Ordinary Differential Equation Solvers with Python3

Paul Simmerling, *Electrical Engineering & Physics*, University of Connecticut

***Abstract*—Numerical methods for solving differential equations are essential for applications in science, engineering, and math. Within this report, Euler's Method, Second-Order, and Fourth-Order Runge-Kutta are presented alongside their application using Python3 and NumPy.**

I. Introduction

DIFFERENTIAL equations are the equations of the world. Classical physics, one of the most universally renowned theories, is see's it's roots in one differential equation, $\mathbf{F} = m\mathbf{a} \rightarrow \frac{d^2\mathbf{x}}{dt^2} = m\frac{d\mathbf{x}}{dt}$. Quantum mechanics is another example of a renowned physics theory based on differential equations, $i\hbar\frac{\partial}{\partial t}\mathbf{\Psi}(\mathbf{r},t) = \left[\frac{-\hbar^2}{2m}\nabla^2 + \mathbf{V}(\mathbf{r},t)\right]\mathbf{\Psi}(\mathbf{r},t)$. Both of these equations can be solved analytically for some scenarios, often degenerate into increasing complexity such that a computer must be used to approximate the solution. To make these approximations, three different methods are often used. Euler's Method, Second-Order Runge-Kutta, and Fourth-Order Runge-Kutta. Within this report, each method will be investigated and compared with each other.

II. Euler's Method

A. *First Order Derivation*

Euler's method is based on a Taylor series approximation of $\mathbf{y}(x)$.

$$\mathbf{y}(x) = \mathbf{y}(a) + \mathbf{y}'(a)(x-a) + \mathbf{y}''(a)\frac{(x-a)^2}{2} + \mathbf{y}'''(a)\frac{(x-a)^3}{6} + \mathbf{y}^{(4)}(a)\frac{(x-a)^4}{24} + \dots \quad (1)$$

Making the following transformations: $x \rightarrow x+h$ & $a \rightarrow x$.

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{y}'(x)((x+h)-x) + \mathbf{y}''(x)\frac{((x+h)-x)^2}{2} + \mathbf{y}'''(x)\frac{((x+h)-x)^3}{6} + \mathbf{y}^{(4)}(x)\frac{((x+h)-x)^4}{24} + \dots \quad (2)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{y}'(x)h + \mathbf{y}''(x)\frac{h^2}{2} + \mathbf{y}'''(x)\frac{h^3}{6} + \mathbf{y}^{(4)}(x)\frac{h^4}{24} + \dots \quad (3)$$

$$\mathbf{y}(x+h) = \sum_{n=0}^{\infty}\mathbf{y}^{(n)}(x)\frac{h^n}{n!} \quad (4)$$

For Euler's method, the first order approximation is used.

$$\mathbf{y}(x+h) \approx \mathbf{y}(x) + \mathbf{y}'(x)h + O(h^2) \quad (5)$$

Equation 5 lends itself to the discretization for use in Python codes.

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h \cdot \mathbf{y}'_i \quad (6)$$

B.

Python Code Using equation 6, a Python code can be created using Numpy.

```
#y_{i+1}=y_i+h*y'_i=y_i+h*F(x,y)
def euler(F, x, y, xStop, h):
  X = np.array(x)
  Y = np.array(y)
  while x < xStop:
    #Avoid overstepping
    h = min(h, xStop - x)
    #y'=F(x,y)
    y = y + h*F(x,y)#Use equation 6
    #to calculate the next y
    x = x + h
    X = np.append(X, x)
    Y = np.append(Y, y)
  return X,Y
```

III. Runge-Kutta Methods

A. *Second-Order Derivation*

To begin, a solution of the form seen in equation 7 can be assumed.

$$\mathbf{y}(x+h) = \mathbf{y}(x) + c_0\mathbf{F}(x,\mathbf{y})h + c_1\mathbf{F}[x+ph, \mathbf{y}+qh\mathbf{F}(x,\mathbf{y})]h \quad (7)$$

The solution to the parameters c_0, c_1, p, and q will be found by matching equation 7 to the Taylor series shown in equation 4.

$$\begin{aligned}\mathbf{y}(x+h) &= \mathbf{y}(x) + \mathbf{y}'(x)h + \mathbf{y}''(x)\frac{h^2}{2} \\ &= \mathbf{y}(x) + \mathbf{F}(x,\mathbf{y})h + \mathbf{F}'(x,\mathbf{y})\frac{h^2}{2} \quad (8)\end{aligned}$$

with,

$$\begin{aligned}\mathbf{F}'(x,\mathbf{y}) &= \frac{\partial \mathbf{F}}{\partial x} + \sum_{i=0}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} y_i' \\ &= \frac{\partial \mathbf{F}}{\partial x} + \sum_{i=0}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y}) \quad (9)\end{aligned}$$

where n is the number of first-order equations. Substituting equation 9 back into equation 9,

$$\begin{aligned}\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{F}(x,\mathbf{y})h + \\ \frac{1}{2}\left(\frac{\partial \mathbf{F}}{\partial x} + \sum_{i=0}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y})\right) h^2 \quad (10)\end{aligned}$$

Applying a Taylor series to the final term in equation 7,

$$\begin{aligned}\mathbf{F}\left[x+ph, \mathbf{y} + qh\mathbf{F}(x,\mathbf{y})\right] = \mathbf{F}(x,\mathbf{y}) \\ + \frac{\partial \mathbf{F}}{\partial x} ph + qh \sum_{i=1}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y}) \quad (11)\end{aligned}$$

Substituting equation 11 into equation 7,

$$\begin{aligned}\mathbf{y}(x+h) = \mathbf{y}(x) + (c_0 + c_1)\mathbf{F}(x,\mathbf{y})h \\ + c_1 \left[\frac{\partial \mathbf{F}}{\partial x} ph + qh \sum_{i=1}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y})\right] h \quad (12)\end{aligned}$$

Equations 10 and 12 are found to be identical if $c_0 + c_1 = 1$, $c_1 p = \frac{1}{2}$, and $c_1 q = \frac{1}{2}$. These constants can be arbitrarily chosen. Moving forward the Modified Euler's method will be used where $c_0 = 0$, $c_1 = 1$, $p = 1/2$, and $q = 1/2$. Substituting back into equation 7 yields,

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{F}\left[x + \frac{h}{2}, \mathbf{y} + \frac{h}{2}\mathbf{F}(x,\mathbf{y})\right] h \quad (13)$$

The integration of differential equation can be found using the following set of formulas,

$$\mathbf{K}_0 = h\mathbf{F}(x,\mathbf{y}) \quad (14)$$

$$\mathbf{K}_1 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_0}{2}\right) \quad (15)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{K}_1 \quad (16)$$

These equations yield themselves to the following discretization,

$$\mathbf{K}_0 = h\mathbf{F}_i(\mathbf{y}_i) \quad (17)$$

$$\mathbf{K}_1 = h\mathbf{F}_{i+1/2}\left(\mathbf{y}_i + \frac{\mathbf{K}_0}{2}\right) \quad (18)$$

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \mathbf{K}_1 \quad (19)$$

This discretization can be seen in the Python3 code below,

```
#y(x+h)=y(x)+F[x+h/2+y+h/2*F(x,y)]h
def RK2(F, x, y, xStop, h):
  X = np.array(x)
  Y = np.array(y)
  while x < xStop:
    h = min(h, xStop - x)
    #dont overstep
    k0 = h*F(x,y)#eq 14
    k1 = h*F(x + h/2, y + k0/2)
    #eq 15
    y = y + k1# eq 16
    x = x + h#increment x
    X = np.append(X, x)
    Y = np.append(Y, y)
  return X,Y
```

B. *Fourth-Order Runge-Kutta*

The fourth-order derivation is very long but follows the same steps as the derivation for the second-order Runge-Kutta. Eventually the solution results in the following set of equations which can be solved to determine the integration of **y**.

$$\mathbf{K}_0 = h\mathbf{F}(x,\mathbf{y}) \quad (20)$$

$$\mathbf{K}_1 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_0}{2}\right) \quad (21)$$

$$\mathbf{K}_2 = h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_1}{2}\right) \quad (22)$$

$$\mathbf{K}_3 = h\mathbf{F}\left(x + h, \mathbf{y} + \mathbf{K}_2\right) \quad (23)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \frac{1}{6}\left(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3\right) \quad (24)$$

which lends itself to the following discrete form,

$$\mathbf{K}_0 = h\mathbf{F}_i(\mathbf{y}_i) \tag{25}$$

$$\mathbf{K}_1 = h\mathbf{F}_{i+1/2}\left(\mathbf{y}_i + \frac{\mathbf{K}_0}{2}\right) \tag{26}$$

$$\mathbf{K}_2 = h\mathbf{F}_{i+1/2}\left(\mathbf{y}_i + \frac{\mathbf{K}_1}{2}\right) \tag{27}$$

$$\mathbf{K}_3 = h\mathbf{F}_{i+1}\left(\mathbf{y}_i + \mathbf{K}_2\right) \tag{28}$$

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{1}{6}\left(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3\right) \tag{29}$$

The Python3 code can be seen below,

```
def RK4(F, x, y, xStop, h):
  X = np.array(x)
  Y = np.array(y)
  while x < xStop:
    h = min(h, xStop − x)#dont
    #overstep
    k0 = h*F(x,y)#eq 20
    k1 = h*F(x + h/2, y + k0/2)
    #eq 21
    k2 = h*F(x + h/2, y + k1/2)
    #eq 22
    k3 = h*F(x + h, y + k2)
    #eq 23
    y = y + (k0+2*k1+2*k2+k3)/6
    #eq 24
    x = x + h
    #increment x for next iteration
    X = np.append(X, x)
    Y = np.append(Y, y)
  return X,Y
```

IV. Testing Code

These codes can be tested on differential equations to show they provide accurate approximations. For example, $y'(x) = F(x, y) = \sin(y(x))$ with the initial value of $y(0) = 1$, range of $x = [0, 2]$, and step size $h = 0.1$.

```
def F(x,y):
                return sin(y)
x, xStop = 0.0, 2.0
y, h = 1.0, 0.1
XE,YE = euler(F,x,y,xStop,h)
X2,Y2 = RK2(F,x,y,xStop,h)
X4,Y4 = RK4(F,x,y,xStop,h)
```

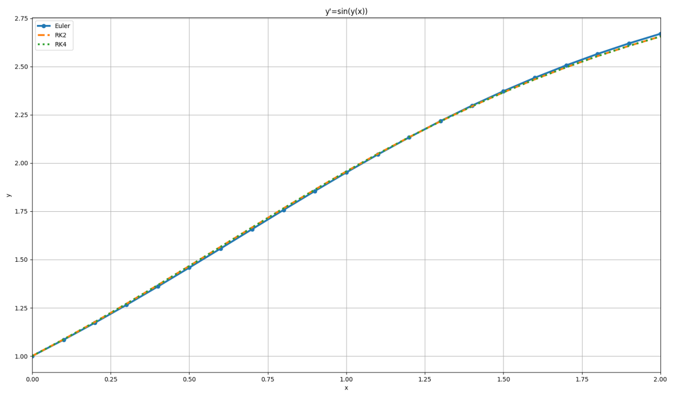

Fig. 1. Various ODE solutions.

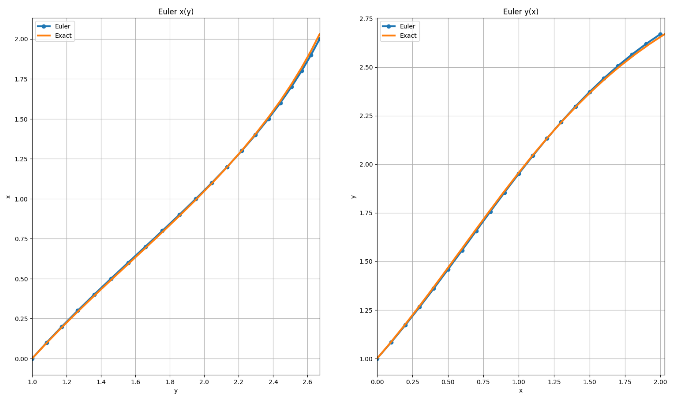

Fig. 2. Euler Solution plotted alongside the exact solution $x(y)$.

A graph of the solution can be see in figure 1.

From this figure, it is evident that the three solutions differ slightly. The exact solution to $y'(x) = \sin(y(x))$ can be found in the inverse form of,

$$x(y) = \ln\left(\csc(y) - \cot(y)\right) + 0.604582 \tag{30}$$

This exact solution can be plotted next to each of the approximated solutions for Euler, Runge-Kutta2, and Runge-Kutta4. These can be seen in figures 2, 3, and 4 respectively. From these figures, it is evident that Euler's method diverges from the correct solution at the upper boundary. A graphical view of the errors can be seen in figure 5. From this figure, the error for Euler's method is seen to be very far from 0 for almost all position except for 0 and 1.25 where the error inverts from over-estimating to under-estimation. Runge-Kutta2 can be seen to have small deviations from the exact solution whereas Runge-Kutta4 appears to be almost perfectly exact for all x. The error squared on a log scale can be seen in figure 6. The errors are much better understood using figure 6. It is evident that

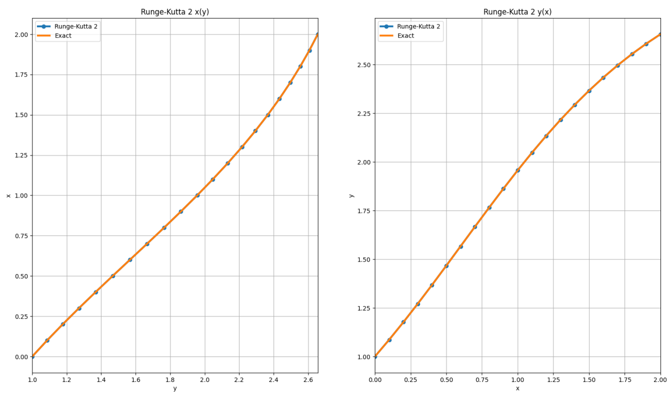

Fig. 3. Second-Order Runge-Kutta Solution plotted alongside the exact solution $x(y)$.

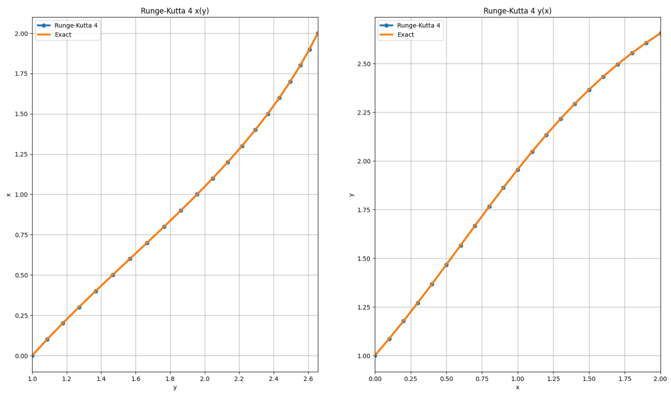

Fig. 4. Fourth-Order Runge-Kutta Solution plotted alongside the exact solution $x(y)$.

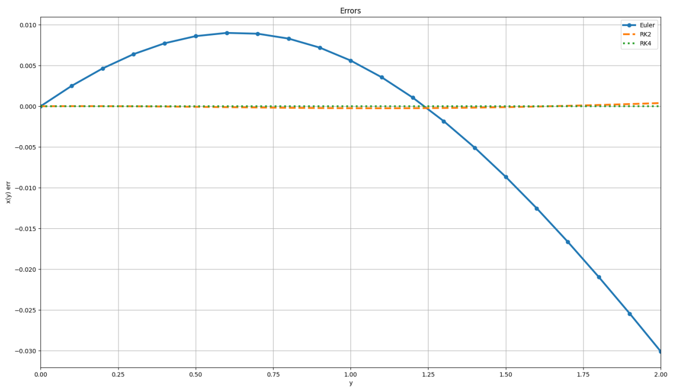

Fig. 5. Associated errors for Euler, Runge-Kutta2, and Runge-Kutta4 with respect to the exact solution $x(y)$.

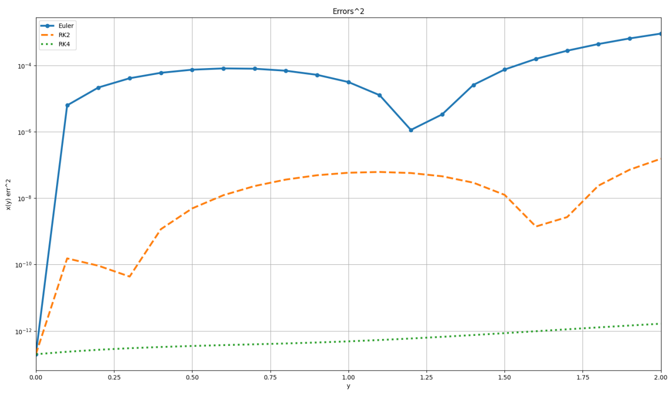

Fig. 6. Associated errors squared for Euler, Runge-Kutta2, and Runge-Kutta4 with respect to the exact solution $x(y)$.

Euler is significantly worse than the Runge-Kutta methods. Fourth-order Runge-Kutta is also seen to be very accurate with an error that is very slowly varying with respect to the other methods. Summing the errors up for each method gives an additional measure of accuracy. This error can be seen in table 1.

$$\chi^2 = \sum_{i=0}^{n} (\hat{y}_i - y_i)^2 \tag{31}$$

$$\sqrt{\chi^2} = \sqrt{\sum_{i=0}^{n} (\hat{y}_i - y_i)^2} \tag{32}$$

$$n = \text{ceil}\left[\frac{x_{max} - x_{min}}{h}\right] \tag{33}$$

From table 1, it is evident that each successive

Method	χ^2	$\sqrt{\chi^2}$
Euler's Method	3.058265×10^{-3}	5.530158×10^{-2}
2nd Order Runge-Kutta	6.451449×10^{-7}	8.032091×10^{-4}
4th Order Runge-Kutta	1.369687×10^{-11}	3.700928×10^{-6}

TABLE I
ERRORS ASSOCIATED WITH EACH METHOD.

method decreases the error($\sqrt{\chi^2}$) associated by about a factor of $\frac{1}{100}$.

V. Conclusion

All three methods did exceptionally well in solving the test differential equation, but it was evident that the fourth-order Runge-Kutta was significantly more precise. Euler's method, while simple to derive and implement, is rarely used due to its inaccuracy.

Higher-order Euler's Method could be used but are computationally inefficient and will still result in a large error. The Runge-Kutta methods, which were also based on the Taylor Series, were much more accurate at almost the same computational complexity level. The Fourth-Order Runge-Kutta, often called *the Runge-Kutta Method*, was shown to be computationally efficient and highly accurate. Even though the step-size was large, the computation error was on the order of $10^{-6} \approx h^6$. As a result of this accuracy, Runge-Kutta is often the de facto method used for numerically solving differential equations.

APPENDIX
CODE FOR PYTHON3 ODE NUMERICAL SOLUTIONS

```python
import numpy as np
from math import *
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

def graph(XE,YE,XRK2,YRK2,XRK4,YRK4,title,xlab='x',ylab='y',\
        clf=True,log=False):
    if clf:
        plt.clf()
    fig, ax = plt.subplots()
    #plt.plot(XX,YX, '-',linewidth=3)
    ax.plot(XE,YE, 'o-',linewidth=3)
    ax.plot(XRK2,YRK2, '--',linewidth=3)
    ax.plot(XRK4,YRK4, ':',linewidth=3)
    plt.title(title)#+" "+ylab+"("+xlab+")")
    plt.xlabel(xlab); plt.ylabel(ylab); ylab = ylab +'\''
    xmin = min(min(XE),min(XRK2),min(XRK2))
    xmax = max(max(XE),max(XRK2),max(XRK2))
    plt.xlim(xmin,xmax)
    plt.legend(["Euler","RK2","RK4"])
    if log:
        ax.set_yscale("log")
        #locmaj = ticker.LogLocator(base=10,numticks=15)
        #ax.yaxis.set_major_locator(locmaj)
        #locmin = ticker.LogLocator(base=10.0,subs=(0.2,0.4,0.6,0.8),
        #    numticks=15)
        #ax.yaxis.set_minor_locator(locmin)
        #ax.yaxis.set_minor_formatter(ticker.NullFormatter())

    plt.grid(True, which='both')
    plt.gcf().set_size_inches(18.5, 10.5)
    plt.savefig("figures/"+title+'.png',bbox_inches='tight')


def graphX(XX,YX,X,Y,title,xlab='x',ylab='y',clf=True):
    if clf:
        plt.clf()
    for i in range(2):
        plt.subplot(1,2,i+1)
        plt.plot([X,Y][i],[Y,X][i],'o-',linewidth=3)
        plt.plot([XX,YX][i],[YX,XX][i],linewidth=3)
        plt.title(title+" "+ylab+"("+xlab+")")
        plt.grid(True)
        plt.xlabel(xlab); plt.ylabel(ylab);
        plt.xlim(min(min([XX,YX][i]),min([X,Y][i])),max(max([XX,YX][i]),
            max([X,Y][i])))
        plt.legend([title,"Exact"])
        plt.gcf().set_size_inches(18.5, 10.5)
        plt.savefig("figures/"+title+'.png',bbox_inches='tight')
        xlab,ylab = ylab,xlab


#Question 1
#Part A
#y_{i+1}=y_i+h*y'_i=y_i+h*F(x,y)
def euler(F, x, y, xStop, h):
    X = np.array(x)
    Y = np.array(y)
    while x < xStop:
      #Avoid overstepping
        h = min(h, xStop - x)
        #y'=F(x,y)
        y = y + h*F(x,y)#Use equation 6
        #to calculate the next y
        x = x + h
        X = np.append(X, x)
        Y = np.append(Y, y)
    return X,Y

#y(x+h)=y(x)+F[x+h/2+y+h/2*F(x,y)]h
def RK2(F, x, y, xStop, h):
    X = np.array(x)
    Y = np.array(y)
    while x < xStop:
        h = min(h, xStop - x)#dont overstep
        k0 = h*F(x,y)#eq 14
        k1 = h*F(x + h/2, y + k0/2)#eq 15
        y = y + k1# eq 16
        x = x + h#increment x
        X = np.append(X, x)
        Y = np.append(Y, y)
    return X,Y

def RK4(F, x, y, xStop, h):
    X = np.array(x)
    Y = np.array(y)
    while x < xStop:
        h = min(h, xStop - x)#dont overstep
        k0 = h*F(x,y)#eq 20
        k1 = h*F(x + h/2, y + k0/2)#eq 21
        k2 = h*F(x + h/2, y + k1/2)#eq 22
        k3 = h*F(x + h, y + k2)#eq 23
        y = y + (k0+2*k1+2*k2+k3)/6#eq 24
        x = x + h#increment x for next iteration
        X = np.append(X, x)
        Y = np.append(Y, y)
    return X,Y
```

```
def Q2A():
    def F(x,y):
        return sin(y)
    x, xStop = 0.0, 2.0
    y, h = 1.0, 0.1
    XE,YE = euler(F,x,y,xStop,h)
    X2,Y2 = RK2(F,x,y,xStop,h)
    X4,Y4 = RK4(F,x,y,xStop,h)
    graph(XE, YE, X2, Y2, X4, Y4, "y'=sin(y(x))")
    return XE,YE,Y2,Y4

def Q2B(XE,YE,Y2,Y4):
    def X(x,y):
        return np.log(1/np.sin(y)-1/np.tan(y)) + 0.604582
    graphX(YE,X(YE,YE),YE,XE,"Euler",'y','x')
    graphX(Y2,X(Y2,Y2),Y2,XE,"Runge-Kutta 2",'y','x')
    graphX(Y4,X(Y4,Y4),Y4,XE,"Runge-Kutta 4",'y','x')
    return

def Q2C(XE,YE,Y2,Y4):
    def X(x,y):
        return np.log(1/np.sin(y)-1/np.tan(y)) + 0.604582
    XXE = X(YE,YE)
    XX2 = X(Y2,Y2)
    XX4 = X(Y4,Y4)
    XEerr = XE - XXE
    X2err = XE - XX2
    X4err = XE - XX4
    graph(XE,XEerr,XE,X2err,XE,X4err,"Errors",'y','x(y) err')
    #graph(YE,XEerr,Y2,X2err,Y4,X4err,"Errors",'y','x(y) err')

    XEerr2 = XEerr**2
    X2err2 = X2err**2
    X4err2 = X4err**2
    print("Euler: ",np.sum(XEerr2))
    print("RK2: ",np.sum(X2err2))
    print("Rk4: ",np.sum(X4err2))
    print("Euler: ",np.sum(XEerr2)**.5)
    print("RK2: ",np.sum(X2err2)**.5)
    print("Rk4: ",np.sum(X4err2)**.5)
    graph(XE,XEerr2,XE,X2err2,XE,X4err2,"Errors^2",'y','x(y) err^2',
        log=True)
    #graph(YE,XEerr2,Y2,X2err2,Y4,X4err2,"Errors^2",'y','x(y) err^2')


    return
if __name__ == "__main__":
    XE,YE,Y2,Y4 = Q2A()
    Q2B(XE,YE,Y2,Y4)
    Q2C(XE,YE,Y2,Y4)
```