CPL103

S16036

1. Ordinary Differential Equations

I. Equations that involve functions and their derivatives are called Ordinary differential equations. They describe the relationship between a function and its rate of change.

II.

ODE	PDE
• Involves functions of a single independent variable	• Involves functions of a multiple independent variable
• Contains ordinary derivatives	• Contains partial derivatives
• Involves in time dependent processes	• Involves in spatial related problems
• Used in population dynamics and electrical circuits	• Used in heat equations and maxwells equations

$$ODE = \frac{d^2y}{dx^2} + A\frac{dy}{dx} + By = 0 \qquad PDE = \frac{\partial U(x,y)}{\partial x} + \frac{\partial U(x,y)}{\partial y} = 0$$

III. Dependent variable, independent variable, derivatives, order of the derivative, initial conditions, boundary conditions

IV. Order of an ordinary differential equation means the highest derivative of the dependent variable that appears in the equation.

V. $\frac{dy}{dx} = f(x,y)$

$\frac{dy}{dx} = xy$

y – dependent variable
x - independent variable
$f(x,y)$ – function of x and y

VI. $\frac{d^2y}{dx^2} = f(x,y,y')$
y – dependent variable
x – independent variable

$f(x, y, y')$ - function of x,y,y'

$$a(x)\frac{d^2y}{dx^2} + b(x)\frac{dy}{dx} + c(x)y = g(x)$$

$$\frac{d^2y}{dx^2} - 3\frac{dy}{dx} + 2y = e^x$$

VII. It is a problem where the solution of the differential equation is required to specify specific initial conditions at a given point.

$$\frac{dy}{dx} = f(x, y)$$

$y(x_0) = y_0$

In here $y=y_0$ is the initial condition when $x= x_0$

2. Eulers method for Solving ODE

I.

a.) Eulers method can be used for straight line functions only furthermore it is a first order method.

b.)

$$y(t + \Delta t) = y(t) + \frac{dy(t)}{dt}\Delta t + \frac{d^2y(t)}{dt^2}\frac{(\Delta t)^2}{2!} + \frac{d^3y(t)}{dt^3}\frac{(\Delta t)^3}{3!} + \cdots$$

c.) The taylor series with remainder term is as following

$$y(t + \Delta t) = y(t) + \Delta t y'(t) + \frac{(\Delta t)^2 y''(t)}{2!} + \frac{(\Delta t)^3 y'''(t)}{3!} + \cdots \frac{(\Delta t)^n y^n(\tau)}{n!}$$

Where τ is an arbitrary number between t and $t + \Delta t$

Taylor series of first term,

$$y(t+\Delta t) = y(t) + \Delta t y'(t) + \frac{1}{2}\Delta t^2 y''(\tau)$$

$$y'(t) = \frac{y(t+\Delta t) - y(t)}{\Delta t} + \frac{1}{2}\Delta t^2 y''(\tau)$$

$$y'(t) = \frac{y(t+\Delta t) - y(t)}{\Delta t} + O(\Delta t)$$

d.) Order of local truncation error = 2
Local truncation error is the difference between the true solution of the differential equation and Eulers method approximation.

$$y(t_{n+1}) = yt_n + \int_{t_n}^{t_{n+1}} f\big(t, y(t)\big)\, dt$$

$$y_{n+1} = y_n + f(t_n, y_n)\Delta t$$

$$Local\ truncation\ error = y(t_{n+1}) - y_{n+1}$$

e.) Order of global truncation error = 2

$$e_n = |y(t_n) - y_n|$$

f.) Yes, we can use Eulers method for 2nd and 3rd order polynomials by converting it to a first order polynomial.

g.) Can be used to solve linear and nonlinear differential equations
Can be used to derive dynamic equations
Can be used for inverse dynamic computations

h.) If the step size is too large, results will not be accurate
If the step size is too small, computation time will be higher
Cannot be solved explicitly

II.

```
#part II

def euler_method(f,x0,t0,dt,num_steps):       #Defining euler method function

    t = [t0] #list of time values
    x = [x0] #list of approximate solution values

    # f= the derivative function
    # x0= initial value of x at t0
    # t0= the initial time value
    # dt= the time step size
    # num_steps= number of steps to take

    for i in range(num_steps):
        t_value = t[i] + dt
        x_value = x[i] + dt*(f(x[i]))

        t.append(t_value)
        x.append(x_value)

    return t,x
```

III.

a.)

$$\frac{dx}{dt} = ax - bx^2$$

$$x(t) = \frac{100e^{0.1t}}{e^{0.1t} - 0.95}$$

$$x(t) = 275.333$$

b.)

```
#part III
#(b)
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return 0.1*x - 0.001*(x**2)

tValue , xValue = euler_method(f,2000,0,0.001,4000)  #calling eulers method function


print("x value:",xValue[-1])
```

```
x value: 275.22806739414483
```

c.)

```
#part III
#(c)
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

percentageError = (np.abs(xValue[-1]-275.333)/275.333)*100  #finding the percentage error
print("Percentage Error:",percentageError)
```

```
Percentage Error: 0.03811116206745901
```

As the percentage error is very less, the answer obtained is accurate for a greater extent.

IV.

a.)

$$I = C\frac{dV}{dt}$$

$$I = \frac{V_{in} - V}{R}$$

$$C\frac{dV}{dt} = \frac{V_{in} - V}{R}$$

$$\frac{dV}{dt} = \frac{V_{in} - V}{RC}$$

b.) RC = 2, V_{in} = 5

$$\frac{dV}{dt} = \frac{5 - V}{2}$$

$$\int \frac{2}{5 - V} dV = \int dt$$

$$-2\ln(5 - V) = t + c$$

t=0, V=0

$$-2\ln(5) = c$$

$$c = -3.2189$$

When t=4,

$$4 - 3.2189 = -2\ln(5 - V)$$

$$V = 4.3234$$

c.)

```
#part IV
#(c)
def f(x):
    return (5-x)/2

tValues , xValues = euler_method(f,0,0,0.01,400)      #calling eulers method function

print("x value:",xValues[-1])
```

```
x value: 4.326709785369929
```

d.)

```
#part IV
#(d)

percentageError = ((xValues[-1]-4.3234)/4.3234)*100        #calculating percentage error
print("Percentage Error:",percentageError)
```

```
Percentage Error: 0.07655515034299104
```

As the percentage error is very less, the answer obtained is accurate.

V. a.)

$$F = -bv - kx$$

$$ma + bv + kx = 0$$

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

b.)

$$when\ b^2 = 4mk\ ,$$

$$roots\ are, \frac{-b}{2m}\ , \frac{-b}{2m}$$

$$x(t) = e^{\frac{-bt}{2m}}(C_1 + C_2)$$

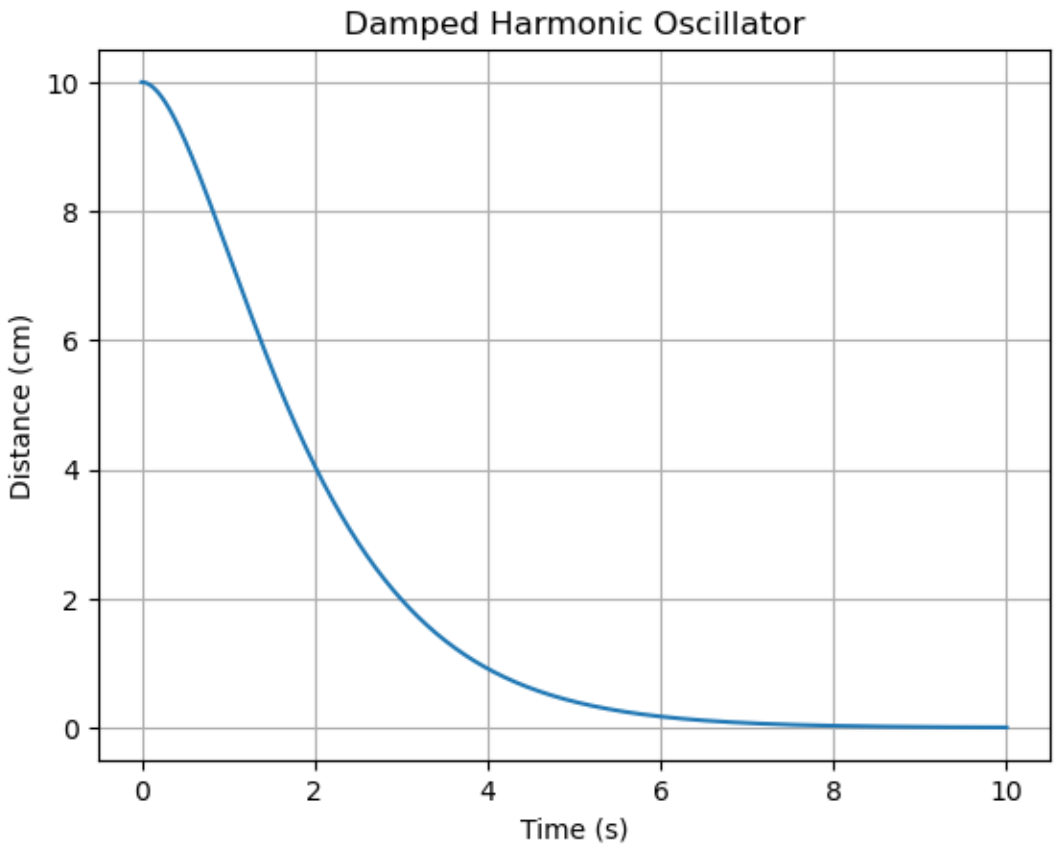
when t=0 and $x(0) = 10$,

$$C_2 = 10$$

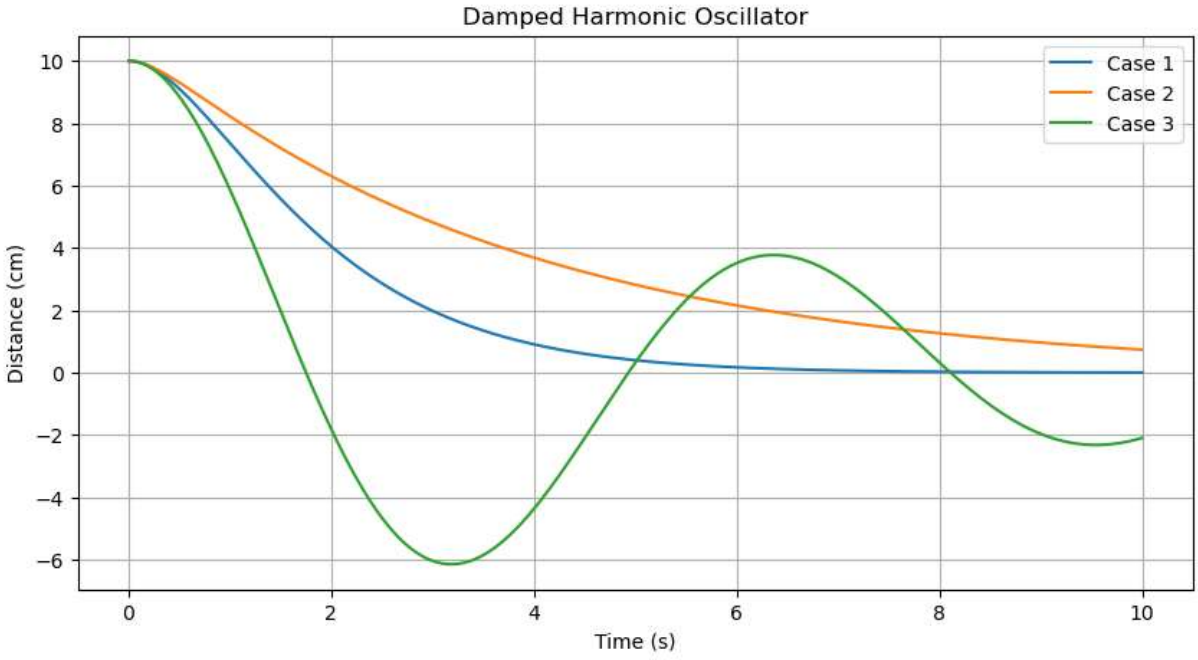
when t=0 and $x'(0) = 0$,

$$C_2 = -10\left(\frac{-b}{2m}\right)$$

```
#part V
#(b)

import numpy as np
import matplotlib.pyplot as plt

def damp(t, C1, C2, b, m):
    return C1 * np.exp(-b/(2*m)*t) + C2 * t * np.exp(-b/(2*m)*t)

t = np.linspace(0, 10, 1000)   #Parameters
m = 1.0
k = 1.0

b = np.sqrt(4 * m * k)  # Damping coefficient of b^2 = 4mk
C1_1 = 10.0
C2_1 = -10 * (-b/(2*m))
x_1 = damp(t, C1_1, C2_1, b, m)


plt.plot(t, x_1)
plt.xlabel('Time (s)')
plt.ylabel('Distance (cm)')
plt.title('Damped Harmonic Oscillator')
plt.grid(True)
plt.show()
```



c.)



Results are very accurate.

3. Second Order Runge-Kutta Method for Solving ODE

I.

a.) $$y(t_{n+1}) = y(t_n) + h\left(\frac{dy}{dt}\right)|_{t_n} + \frac{h^2}{2}\left(\frac{d^2y}{dt^2}\right)|_{t_n} + O(h^3)$$

$$\frac{dy}{dt} = f(y,t)$$

$$\frac{d^2y}{dt^2} = \frac{df(y,t)}{dt} = \frac{\partial y}{\partial t} + \frac{\partial y}{dt}\left(\frac{dy}{dt}\right) = \frac{\partial f}{\partial t} + f\left(\frac{\partial f}{\partial y}\right)$$

$$y_{n+1} = y_n + hf(y_n,t_n) + \frac{h^2}{2}\left[\frac{\partial f}{\partial t} + f\left(\frac{\partial f}{\partial y}\right)\right](y_n,t_n) + O(h^3)$$

$$k_2 = hf(y_n + \beta k_1, t_n + \alpha h)$$

$$= h(f(y_n,t_n) + \alpha h\frac{\partial f}{\partial t}(y_n,t_n) + \beta k_1\frac{\partial f}{\partial t}(y_n,t_n)$$

substituting for k_2,

$$y_{n+1} = y_n + (a+b)hf(y_n,t_n) + bh^2\left[\alpha\frac{\partial f}{\partial t} + \beta f\left(\frac{\partial f}{\partial y}\right)\right](y_n,t_n) + O(h^3)$$

Comparing the terms,

$$a + b = 1, \alpha b = \frac{1}{2}, \beta b = \frac{1}{2}$$

$$k_1 = hf(y_n,t_n)$$

$$k_2 = hf(y_n + k_1, t_n + h)$$

Second order runge kutta method,

$$y_{n+1} = y_n + \frac{k_1 + k_2}{2}$$

b.) Truncation error = $O(h^3)$

c.) Due to the higher degree of accuracy, runge kutta method has high convergence than that of eulers method

d.) Local truncation error: Error which occurs in each step which is = $O(h^3)$
Global truncation error: Errors which occurs in all steps

e.) When the step size decreases, accuracy increases as the step size represents sub intervals

II.

```
#part2
#defining second order runge kutta method

def second_order_runge_kutta(f, x0, y0, h, num_steps):
    xValues = [x0]
    yValues = [y0]

    for i in range(num_steps):
        x = xValues[-1]
        y = yValues[-1]

        k1 = h * f(y)
        k2 = h * f(y + k1/2)

        xNew = x + h
        yNew = y + k2

        xValues.append(xNew)
        yValues.append(yNew)

    return xValues, yValues
```

III.

a.) Analytical value = 275.3335

b.)

```
#partIII
#b
def f(x):
    return (0.1 * x) - (0.001 * (x**2))

def solve_bacteria_population(x0, num_st):
    h = 0.01  # Step size

    xValues, yValues = second_order_runge_kutta(f, 0, x0, h, num_st)

    final_population = yValues[-1]
    return final_population

x0 = 2000
num_st = 400

final_population1 = solve_bacteria_population(x0, num_st)
print("Amount of bacteria after 4 hours:", final_population1)
```

```
Amount of bacteria after 4 hours: 275.33985386727664
```

c.)

```
#PartIII
#c

error = ((final_population1-275.3335)/275.3335)*100    #percentage error
print("Percentage error:",error)
```

```
Percentage error: 0.0023076985824935572
```

IV.

c.)

```
#PartIV
#c

def f(x):
    return (5-x)/2

tValues , xValues = second_order_runge_kutta(f, 0 , 0, 0.01, 400)


print("x value:",xValues[-1])
```

```
x value: 4.323317923634698
```

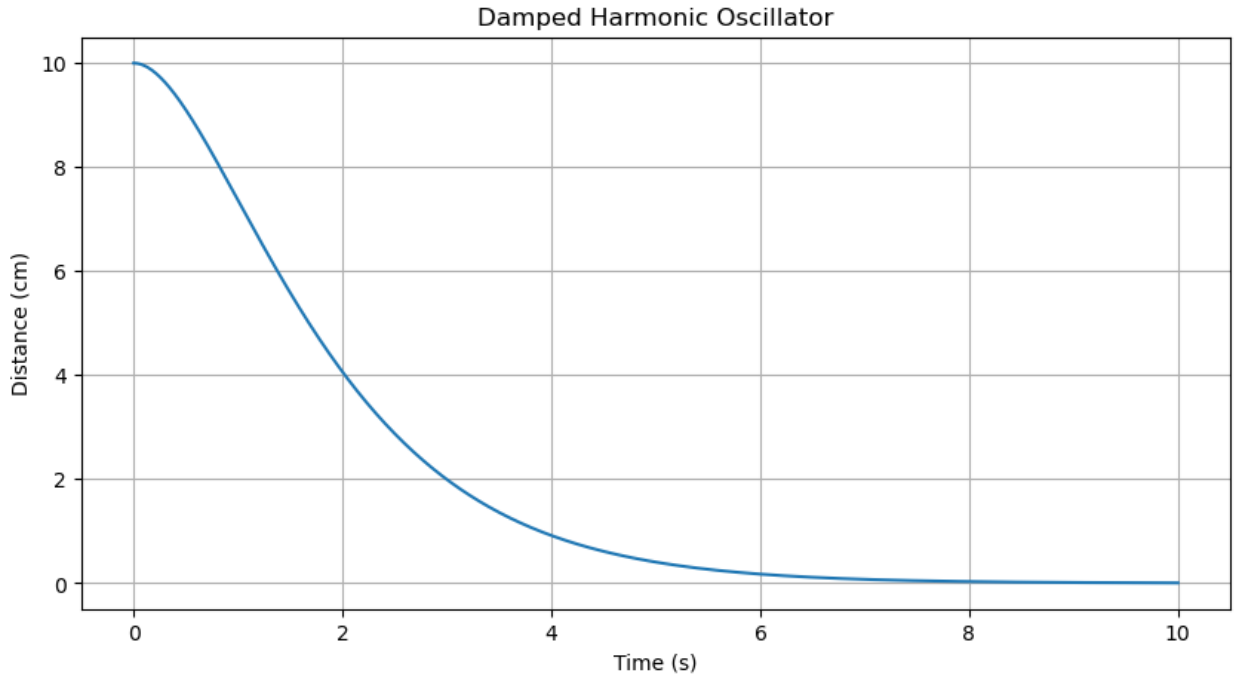
d.)

```
#PartIV
#d
#percentage error
import numpy as np

error = (np.abs(xValues[-1]-4.3234)/4.3234)*100
print("Percentage error:",error)
```

```
Percentage error: 0.0018984217352648 71
```

Accuracy has increased more than eulers method

v.)



4. Fourth Order Runge-Kutta Method for Solving ODE

I.

a.)

$$k_1 = hf(y_n, t_n)$$

$$k_2 = hf(y_n + \frac{k_1}{2}, t_n + \frac{h}{2})$$

$$k_3 = hf(y_n + \frac{k_2}{2}, t_n + \frac{h}{2})$$

$$k_4 = hf(y_n + k_3, t_n + h)$$

$$y_{n+1} = y_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

b.)

Truncation error = $O(h^5)$

c.) Accuracy is very high compared to Eulers method
Fourth order runge kutta method can handle non linear functions also

d.) As the order of truncation error is high, the accuracy is also higher

II.

```
#defining fourth order runge kutta method

def fourth_order_runge_kutta(f, x0, y0, h, num_steps):
    xValues = [x0]
    yValues = [y0]

    for i in range(num_steps):
        x = xValues[-1]
        y = yValues[-1]

        k1 = h * f(y)
        k2 = h * f(y + k1/2)
        k3 = h * f(y + k2/2)
        k4 = h * f(y + k3)

        xNew = x + h
        yNew = y + (1/6)*(k1 + 2*k2 + 2*k3 + k4)

        xValues.append(xNew)
        yValues.append(yNew)

    return xValues, yValues
```

III.

IV.

b.)

```
#part3
#b
def f(x):
    return (0.1 * x) - (0.001 * (x**2))

def solve_bacteria_population(x0, num_st):
    h = 0.01  # Step size

    xValues, yValues = fourth_order_runge_kutta(f, 0, x0, h, num_st)

    final_population = yValues[-1]
    return final_population

x0 = 2000
num_st = 400

finalPopulation = solve_bacteria_population(x0, num_st)
print("Amount of bacteria after 4 hours:", finalPopulation)
```

```
Amount of bacteria after 4 hours: 275.3334619861144
```

c.)

```
#part3
#c
import numpy as np
#percentage error

error = (np.abs(finalPopulation-275.3335)/275.3335)*100
print("Percentage error:",error)
```

```
Percentage error: 1.3806487628428267e-05
```

IV.)

c.)

```
#part4
#c
def f(x):
    return (5-x)/2

tValues , xValues = fourth_order_runge_kutta(f, 0 , 0, 0.01, 400)


print("x value:",xValues[-1])
```

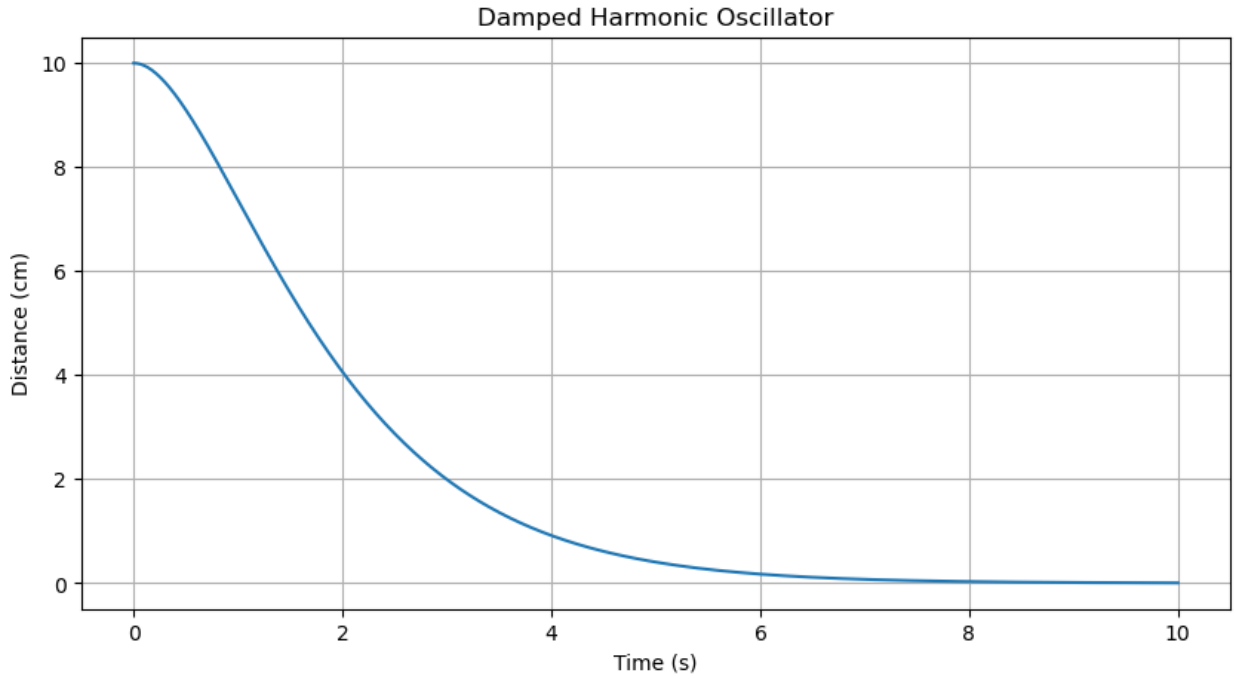
```
x value: 4.323323583809859
```

d.)

```
##part4
#c
#percentage error

error = (np.abs(xValues[-1]-4.3234)/4.3234)*100
print("Percentage error:",error)
```

```
Percentage error: 0.0017675022006055905
```

V.)



5. Python ODE Solvers

I.)
a.) 275.3335

b.)

```
from scipy.integrate import solve_ivp

def bacterial_population(t,x):
    a = 0.1
    b = 0.001
    f = a * x - b * x**2
    return f

#initial conditions
x0 = 2000
t_span = (0, 4)

# Solving the differential equation
solution = solve_ivp(bacterial_population, t_span, [x0])

# Extract the solution at t = 4
t = 4
bacteriaCount = solution.y[0][-1]

print("Amount of bacteria after 4 hours:",bacteriaCount)
```

```
Amount of bacteria after 4 hours: 275.8245389607119
```

```
: import numpy as np
  #percentage error

  error = (np.abs(275.8245-275.3335)/275.3335)*100
  print("Percentage error:",error)
```

```
Percentage error: 0.17832918987336646
```

II.)

We can observe that the accuracy of the Runge-Kutta methods begins to rise when compared to Euler's approach while employing the Second Order Runge-Kutta method, the Fourth Order Runge-Kutta method, and Euler's method. It's also important to remember that the Runge-Kutta method in fourth order has more accuracy than the Runge-Kutta method in second order. The fourth order Runge Kutta method is preferable in question (III) because of its exceptionally high accuracy and incredibly low error percentage. All four approaches, including the solve_ivp() method, yield satisfactory results. It is preferable to employ the second order Runge-Kutta method due to its efficiency.