PROJECT 1

NUMN12 IVP PROJECT

Jose Mendez-Villanueva

Lund University
NUMN12: Numerical Analysis for Differential Equations
Professor Tony Stillfjord

Brief Theoretical & Methodical Foundations

To be able to calculate formulas that do not have exact solutions, we use numerical methods. By using these methods, we start off with some initial values and bounds and approximate values from what is given and circulate from there on out. Some of these methods are Euler and Runge Kutta. The difference in methods and uses are very particular. Some methods are more accurate than others and some computationally more expensive than others as well. The method that we focus on in this project is primarily Runge Kutta 4th Order and some of its derivations as it is a more accurate method. The form of this base method that we use is as follows:

Runge Kutta 4th Order

$$k_1 = \dot{x}(t, y)$$

$$k_2 = \dot{x}(t + \frac{h}{2}, y + \frac{hk_1}{2})$$

$$k_3 = \dot{x}(t + \frac{h}{2}, y + \frac{hk_2}{2})$$

$$k_4 = \dot{x}(t + h, y + hk_{y3})$$

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

The reason for using this method is because of the use of different points which gives us more accuracy and stability to calculate four stage derivatives (which are the k's) to be able to compute the new value. This method works well up to certain point. By this I mean, if you would want a more accurate solution you would either use a higher order method or in my case, one that accounts for the error and controls it while doing the approximation (Adaptive Runge Kutta). Here we introduce the Runge Kutta 34 method which leads to this Adaptive Runge Kutta Method:

Runge Kutta 34

$$k_1 = \dot{x}(t, y)$$

$$k_2 = \dot{x}(t + \frac{h}{2}, y + \frac{hk_1}{2})$$

$$k_3 = \dot{x}(t + \frac{h}{2}, y + \frac{hk_2}{2})$$

$$z_3 = \dot{x}(t + h, y - hk_1 + 2hk_2)$$

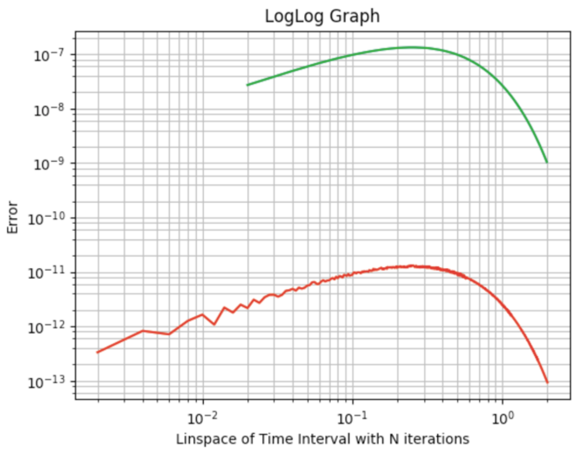
$$k_4 = \dot{x}(t + h, y + hk_{y3})$$

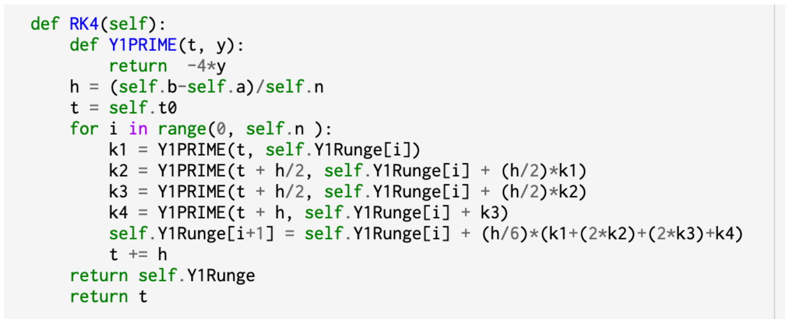
$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$l_{i+1} = \frac{h}{6}(2k_1 + z_3 - 2k_3 - k_4)$$

Where it is almost the same as the 4th order Runge Kutta method but in which we include a 3rd Order approximation (z_3) to our 4th order approximation. The value l_{i+1} is calculating the local error of the approximation using both the 3rd order and 4th order approximations to generate this. Thus, from this local error we can construct a new way of keeping our error under control by creating a tolerance to adapt our step size in accordance to our error, where our step size is the variable h. We will take the norm of the local error that we will denote as R. Thus we set a tolerance for R and while R is within our range of tolerance, we then update our values and see if our step size is within our set range as well because we will be adapting our step size in order to see if we need to make it smaller or larger depending on our error and the fact that it doesn't pass over out time interval. This is to give us a more accurate solution. We then can construct our new step size that is dependent off of R and the tolerance of R to continue. The reason for using this Adaptive Runge Kutta 34 method is because of the fact that it is of Order 4. Order of convergence is the estimate of the speed at which the error will go to zero thus the higher the Order of convergence the better. Continuing from this, you can only use Runge Kutta for 1st Order Equations thus when given a Higher Order Equation you have to reduce it to multiple 1st Order Equations in which you then have to modify the method to a multiple dimension Runge Kutta.

Task 2

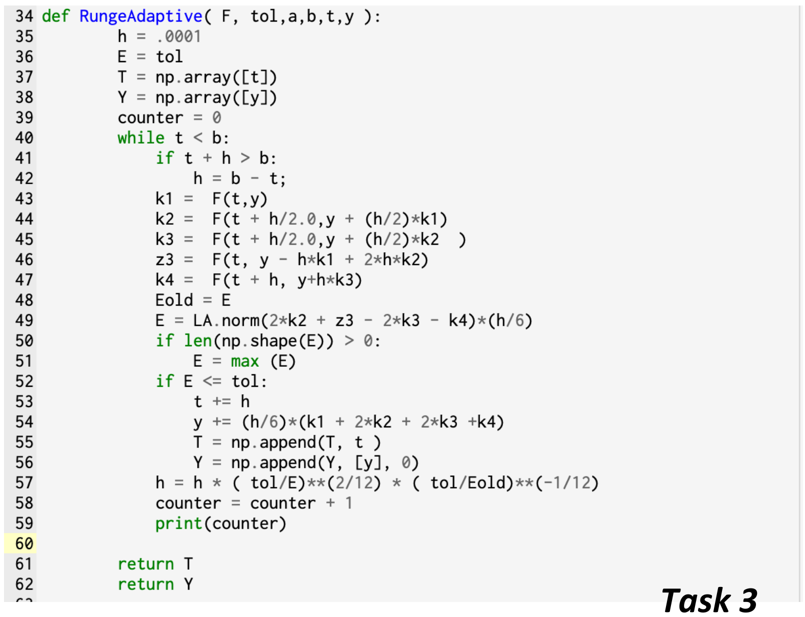




```python
def RK4(self):
    def Y1PRIME(t, y):
        return  -4*y
    h = (self.b-self.a)/self.n
    t = self.t0
    for i in range(0, self.n ):
        k1 = Y1PRIME(t, self.Y1Runge[i])
        k2 = Y1PRIME(t + h/2, self.Y1Runge[i] + (h/2)*k1)
        k3 = Y1PRIME(t + h/2, self.Y1Runge[i] + (h/2)*k2)
        k4 = Y1PRIME(t + h, self.Y1Runge[i] + k3)
        self.Y1Runge[i+1] = self.Y1Runge[i] + (h/6)*(k1+(2*k2)+(2*k3)+k4)
        t += h
    return self.Y1Runge
    return t
```

This graph above is depicting a graph of Error vs. the Linspace of Time for Runge Kutta 4 on the interval [0,2] with 100 iterations on the top and 1000 iterations on the bottom. This graph lets us know the order of convergence for the method that we are using for the equation/equations we plug into the code. You are able to find the order of convergence by finding the logarithm of the difference of two Error points divided by the logarithm of the difference of two time points. The Order of Convergence for this method is closer to 3 with the slope being 3.46 after choosing two points on the graph. The right-hand side picture is the main block of code for the Runge Kutta 4 Method in which I use to give an approximation to the equation of $y' = -4y$.

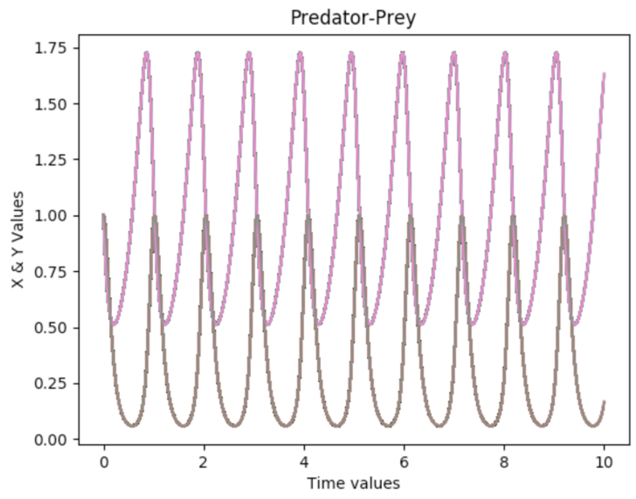
For the rest of the task, I decided to complete them a little different than how the instructions hinted to complete them. Instead of doing solely smaller functions to lead up to a bigger one, I did a combination of both as I took an object-oriented approach in Python. I ran into a few problems when trying to implement a few requirements for the Adaptive Runge Kutta 34 Method. The issues I had were starting with an h_0 equaling to $\frac{|t_f - t_0| TOL^{\frac{1}{4}}}{100(1+\|f(y_0)\|)}$. The issues that I had with this was implementing the norm of $f(y_0)$. I kept getting error messages when I tried applying the equation f which was vector valued, this is what seemed to be the problem. After many attempts of working on this, I decided to implement a small enough h_0 because it was not taking the norm value because it wanted a scalar/1 dimension array instead (the norm should've turned it into a scalar but for some reason it would not). Thus, to be able to run my code I set my h_0 to .0001 which is still fairly small enough as the other h_0 gives us a small value as well. I kept the rest of the values the same.

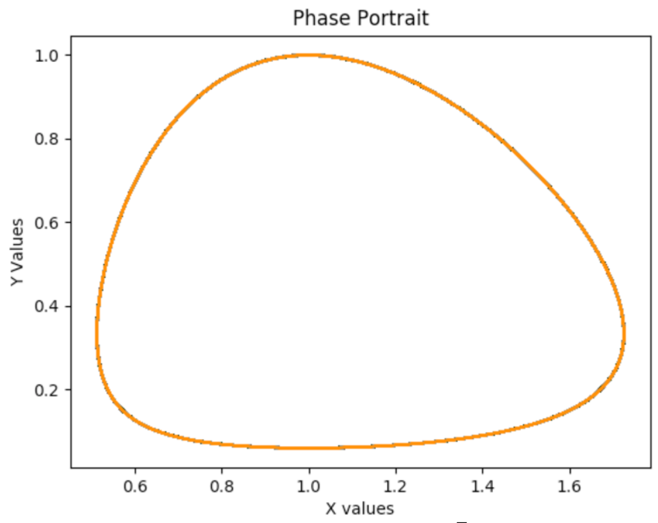


```
def RungeAdaptive( F, tol,a,b,t,y ):
        h = .0001
        E = tol
        T = np.array([t])
        Y = np.array([y])
        counter = 0
        while t < b:
            if t + h > b:
                h = b - t;
            k1 =  F(t,y)
            k2 =  F(t + h/2.0,y + (h/2)*k1)
            k3 =  F(t + h/2.0,y + (h/2)*k2  )
            z3 =  F(t, y - h*k1 + 2*h*k2)
            k4 =  F(t + h, y+h*k3)
            Eold = E
            E = LA.norm(2*k2 + z3 - 2*k3 - k4)*(h/6)
            if len(np.shape(E)) > 0:
                E = max (E)
            if E <= tol:
                t += h
                y += (h/6)*(k1 + 2*k2 + 2*k3 +k4)
                T = np.append(T, t )
                Y = np.append(Y, [y], 0)
            h = h * ( tol/E)**(2/12) * ( tol/Eold)**(-1/12)
            counter = counter + 1
            print(counter)

        return T
        return Y
```

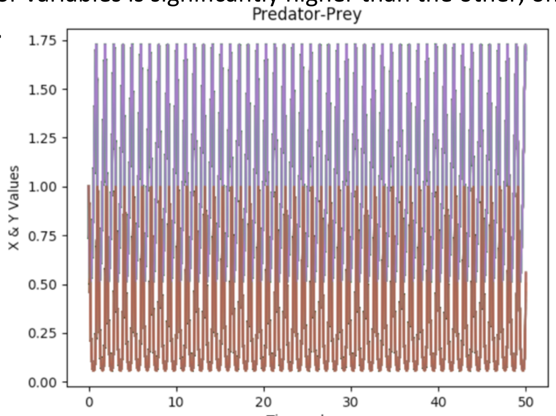
From the code on the left we see how to implement the above into the code. Here you can see how I use the h_0 instead of the one implied but it is still small enough to start off the calculations. Then you can see how the Error I used agrees with the guidelines. Here error is denoted with E. F here is vector valued, thus it can take a single equation or a system of equations.

Task 3

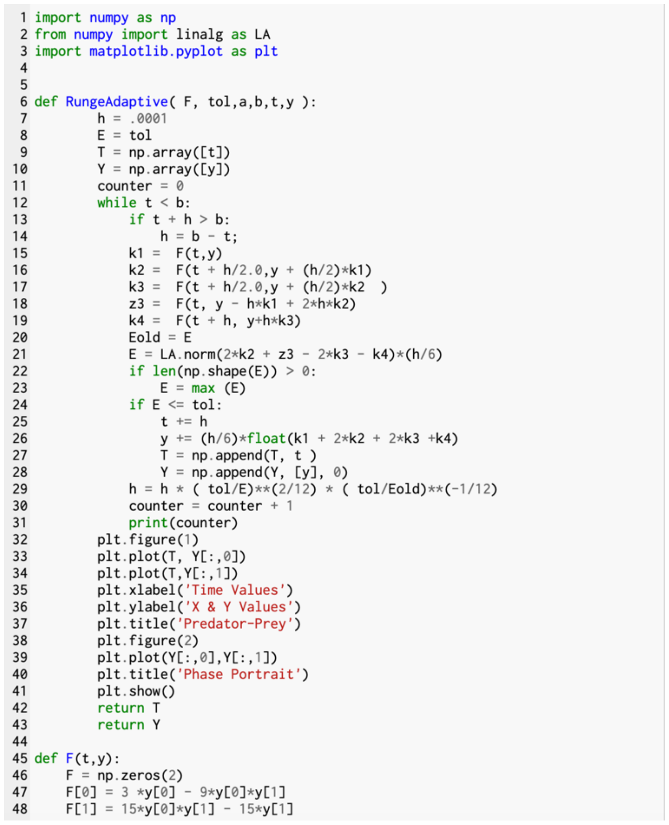




Using the given parameters of a, b, c, d as $[3, 9, 15, 15]$ with initial conditions of $t_0 = 0$ and $y(t_0) = [1,1]^T$ we get the above graphs. From these graphs we can see that the solutions are periodic as it has been claimed. Looking at the left graph you can see how at each second it completes one entire period and looking at the right graph we can see how it is periodic since it completes an entire orbit. For the sake of being able to see the graph clearly there are not any more periods included. In terms of time, the period tends to be 1 second long. The period was set from $[0,10]$. Changing the parameter values give you the same type of graph that is on the left side just with different variations in height. If the parameters are changed, the period does change as well. Looking at the graph below for a longer period of time shows that it keeps the integrity of the solution and that one variable doesn't dissolve. If one of either the Prey/Predator variables is significantly higher than the other, one would either increase drastically or go towards zero.

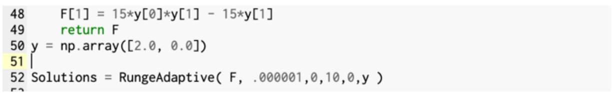


The values as they increase over time will not drift away from the equilibrium. They will continue typically in the same pattern as they have been going. The cycle is a loop (closed orbit) as has been seen in the phase portrait graph in which one variable goes down but then goes back up and vice versa. The only way it will change is how it was mentioned above, one variable would have to be significantly higher than the other.



```python
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt


def RungeAdaptive( F, tol,a,b,t,y ):
        h = .0001
        E = tol
        T = np.array([t])
        Y = np.array([y])
        counter = 0
        while t < b:
            if t + h > b:
                h = b - t;
            k1 =  F(t,y)
            k2 =  F(t + h/2.0,y + (h/2)*k1)
            k3 =  F(t + h/2.0,y + (h/2)*k2  )
            z3 =  F(t, y - h*k1 + 2*h*k2)
            k4 =  F(t + h, y+h*k3)
            Eold = E
            E = LA.norm(2*k2 + z3 - 2*k3 - k4)*(h/6)
            if len(np.shape(E)) > 0:
                E = max (E)
            if E <= tol:
                t += h
                y += (h/6)*float(k1 + 2*k2 + 2*k3 +k4)
                T = np.append(T, t )
                Y = np.append(Y, [y], 0)
            h = h * ( tol/E)**(2/12) * ( tol/Eold)**(-1/12)
            counter = counter + 1
            print(counter)
        plt.figure(1)
        plt.plot(T, Y[:,0])
        plt.plot(T,Y[:,1])
        plt.xlabel('Time Values')
        plt.ylabel('X & Y Values')
        plt.title('Predator-Prey')
        plt.figure(2)
        plt.plot(Y[:,0],Y[:,1])
        plt.title('Phase Portrait')
        plt.show()
        return T
        return Y

def F(t,y):
    F = np.zeros(2)
    F[0] = 3 *y[0] - 9*y[0]*y[1]
    F[1] = 15*y[0]*y[1] - 15*y[1]
```

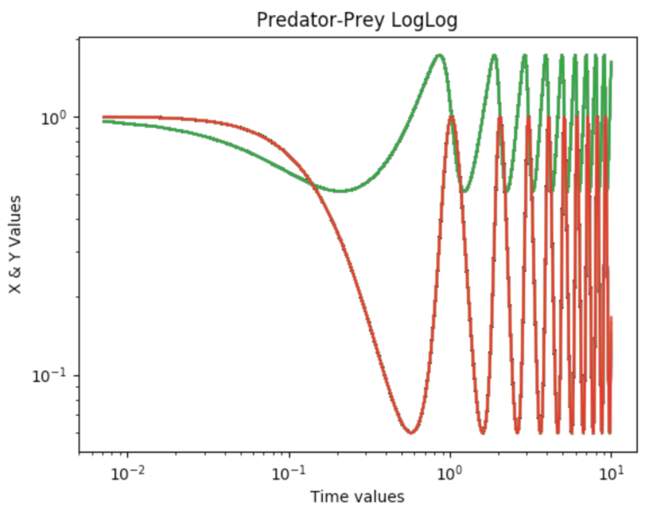
Code: Part 1



```python
    F[1] = 15*y[0]*y[1] - 15*y[1]
    return F
y = np.array([2.0, 0.0])

Solutions = RungeAdaptive( F, .000001,0,10,0,y )
```

Code: Part 2

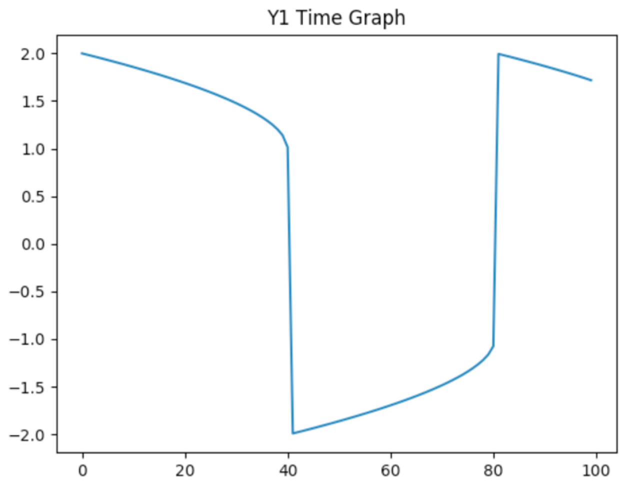
Looking at the above code, we denote $F(t, y)$ to be our Lotka Volterra Equation (Prey-Predator). We set the parameters in this equation and then eventually call it back with our $Solutions$ value denoted at the end of the code to be able to call the equation and use it. In this code, I had to adapt the Runge Kutta Method to be able to work for the System of ODE's as the Runge Kutta Method has to be adapted to be able to give approximations for several equations instead of one. The $k_{1,2,3,4}$ variables are the stage derivatives for both equations, where the $k's$ are for both equations. Even though we do not use t in the equation, I update t as the need for t still remains, just not in the stage derivatives. Time will keep on getting updated as we will still need this to keep our approximations within our time bounds and to keep on updating our step sizes. Thus, we need to proceed by taking the norm of the error for our new error and our initial error will be the tolerance.

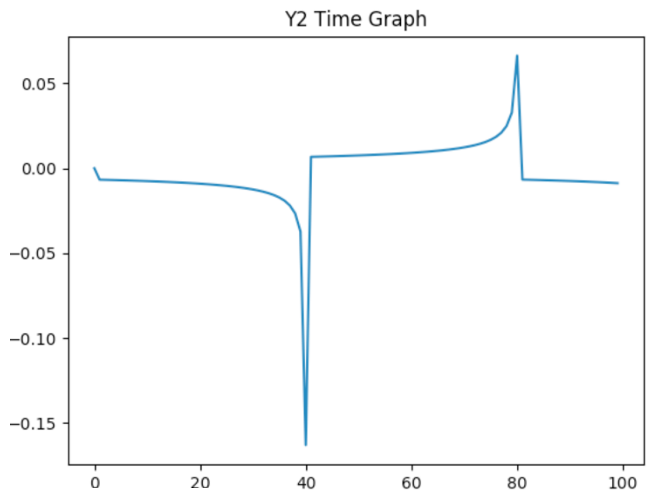


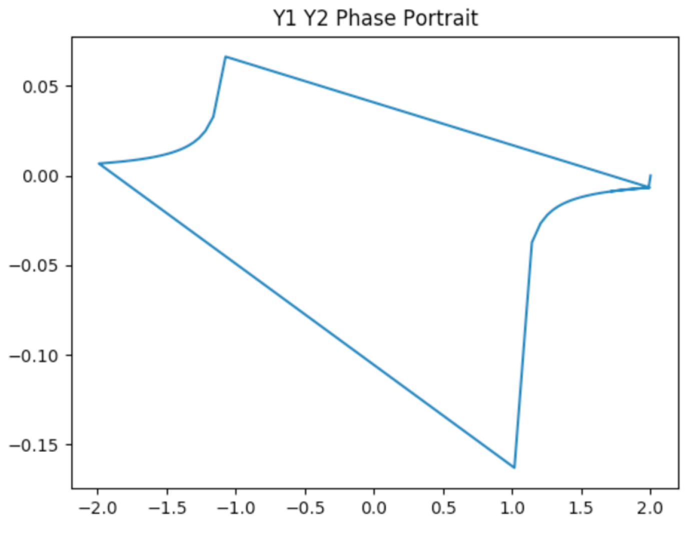
This graph above is of the same parameters as the ones earlier but using the loglog graph. The reason for including the loglog graph is to show that it does not show anything new or is needed at all for this case. Since there can be no negative values for this equation as you cannot have negative amounts of prey or predators this type of graph works as it would not accepts negative values to begin with but overall pointless in this case.

Task 4

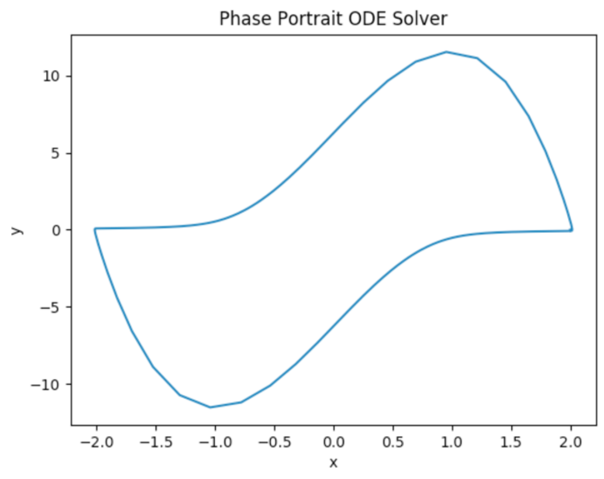
The following graphs will be for y_1 as a function of time, y_2 as a function of time, and then y_2 as a function of y_1 (phase portrait).



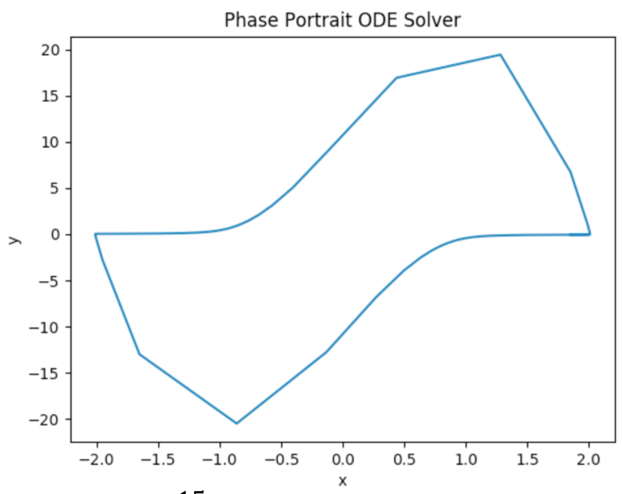




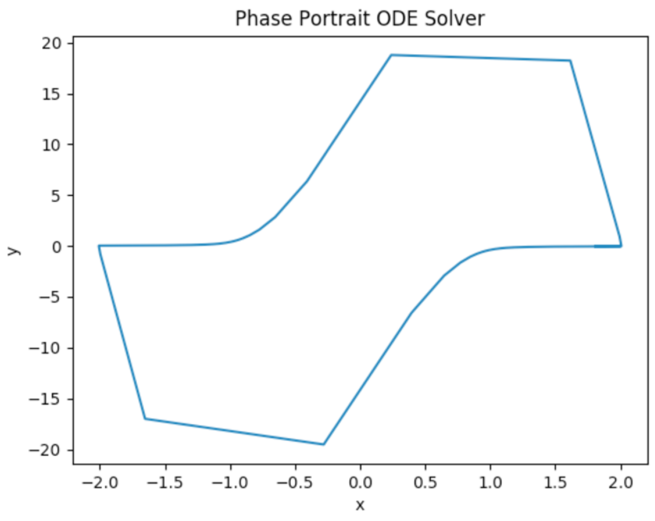
These graphs are for $\mu = 100$, interval of $[0, 200]$, $y(0) = [2,0]^T$. The variable μ describes the strength of the damping that is happening. The value of μ changes the look of the graph drastically, with smaller values making it look similar to a sine curve (similar to the graph of the Lotka-Volterra) and larger values making it look similar to the "Y1 Time Graph". This meaning that depending on the value of μ, we can anticipate towards what oscillation it will tend to. Now after much research and trying to implement the ODE solver in many ways, I found that as μ increases, the graph begins to distort compared to the graph we should have. In trying to figure out why this was happening, I found out that there are many solvers for ODE's but many of them being in MATLAB with many of them being for stiff and non-stiff problems. After much testing and trying to figure out and implement new things I noticed that as μ goes over 20 the ODE solver starts to fail in python for the phase portrait graphs which implies in accurate results. I used ode and odeint in python and neither one of those would give the graph that is needed. Originally I thought it could've been because of the fact of how I was using the function $Vanderpol(X, t)$ where the X and t are reversed but looking online, that's how the order should be for this solver. Looking online I would only see examples of μ with small values for solvers in python and could not find a stiff ode solver that would work. From the following we can see how when $\mu = 8$ we get the desired phase portrait but after that is begins to distort significantly.

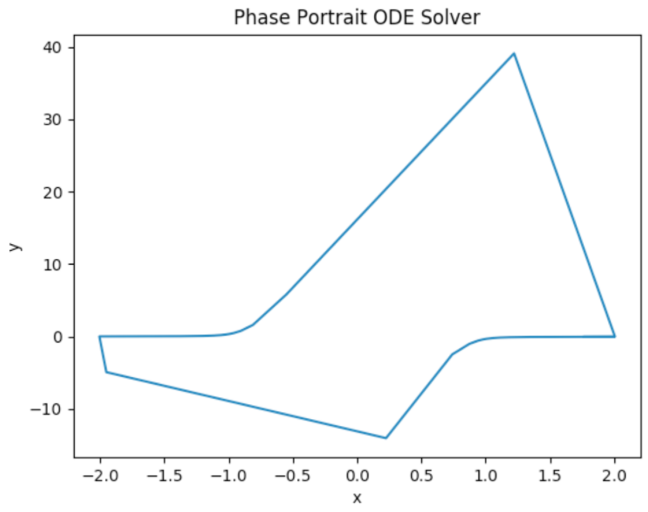


$\mu = 8$



$\mu = 15$





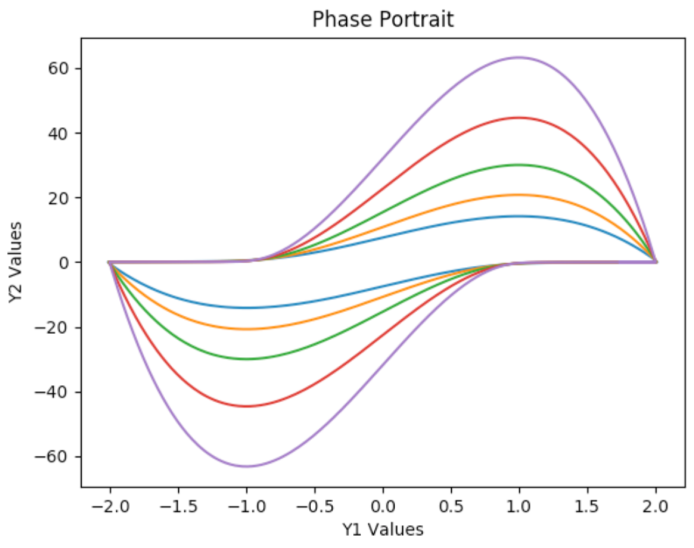
$\mu = 20$

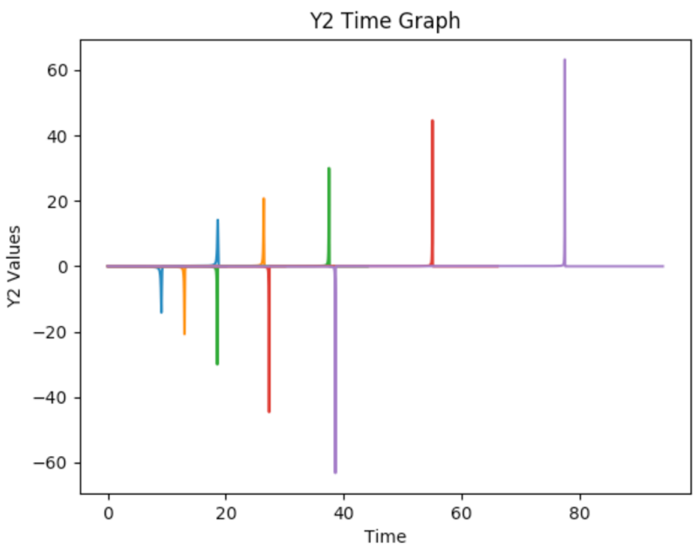
$\mu = 30$

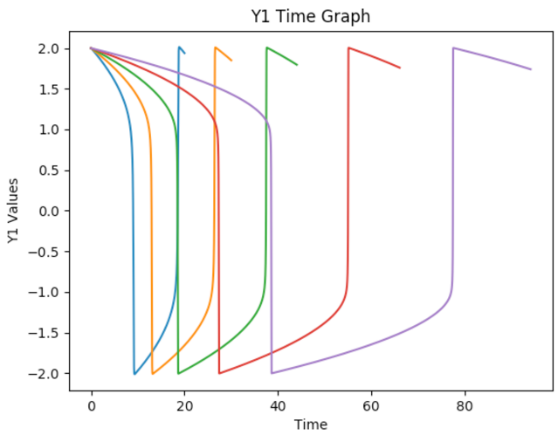
I now used my code to run the system with several different values for μ with certain parameters. The parameters that I used in my code were to set $tol = .000001$. Then I also set $h_0 = .0001$, time interval was set to $[0, 2(\mu)]$, and initial conditions were set to $y(0) = [2,0]^T$. Then I was able to compute μ for the values of 10,15,22,33,47,68 but after that I was not able to go any further because of run time. The number of iterations grew significantly as μ grew. The following chart will depict this especially between 47 and 68:

$\boldsymbol{\mu}$ Values	# of iterations
10	31822
15	34654
22	37416
33	41164
47	45031
68	8 million +
100	~~~

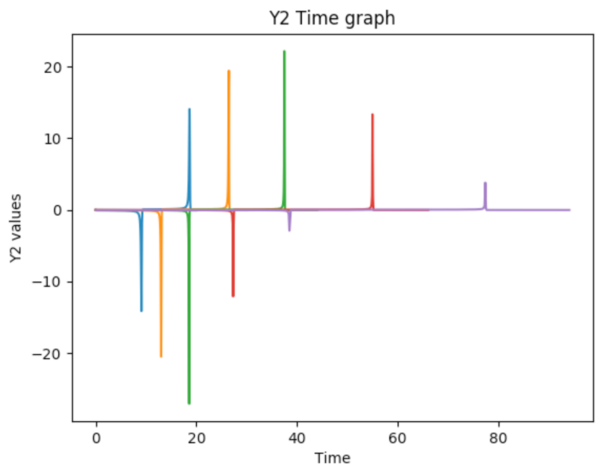
By looking at bigger periods and whole graphs of the van der Pol equation and from the data that I have collected, the larger the value of μ is, the stiffer the oscillation becomes. This is why as μ has increased so has the number of iterations needed as seen above. Thus, from this we see that μ and stiffness go hand in hand. Looking at the number of iterations compared to μ you can see that $N \sim C\mu^q$ with q being of power 3 but then you would have to see what your C is to make it equivalent to that.

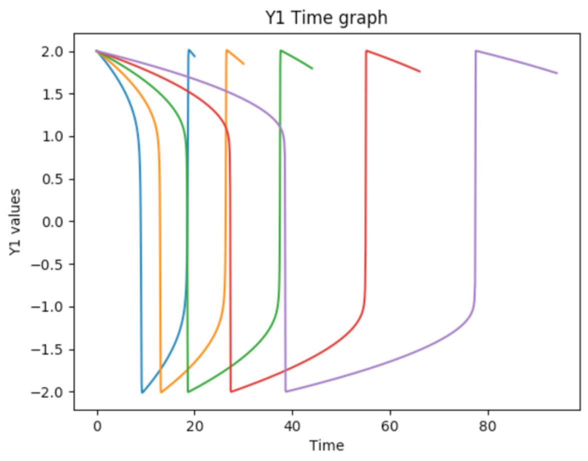






Here the graphs are using my approximation and the same parameters as I mentioned above but only for 5 values of μ. As you can tell, the graph from left to right has increasing values of μ. The values that I used for μ were 10,15,22,33, and 47 with 10 being the left most and 47 the right most. As the value of μ grows, so does the number of iterations and time interval as seen above. The reason why I chose this graph versus a loglog graph was that if you see above, there are negative values for my output and thus a loglog graph would not work for this. To verify that this is accurate, I reviewed other cases of van der Pol oscillator graphs to see if there were negative values in which I did see that there were. Not only this but to verify that my three graphs were accurate, and that they had the same general shape as the graphs that I saw.





These graphs above are using the same values of μ and parameters as the two above. The difference between them is that the set of new graphs are using an ODE solver to give the solution versus my approximation. If you compare both of these graph's they are very similar with minor differences but when bringing up the Phase Portrait, that is where we see a big difference as shown above. Using the ODE solver for this same equation I was able to solve for higher values of μ compared to my code as it would take too long. I was able to run values higher than 1,000 for μ with very fast computing speeds whereas my code took a while to be able to process the approximations, but with the fact that it is not plotting an accurate phase portrait signifies that the results it is giving are not accurate as the problem becomes too stiff for it to handle.

The ODE solver should compute better and faster as there are many solvers which are made to compute stiff problems. For example, some solvers which are not adapted to stiff problems are forced to take very small step sizes in which it can lead to the integration failing or taking a very long time such as was the case for my approximations as it was running many iterations. Whereas, there are solves made for stiff problems which do more work for each step but by doing this they are able to take larger step sizes and have more stability which is why they compute faster.

The code for this is very similar to the Task 3 one and thus I will include it at the very end under Appendix. Now to discuss a few things that I learned from this computer project is that there are many methods for numerical approximation because of the cost and efficiency of them. Having to run my own code I could see that for high numbers of iteration it does take a while to compute and can imagine how tedious and long this would be for a big system. I also learned that there are many ways that you can code a method and to not overcomplicate things at times as that can lead to further issues.

References

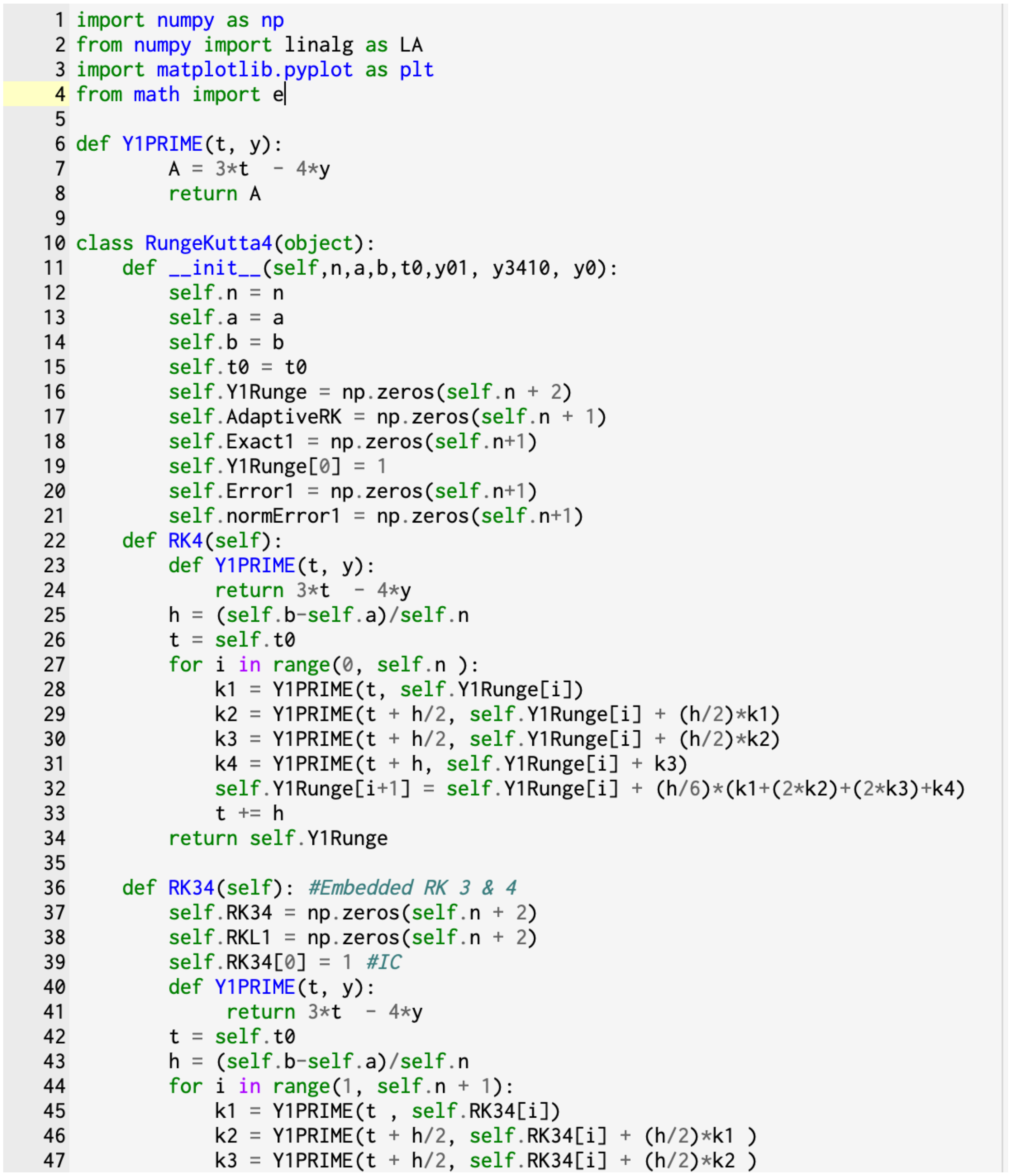
Burden, Richard L., et al. *Numerical Analysis*. Cengage Learning, 2016.

Runge-Kutta-Fehlberg Method (RKF45). maths.cnam.fr/IMG/pdf/RungeKuttaFehlbergProof.pdf.

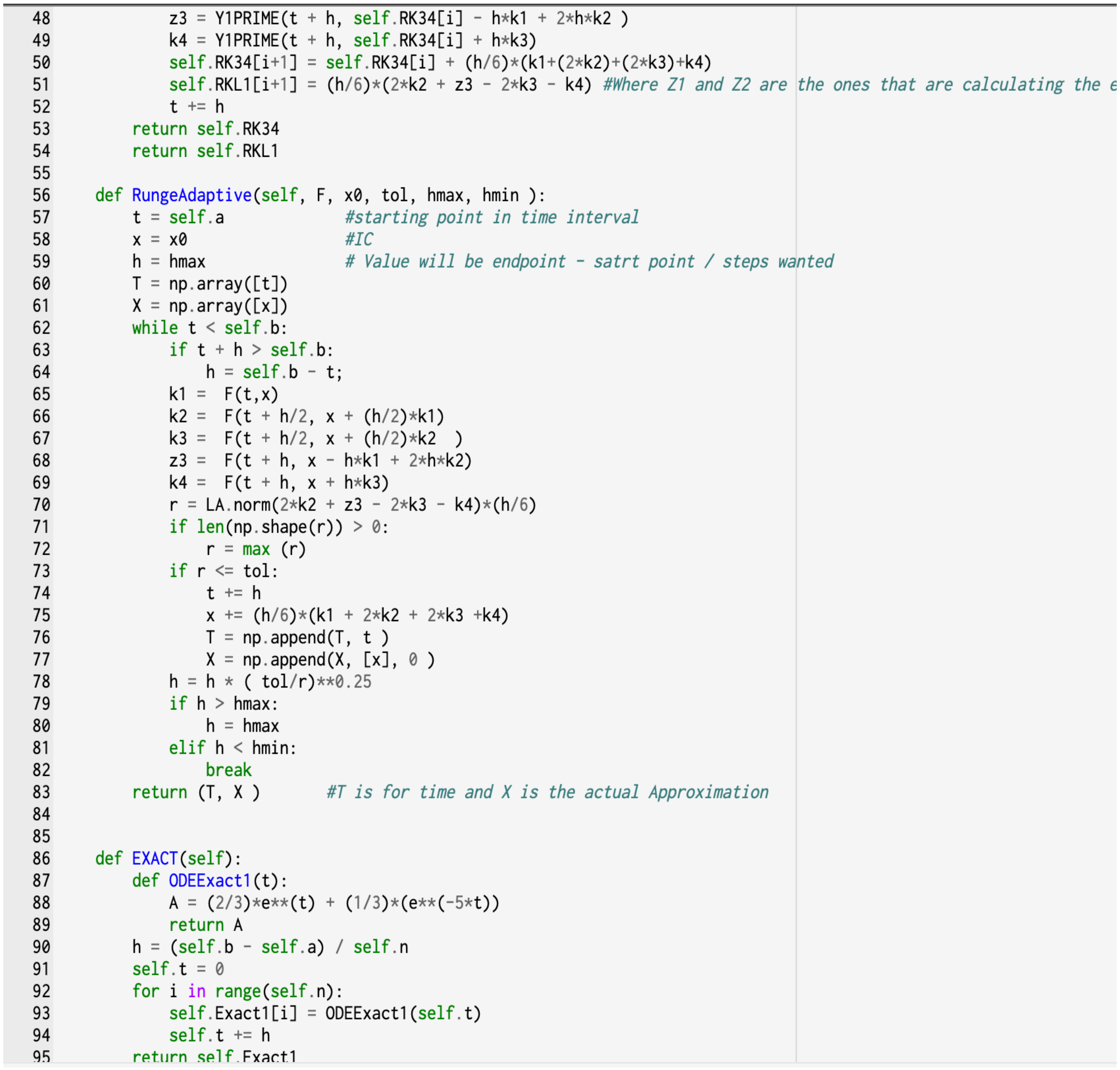
“Runge–Kutta Methods.” *Wikipedia*, Wikimedia Foundation, 8 Nov. 2019, en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods.

Appendix

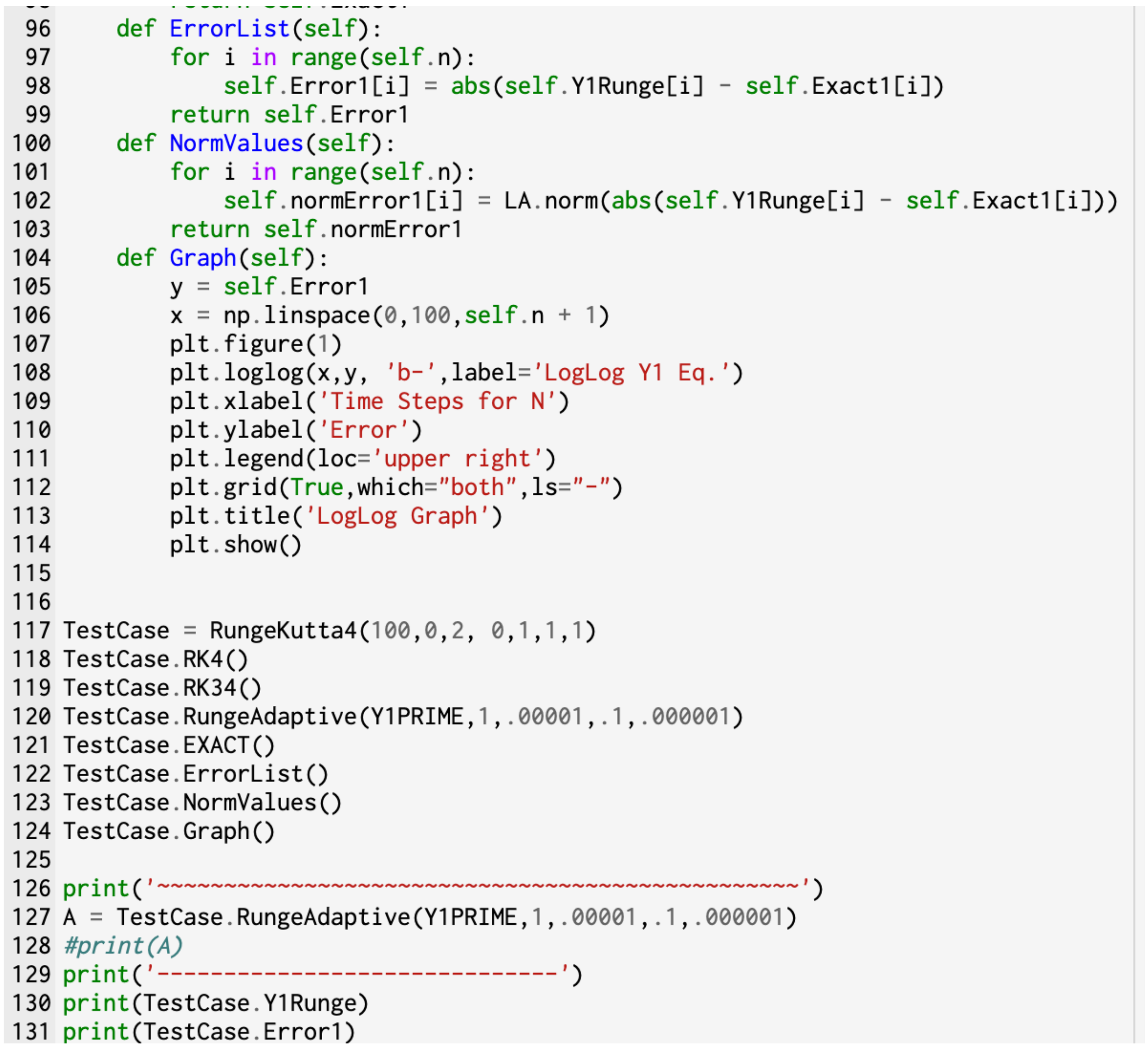
Task 2 Code:



```python
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
from math import e

def Y1PRIME(t, y):
        A = 3*t  - 4*y
        return A

class RungeKutta4(object):
    def __init__(self,n,a,b,t0,y01, y3410, y0):
        self.n = n
        self.a = a
        self.b = b
        self.t0 = t0
        self.Y1Runge = np.zeros(self.n + 2)
        self.AdaptiveRK = np.zeros(self.n + 1)
        self.Exact1 = np.zeros(self.n+1)
        self.Y1Runge[0] = 1
        self.Error1 = np.zeros(self.n+1)
        self.normError1 = np.zeros(self.n+1)
    def RK4(self):
        def Y1PRIME(t, y):
            return 3*t  - 4*y
        h = (self.b-self.a)/self.n
        t = self.t0
        for i in range(0, self.n ):
            k1 = Y1PRIME(t, self.Y1Runge[i])
            k2 = Y1PRIME(t + h/2, self.Y1Runge[i] + (h/2)*k1)
            k3 = Y1PRIME(t + h/2, self.Y1Runge[i] + (h/2)*k2)
            k4 = Y1PRIME(t + h, self.Y1Runge[i] + k3)
            self.Y1Runge[i+1] = self.Y1Runge[i] + (h/6)*(k1+(2*k2)+(2*k3)+k4)
            t += h
        return self.Y1Runge

    def RK34(self): #Embedded RK 3 & 4
        self.RK34 = np.zeros(self.n + 2)
        self.RKL1 = np.zeros(self.n + 2)
        self.RK34[0] = 1 #IC
        def Y1PRIME(t, y):
             return 3*t  - 4*y
        t = self.t0
        h = (self.b-self.a)/self.n
        for i in range(1, self.n + 1):
            k1 = Y1PRIME(t , self.RK34[i])
            k2 = Y1PRIME(t + h/2, self.RK34[i] + (h/2)*k1 )
            k3 = Y1PRIME(t + h/2, self.RK34[i] + (h/2)*k2 )
```

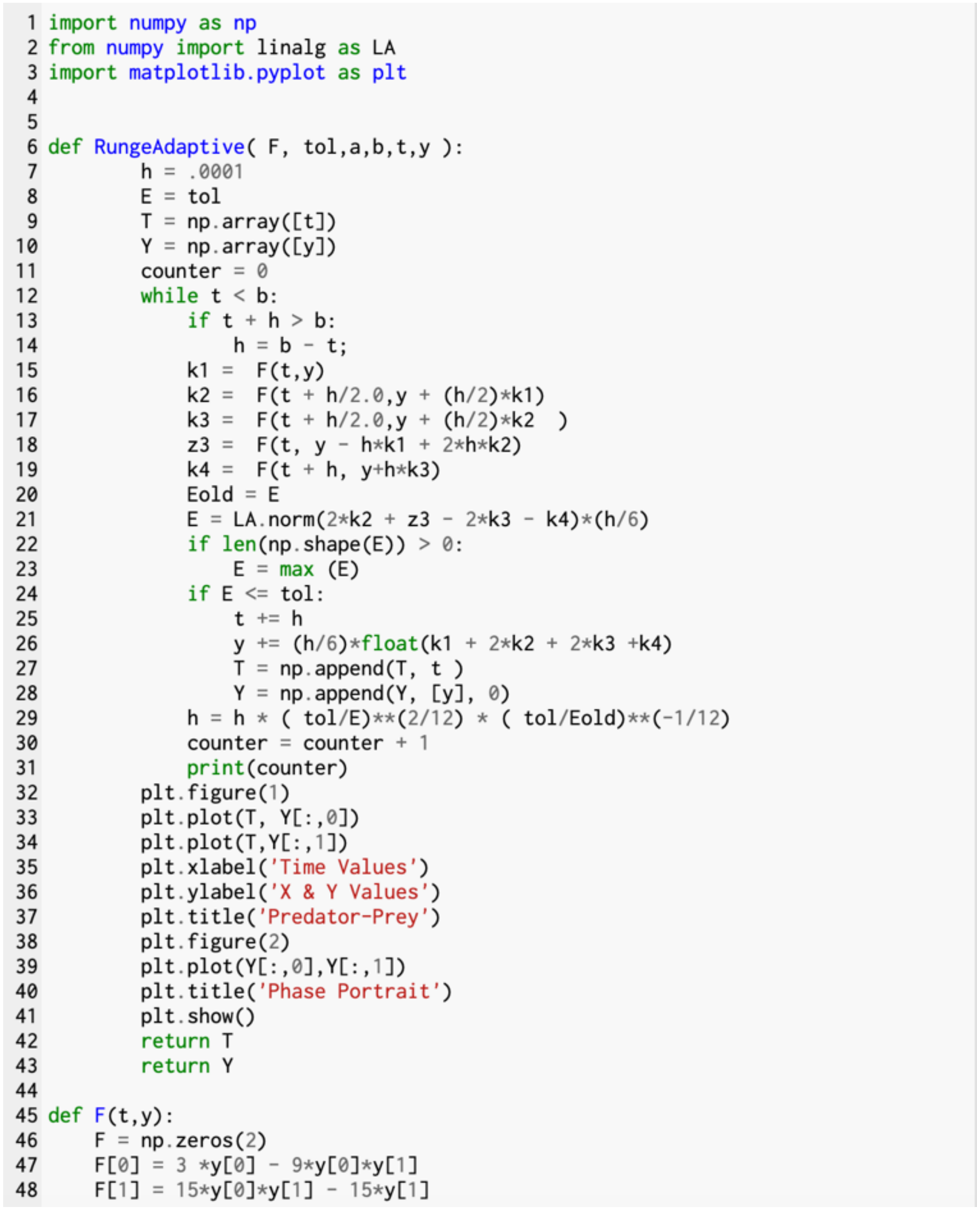


```python
            z3 = Y1PRIME(t + h, self.RK34[i] - h*k1 + 2*h*k2 )
            k4 = Y1PRIME(t + h, self.RK34[i] + h*k3)
            self.RK34[i+1] = self.RK34[i] + (h/6)*(k1+(2*k2)+(2*k3)+k4)
            self.RKL1[i+1] = (h/6)*(2*k2 + z3 - 2*k3 - k4) #Where Z1 and Z2 are the ones that are calculating the e
            t += h
        return self.RK34
        return self.RKL1

    def RungeAdaptive(self, F, x0, tol, hmax, hmin ):
        t = self.a             #starting point in time interval
        x = x0                 #IC
        h = hmax               # Value will be endpoint - satrt point / steps wanted
        T = np.array([t])
        X = np.array([x])
        while t < self.b:
            if t + h > self.b:
                h = self.b - t;
            k1 =  F(t,x)
            k2 =  F(t + h/2, x + (h/2)*k1)
            k3 =  F(t + h/2, x + (h/2)*k2  )
            z3 =  F(t + h, x - h*k1 + 2*h*k2)
            k4 =  F(t + h, x + h*k3)
            r = LA.norm(2*k2 + z3 - 2*k3 - k4)*(h/6)
            if len(np.shape(r)) > 0:
                r = max (r)
            if r <= tol:
                t += h
                x += (h/6)*(k1 + 2*k2 + 2*k3 +k4)
                T = np.append(T, t )
                X = np.append(X, [x], 0 )
            h = h * ( tol/r)**0.25
            if h > hmax:
                h = hmax
            elif h < hmin:
                break
        return (T, X )       #T is for time and X is the actual Approximation


    def EXACT(self):
        def ODEExact1(t):
            A = (2/3)*e**(t) + (1/3)*(e**(-5*t))
            return A
        h = (self.b - self.a) / self.n
        self.t = 0
        for i in range(self.n):
            self.Exact1[i] = ODEExact1(self.t)
            self.t += h
        return self.Exact1
```

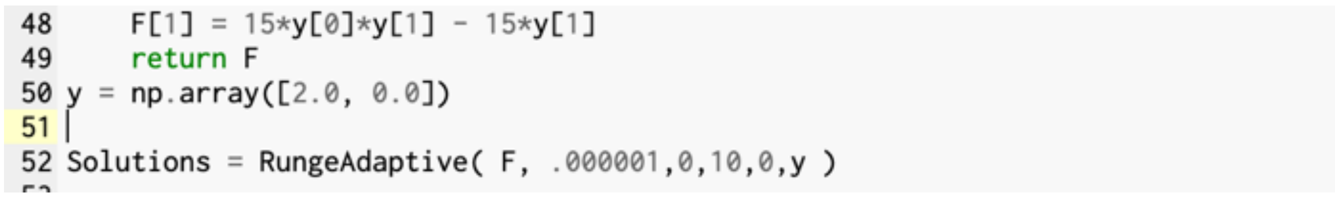


```
    def ErrorList(self):
        for i in range(self.n):
            self.Error1[i] = abs(self.Y1Runge[i] - self.Exact1[i])
        return self.Error1
    def NormValues(self):
        for i in range(self.n):
            self.normError1[i] = LA.norm(abs(self.Y1Runge[i] - self.Exact1[i]))
        return self.normError1
    def Graph(self):
        y = self.Error1
        x = np.linspace(0,100,self.n + 1)
        plt.figure(1)
        plt.loglog(x,y, 'b-',label='LogLog Y1 Eq.')
        plt.xlabel('Time Steps for N')
        plt.ylabel('Error')
        plt.legend(loc='upper right')
        plt.grid(True,which="both",ls="-")
        plt.title('LogLog Graph')
        plt.show()


TestCase = RungeKutta4(100,0,2, 0,1,1,1)
TestCase.RK4()
TestCase.RK34()
TestCase.RungeAdaptive(Y1PRIME,1,.00001,.1,.000001)
TestCase.EXACT()
TestCase.ErrorList()
TestCase.NormValues()
TestCase.Graph()

print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
A = TestCase.RungeAdaptive(Y1PRIME,1,.00001,.1,.000001)
#print(A)
print('------------------------------')
print(TestCase.Y1Runge)
print(TestCase.Error1)
```

Task 3 Code:

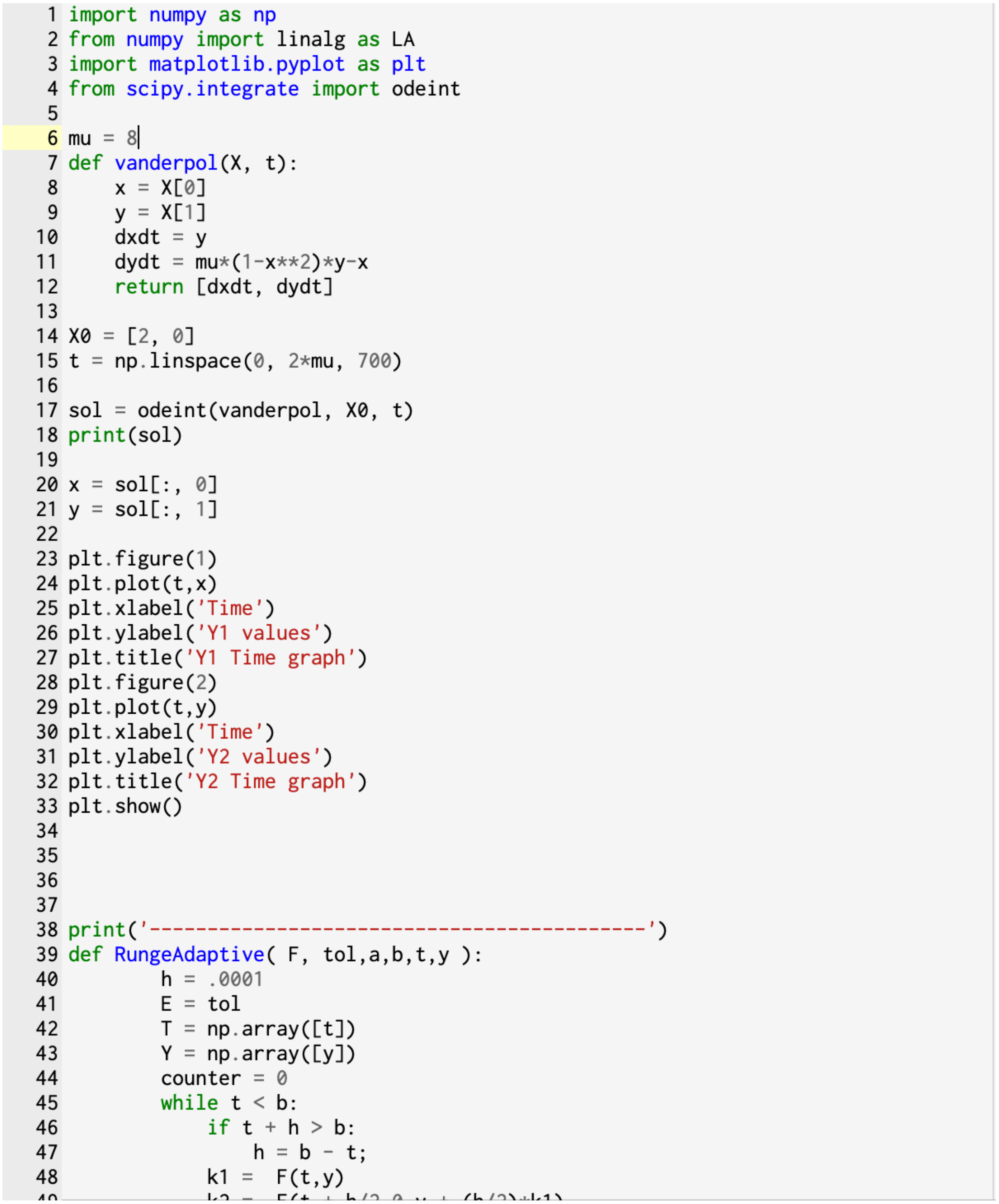


```python
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt


def RungeAdaptive( F, tol,a,b,t,y ):
        h = .0001
        E = tol
        T = np.array([t])
        Y = np.array([y])
        counter = 0
        while t < b:
            if t + h > b:
                h = b - t;
            k1 =  F(t,y)
            k2 =  F(t + h/2.0,y + (h/2)*k1)
            k3 =  F(t + h/2.0,y + (h/2)*k2  )
            z3 =  F(t, y - h*k1 + 2*h*k2)
            k4 =  F(t + h, y+h*k3)
            Eold = E
            E = LA.norm(2*k2 + z3 - 2*k3 - k4)*(h/6)
            if len(np.shape(E)) > 0:
                E = max (E)
            if E <= tol:
                t += h
                y += (h/6)*float(k1 + 2*k2 + 2*k3 +k4)
                T = np.append(T, t )
                Y = np.append(Y, [y], 0)
            h = h * ( tol/E)**(2/12) * ( tol/Eold)**(-1/12)
            counter = counter + 1
            print(counter)
        plt.figure(1)
        plt.plot(T, Y[:,0])
        plt.plot(T,Y[:,1])
        plt.xlabel('Time Values')
        plt.ylabel('X & Y Values')
        plt.title('Predator-Prey')
        plt.figure(2)
        plt.plot(Y[:,0],Y[:,1])
        plt.title('Phase Portrait')
        plt.show()
        return T
        return Y

def F(t,y):
    F = np.zeros(2)
    F[0] = 3 *y[0] - 9*y[0]*y[1]
    F[1] = 15*y[0]*y[1] - 15*y[1]
```



```
    F[1] = 15*y[0]*y[1] - 15*y[1]
    return F
y = np.array([2.0, 0.0])

Solutions = RungeAdaptive( F, .000001,0,10,0,y )
```

Task 4 Code:



```
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
from scipy.integrate import odeint

mu = 8
def vanderpol(X, t):
    x = X[0]
    y = X[1]
    dxdt = y
    dydt = mu*(1-x**2)*y-x
    return [dxdt, dydt]

X0 = [2, 0]
t = np.linspace(0, 2*mu, 700)

sol = odeint(vanderpol, X0, t)
print(sol)

x = sol[:, 0]
y = sol[:, 1]

plt.figure(1)
plt.plot(t,x)
plt.xlabel('Time')
plt.ylabel('Y1 values')
plt.title('Y1 Time graph')
plt.figure(2)
plt.plot(t,y)
plt.xlabel('Time')
plt.ylabel('Y2 values')
plt.title('Y2 Time graph')
plt.show()




print('--------------------------------------------')
def RungeAdaptive( F, tol,a,b,t,y ):
        h = .0001
        E = tol
        T = np.array([t])
        Y = np.array([y])
        counter = 0
        while t < b:
            if t + h > b:
                h = b - t;
            k1 =  F(t,y)
```

```
      k2 =  F(t + h/2.0,y + (h/2)*k1)
      k3 =  F(t + h/2.0,y + (h/2)*k2  )
      z3 =  F(t, y - h*k1 + 2*h*k2)
      k4 =  F(t + h, y+h*k3)
      Eold = E
      E = LA.norm(2*k2 + z3 - 2*k3 - k4)*(h/6)
      if len(np.shape(E)) > 0:
        E = max (E)
      if E <= tol:
        t += h
        y += (h/6)*(k1 + 2*k2 + 2*k3 +k4)
        T = np.append(T, t )
        Y = np.append(Y, [y], 0)
      h = h * ( tol/E)**(2/12) * ( tol/Eold)**(-1/12)
      counter = counter + 1
      print(counter)
    plt.figure(1)
    plt.plot(T, Y[:,0])
    plt.xlabel('Time')
    plt.ylabel('Y1 Values')
    plt.title('Y1 Time Graph')
    plt.figure(2)
    plt.plot(Y[:,0],Y[:,1])
    plt.xlabel('Y1 Values')
    plt.ylabel('Y2 Values')
    plt.title('Phase Portrait')
    plt.figure(3)
    plt.plot(T,Y[:,1])
    plt.xlabel('Time')
    plt.ylabel('Y2 Values')
    plt.title('Y2 Time Graph')
    plt.show()
    return T
    return Y


mu =  10
def F(t,y):
  F = np.zeros(2)
  # Ley y0 = y and y1 = y'
  F[0] = y[1]
  F[1] = mu*(1-y[0]**2)*y[1] - y[0]
  return F
y = np.array([2.0, 0.0])
Solutions = RungeAdaptive( F, .000001,0,.7*mu,0,y )
print(Solutions)
```