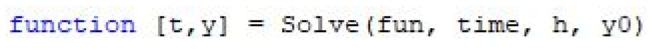
Write a code for numerical integration of the IVP (2), and verify that it is second-order accurate. Use the vector ∞-norm to measure errors. You will have to decide on an appropriate value for the final time, ideally neither too small nor too large. Document your results to convince the reader that your code is correct.

Code for IVP(2)



```
% IVP(2)


% rhs of th eproblem
f = @(t,y) [-y(2); y(1)];
% initial condition
y0 = [1;0];

% time\\
time = [0 10];
% stepping time
h = 0.1;
% putting into solve function
[t,y] = Solve(f,time,h,y0);
figure % another figure file
% plotting the solution
plot(y(1,:),y(2,:))
% tidying up
xlabel('y1');
ylabel('y2');
```

Put into this function which is based off the code given in the project details

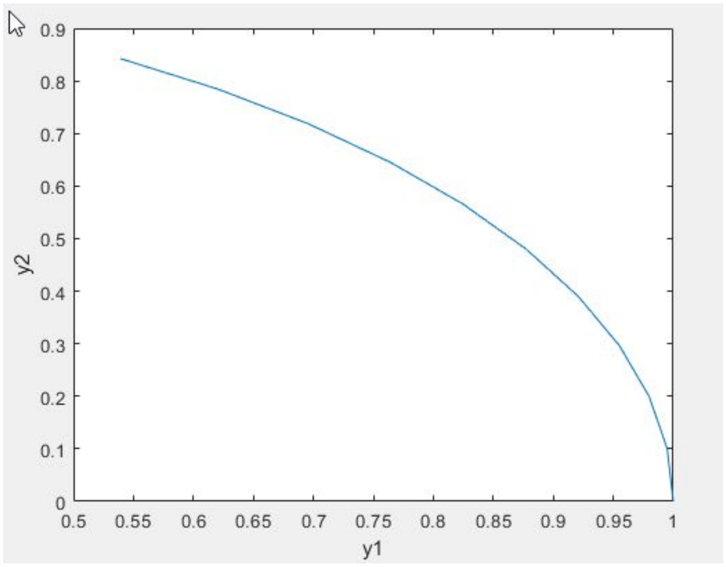


```
function [t,y] = Solve(fun, time, h, y0)
```

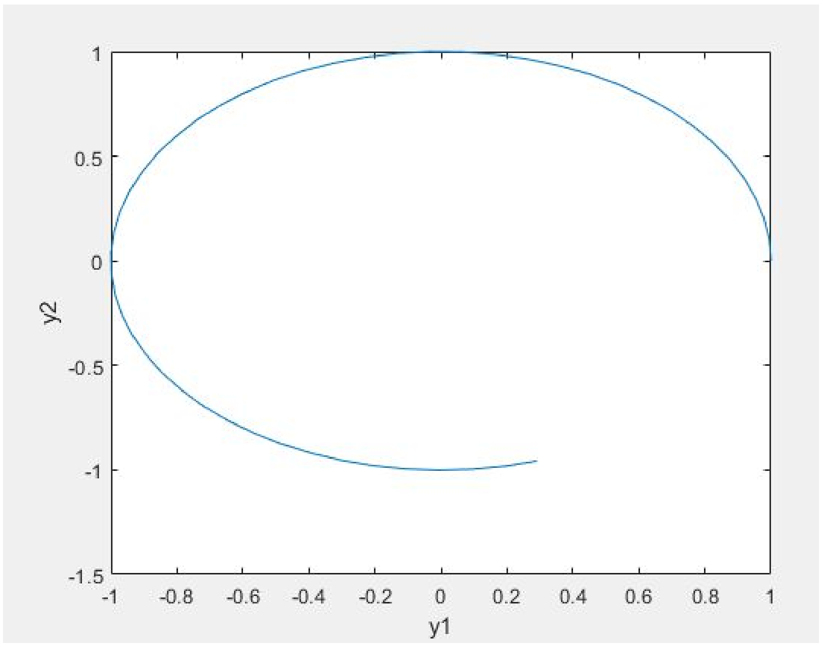
Fun is function -y2 = y1
Time is a vertical matrix S = [0 15] We are using 15 : more details down below
H is the step that will be calculated in advance

y0 is just a simple solve matrix with [1 ; 0]
Instead of using a while loop I just did a for loop of 1 to N-1

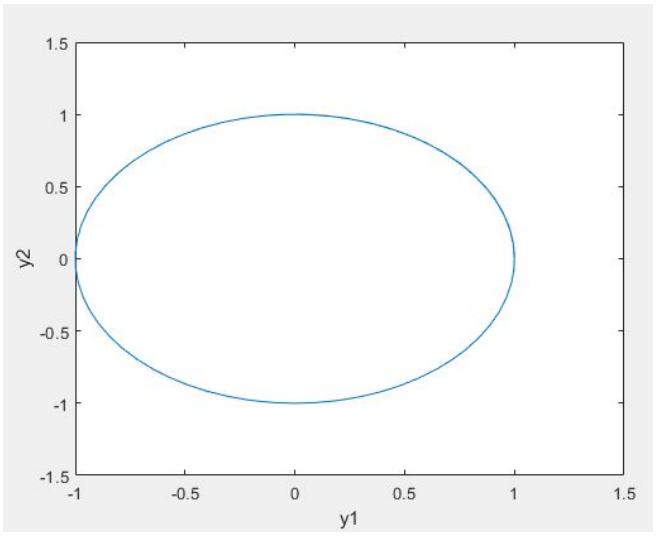
The Plot of IVP(2) Time [0 1]



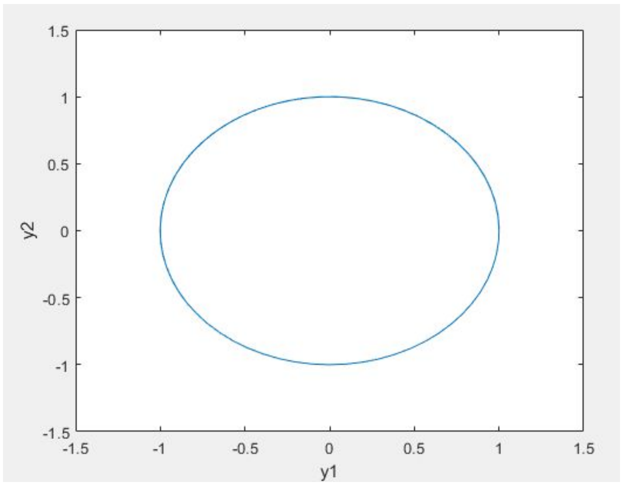
The plot of IVP(2) Time [0 5]



The Plot of IVP(2) Time [0 8]



The Plot of IVP(2) Time [0 15]

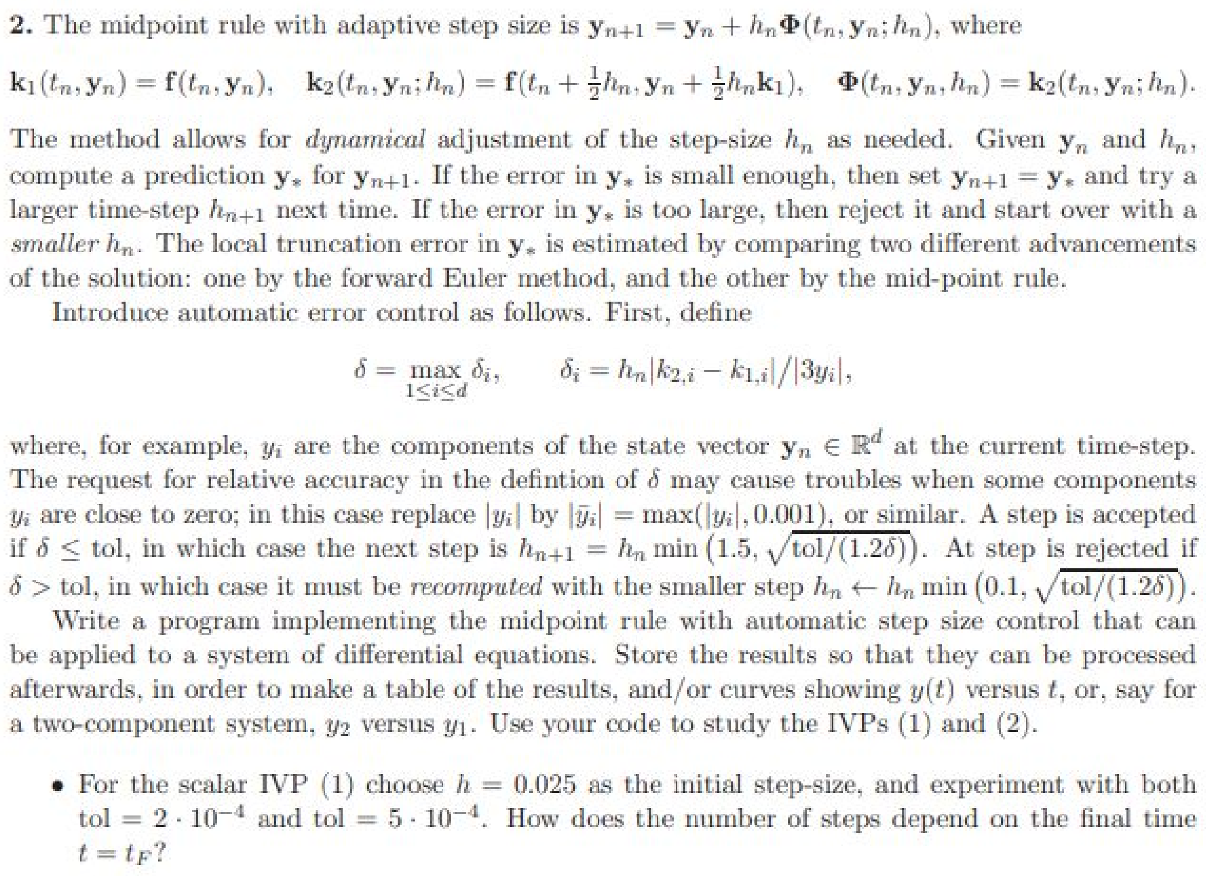


So, time [0 15] Seems the most well rounded and that is what we will be using.

```
first = fun(t(n),y(:,n));
second = fun(t(n)+h/2,y(:,n)+h*first/2);
y(:,n+1) = y(:,n) + h*second;
```

This is the second order algorithm that was used

2 Second Problem



2. The midpoint rule with adaptive step size is $\mathbf{y}_{n+1} = \mathbf{y}_n + h_n \mathbf{\Phi}(t_n, \mathbf{y}_n; h_n)$, where

$$\mathbf{k}_1(t_n, \mathbf{y}_n) = \mathbf{f}(t_n, \mathbf{y}_n), \quad \mathbf{k}_2(t_n, \mathbf{y}_n; h_n) = \mathbf{f}(t_n + \tfrac{1}{2}h_n, \mathbf{y}_n + \tfrac{1}{2}h_n\mathbf{k}_1), \quad \mathbf{\Phi}(t_n, \mathbf{y}_n, h_n) = \mathbf{k}_2(t_n, \mathbf{y}_n; h_n).$$

The method allows for *dynamical* adjustment of the step-size h_n as needed. Given $\mathbf{y}_n$ and h_n, compute a prediction $\mathbf{y}_*$ for $\mathbf{y}_{n+1}$. If the error in $\mathbf{y}_*$ is small enough, then set $\mathbf{y}_{n+1} = \mathbf{y}_*$ and try a larger time-step h_{n+1} next time. If the error in $\mathbf{y}_*$ is too large, then reject it and start over with a *smaller* h_n. The local truncation error in $\mathbf{y}_*$ is estimated by comparing two different advancements of the solution: one by the forward Euler method, and the other by the mid-point rule.

Introduce automatic error control as follows. First, define

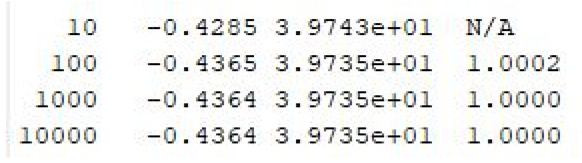
$$\delta = \max_{1 \le i \le d} \delta_i, \qquad \delta_i = h_n|k_{2,i} - k_{1,i}|/|3y_i|,$$

where, for example, y_i are the components of the state vector $\mathbf{y}_n \in \mathbb{R}^d$ at the current time-step. The request for relative accuracy in the defintion of δ may cause troubles when some components y_i are close to zero; in this case replace $|y_i|$ by $|\bar{y}_i| = \max(|y_i|, 0.001)$, or similar. A step is accepted if $\delta \le \text{tol}$, in which case the next step is $h_{n+1} = h_n \min\left(1.5, \sqrt{\text{tol}/(1.2\delta)}\right)$. At step is rejected if $\delta > \text{tol}$, in which case it must be *recomputed* with the smaller step $h_n \leftarrow h_n \min\left(0.1, \sqrt{\text{tol}/(1.2\delta)}\right)$.

Write a program implementing the midpoint rule with automatic step size control that can be applied to a system of differential equations. Store the results so that they can be processed afterwards, in order to make a table of the results, and/or curves showing $y(t)$ versus t, or, say for a two-component system, y_2 versus y_1. Use your code to study the IVPs (1) and (2).

- For the scalar IVP (1) choose $h = 0.025$ as the initial step-size, and experiment with both $\text{tol} = 2 \cdot 10^{-4}$ and $\text{tol} = 5 \cdot 10^{-4}$. How does the number of steps depend on the final time $t = t_F$?

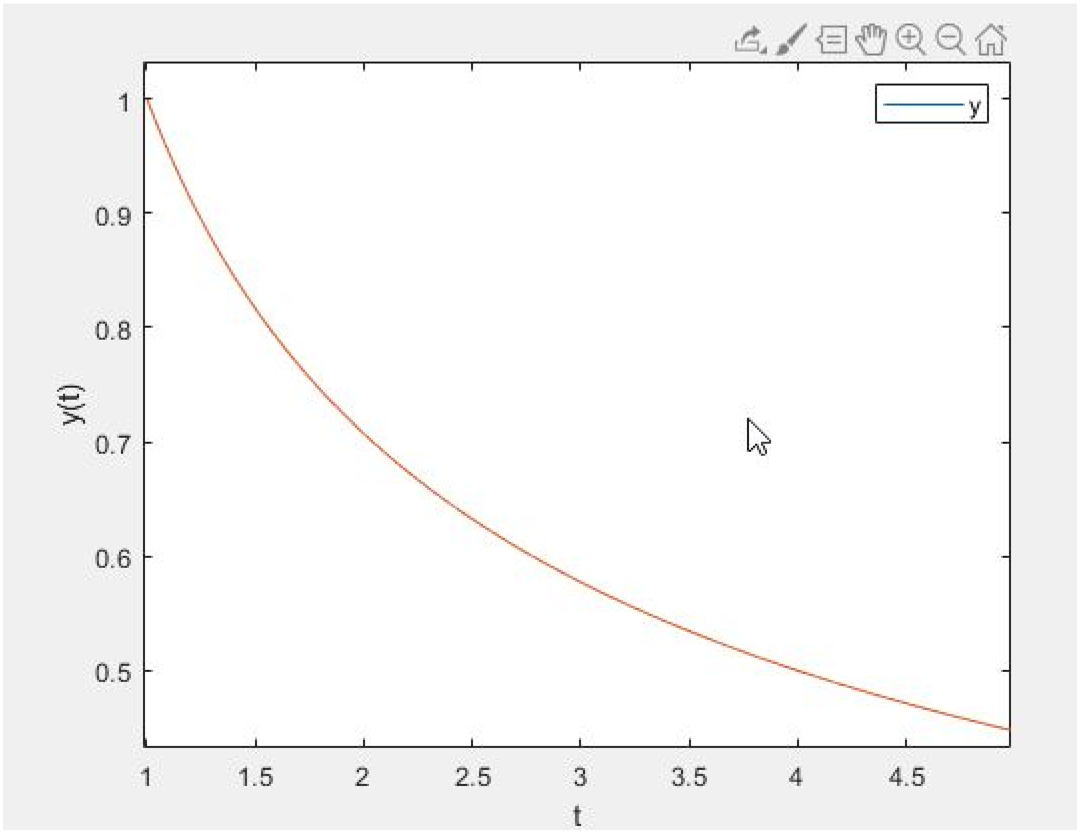
The more steps, the fast it will decline.



```
   10    -0.4285 3.9743e+01   N/A
  100    -0.4365 3.9735e+01   1.0002
 1000    -0.4364 3.9735e+01   1.0000
10000    -0.4364 3.9735e+01   1.0000
```

There is an eventual stopping point though at -0.4365

https://i.imgur.com/g4QF9Rk.gifv see animation of it happening here.



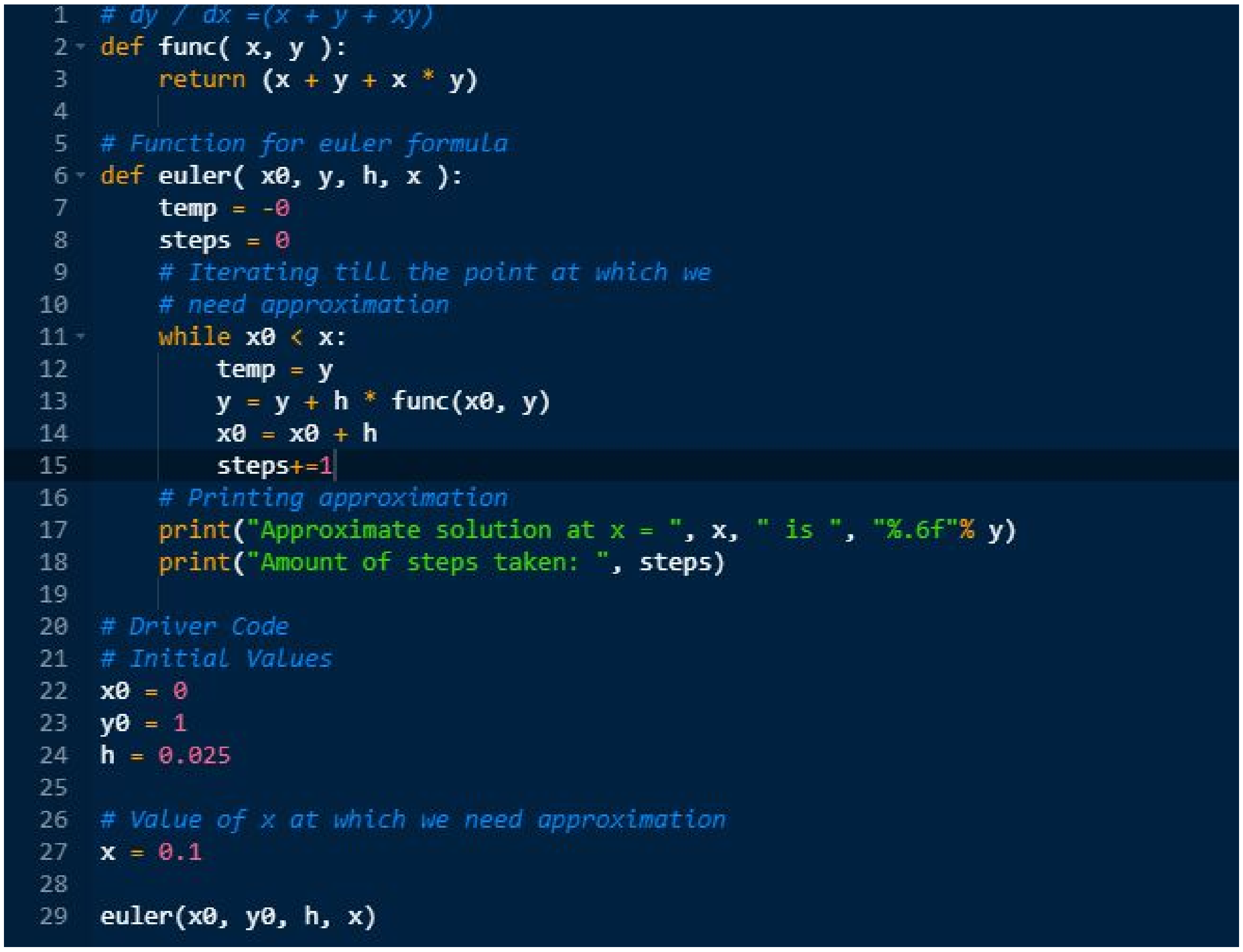
Part 2

- For the "circle test" (2) stop the computations after 10 revolutions ($t = 20\pi$). Experiment with different tolerances, and determine how small the tolerance has to be in order that the circle on the screen should not become "thick". Discuss the total number of steps needed. Provide plots as necessary to explain your results.

The guess steps are set at 10,000 but the actual number of steps needed to solve IVP(2) is seen below to explain the reasoning.

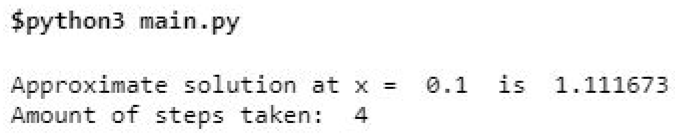
With python, you can see a simple variation of the program where it will count the steps for a linear IVP solving process but solving an IVP with a step size that can change, is a little more difficult to pinpoint.

Here is the basic python script that allows to solve for some X and uses stepping to solve it by using eulers formula for solving an IVP.



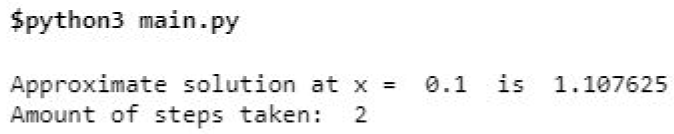
```python
# dy / dx =(x + y + xy)
def func( x, y ):
    return (x + y + x * y)

# Function for euler formula
def euler( x0, y, h, x ):
    temp = -0
    steps = 0
    # Iterating till the point at which we
    # need approximation
    while x0 < x:
        temp = y
        y = y + h * func(x0, y)
        x0 = x0 + h
        steps+=1
    # Printing approximation
    print("Approximate solution at x = ", x, " is ", "%.6f"% y)
    print("Amount of steps taken: ", steps)

# Driver Code
# Initial Values
x0 = 0
y0 = 1
h = 0.025

# Value of x at which we need approximation
x = 0.1

euler(x0, y0, h, x)
```

When H = .0025



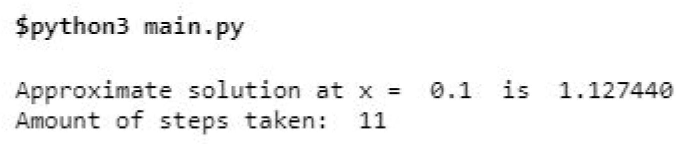
```
$python3 main.py

Approximate solution at x =  0.1  is  1.111673
Amount of steps taken:  4
```

When H = .0050   Steps = 2



```
$python3 main.py

Approximate solution at x =  0.1  is  1.107625
Amount of steps taken:  2
```

When H = .001   Steps = 11

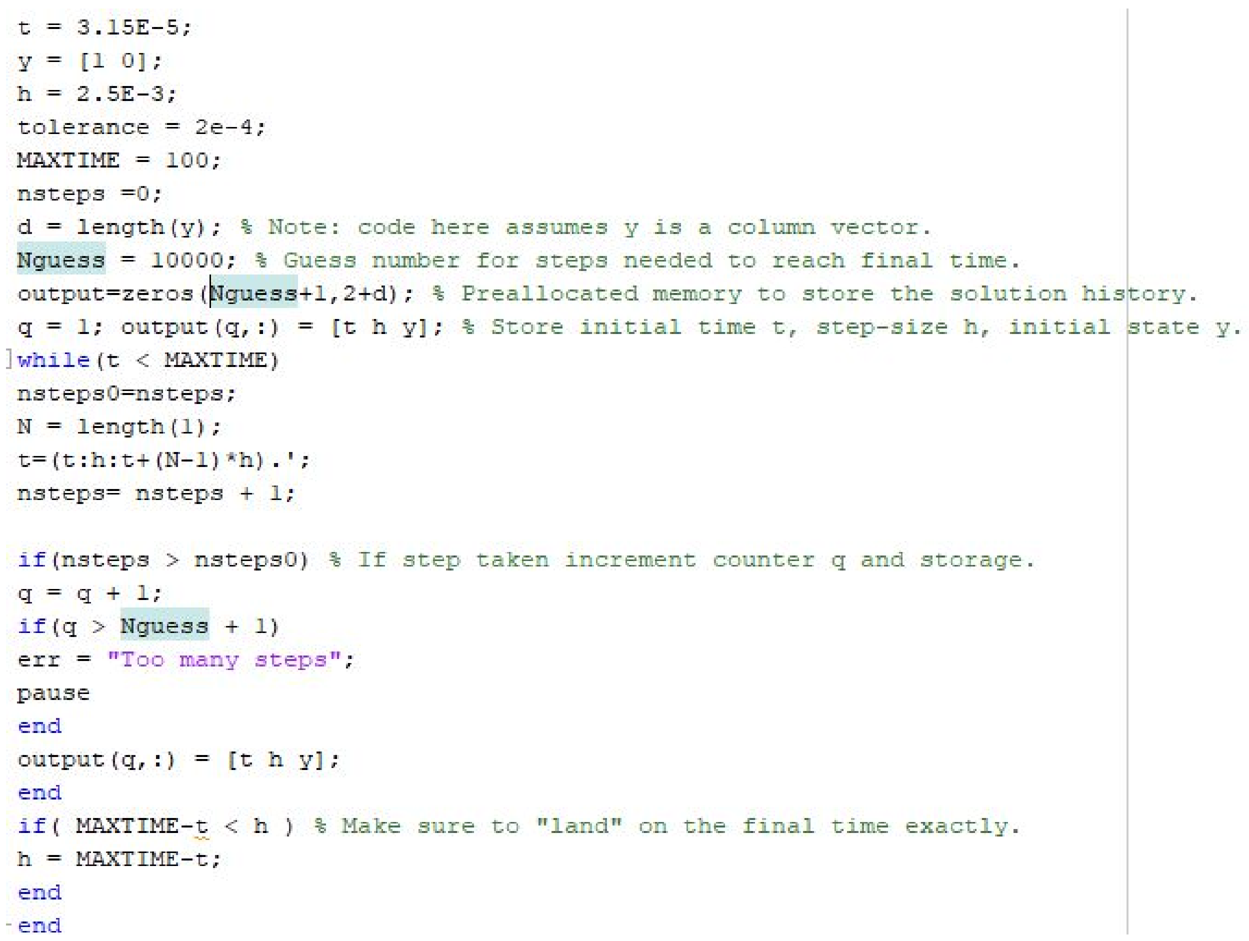


```
$python3 main.py

Approximate solution at x =  0.1  is  1.127440
Amount of steps taken:  11
```

Based off this assumption, the Smaller the initial stepsize, the more steps it will take to reach the solution.

You can find that by looking at the local truncation error that is from the approximation.

Source:
http://www1.maths.leeds.ac.uk/~kersale/2600/Notes/runge-kutta.pdf



```
t = 3.15E-5;
y = [1 0];
h = 2.5E-3;
tolerance = 2e-4;
MAXTIME = 100;
nsteps =0;
d = length(y); % Note: code here assumes y is a column vector.
Nguess = 10000; % Guess number for steps needed to reach final time.
output=zeros(Nguess+1,2+d); % Preallocated memory to store the solution history.
q = 1; output(q,:) = [t h y]; % Store initial time t, step-size h, initial state y.
while(t < MAXTIME)
nsteps0=nsteps;
N = length(1);
t=(t:h:t+(N-1)*h).';
nsteps= nsteps + 1;

if(nsteps > nsteps0) % If step taken increment counter q and storage.
q = q + 1;
if(q > Nguess + 1)
err = "Too many steps";
pause
end
output(q,:) = [t h y];
end
if( MAXTIME-t < h ) % Make sure to "land" on the final time exactly.
h = MAXTIME-t;
end
end
```

Credits

https://books.google.com/books?id=9y82DwAAQBAJ&pg=PT91&lpg=PT91&dq=K1+%3D+fun(t(n),y(:,n));+K2+%3D+fun(t(n)%2Bh/2,y(:,n)%2Bh*K1/2);+y(:,n%2B1)+%3D+y(:,n)+%2B+h*K2;&source=bl&ots=9BMtenn2gg&sig=ACfU3U2tKO9JuZ-M53c58TTVYrxX7A5-ig&hl=en&sa=X&ved=2ahUKEwi4q_j-hrfpAhVQCs0KHcQHDnQQ6AEwAHoECAQQAQ#v=onepage&q=K1%20%3D%20fun(t(n)%2Cy(%3A%2Cn))%3B%20K2%20%3D%20fun(t(n)%2Bh%2F2%2Cy(%3A%2Cn)%2Bh*K1%2F2)%3B%20y(%3A%2Cn%2B1)%20%3D%20y(%3A%2Cn)%20%2B%20h*K2%3B&f=false Where I found the second order algorithm