CPL102

S16036

1. First Derivatives

I.

a.) $f(x) = \frac{f(a)(x-a)^0}{0!} + \frac{f'(a)(x-a)^1}{1!} + \frac{f''(a)(x-a)^2}{2!} + \cdots \ldots \ldots \ldots \ldots \ldots + \frac{f^n(a)(x-a)^n}{n!}$

$$f(x) = \frac{f(a)}{1} + \frac{f'(a)(x-a)^1}{1!} + \frac{f''(a)(x-a)^2}{2!} + \cdots \ldots \ldots \ldots \ldots \ldots + \frac{f^n(a)(x-a)^n}{n!}$$

b.)

1. Forward difference:

The Taylor series polynomial can be written as follows

$$f(x+h) = f(x) + \frac{hf'(x)}{1!} + \frac{h^2f''(x)}{2!} + \frac{h^3f'''(x)}{3!} + \cdots$$

$$f(x+h) - f(x) = \frac{hf'(x)}{1!} + \frac{h^2f''(x)}{2!} + \frac{h^3f'''(x)}{3!} + \cdots$$

$$\frac{f(x+h) - f(x)}{h} = \frac{f'(x)}{1!} + \frac{hf''(x)}{2!} + \frac{h^2f'''(x)}{3!} + \cdots$$

$$\frac{f(x+h) - f(x)}{h} = \frac{f'(x)}{1!} + \underbrace{h\left[\frac{f''(x)}{2!} + \frac{hf'''(x)}{3!} + \cdots\right]}_{O(h)}$$

$$\frac{f(x+h) - f(x)}{h} = \frac{f'(x)}{1!} + O(h)$$

$$f'(x) = \frac{f(x+h) - f(x)}{h} - O(h)$$

2. Backward difference:

The Taylor series polynomial can be written as follows

$$f(x-h) = f(x) - \frac{hf'(x)}{1!} + \frac{h^2f''(x)}{2!} - \frac{h^3f'''(x)}{3!} + \cdots$$

$$f(x-h) - f(x) = -\frac{hf'(x)}{1!} + \frac{h^2f''(x)}{2!} - \frac{h^3f'''(x)}{3!} + \cdots$$

$$\frac{f(x-h)-f(x)}{-h} = \frac{f'(x)}{1!} - \frac{hf''(x)}{2!} + \frac{h^2 f'''(x)}{3!} + \cdots$$

$$\frac{f(x-h)-f(x)}{-h} = \frac{f'(x)}{1!} - \underbrace{h\left[\frac{f''(x)}{2!} - \frac{hf'''(x)}{3!} + \cdots\right]}_{O(h)}$$

$$\frac{f(x-h)-f(x)}{-h} = \frac{f'(x)}{1!} - O(h)$$

$$f'(x) = \frac{f(x-h)-f(x)}{-h} + O(h)$$

$$f'(x) = \frac{f(x)-f(x-h)}{h} + O(h)$$

---

3. Central difference:

$$f(x+h) = f(x) + \frac{hf'(x)}{1!} + \frac{h^2 f''(x)}{2!} + \frac{h^3 f'''(x)}{3!} + \cdots \longrightarrow \text{Equation 1}$$

$$f(x-h) = f(x) - \frac{hf'(x)}{1!} + \frac{h^2 f''(x)}{2!} - \frac{h^3 f'''(x)}{3!} + \cdots \longrightarrow \text{Equation 1}$$

(1) – (2),

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2h^3 f'''(x)}{3!} + \cdots$$

$$f(x+h) - f(x-h) = 2h\left[f'(x) + \frac{h^2 f'''(x)}{3!} + \cdots\right]$$

$$\frac{f(x+h)-f(x-h)}{2h} = f'(x) + \underbrace{\frac{h^2 f'''(x)}{3!} + \cdots}_{O(h)}$$

$$\frac{f(x+h)-f(x-h)}{2h} = f'(x) + O(h)$$

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} - O(h)$$

c.) When we are doing numerical simplifications using large series functions of Taylor polynomial, on most occasions we remove the end part of the polynomial. The error which occurs because of that is called as the truncation error. Value of the truncation error is equal to the difference of the numerical approximation and the analytical derivative. Truncation error is indicated by the big O notation.

d.) When the order of the polynomial increases, the truncation error gets lower as when h approaches to zero. Truncation of forward difference and backward difference formula are in the order of O(h) and the truncation error of central difference formula are in the order of $O(h^2)$. As the degree of truncation error of central difference formula is larger, it has the lowest truncation error.

e.)

Backward difference method	Forward difference method	Central difference method
• Method is easy and simple • Truncation error is higher than central difference method • Accuracy is less than central difference method • Used to approximate future data when future data is not available	• Method is easy and simple • Truncation error is higher than central difference method • Accuracy is less than central difference method • Used in single step differential equations and in predictor corrector methods • This method is suitable for functions with known data points.	• Has the highest accuracy • Number of mathematical computations are it higher • Used when both past and future values are known. • Used to solve partial differential equations

f.) Step size sets the accuracy of the order of approximation.
When the step size is smaller, it produces a small truncation error and thereby it increases the accuracy. But if the step size is very small, there can be calculation errors and rounding off errors as well. So, we should select a step size which is not very large and not very low.

g.) Sources of errors of the above methods are step size, round off error, truncation error.

When the step size decreases, truncation error also decreases but lower step size is sensitive to noise and calculation errors.
If the function is smooth, the vales of the function around the point of differentiation are also accurate. So, the error can be reduced if the above function is smoother.

II.

```
#Question II

def num_diff(f,x,h,method):
    if method == "forward":
        df = (f(x+h)-f(x))/h
    elif method == "backward":
        df = (f(x)-f(x-h))/h
    elif method == "central":
        df = (f(x+h)-f(x-h))/(2*h)
    else:
        print("method error")
        return
    return df
```

III.

h	y' (forward)	y' (backward)	y' (central)	Expected solution	Error % (forward)	Error % (Backward)	Error % (Central)
0.5	17.25	11.25	14.25	14	23.21	-19.64	1.78
0.2	15.24	12.84	14.04	14	8.85	-8.28	0.28
0.1	14.61	13.41	14.01	14	4.35	-4.21	0.07
0.05	14.30	13.70	14.00	14	2.16	-2.12	0.01
0.01	14.06	13.94	14.00	14	0.43	-0.42	0.00
0.001	14.00	13.99	14.00	14	0.04	-0.04	0.00

```python
#Question III

import numpy as np
import math

def f(x):
    return (x**3)+(2*x)-1     #defining the function f(x)

def error_perc(df):
    return ((math.sqrt((df-14)**2))/14)*100          #defining the error function of f(x)

h = np.array([0.5,0.2,0.1,0.05,0.01,0.001])       #Defining the step sizes

forwardDifference  = []
backwardDifference = []
centralDifference  = []

forwardDifferenceError = []      # percentage errors
backwardDifferenceError = []
centralDifferenceError = []

for i in h:                      #forward difference method
    yf = num_diff(f,2,i,"forward")
    yf = round(yf,4)
    forwardDifference.append(yf)
print("Forward difference derivatives:")
print(forwardDifference)
print("")

for i in forwardDifference:      #Percentage errors of forward difference derivatives
    err_f = error_perc(i)
    err_f = round(err_f,3)
    forwardDifferenceError.append(err_f)
print("Percentage errors of forward difference derivatives:")
print(forwardDifferenceError)
print("")
```

```
for i in h:                          #backward difference method
    yb = num_diff(f,2,i,"backward")
    yb = round(yb,4)
    backwardDifference.append(yb)
print("Backward difference derivatives:")
print(backwardDifference)
print("")

for i in backwardDifference:     #Percentage errors of backward difference derivatives
    err_b = error_perc(i)
    err_b = round(err_b,3)
    backwardDifferenceError.append(err_b)
print("Percentage errors of backward difference derivatives:")
print(backwardDifferenceError)
print("")

for i in h:                          #central difference method
    yc = num_diff(f,2,i,"central")
    yc = round(yc,4)
    centralDifference.append(yc)
print("Central difference derivatives:")
print(centralDifference)
print("")

for i in centralDifference:      #Percentage errors of central difference derivatives
    err_c = error_perc(i)
    err_c = round(err_c,3)
    centralDifferenceError.append(err_c)
print("Percentage errors of central difference derivatives:")
print(centralDifferenceError)
print("")
```

```
Forward difference derivatives:
[17.25, 15.24, 14.61, 14.3025, 14.0601, 14.006]

Backward difference derivatives:
[11.25, 12.84, 13.41, 13.7025, 13.9401, 13.994]

Central difference derivatives:
[14.25, 14.04, 14.01, 14.0025, 14.0001, 14.0]
```

2. Second Order Derivatives

I.

a.)

$$f(x+h) = f(x) + \frac{hf'(x)}{1!} + \frac{h^2 f''(x)}{2!} + \frac{h^3 f'''(x)}{3!} + \cdots \longrightarrow \text{Equation 1}$$

$$f(x-h) = f(x) - \frac{hf'(x)}{1!} + \frac{h^2 f''(x)}{2!} - \frac{h^3 f'''(x)}{3!} + \cdots \longrightarrow \text{Equation 2}$$

1 + 2,

$$f(x+h) + f(x-h) = 2f(x) + \frac{h^2 f''(x)}{1} + \frac{h^4 f''''(x)}{12} + \cdots$$

$$\frac{f(x+h) + f(x-h) - 2f(x)}{h^2} = f''(x) + \frac{h^2 f''''(x)}{12} + \cdots$$

$$\frac{f(x+h) + f(x-h) - 2f(x)}{h^2} = f''(x) + h^2 \left[\frac{f'''(x)}{12} + \cdots\right]$$

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} - h^2 \left[\frac{f'''(x)}{12} + \cdots\right]$$

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + O(h^2)$$

b.)

$$Truncation\ error = h^2 \left[\frac{f'''(x)}{12} + \cdots\right]$$

Its order is 2.

c.) No, as we assumed that the step size is uniform when deriving the equations we cannot use the above method to compute the second derivative if the step size is not uniform. By adjusting the step size, we can use the above method to compute the second derivative although the step size is not uniform.

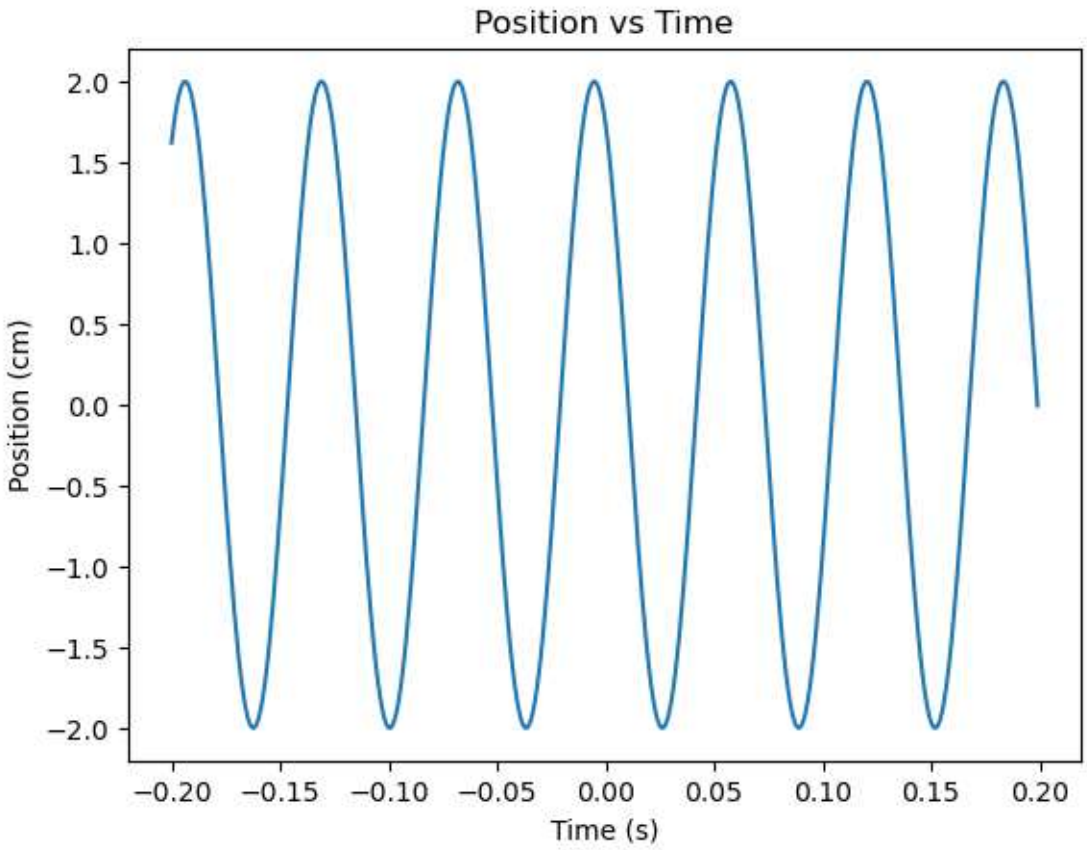
d.) Accuracy will drop if the function is discontinuous. At sharp edges function will change rapidly. When there are high frequency components, the function may not capture the true behavior.

II.

```
#Question 2
#part 2

def central_diff(f, x, h, order=1):
    if order == 1:
        return (f(x+h) - f(x-h)) / (2*h)
    elif order == 2:
        return (f(x + h) - 2 * f(x) + f(x - h)) / (h ** 2)
    else:
        return ("Error in the order")
```

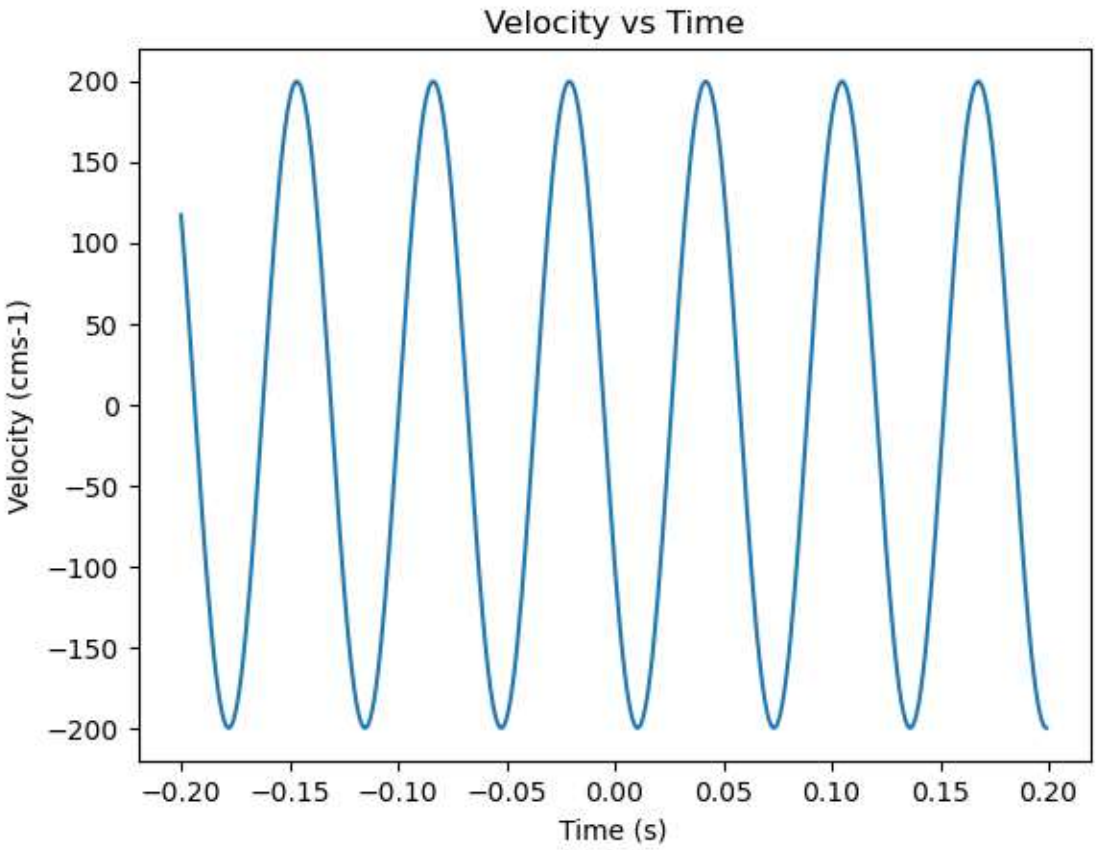
III. a)

```
#Question 2
#part 3
#a)

import numpy as np
import matplotlib.pyplot as plt

A = 2
omega = 100
O = np.pi/6

t = np.arange(-0.2,0.2,0.001)

x = A*np.cos(omega*t+O)

plt.plot(t1, x)
plt.xlabel('Time (s)')
plt.ylabel('Position (cm)')
plt.title('Position vs Time')
plt.show()
```



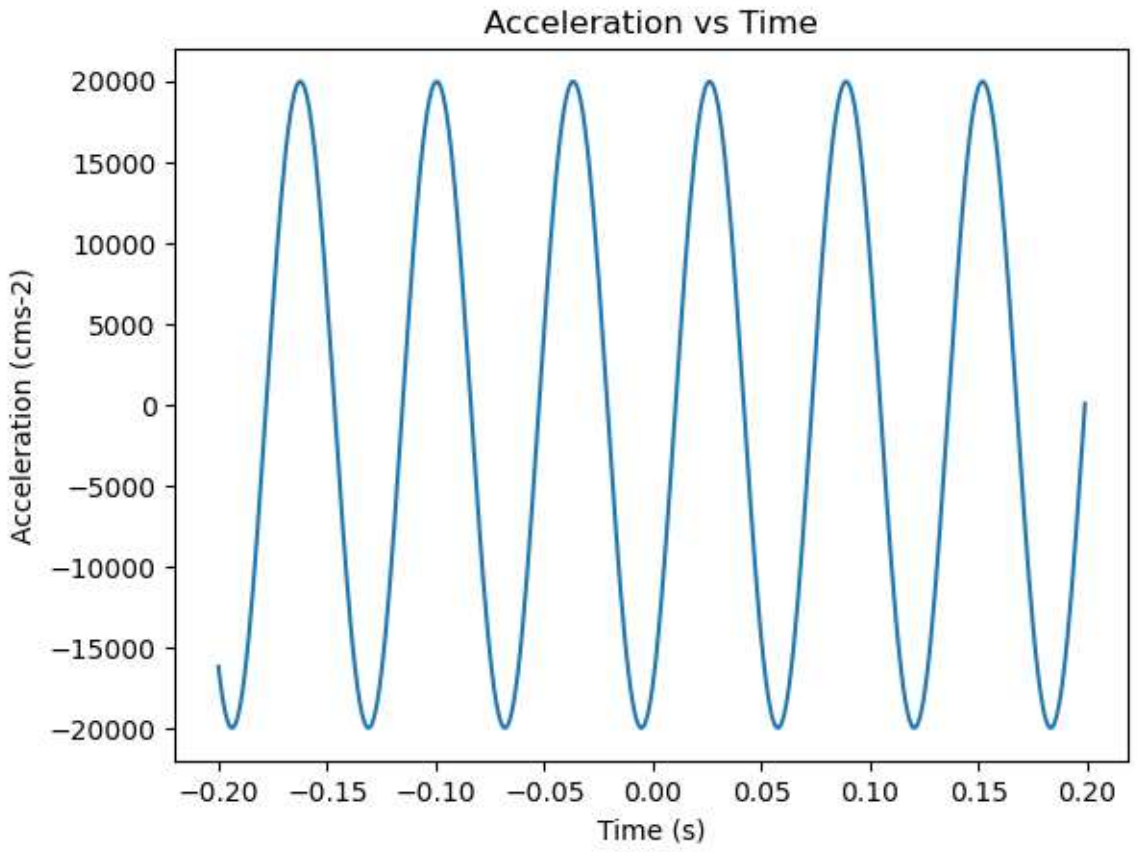
b.)

```
#b)
def f(x):
    return 2*np.cos((100*x)+(np.pi/6))

t = np.arange(-0.2,0.2,0.001)
velocity = central_diff(f,t,0.001)

plt.plot(t, velocity)
plt.xlabel('Time (s)')
plt.ylabel('Velocity (cms-1)')
plt.title('Velocity vs Time')
plt.show()
```



c.)

```
#c)
#t = np.arange(-4, 4, 0.01)
t = np.arange(-0.2,0.2,0.001)
acceleration = central_diff(f,t,0.001,2)

plt.plot(t, acceleration)
plt.xlabel('Time (s)')
plt.ylabel('Acceleration (cms-2)')
plt.title('Acceleration vs Time')
plt.show()
```



3. Integration: Trapezoidal rule

I.

a.) $$\int_a^b f(x)dx = \sum_{i=0}^{n-1} h\left(\frac{f(x_i)+f(x_i+1)}{2}\right)$$

$$x_i \epsilon [a, b]$$

b.) Trapezoidal rule is very simple, and it is very accurate as its order of error is 3. So, it is very smooth. Furthermore, it is smooth. So, it can be used for many integration tasks as it is numerically stable also.

c.) Order of accuracy = 3

d.) Large number of intervals are required for more accurate results.

Less accurate with the functions having rapid changes.

Can be used only for a limited interval and not up to infinity.

e.) First a piecewise continuous function should be made using the discontinuous function. After that trapezoidal rule must be applied for all continuous functions in it.

f.) Decrease the step size and increase the number of sub intervals

II.

```
#Question 3
#part 2

import numpy as np

def trapezoidal_integration(f,a,b,h):

    n = int((b-a)/h) + 1
    x = np.linspace(a, b, n)
    integral_value=((h/2)*(f(x[0]) + 2*sum(f(x[1:n-1])) + f(x[n-1])))

    return integral_value
```

III.

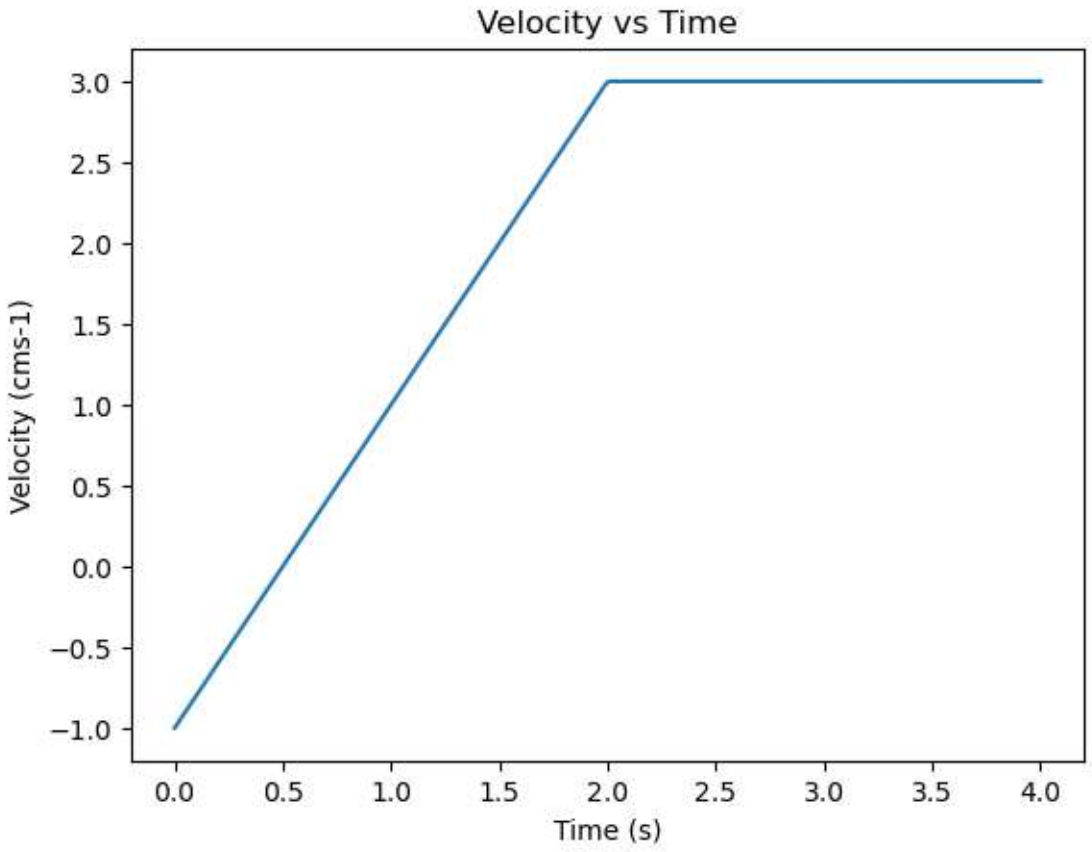
n	m(Trapezoid)	m(Expected)	Error (%)
0.5	21.23	20.31	4.57
0.1	20.35	20.31	0.18
0.05	20.32	20.31	0.05
0.01	20.31	20.31	0.00
0.001	20.31	20.31	0.00

```python
#Question 3
#part 3

import sympy as sym
import matplotlib.pyplot as plt

x = sym.symbols('x')
m = 2*np.pi*x*x*sym.sqrt(x)

expectedSolution = float(sym.integrate(m, (x, 0, 2)))
print('Expected solution:',expectedSolution)

def m(x):       #mass radi
    m=2*np.pi*x*x*np.sqrt(x)
    return m


print(f'h   \t\t m    \t\t % Error ')

for i in ([0.5, 0.1, 0.05, 0.01, 0.001]):
    myVal = trapezoidal_integration(m, 0, 2, i)
    print(f'{i} \t {myVal}  \t {((myVal-expectedSolution)/expectedSolution)*100}')
```

```
Expected solution: 20.310322003009674
h                m                % Error
0.5      21.2399302507689         4.57702368097105
0.1      20.347361792981328       0.18236928969499175
0.05     20.319579432565266       0.04557992509533288
0.01     20.310692248490167       0.0018229424449211632
0.001    20.31032570541382        1.822917502870116e-05
```

IV.

```python
#Question 3
#part 4

def v(t):
    if t >= 2:
        return 3
    else:
        return 2*t -1

t = np.linspace(0, 4, 250)
vt = np.zeros_like(t)

for i in range(len(t)):
    vt[i] = v(t[i])

plt.plot(t, vt)
plt.title('Velocity vs Time')
plt.xlabel('Time (s)')
plt.ylabel('Velocity (cms-1)')

n_values = np.array([4,20,40,200,2000])

displacements = []

for i in n_values:
    value = trapezoidal_integration(vt,0,4,i)
    displacements.append(displacements)

print(displacements)
```



4. Integration: Simpsons rule

I.

a.)

$$\int_a^b f(x)dx = \frac{h}{3}\{f(x_0) + 4[f(x_1) + f(x_3) + \cdots + f(x_{n-1})] + 2[f(x_2) + f(x_4) + \cdots + f(x_{n-2})] + f(x_n)\}$$

b.) Accuracy of Simpsons rule is higher as order of error is larger than others. It converges towards the true value faster when integrating.

c.) Order of accuracy = 3

d.) Computational cost is higher.

Sensitive to end points
Can be used only when there are even number of intervals.

e.) Convert the discontinuous function into a piece wise and then apply Simpsons rule.

f.) Divide the integrating function into sub intervals and treat each interval as a different integrating problem.

II.

```
#Question 4
#part 2

from sympy import *
import numpy as np
import matplotlib.pyplot as plt

def simpsons_rule(f, a, b, h):
    n = int((b-a)/h) + 1
    x = np.linspace(a, b, n)
    return ((h/3)*(f(x[0]) + 2*sum(f(x[2:-1:2])) + 4*sum(f(x[1:-1:2])) + f(x[-1])))
```

III.

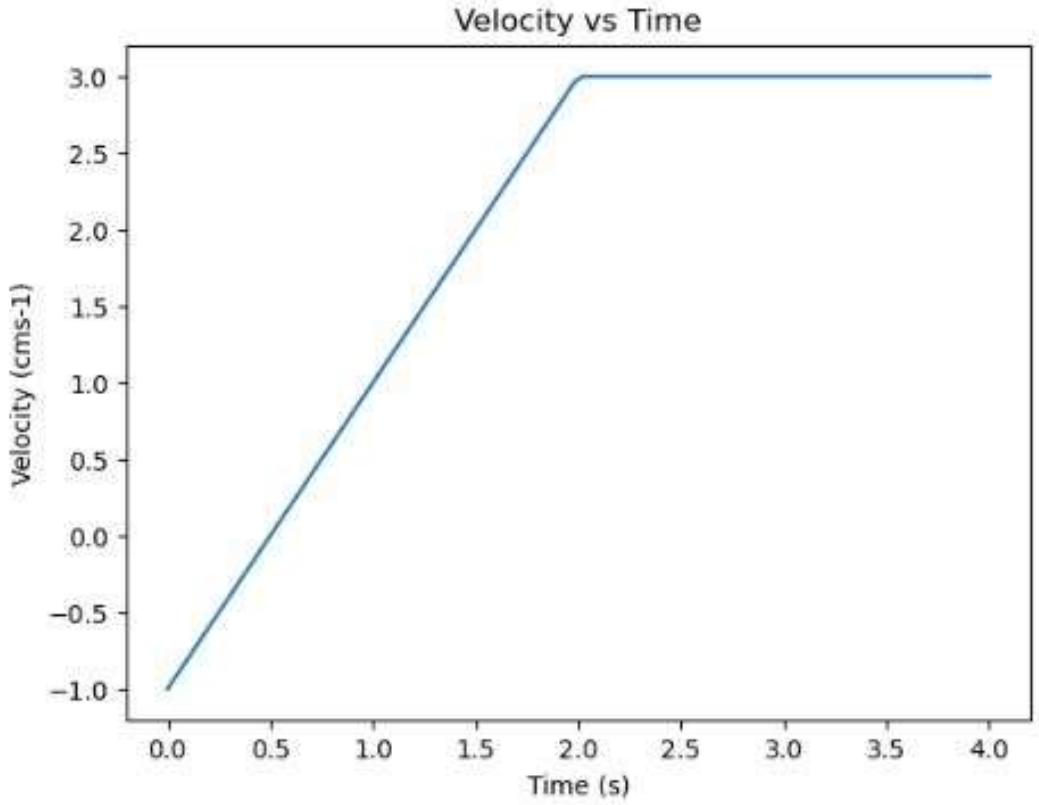
h	m(Trapezoid)	m(Expected)	Error (%)
0.5	20.3017	20.3103	0.046
0.1	20.3103	20.3103	0.000
0.05	20.3103	20.3103	1.652
0.01	20.3103	20.3103	6.195
0.001	20.3103	20.3103	2.008

```python
#Question 4
#part 3

x = symbols('x')
g = 2*np.pi*x*x*sqrt(x)

expectedSolution = float(integrate(g, (x, 0, 2)))
print('Expected solution:',expectedSolution)

g = lambda x: 2*np.pi*x*x*np.sqrt(x)

def _err(y, y_exp):
    return ((y - y_exp)/y_exp)*100

print(f'h   \t\t m    \t\t\t % Error ')

for i in ([0.5, 0.1, 0.05, 0.01, 0.001]):
    myVal = simpsons_rule(g, 0, 2, i)
    print(f'{i} \t {myVal}   \t {_err(myVal, expectedSolution)}')
```

```
Expected solution: 20.310322003009674
h                m                          % Error
0.5      20.30166798108752       -0.04260898434240632
0.1      20.31028537472705       -0.00018034319011115462
0.05     20.310318645759914      -1.6529771214235794e-05
0.01     20.310321990426424      -6.195495076675045e-08
0.001    20.310322003005595      -2.0080997744194322e-11
```

IV.

```python
#Question 4
#part 4

def v(t):
    if t<2:
        return (2*t)-1
    else:
        return 3

t_val = np.linspace(0, 4, 100)
v_val = []

for i in t_val:
    x4 = v(i)
    v_val.append(x4)

plt.plot(t_val, v_val)
plt.xlabel('Time (s)')
plt.ylabel('Velocity (cms-1)')
plt.title('Velocity vs Time')
plt.show()

t = symbols('t')
expectedSolution = integrate(2*t-1, (t, 0, 2)) + integrate(3, (t, 2, 4))
print('Expected solution:',expectedSolution)

for i in ([0.5, 0.1, 0.05, 0.01, 0.001]):
    myVal = simpsons_rule(lambda t: 2*t-1, 0, 2, i) + simpsons_rule(lambda t: 3*t/t, 2, 4, i)
    print(f'h=  {i} \t s = {myVal}    \t % Error = {_err(myVal, expectedSolution)}')
```




```
Expected solution: 8
h=  0.5          s = 8.0          % Error = 0
h=  0.1          s = 8.0          % Error = 0
h=  0.05         s = 8.0          % Error = 0
h=  0.01         s = 8.0          % Error = 0
h=  0.001        s = 8.0          % Error = 0
```

h	m (Simpsons rule)	m(Expected)	Error (%)
0.5	8.0	8.0	0
0.1	8.0	8.0	0
0.05	8.0	8.0	0
0.01	8.0	8.0	0
0.001	8.0	8.0	0