Christopher Allred A02233404

# MAE 3210 - Spring 2020 - Homework 6

1. (a) Develop an algorithm which, for a given function of two variables $f(x, y)$, interval bounds $a$ and $b$ with $a < b$, and $c$ and $d$ with $c < d$, and input integer $n \geq 1$, does the following:

   (i) If $n$ is odd, it applies the multiple-application trapezoidal rule in each dimension to approximate $I = \int_c^d \left( \int_a^b f(x, y)\, dx \right) dy$.

   (ii) If $n$ is even, it applies the multiple-application Simpson's 1/3 rule in each dimension to approximate $I = \int_c^d \left( \int_a^b f(x, y)\, dx \right) dy$.

   (b) Suppose the temperature T (°C) at a point $(x, y)$ on a 16 $\text{m}^2$ rectangular heated plate is given by

$$T(x, y) = x^2 - 3y^2 + xy + 72,$$

   where $-2 \leq x \leq 2$ and $0 \leq y \leq 4$ (here $x$ and $y$ are measured in meters about a reference point at $(0, 0)$). Determine the average temperature of the plate:

   (i) Analytically, to obtain a true value.

   (ii) Numerically, using the algorithm you developed in question 1(a) above, and plot the true percent relative error $\epsilon_t$ as a function of $n$ for $1 \leq n \leq 5$. Provide some interpretation of the results.

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

# this functio nreorders the bound if out of order assumed
def testBound(a,b):
    if a > b:
        temp = a
        a = b
        b = temp
        return
    elif a == b:
        print( "Error : the Lower bound is the smae as the upper bound")
        exit(-1)
```

```
def trapezoidal(a,b,y,n,funct):
    if n== 0:
        return 0
    testBound(a,b)

    step_x = (b-a) / float(n)

    sumVal = (funct(a,y) + funct(b, y))

    for i in np.arange(1,n,1):
        sumVal += 2 *funct(a + i*step_x,y)

    return (b - a) * sumVal / (2 * n)#step_x*sumVal

# applies the multiple-application trapezoidal rule
# 1st bound a-b
# then
# 2nd bound c-c
def dub_trap(a,b,c,d,n,funct):
    sumtot = 0
    if n==0:
        return 0
    step_y = (d - c) / n

    y_list = np.arange(c + step_y, d, step_y)

    sumtot = trapezoidal(a,b,c,n,funct) + trapezoidal(a,b,d,n,funct)

    for i in range(len(y_list)):
        sumtot +=2* trapezoidal(a,b,y_list[i],n,funct) #+ trapezoidal(c,d,y_list[i],n,funct)

    return (d - c) * sumtot / (2 * n) #sumtot

def simpsons_1_3(a,b,y,n,funct):
    if n== 0:
        return 0
    testBound(a,b)

    step = float( b - a) /n
```

```
    sumVal = 0
    xList = np.arange(a + step, b,step)
    for i in range(len(xList)):

        if i == 0 or i % 2 == 0:
            sumVal += 4 * funct(xList[i],y)
        else:
            sumVal += 2 * funct(xList[i],y)

    return ( (b - a) * (funct(a,y) + sumVal + funct( a,y )) / (3*n))


# applies the multiple-application Simpson’s 1/3 rule
# 1st bound a-b
# then
# 2nd bound c-c
def dub_Simp(a,b,c,d,n,funct):
    sumtot = 0
    if n==0:
        return 0
    step = float(d - c) / n
    y_list = np.arange(c + step, d, step)
    for i in range(len(y_list)):
        if i == 0 or i % 2 == 0:
            sumtot += 4 * simpsons_1_3(a,b,y_list[i],n,funct)
        else:
            sumtot += 2 * simpsons_1_3(a,b,y_list[i],n,funct)


    sumtot +=simpsons_1_3(a,b,c,n,funct)
    sumtot += simpsons_1_3(a,b,d,n,funct)
    return (d - c) *sumtot/ (3*n)


CorectVal = (2752/3.0)/(4.0*4) # = 57.33333333



def pob1a(a,b,c,d,max_n,funct):
    print("Starting Problem 1.a  ")
```

```python
# n_list = np.arange(0,max_n,1).tolist()
y_errorTrap =[]
y_errorSim =[]
y_combo =[]
x_nTrap=[]
x_nSim = []
x_nCombo = []
for n in range(0,max_n):
    curVal = 0
    if n<=0:
        y_errorTrap.append(100)
        y_errorSim.append(100)
        y_combo.append(100)
        x_nCombo.append(n)
        x_nTrap.append(n)
        x_nSim.append(n)
        continue
    if n%2 !=0: #odd
        curVal = dub_trap(a,b,c,d,n,funct)/((b-a)*(d-c))
        y_errorTrap.append(100* abs( curVal - CorectVal )/CorectVal)
        x_nTrap.append(n)
        y_combo.append(100* abs( curVal - CorectVal )/CorectVal)
        x_nCombo.append(n)

    else: # even
        curVal = dub_Simp(a,b,c,d,n,funct) /((b-a)*(d-c))
        y_errorSim.append(100* abs( curVal - CorectVal )/CorectVal)
        x_nSim.append(n)
        y_combo.append(100* abs( curVal - CorectVal )/CorectVal)
        x_nCombo.append(n)


plt.plot(x_nTrap, y_errorTrap,color='blue', label="Percent Trapizodal Error")
plt.plot(x_nSim, y_errorSim,color='green',label="Percent Simpsons Error")
plt.plot(x_nCombo, y_combo,color='red',label="Percent Composite Error")

plt.title(" Percent Error vs n")
plt.ylabel("Percent Error")
plt.xlabel("n")
plt.legend()
```

```
    plt.show()
    print('Done with Prob1')

def funct_temp_1A(x,y):
    return x**2 - 3*y**2 + x*y + 72

if __name__ == "__main__":
    print("Starting Application: ")
    p1.pob1a(-2,2,0,4,6,funct_temp_1A)
```

$a := -2$

$$a := -2 \quad (1)$$

$b := 2$

$$b := 2 \quad (2)$$

$c := 0$

$$c := 0 \quad (3)$$

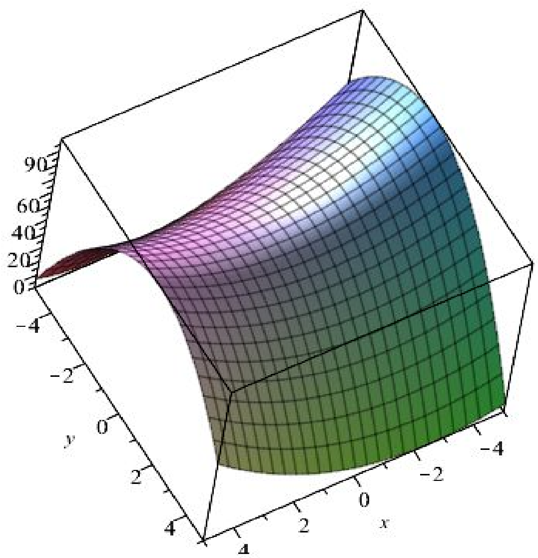
$d := 4$

$$d := 4 \quad (4)$$

$Temp := x^2 - 3 * y^2 + x * y + 72$

$$Temp := x^2 + x\,y - 3\,y^2 + 72 \quad (5)$$

$smartplot3d[x, y]($ **(5)** $)$

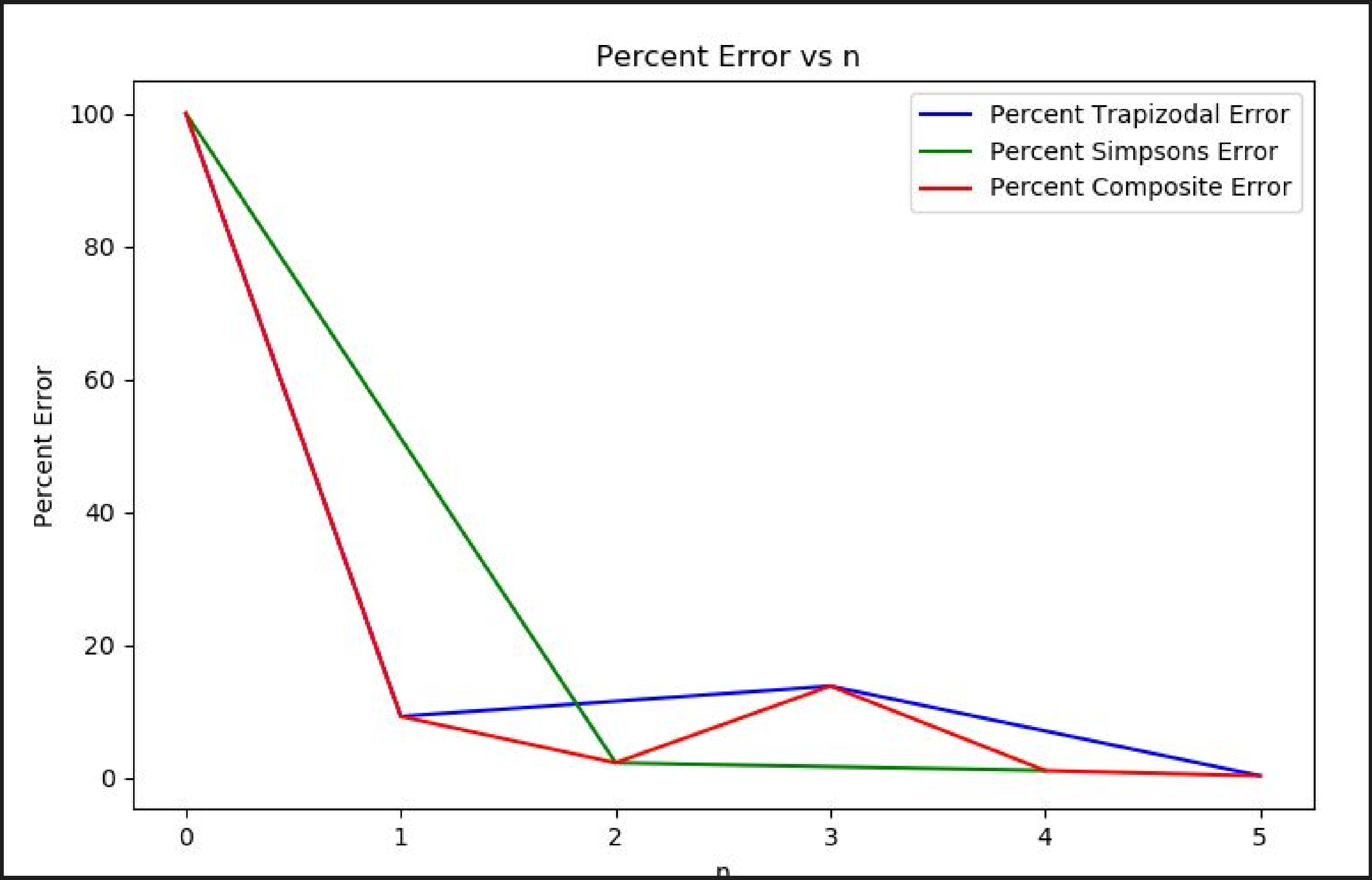


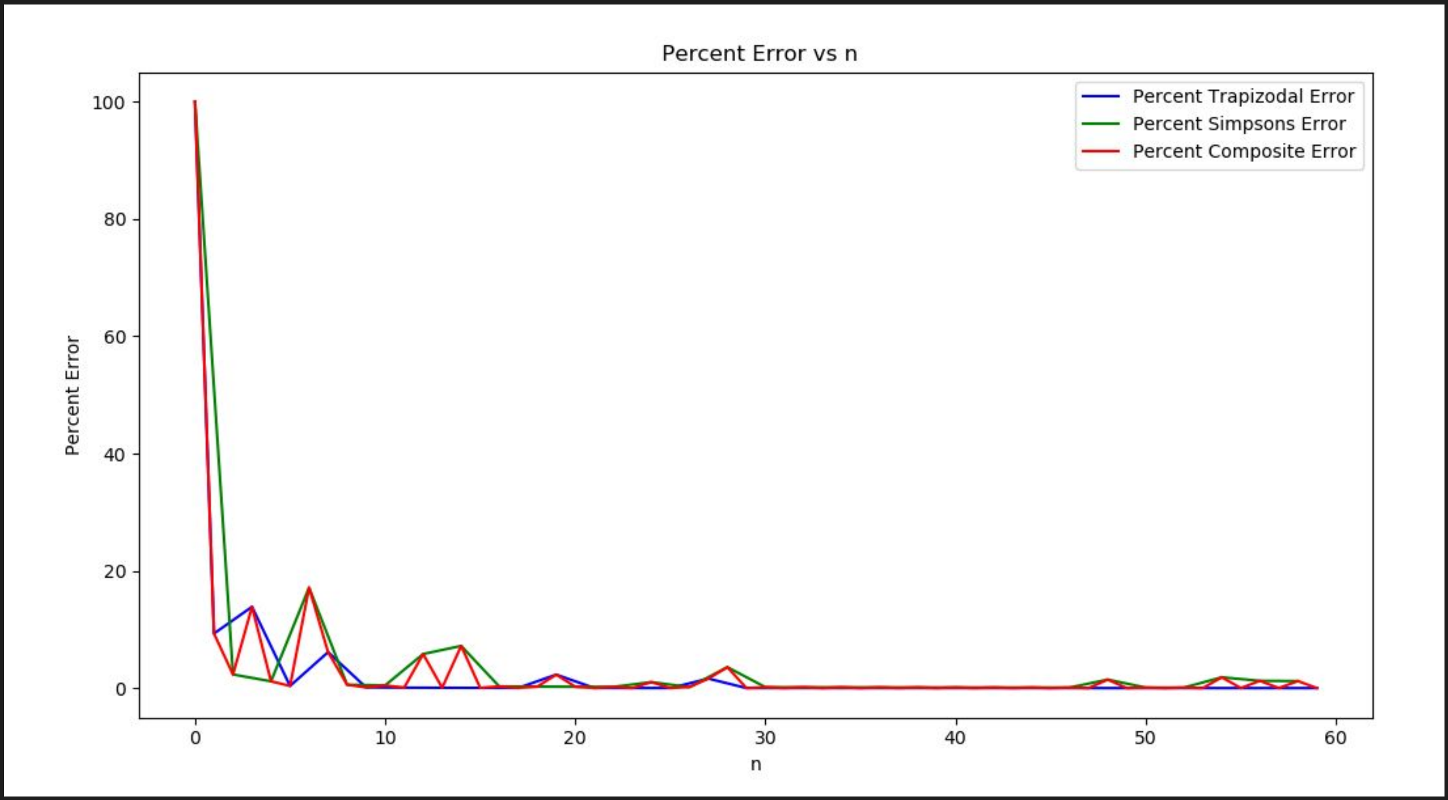

$$\frac{evalf\left(\int_c^d \int_a^b Temp \, dx \, dy\right)}{(b-a) * (d-c)}$$

$$57.33333332 \quad (6)$$

$$\frac{\left(\frac{2752}{3.0}\right)}{(b-a) * (d-c)}$$

$$57.33333333 \quad (7)$$




## Interpretation of Results

It seems after about 2 iteration of n the error significantly decreases asymptotically approaching zero. Trapezoidal method seems to take longer to reach a small amount of errors but it seems to converge to the actual value. After about 5 iterations the error is almost zero but occasionally the system will get a little error as seen in the second plot.

---

2. Write code for two separate algorithms to implement (a) Euler's method and (b) the standard 4th order Runge-Kutta method, for solving a given first-order **one-dimensional** ODE. Design the code to solve the ODE over a prescribed interval with a prescribed step size, taking the initial condition at the left end point of the interval as an input variable.

```
# # http://code.activestate.com/recipes/577647-ode-solver-using-euler-method/
# (xa, ya) are a know solution point
# from xa to xb
# n is the number of steps
def euler(f, xa, xb, ya, n):
    # step size
    step = (xb - xa) / float(n)
    # x inital
    x = xa
    # y intal
    yb = ya
    for i in range(n):
        yb += step * f(x)#f(x, yb)
        x += step
    return yb # this is the value at xb
```

```python
# help from:
#  https://github.com/twright/Python-Examples/blob/master/runge-kutta-method.py

def runge_kutta(timei, y0, step,fun1):
    y_current = y0
    timeCer = timei
    while  y_current > 0:

        vF1 = fun1(timeCer)
        vF2 = fun1(timeCer + step/2)
        vF3 = fun1(timeCer + step/2)
        vF4 = fun1(timeCer + step)
        y_current = y_current + (step/6)*(vF1 + 2*(vF2 + vF3) + vF4)
        timeCer = timeCer + step

    return timeCe
```

3. The drag force $F_d$ (N) exerted on a falling object can be modeled as proportional to the square of the objects downward velocity $v$ (m/s), with a constant of proportionality $c_d$ (kg/m).

   (a) Assume that a falling object has mass $m = 100$ (kg) with a drag coefficient of $c_d = 0.25$ kg/m, and let $g = 9.81$ (m/s$^2$) denote the constant downward acceleration due to gravity near the surface of the earth. Starting from Newton's second law, explain the derivation of the following ODE for the downward velocity $v = v(t)$ of the falling object:

$$\frac{dv}{dt} = 9.81 - 0.0025v^2. \tag{1}$$

derivation of the velocity at a point in time due to drag force

$F_{tot} = F[grav] + F[air]$

$$F_{tot} = F_{grav} + F_{air} \tag{1}$$

$F_{grav} := m \cdot g$

$$F_{grav} := m\,g \tag{2}$$

$F_{air} := -k \cdot v(t)$

$$F_{air} := -k\,v(t) \tag{3}$$

$F_{tot} := m \cdot \frac{dv}{dt}$

$$F_{tot} := m\left(\frac{d}{dt}\,v(t)\right) \tag{4}$$

$F_{tot} = F_{grav} + F_{air}$

$$m\left(\frac{d}{dt}\,v(t)\right) = m\,g - k\,v(t) \tag{5}$$

$a := \frac{dv}{dt}$

$$a := \frac{d}{dt}\,v(t) \tag{6}$$

$F_{tot} = F_{grav} + F_{air}$

$$m\left(\frac{d}{dt}\,v(t)\right) = m\,g - k\,v(t) \tag{7}$$

$\left(\frac{d}{dt}\,v(t)\right) = \frac{m \cdot g - k \cdot v(t)}{m}$

$$\frac{d}{dt}\,v(t) = \frac{m\,g - k\,v(t)}{m} \tag{8}$$

$\frac{d}{dt}\,v(t) = g - \frac{(k\,v(t))}{m}$

$$\frac{d}{dt}\,v(t) = g - \frac{k\,v(t)}{m} \tag{9}$$

$\frac{d\,v(t)}{g - \frac{(k\,v(t))}{m}} = dt$

$$\frac{d\,v(t)}{g - \frac{k\,v(t)}{m}} = dt \tag{10}$$

*#Separating the variables*

$\int_0^v \frac{1}{g - \frac{k \cdot v}{m}}\,du = t$

$$\frac{v}{g - \frac{k\,v}{m}} = t \tag{11}$$

$\ln\left(\frac{\left(g - \frac{k \cdot u}{m}\right)}{g}\right) = -\frac{k}{m} \cdot t$

$$\ln\left(\frac{g - \frac{k\,u}{m}}{g}\right) = -\frac{k\,t}{m} \tag{12}$$

$1 - \frac{k}{mg} \cdot v = e^{-\frac{k}{m} \cdot t}$

$$1 - \frac{k\,v}{mg} = e^{-\frac{k\,t}{m}} \tag{13}$$

$solve\left(1 - \frac{k}{mg} \cdot v = e^{-\frac{k}{m} \cdot t}, v\right)$

$$-\frac{\left(e^{-\frac{k\,t}{m}} - 1\right) mg}{k} \tag{14}$$

$v := t \rightarrow -\frac{m \cdot g}{k} \cdot \left(e^{-\frac{k\,t}{m}} - 1\right)$

$$v := t \rightarrow -\frac{m\,g\left(e^{-\frac{k\,t}{m}} - 1\right)}{k} \tag{15}$$

(b) Suppose that this same object is dropped from an initial height of $y_0 = 2$ km. Determine when the object hits the ground by solving the ODE you derived in question 3(a) using

(i) Euler's method.

(ii) the standard 4th order Runge-Kutta method.

**HINT:** Note that, with the velocity $v$ oriented downward, the height $y = y(t)$ satisfies $\dfrac{dy}{dt} = -v$. You are asked to find the final time $t_f$ when the height $y$ of the falling object reaches zero, i.e. when $y(t_f) = 0$. There are two ways to solve this problem.

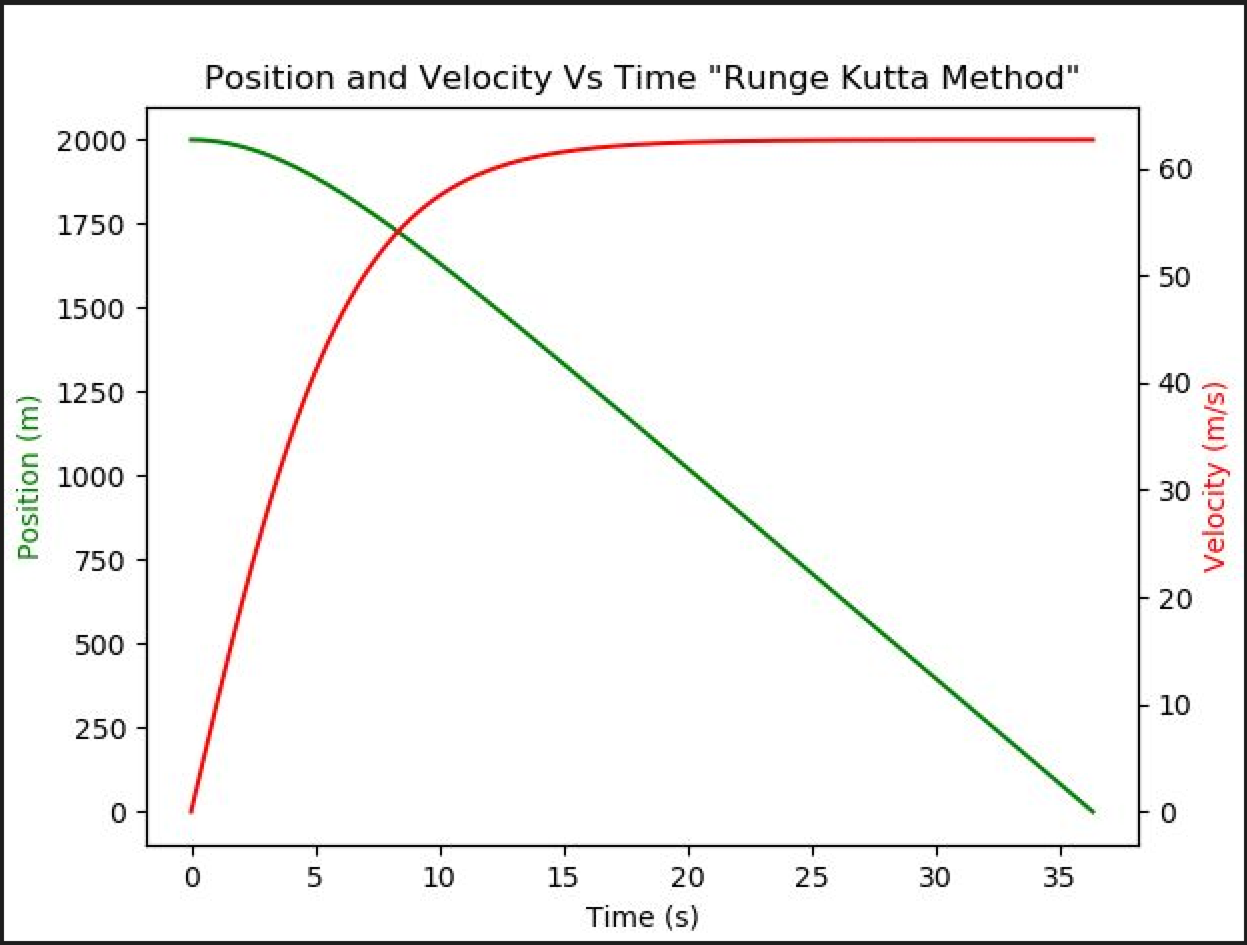
option A

A. You can use your algorithm for solving one-dimensional ODEs (Euler and Runge-Kutta 4) from question 2 to solve the ODE (1) to find $v = v(t)$ (at discrete time points) with initial condition $v(0) = 0$. Then, you can use your one-dimensional ODE algorithms, again, to solve $\dfrac{dy}{dt} = -v$ with initial condition $y(0) = 2000$ m, and try to identify when $y(t_f) = 0$.

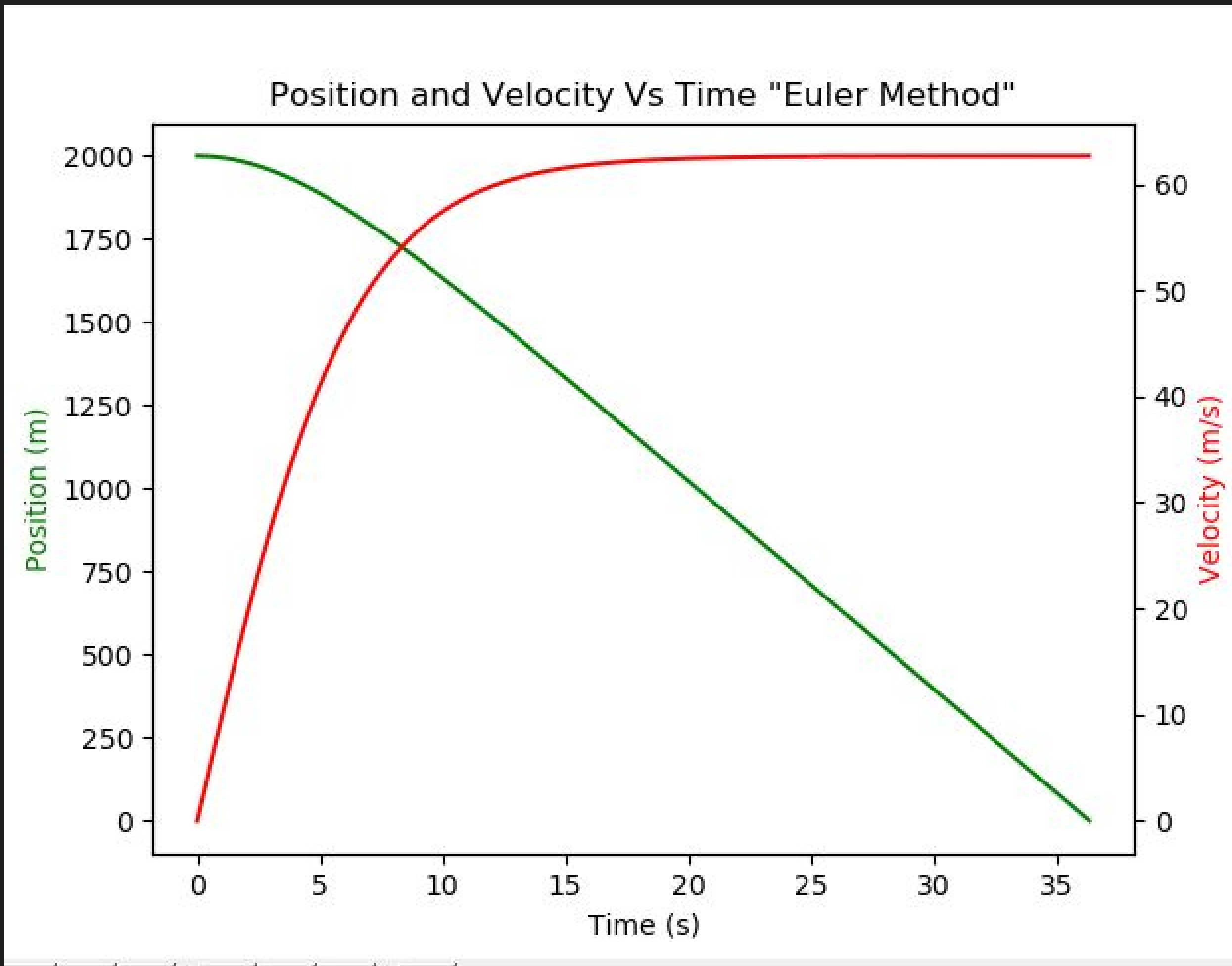
```
def fallAccel(v):
    return 9.81 - 0.0025 * v**2



# help from:
#  https://github.com/twright/Python-Examples/blob/master/runge-kutta-method.py


def rungeKuttaMethod(xi,yi,step):

    time = [0]

    velosList = [xi]
    positionList = [yi]
    y_current = yi

    i = 0
    while y_current > 0:
        #F1
```

```
        vF1 = fallAccel(velosList[i])
        pF1 = fallVel(velosList[i])

        #F2
        vF2 = fallAccel(velosList[i]+(1/2)*step * vF1)
        pF2 = fallVel(velosList[i]+(1/2)*step * vF1)


        #F3
        vF3 = fallAccel(velosList[i]+(1/2)*step * vF2)
        pF3 = fallVel(velosList[i]+(1/2)*step * vF2)


        #F4
        vF4 = fallAccel(velosList[i]+step * vF3)
        pF4 = fallVel(velosList[i]+step * vF3)

        velosList.append( velosList[i] + (vF1+2*(vF2+vF3)+vF4)*step/6)
        positionList.append(  positionList[i] + ( pF1+2*(pF2+pF3)+pF4)*step/6)
        y_current = positionList[i+1]
        time.append(i*step)
        i += 1


    plt.plot(time,positionList,color='green', label="Position")
    plt.ylabel("Position (m)",color='green')
    # plt.legend()
    plt.xlabel("Time (s)")

    plt.twinx()
    plt.plot(time,velosList,color='red', label="Velocity")
    plt.ylabel("Velocity (m/s)",color ="red")
    plt.title("Position and Velocity Vs Time \"Runge Kutta Method\"")
    # plt.legend()
    print(velosList[-1])
    print("the object reaches Position: ",positionList[-2],"m  at the time: ",time[-2],"s")
    plt.show()
    return
```

```
    #this shows how to twin axis:
    '''
https://matplotlib.org/3.1.1/gallery/subplots_axes_and_figures/two_scales.html#sphx-glr-galle
ry-subplots-axes-and-figures-two-scales-py
    '''
```



```python
def eulerMethod (xi,yi,step):

    time = [0]

    velosList = [xi]
    positionList = [yi]
    y_current = yi

    i = 0
    while y_current > 0:

        velosList.append( velosList[i] + fallAccel(velosList[i])*step)
        positionList.append(  positionList[i] -(velosList[i])*step)
        y_current = positionList[i+1]
        time.append(i*step)
        i += 1
    print(i)
    plt.plot(time,positionList,color='green', label="Position")
    plt.ylabel("Position (m)",color='green')
    # plt.legend()
    plt.xlabel("Time (s)")
    plt.twinx()
    plt.plot(time,velosList,color='red', label="Velocity")
    plt.ylabel("Velocity (m/s)",color ="red")
    plt.title("Position and Velocity Vs Time \"Euler Method\"")
    # plt.legend()
    print("Final Velocity: ",velosList[-1])
    print("the object reaches Position: ",positionList[-2],"m  at the time: ",time[-2],"s")
    plt.show()
    return
```




```
.python-2020.4.74986\pythonFiles\lib\python\debugpy\wheels\debugpy\launcher' 'd:\Christopher Allred\Documents\GitH
ub\NumMethods\HW\HW6\prob2.py'
36354
Final Velocity:  62.6404194876307
Euler Method: the object reaches Position:  0.04766344085489345 m  at the time:  36.352000000000004 s
Final Velocity:  62.64041726233247
Runge Kutta Method: the object reaches Position:  0.03555623593237381 m  at the time:  36.352000000000004 s
PS D:\Christopher Allred\Documents\GitHub\NumMethods\HW\HW6>
on: Current File (HW6)                    Ln 78, Col 12   Spaces: 4   UTF-8   CRLF   Python
```

Interpretation of Results:

The object looks like it hits the ground going about 62.6404194876307 m/s at the time 36.352000000000004 27seconds.

4. Write code for two separate algorithms to implement (a) Euler's method and (b) the standard 4th order Runge-Kutta method, for solving a given first-order **two-dimensional** system of ODEs. Design the code to solve the system of ODEs over a prescribed interval with a prescribed step size.

```
def rungeKuttaMethod(dim1,dim2,step,fun1,fun2):

    time = [0]

    dim1List = [dim1]
    dim2List = [dim2]
    dim2_current = dim2

    i = 0
    while dim2_current > 0:
        #F1
        vF1 = fun1(dim1List[i])
        pF1 = fun2(dim1List[i])

        #F2
        vF2 = fun1(dim1List[i]+(1/2)*step * vF1)
        pF2 = fun2(dim1List[i]+(1/2)*step * vF1)


        #F3
        vF3 = fun1(dim1List[i]+(1/2)*step * vF2)
        pF3 = fun2(dim1List[i]+(1/2)*step * vF2)


        #F4
        vF4 = fun1(dim1List[i]+step * vF3)
        pF4 = fun2(dim1List[i]+step * vF3)

        dim1List.append( dim1List[i] + (vF1+2*(vF2+vF3)+vF4)*step/6)
        dim2List.append(  dim2List[i] + ( pF1+2*(pF2+pF3)+pF4)*step/6)
        dim2_current = dim2List[i+1]
        time.append(i*step)
        i += 1
```

```
    print("Final dim1List: ",dim1List[-1])
    print("Runge Kutta Method: the object reaches Position: ",dim2List[-2],"m  at the time:
",time[-2],"s")

    return

def eulerMethod (dim1,dim2,step,fun1,fun2):

    time = [0]

    dim1List = [dim1]
    dim2List = [dim2]
    dim2_current = dim2

    i = 0
    while dim2_current > 0:

        dim1List.append( dim1List[i] + fun1(dim1List[i])*step)
        dim2List.append(  dim2List[i] +fun2(dim1List[i])*step)
        dim2_current = dim2List[i+1]
        time.append(i*step)
        i += 1

    plt.plot(time,dim2List,color='green', label="dim2List")
    plt.ylabel("Dimension 2",color='green')
    # plt.legend()
    plt.xlabel("Time (s)")
    plt.twinx()
    plt.plot(time,dim1List,color='red', label="dim1List")
    plt.ylabel("dimintion",color ="red")
    plt.title("Position and Velocity Vs Time \"Runge Kutta Method\"")
    # plt.legend()
    plt.show()
    print("Final dim1List: ",dim1List[-1])
    print("Euler Method: the object reaches Position: ",dim2List[-2],"m  at the time:
",time[-2],"s")

    return
```

5. The motion of a damped mass spring is described by the following ODE

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0, \tag{2}$$

where $x$ = displacement from equilibrium position (m), $t$ = time (s), $m$ = mass (kg), $k$ = stiffness constant (N/m) and $c$ = damping coefficient (N·s/m).

(a) Rewrite the 2nd order ODE (2) as a two-dimensional system of first order ODEs for the displacement $x = x(t)$ and velocity $v = v(t)$ of the mass attached to the spring.

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

#GlobalVars
c = 0  # Damping
k = 0  # Spring Const
m = 0  # Mass
def funAccl(v, x):
    return -(c*v + k*x) / m
def velos(v):
    return v


def rungeKuttaMethod(dim1,dim2,step,endTime, fun1,fun2):

    time = [0]

    dim1List = [dim1]
    dim2List = [dim2]

    i = 0
    while time[-1] < endTime:
        #F1
        vF1 = fun1(dim1List[i],dim2List[i])
        pF1 = fun2(dim1List[i])

        #F2
        vF2 = fun1(dim1List[i]+(1/2)*step * vF1, dim2List[i]+(1/2)*step * pF1)
```

```
        pF2 = fun2(dim1List[i]+(1/2)*step * vF1)


        #F3
        vF3 = fun1(dim1List[i]+(1/2)*step * vF2, dim2List[i]+(1/2)*step * pF1)
        pF3 = fun2(dim1List[i]+(1/2)*step * vF2)


        #F4
        vF4 = fun1(dim1List[i]+step * vF3, dim2List[i] + step * pF1)
        pF4 = fun2(dim1List[i]+step * vF3)

        dim1List.append( dim1List[i] + (vF1+2*(vF2+vF3)+vF4)*step/6)
        dim2List.append(  dim2List[i] + ( pF1+2*(pF2+pF3)+pF4)*step/6)

        time.append(i*step)
        i += 1

    plt.plot(time,dim2List,color='green', label="dim2List")
    plt.ylabel("Position (m)",color='green')
    # plt.legend()
    plt.xlabel("Time (s)")
    plt.twinx()
    plt.plot(time,dim1List,color='red', label="dim1List")
    plt.ylabel("Velocity (m/s)",color ="red")
    plt.title("dim2List and dim1List Vs Time \"Runge Kutta Method\"")
    # plt.legend()
    print("Final Velocity: ",dim1List[-1])
    print("Runge Kutta Method: the object reaches Position: ",dim2List[-2],"m  at the time:
",time[-2],"s")
    plt.show()
    return


def eulerMethod (dim1,dim2,step,endTime,fun1,fun2):

    time = [0]

    dim1List = [dim1]
    dim2List = [dim2]
```

```
    i = 0
    while time[-1] < endTime:

        dim1List.append( dim1List[i] + fun1(dim1List[i],dim2List[i])*step)
        dim2List.append(  dim2List[i] +fun2(dim1List[i])*step)
        time.append(i*step)
        i += 1

    plt.plot(time,dim2List,color='green', label="dim2List")
    plt.ylabel("Position",color='green')
    # plt.legend()
    plt.xlabel("Time (s)")
    plt.twinx()
    plt.plot(time,dim1List,color='red', label="dim1List")
    plt.ylabel("Velocity",color ="red")
    plt.title("Position and Velocity Vs Time \"Euler Method\"")
    # plt.legend()
    plt.show()
    print("Final dim1List: ",dim1List[-1])
    print("Euler Method: the object reaches Position: ",dim2List[-2],"m  at the time: 
",time[-2],"s")

    return
```

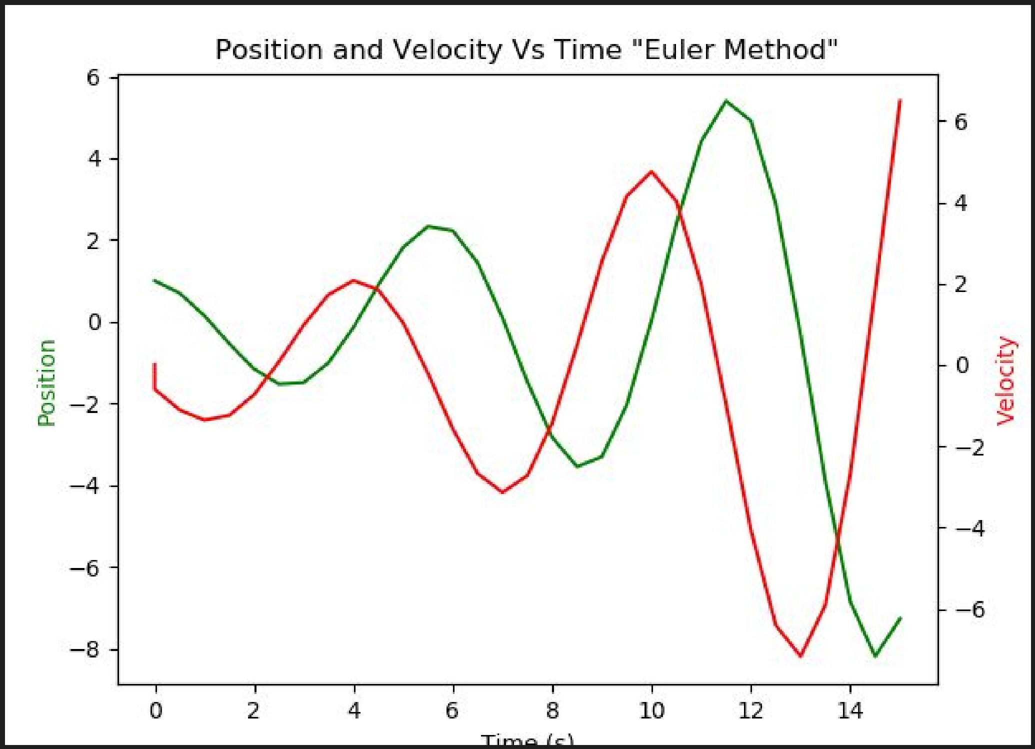
(b) Assume that the mass is $m = 10$ kg, the stiffness $k = 12$ N/m, the damping coefficient is $c = 3$ N·s/m, the initial velocity of the mass is zero ($v(0) = 0$), and the initial displacement is $x = 1$ m ($x(0) = 1$). Solve for the displacement and velocity of the mass over the time period $0 \leq t \leq 15$, and plot your results for the displacement $x = x(t)$,

```
# 5.b
c = 3   # Damping
k = 12  # Spring Const
m = 10  # Mass
```

(i) using Euler's method with step size $h = 0.5$, and then with step size $h = 0.01$.

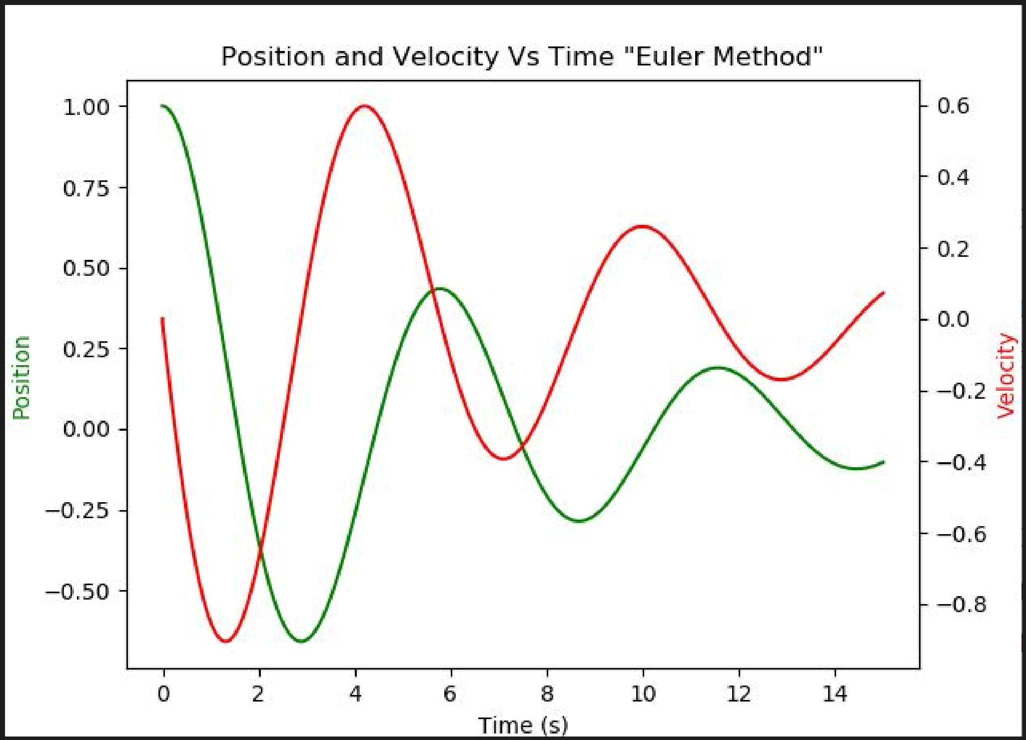
```
# 5.b.i
```

```
eulerMethod(0, 1, .5, 15, funAccl,velos)
```




```
Final dim1List:  6.48357336245382
Euler Method: the object reaches Position:  -8.194935836188012 m  at the time:  14.5 s
```

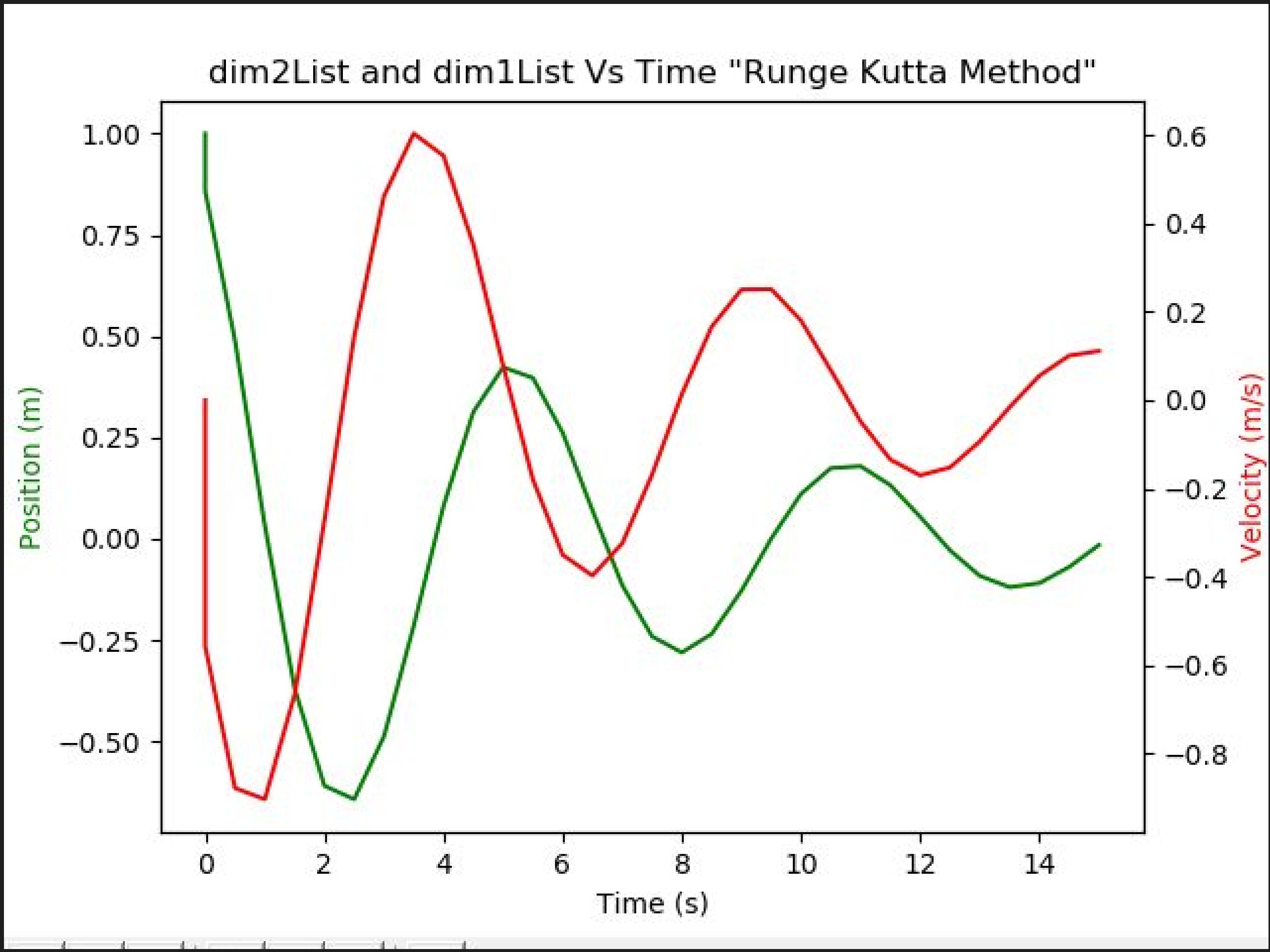
```
eulerMethod(0, 1, .01, 15, funAccl,velos)
```




```
Final dim1List:  0.07206169870200059
Euler Method: the object reaches Position:  -0.10423203804492814 m  at the time:  14.99 s
```

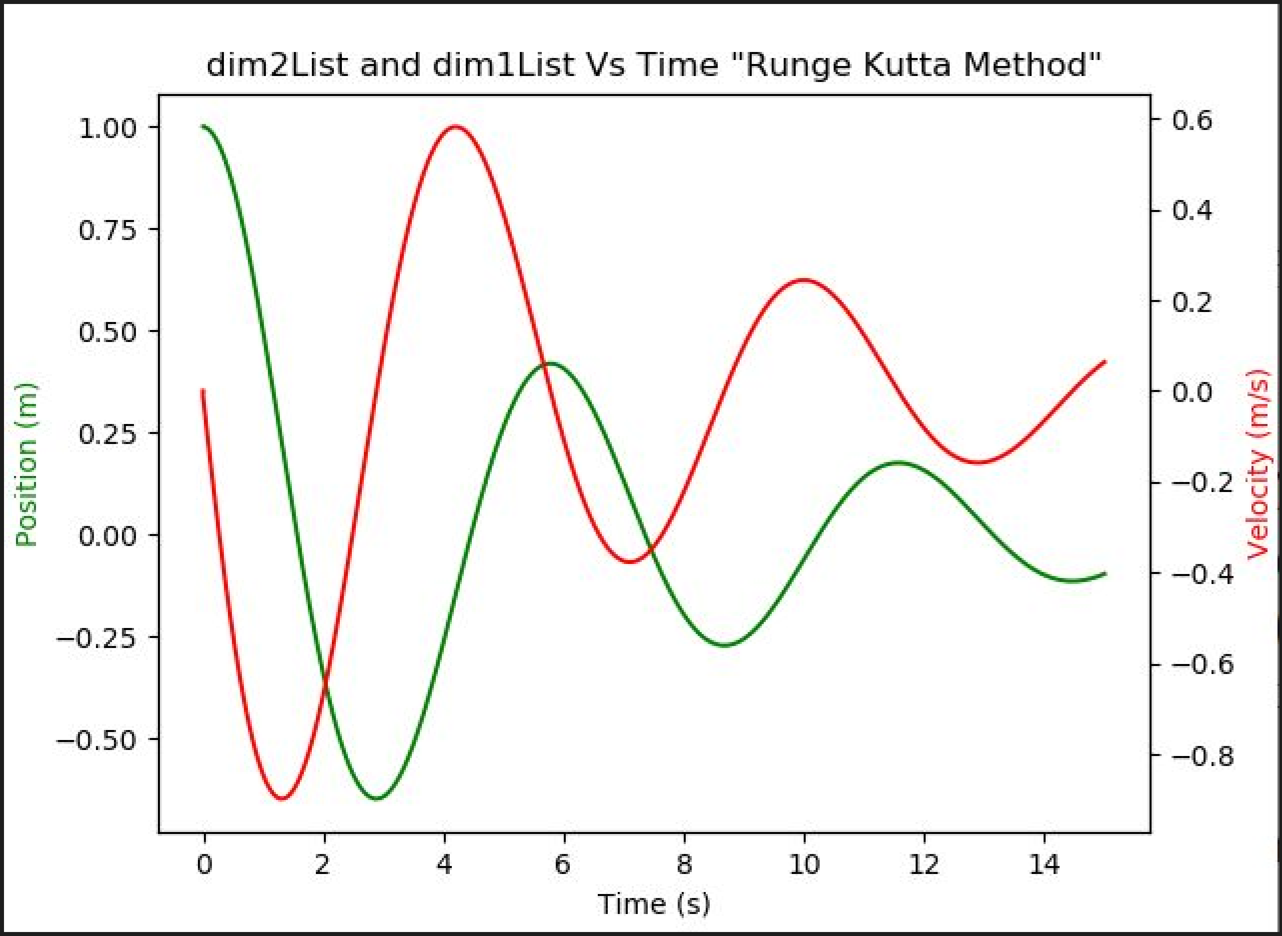
(ii) using the standard 4th order Runge-Kutta method with step size $h = 0.5$, and then with step size $h = 0.01$.

```
# 5.b.ii
rungeKuttaMethod(0, 1, .5,15, funAccl,velos)
```




```
Final Velocity:  0.11109639597455334
Runge Kutta Method: the object reaches Position:  -0.0696890787361816 m  at the time:  14.5 s
```

```
rungeKuttaMethod(0, 1, .01, 15, funAccl,velos)
```

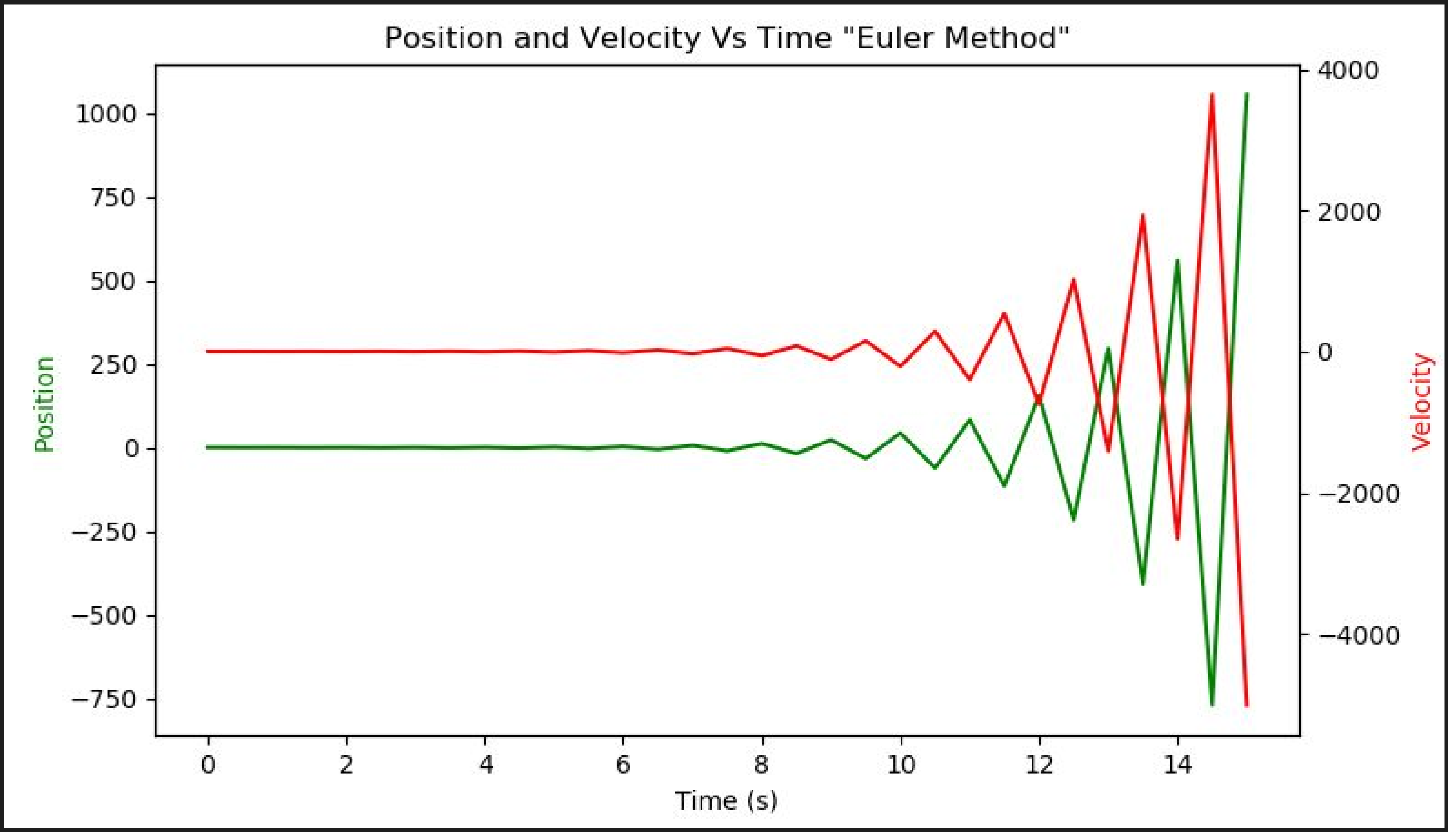



```
Final Velocity:  0.06377312156592889
Runge Kutta Method: the object reaches Position:  -0.09663397464106124 m  at the time:  14.99
s
```

(c) Assume that the mass is $m = 10$ kg, the stiffness $k = 12$ N/m, the damping coefficient is $c = 50$ N·s/m, the initial velocity of the mass is zero ($v(0) = 0$), and the initial displacement is $x = 1$ m ($x(0) = 1$). Solve for the displacement and velocity of the mass over the time period $0 \le t \le 15$, and plot your results for the displacement $x = x(t)$,

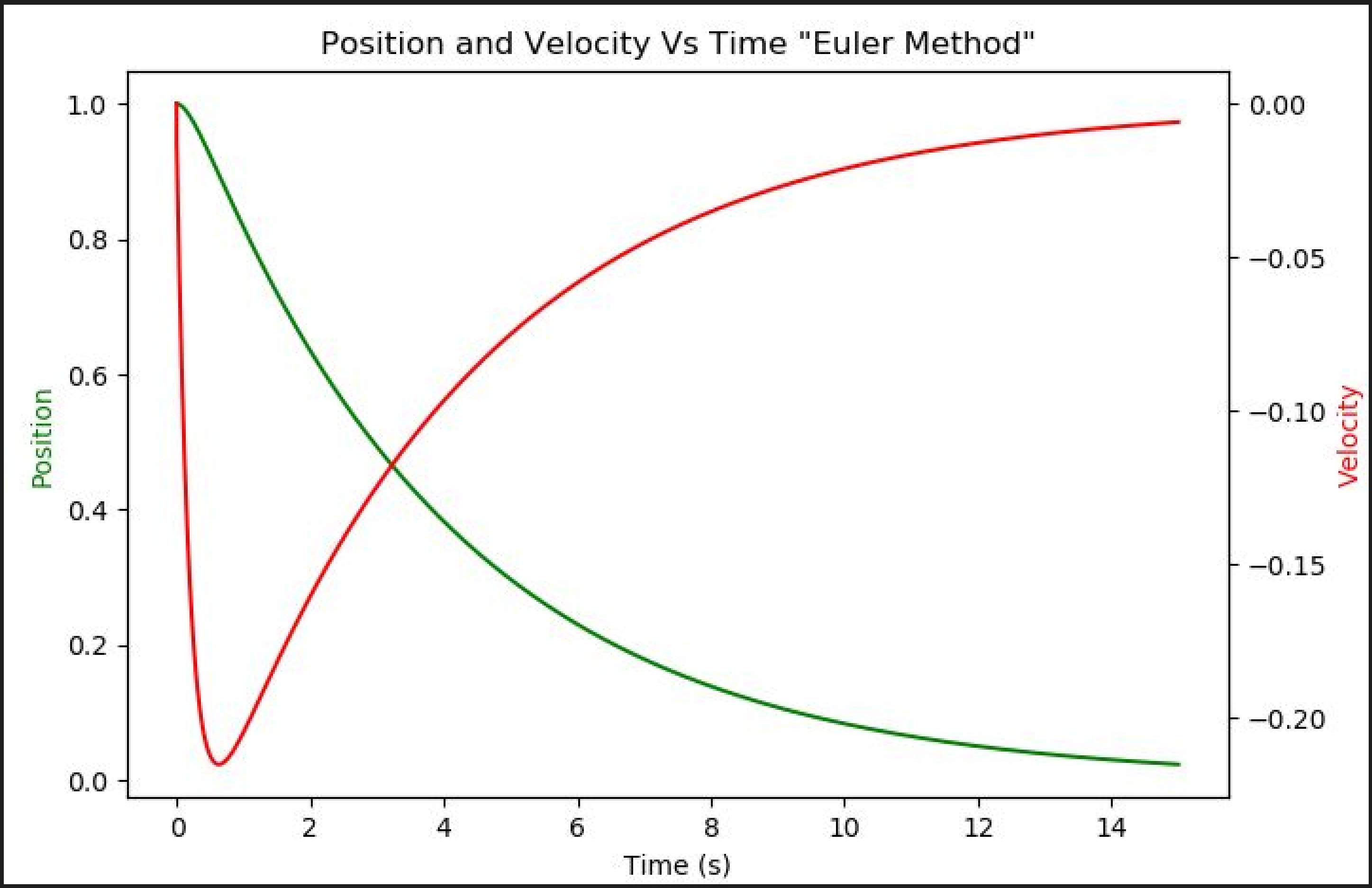
(i) using Euler's method with step size $h = 0.5$, and then with step size $h = 0.01$.

```
# 5.c
c = 50  # Damping
k = 12  # Spring Const
m = 10  # Mass

# 5.c.i
eulerMethod(0, 1, .5, 15, funAccl,velos)
```




```
Final dim1List:  -5015.048630697396
Euler Method: the object reaches Position:  -769.0622873406406 m  at the time:  14.5 s
```

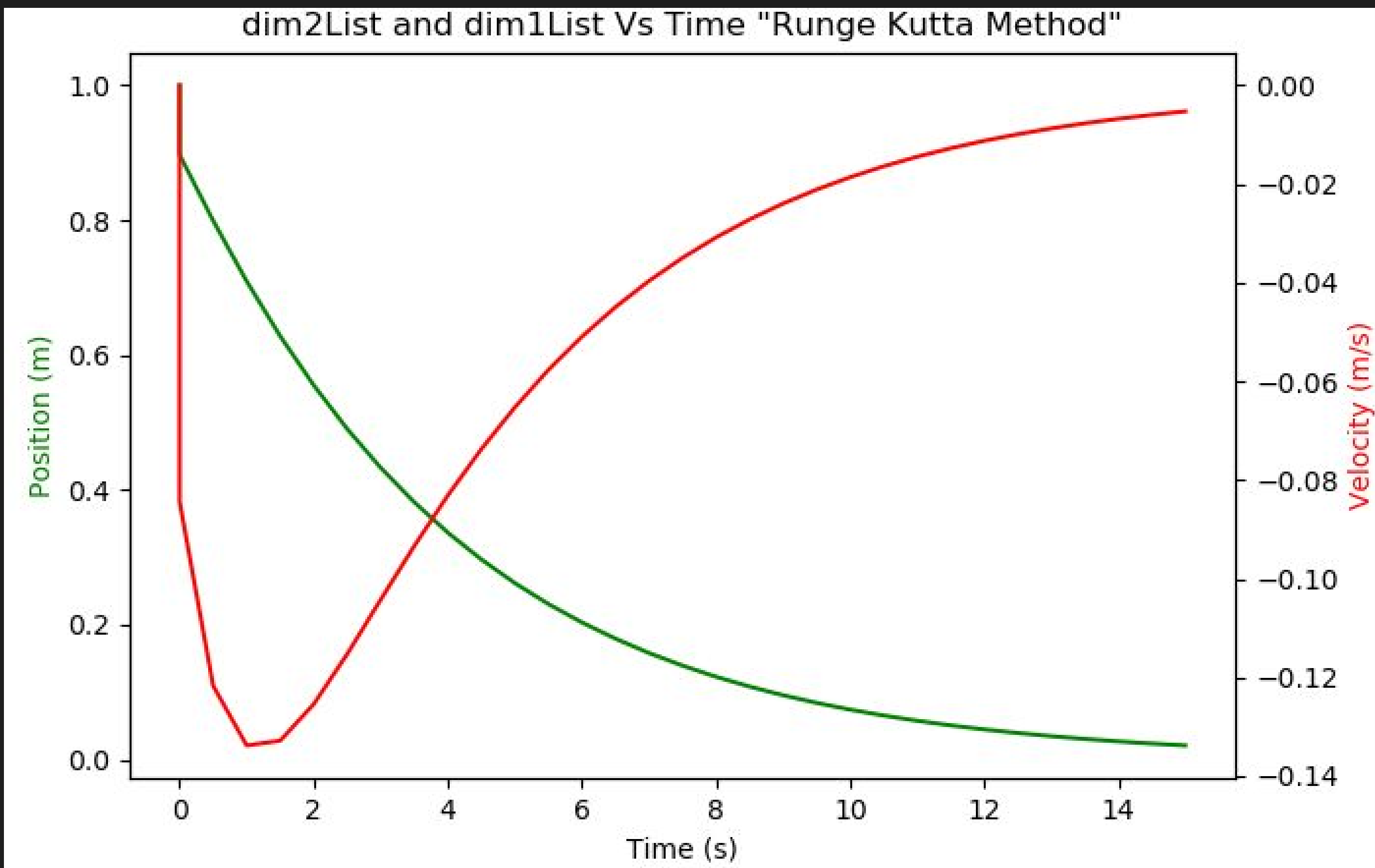
```
eulerMethod(0, 1, .01, 15, funAccl,velos)
```




```
Final dim1List:  -0.005978751207227683
Euler Method: the object reaches Position:  0.023711980965456906 m  at the time:  14.99 s
```

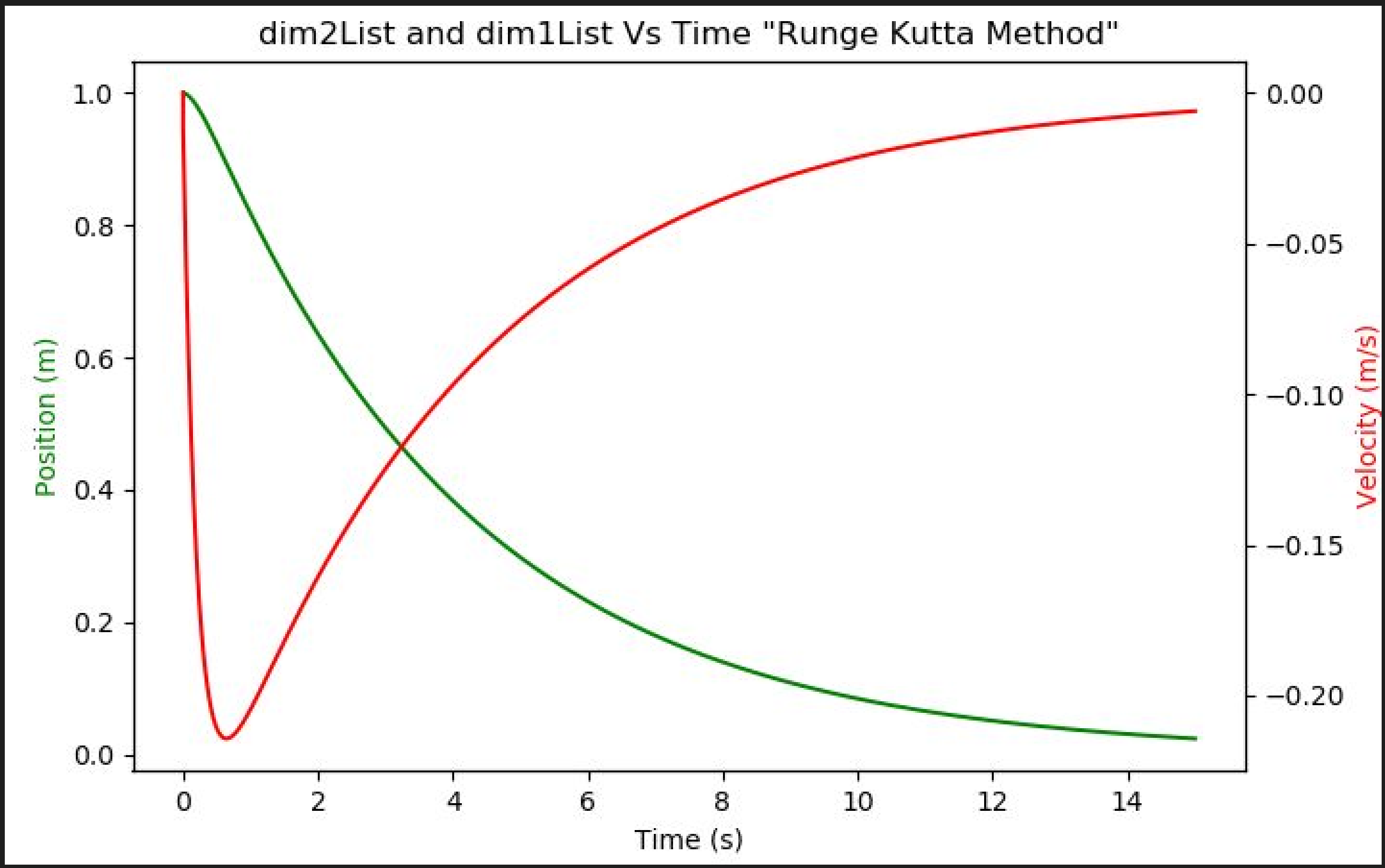
(ii) using the standard 4th order Runge-Kutta method with step size $h = 0.5$, and then with step size $h = 0.01$.

```
# 5.c.ii
rungeKuttaMethod(0, 1, .5, 15, funAccl,velos)
```




```
Final Velocity:  -0.005283506327920375
Runge Kutta Method: the object reaches Position:  0.02388558833470731 m  at the time:  14.5 s
```

```
rungeKuttaMethod(0, 1, .01, 15, funAccl,velos)
```




Final Velocity: -0.0060075510575223175

Runge Kutta Method: the object reaches Position: 0.023826153208791327 m at the time: 14.99 s