EXERCISE 2.1: INTEGRATION USING SIMPSON'S RULE

a) Starting with Example 2.3 in the notes, add a function to perform Simpson's rule integration.

Code :

```
import numpy as np
def Simpson(function, a, b, N):
    '''
    Parameters
    ----------
    function : function of a single variable
    a, b     : interval of integration [a,b]
    N        : number of subintervals of [a,b]


    Returns
    ----------
    Approximation of the integral of f(x) from a to b using the
        Simpson rule with N subintervals of equal length.

    Examples
    ----------
    >>> Simpson(lambda x: x**4 - 2*x + 1, 0.0, 2.0, 10)
    4.50656
    >>> Simpson(lambda x: x**4 - 2*x + 1, 0.0, 2.0, 1000)
    4.40001066667
    >>> Simpson(lambda x: np.sin(x), 0.0, 2.0, 10)
    1.4161463644981664
    '''
    h = (b - a)/N
    s = 0.5*function(a) + 0.5*function(b)
    for k in range(1, N):
        s += function(a+k*h)
    return h*s
```

b) Use the integral $\int_0^2 (x^4 - 2x + 1)dx$, for which the exact value is 4.4, to compare the accuracy of the trapezium rule and Simpsons rule for the same values of N.

Code implementation for Trapezium's rule:

```
def Trapezium(function, a, b, N):
    '''
    Parameters
    ----------
    function : function of a single variable
    a , b    : interval of integration [a,b]
    N        : number of subintervals of [a,b]

    Returns
    -------
    Approximation of the integral of f(x) from a to b using the
        Trapezium rule with N subintervals of equal length.

    Examples
    --------
    >>> Trapezium(lambda x: x**4 - 2*x + 1, 0.0, 2.0, 10)
    4.50656
    >>> Trapezium(lambda x: x**4 - 2*x + 1, 0.0, 2.0, 1000)
    4.4000106666656
    '''
    x = np.linspace(a, b, N+1)
    y = function(x)
    y_right = y[1:] # right endpoints
    y_left = y[:-1] # left endpoints
    dx = (b - a)/N
    T = (dx/2) * np.sum(y_right + y_left)
    return T
```

Conclusion : when increasing N(number of subintervals of [a,b]), in the case of quadratic functions, the Simpsons method gave the best approximation and the Trapezoidal provided the worst.

EXERCISE 2.2 MAXWELL SPEED DISTRIBUTION

a) Write a function that returns a value of f(v) from Maxwell Speed distribution, given an input of v, m and T

Code :

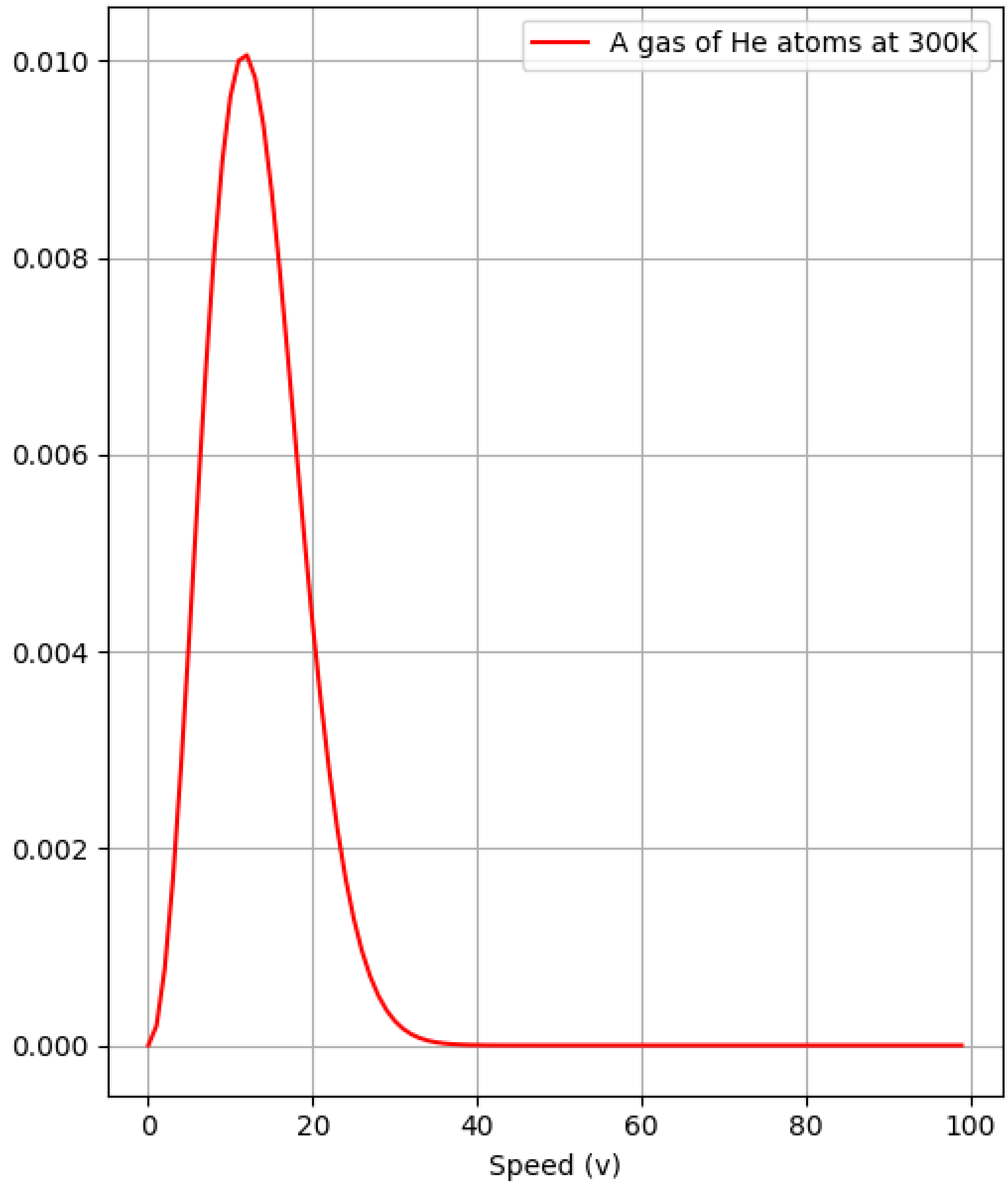
```
import math
import numpy as np
import matplotlib.pyplot as plt

# EXERCISE 2.2 MAXWELL SPEED DISTRIBUTION

# Constant
kB = 1.38e-23 # J/K^-1


# a)
def MaxwellSpeedDistribution(v, m, T):
    return 4*np.pi*((m/(2*np.pi*kB*T))**(1.5))*v*v*math.exp(-(m*v*v)/(2*kB*T))
```

b) Calculate and plot f(v) at the temperatures of 300K an 1000K for:
   (i) A gas of He atoms, m = 6.65×10^{-27} kg
   Use above function, we have :
   $f(v) = 1.6243 \times 10^{-9}\, v^2 \exp(-8.0314 \times 10^{-7}\, v^2)$

Code :

```
# (i)
v = np.arange(0, 100)
def f1(v):
    return 1.6243*np.exp(-9)*(v**2)*np.exp(-8.0314*np.exp(-7)*v**2)
plt.subplot(1, 2, 1)
plt.plot(v, f1(v), 'r', label = 'A gas of He atoms at 300K')
plt.xlabel('Speed (v)')
plt.grid()
plt.legend()
```

Graph :



A gas of He atoms at 300K
0.010
0.008
0.006
0.004
0.002
0.000
0
20
40
60
80
100
Speed (v)

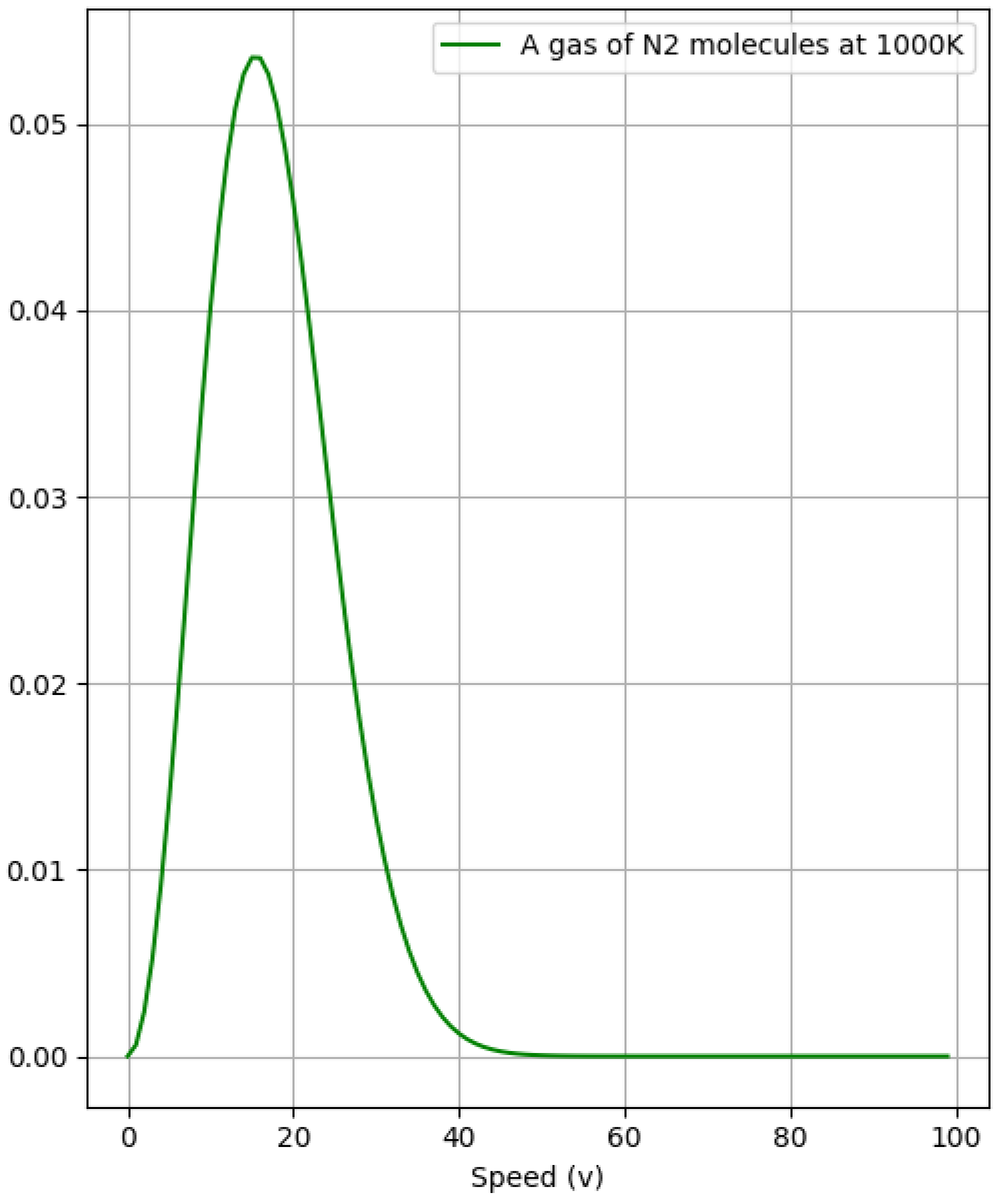
(ii) A gas of N_2 molecules, $m = 4.65 \times 10^{-26}$ kg

$f(v) = 4.9352 \times 10^{-9}\, v^2 \exp(-1.6848 \times 10^{-6}\, v^2)$

Code :

```
# (ii)
def f2(v):
    return 4.9352*np.exp(-9)*(v**2)*np.exp(-1.6848*np.exp(-6)*v**2)
plt.subplot(1, 2, 2)
plt.plot(v, f2(v), 'g', label = 'A gas of N2 molecules at 1000K')
plt.xlabel('Speed (v)')
plt.grid()
plt.legend()
```

Graph :



c) Calculate the probability of molecule having a speed between v_1 and v_2 using

(i) Simpson's rule (Exercis 2.1)

A gas of He atoms, m = 6.65 x 10^{-27} kg

⇨ $f(v) = 1.6243 \times 10^{-9} v^2 \exp(-8.0314 \times 10^{-7} v^2)$

⇨ $P(15 < v < 25) = 0.045484531105908416$

A gas of N_2 molecules, m = 4.65 x 10^{-26} kg

⇨ $f(v) = 4.9352 \times 10^{-9} v^2 \exp(-1.6848 \times 10^{-6} v^2)$

⇨ $P(15 < v < 25) = 0.44146966504554$

Code :

```
# (i) Simpson's rule
from Exercise_2_1 import *
print(Simpson(f1, 15.0, 25.0, 1000))
print(Simpson(f2, 15.0, 25.0, 1000))
```

Result :

```
administrator@PC00025:~/Desktop/RemoteWorplace/Project-ID-30928060$ python Exercise_2_2.py
0.045484531105908416
0.44146966504554
```

(ii) Scipy.integrate.quad ()

$P(15 < v < 25) = 0.04548452792973004$

$P(15 < v < 25) = 0.44146969867934865$

Code :

```
# (ii) scipy.integrate.quad()
from scipy import integrate
print(integrate.quad(f1, 15, 25)[0])
print(integrate.quad(f2, 15, 25)[0])
plt.show()
```

Result :

```
administrator@PC00025:~/Desktop/RemoteWorplace/Project-ID-30928060$ python Exercise_2_2.py
0.045484531105908416
0.44146966504554
0.0454845279297
0.441469698679
```

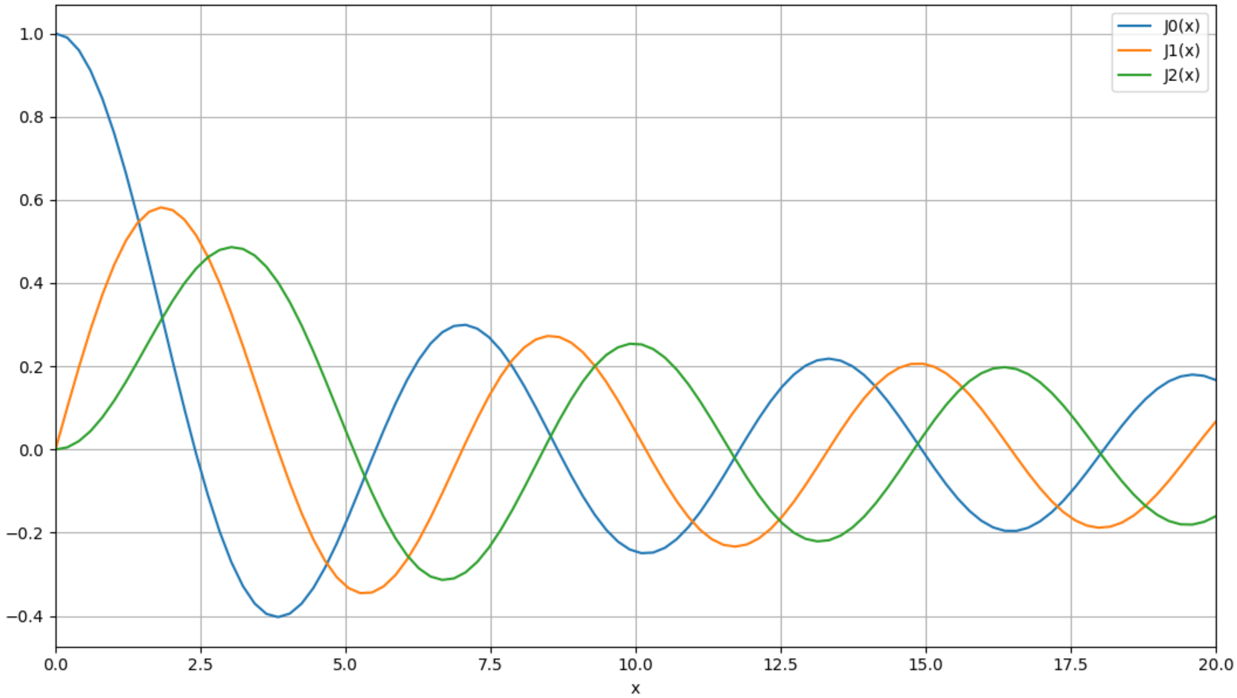
EXERCISE 2.4: DIFFRACTION LIMIT OF A TELESCOPE

Write a python function J(m,x) that calculates the value of Jm(x) from Eqn. (5)

Code :

```
from scipy.integrate import quad
from numpy import sqrt, sin, cos, pi
import numpy as np
import math
import matplotlib.pyplot as plt

def Bessel(theta, m, x):
    return (1/pi)*(cos(m*theta - x*sin(theta)))
def J(m, x):
    return quad(Bessel, 0, pi, args=(m, x))[0]

#print J(1.0, 5.0)

# a)
x = np.linspace(0, 20, 100)
J0 = [J(0, _) for _ in x]
J1 = [J(1, _) for _ in x]
J2 = [J(2, _) for _ in x]
plt.xlim(0, 20)
plt.plot(x, J0, label = 'J0(x)')
plt.plot(x, J1, label = 'J1(x)')
plt.plot(x, J2, label = 'J2(x)')
plt.grid()
plt.legend()
plt.show()
```

a) Plot $J_0(x)$, $J_1(x)$ and $J_2(x)$ over the range from x = 0 to x = 20



EXERCISE 2.5: ERRORS ON INTEGRALS AND ADAPTIVE INTEGRATION

a) Write a user defined function trapezium_adaptive(f, a, b, eta, *args)

Code :

```
from scipy import integrate
from Exercise_2_1 import *


def trapezium_adaptive(f, a, b, eta, *args):
    return Trapezium(f, a, b, eta), abs(integrate.quad(f, a, b)[0] - Trapezium(f, a, b, eta))
I = trapezium_adaptive(lambda x: x**2, 0.0, 2.0, 5)
print 'I = trapezium_adaptive(lambda x: x**2, 0.0, 2.0, 5) = ', I
i, e = trapezium_adaptive(lambda x: x**2, 0.0, 2.0, 5)
print 'i,e = trapezium_adaptive(lambda x: x**2, 0.0, 2.0, 5), i = ',i, ' e = ', e
```

Result :

```
administrator@PC00025:~/Desktop/RemoteWorplace/Project-ID-30928060$ python Exercise_2_5.py
I = trapezium_adaptive(lambda x: x**2, 0.0, 2.0, 5) =  (2.7200000000000006, 0.05333333333333368)
i,e = trapezium_adaptive(lambda x: x**2, 0.0, 2.0, 5), i =  2.7200000000000006  e =  0.05333333333333368
```

b) Write a user-defined function simpson_adaptive(f, a, b, eta, *args)

Code :

```
def simpson_adaptive(f, a, b, eta, *args):
    return Simpson(f, a, b, eta), abs(integrate.quad(f, a, b)[0] - Simpson(f, a, b, eta))

II = simpson_adaptive(lambda x: x**2, 0.0, 2.0, 5)
print 'I = simpson_adaptive(lambda x: x**2, 0.0, 2.0, 5) = ', II
ii, ee = simpson_adaptive(lambda x: x**2, 0.0, 2.0, 5)
print 'i,e = simpson_adaptive(lambda x: x**2, 0.0, 2.0, 5), i = ', ii, ' e = ', ee
```

Result :

```
I = simpson_adaptive(lambda x: x**2, 0.0, 2.0, 5) =  (2.7200000000000006, 0.05333333333333368)
i,e = simpson_adaptive(lambda x: x**2, 0.0, 2.0, 5), i =  2.72  e =  0.0533333333333
administrator@PC00025:~/Desktop/RemoteWorplace/Project-ID-30928060$
```

c) Compare the time it takes for the two functions to calculate a given integral function.

Code :

```
t1 = time()
i, e = trapezium_adaptive(lambda x: x**2, 0.0, 2.0, 5)
t2 = time()
print 'trapezium_adaptive takes ', t2 - t1, 's to be completed.'
print
print
t3 = time()
i, e = simpson_adaptive(lambda x: x**2, 0.0, 2.0, 5)
t4 = time()
print 'simpson_adaptive takes ', t4 - t3, 's to be completed.'
```

Result :

```
trapezium_adaptive takes  0.000128030776978 s to be completed.


simpson_adaptive takes  2.50339508057e-05 s to be completed.
```

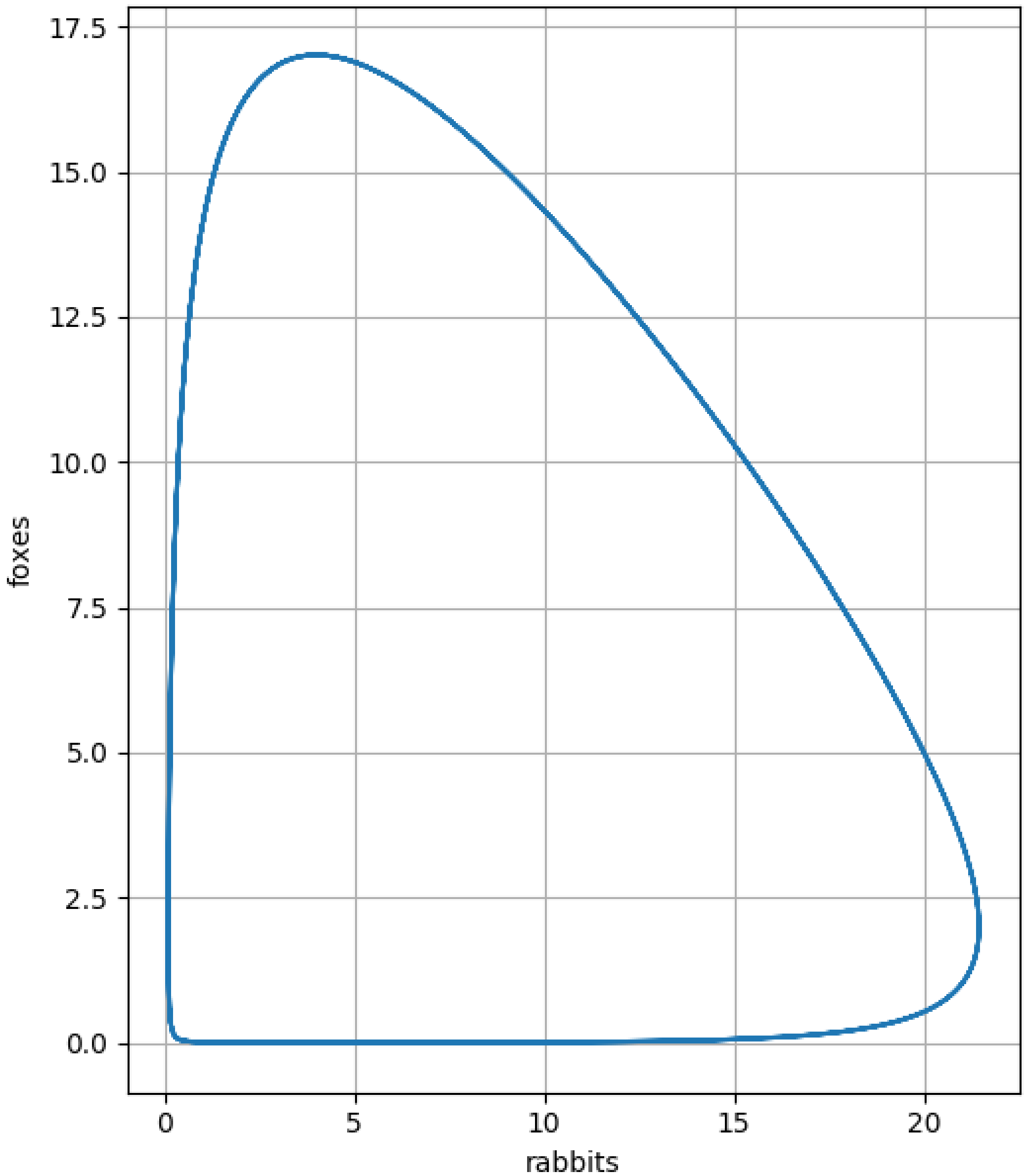
EXERCISE 3 : THE LOTKA-VOLTERRA EQUATIONS (COUPLED ODES)

a) Write a program to solve these equations using the fourth-order Runge-Kutta method for the case $\alpha = 1$, $\beta = \gamma = 0.5$, $\delta = 2$.

Code :

```
import numpy as np
import matplotlib.pyplot as plt

# a) Solve equations
def LotkaVolteraModel(x, alpha, beta, gamma, delta):
    return np.array([alpha*x[0] - beta*x[0]*x[1], delta*x[0]*x[1] - gamma*x[1]])

def RungeKutta4(f, x0, t0, tf, dt):
    t = np.arange(t0, tf, dt)
    nt = t.size
    nx = x0.size
    x = np.zeros((nx, nt))
    x[:, 0] = x0
    for k in range(nt - 1):
        k1 = dt*f(t[k], x[:,k])
        k2 = dt*f(t[k] + dt/2, x[:,k] + k1/2)
        k3 = dt*f(t[k] + dt/2, x[:,k] + k2/2)
        k4 = dt*f(t[k] + dt, x[:,k] + k3)
        dx = (k1 + 2*k2 + 2*k3 +k4)/6
        x[:, k+1] = x[:, k] + dx
    return x, t

f = lambda t, x : LotkaVolteraModel(x, 1, 0.5, 2, 0.5)
x0 = np.array([20, 5])
t0 = 0
tf = 30
dt = 0.001

x, t = RungeKutta4(f, x0, t0, tf, dt)

plt.subplot(1, 2, 1)
plt.plot(x[0,:], x[1,:])
plt.xlabel('rabbits')
plt.ylabel('foxes')
plt.grid()
```

Result :



b) Have the program make a graph showing both x and y as a function of time on the same axes from t = 0 to t = 30, start from the initial condition x = y = 2

Code :

```
# model parameters
a = 1
b = 0.5
c = 2
d = 0.5
dt = 0.0001
limitTime = 30

# initial time and populations
t = 0
x = 2
y = 2

t_list = []
x_list = []
y_list = []

# initialize lists
t_list.append(t)
x_list.append(x)
y_list.append(y)

while t < limitTime:
    t = t + dt
    x = x + (a*x - b*x*y)*dt
    y = y + (-c*y + d*x*y)*dt
    t_list.append(t)
    x_list.append(x)
    y_list.append(y)

plt.subplot(1, 2, 2)
plt.plot(t_list, x_list, 'r', label = 'rabbits')
plt.plot(t_list, y_list, 'g', label = 'foxes')
plt.xlabel('Time (t)')
plt.xlim(0, 30)
plt.grid()
plt.legend()
plt.show()
```

Graph :