Table of Index

Patuakhali Science and Technology University (PSTU)

Faculty of Computer Science and Engineering (CSE)

Lab Report on

‘Implementation of Numerical Methods’.

Course Code: CCE-312

Course title: Numerical Methods Sessional.

<u>Submitted to:</u>

Professor Dr. Md. Samsuzzaman

Professor,

Department of Computer & Communication Engineering, PSTU.

***<u>Submitted by</u>*:**

Hasan Ahammad

ID No:1902073

Reg No: 08779

Session:2019-2020

.

Bisection Method.

1. **Use the *Bisection Method* to determine the drag coefficients 'c' needed for a Parachutist of mass m=68.1 kg to have a velocity of 40 m/s after free-falling for timer t=10 s after that Implement it by *Python*. In Python, Result must be shown both *Graphical and Tabular format*. Note: The acceleration due to gravity is 9.81 m/s^2.**

The equation is $f(c)=\frac{gm}{c}\left(1-e^{-\left(\frac{c}{m}\right)t}\right)-v$

Sol^n: Put the given value in the equation we find:

$f(c)=\frac{667.38}{c}(1-e^{-0.146843c})-40$

initial guess interval is a=12 and b=16 because f(12) x f(16)<0.

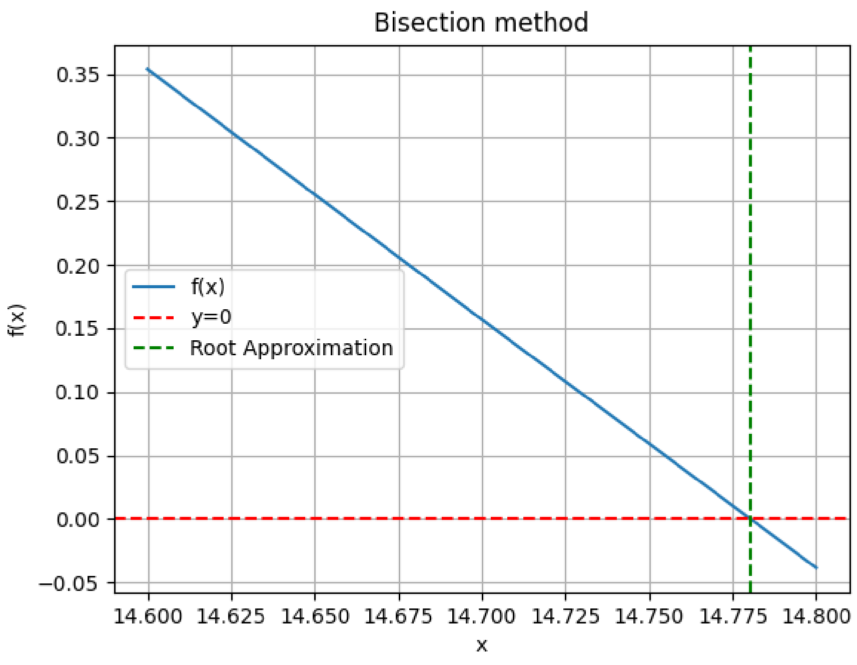
Now use table for further calculation:

Iteration	x_l	x_u	x_r	ε_a (%)	ε_t (%)
1	12	16	14		5.413
2	14	16	15	6.667	1.344
3	14	15	14.5	3.448	2.035
4	14.5	15	14.75	1.695	0.345
5	14.75	15	14.875	0.840	0.499
6	14.75	14.875	14.8125	0.422	0.077

Implement by Python:

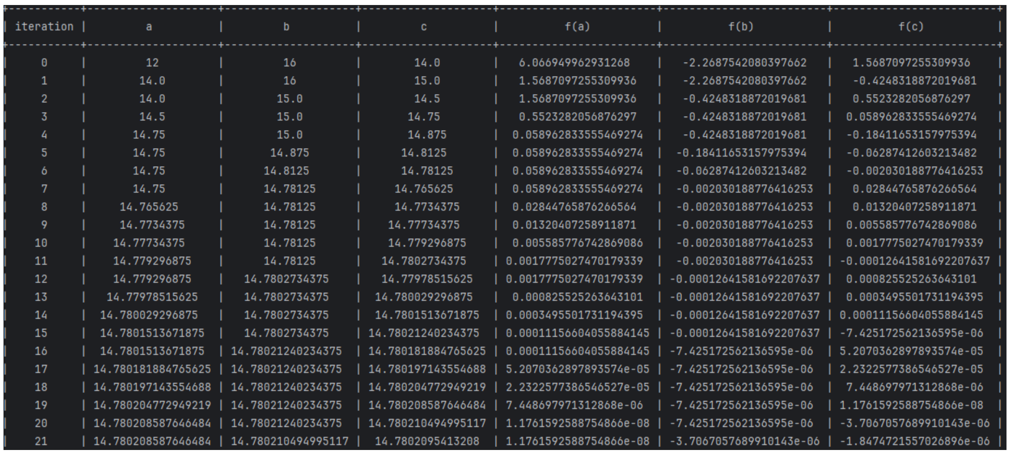
```
import matplotlib.pyplot as plt
from tabulate import tabulate
import math
def bisection(func,a,b,tol=1e-6,max_iter=100):
    iterations=[]
    a_values=[]
    b_values=[]
    c_values=[]
    f_a_values=[]
    f_b_values=[]
    f_c_values=[]

    if func(a)*func(b)>=0:
        raise ValueError("the value is not accurate")
    for i in range(max_iter):
        c=(a+b)/2
        iterations.append(i)
        a_values.append(a)
        b_values.append(b)
        c_values.append(c)
        f_a=func(a)
        f_b=func(b)
        f_c=func(c)
        f_a_values.append(f_a)
        f_b_values.append(f_b)
        f_c_values.append(f_c)

        if f_c==0 or abs(b-a)/2<tol:
            break
        elif f_a * f_c<0:
            b=c
        else:
            a=c

results=list(zip(iterations,a_values,b_values,c_values,f_a_values,f_b
_values,f_c_values))

table=tabulate(results,headers=["iteration","a","b","c","f(a)","f(b)"
,"f(c)"],tablefmt="pretty")
    print(table)

    x=[i/10 for i in range(int(10*min(a,b))-1,int(10*max(a,b))+2)]
    y=[func(xi) for xi in x]

    plt.plot(x,y,label='f(x)')
    plt.axhline(0,color='red',linestyle='--',label='y=0')
    plt.axvline(c,color='green',linestyle='--',label='Root
Approximation')

    plt.title('Bisection method')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.grid(True)
    plt.show()
    return c



def function(c):
    return ((667.38 /c) * (1 - math.exp(-0.146843 * c))) - 40
a=12
b=16
root=bisection(function,a,b)
print(f"The approximate root is:{root:.6f}")
```

- **Result:**
  - **Graph:**



+

➢ **Table:**



```
+-----------+--------------------+--------------------+--------------------+-----------------------+------------------------+------------------------+
| iteration |         a          |         b          |         c          |         f(a)          |          f(b)          |          f(c)          |
+-----------+--------------------+--------------------+--------------------+-----------------------+------------------------+------------------------+
|     0     |         12         |         16         |        14.0        |   6.066949962931268   |  -2.2687542080397662   |   1.5687097255309936   |
|     1     |        14.0        |         16         |        15.0        |  1.5687097255309936   |  -2.2687542080397662   |  -0.4248318872019681   |
|     2     |        14.0        |        15.0        |        14.5        |  1.5687097255309936   |  -0.4248318872019681   |   0.5523282056876297   |
|     3     |        14.5        |        15.0        |       14.75        |  0.5523282056876297   |  -0.4248318872019681   |  0.05896283355469274   |
|     4     |       14.75        |        15.0        |       14.875       |  0.05896283355469274  |  -0.4248318872019681   |  -0.18411653157975394  |
|     5     |       14.75        |       14.875       |      14.8125       |  0.05896283355469274  |  -0.18411653157975394  |  -0.06287412603213482  |
|     6     |       14.75        |      14.8125       |      14.78125      |  0.05896283355469274  |  -0.06287412603213482  | -0.0020301887764l6253  |
|     7     |       14.75        |      14.78125      |     14.765625      |  0.05896283355469274  | -0.002030188776416253  |  0.02844765876266564   |
|     8     |     14.765625      |      14.78125      |     14.7734375     |  0.02844765876266564  | -0.002030188776416253  |  0.01320407258911871   |
|     9     |     14.7734375     |      14.78125      |    14.77734375     |  0.01320407258911871  | -0.002030188776416253  |  0.005585776742869086  |
|    10     |    14.77734375     |      14.78125      |    14.779296875    | 0.005585776742869086  | -0.002030188776416253  |  0.001777502747017 9339 |
|    11     |    14.779296875    |      14.78125      |   14.7802734375    | 0.0017775027470179339 | -0.002030188776416253  | -0.00012641581692207637 |
|    12     |    14.779296875    |   14.7802734375    |   14.77978515625   | 0.0017775027470179339 | -0.00012641581692207637 |  0.000825525263643101  |
|    13     |   14.77978515625   |   14.7802734375    |  14.780029296875   | 0.000825525263643101  | -0.00012641581692207637 | 0.0003495501731194395  |
|    14     |  14.780029296875   |   14.7802734375    |  14.7801513671875  | 0.0003495501731194395 | -0.00012641581692207637 | 0.00011156604055884145 |
|    15     |  14.7801513671875  |   14.7802734375    | 14.78021240234375  | 0.00011156604055884145 | -0.00012641581692207637 | -7.425172562136595e-06 |
|    16     |  14.7801513671875  | 14.78021240234375  | 14.780181884765625 | 0.00011156604055884145 | -7.425172562136595e-06 | 5.207036289789357 4e-05 |
|    17     | 14.780181884765625 | 14.78021240234375  | 14.780197143554688 | 5.2070362897893574e-05 | -7.425172562136595e-06 | 2.2322577386546527e-05 |
|    18     | 14.780197143554688 | 14.78021240234375  | 14.780204772949219 | 2.2322577386546527e-05 | -7.425172562136595e-06 |  7.448697971312868e-06 |
|    19     | 14.780204772949219 | 14.78021240234375  | 14.780208587646484 | 7.448697971312868e-06 | -7.425172562136595e-06 | 1.1761592588754866e-08 |
|    20     | 14.780208587646484 | 14.78021240234375  | 14.780210494995117 | 1.1761592588754866e-08 | -7.425172562136595e-06 | -3.7067057689910143e-06 |
|    21     | 14.780208587646484 | 14.780210494995117 |  14.7802095413208  | 1.1761592588754866e-08 | -3.7067057689910143e-06 | -1.8474721557026896e-06 |
```

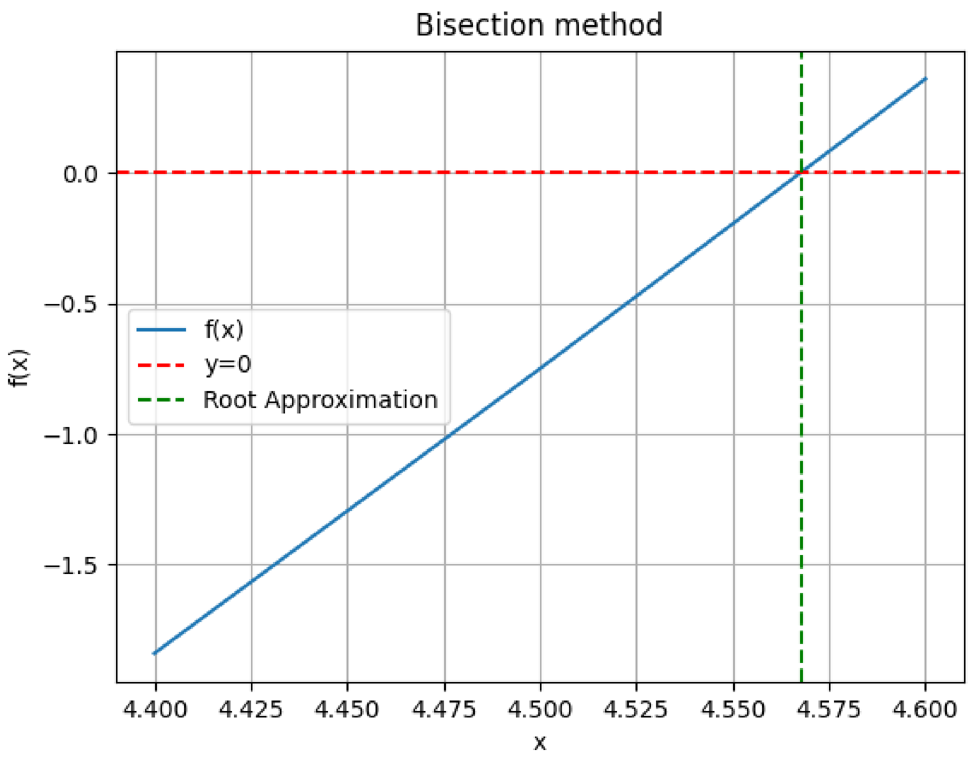
2. **Suppose you own a company that produces a certain quantity of items, and you want to determine the optimal production level that maximizes your profit. Let's model the profit function as a quadratic equation and use the _**bisection method**_ to find the production quantity that yields the maximum profit.**
   The profit(P) can be modeled as a quadratic function of the production quantity(Q): $P(Q)=Q^2+2Q-30$. Here, Q is the Production quantity. Consider an initial production interval [4, 5] since you expect the optimal production to lie in this range. After that _ Implement it by Python**_. In Python, result must be contain both _**graphical and tabular**_ format.**

✪ **Implement Using Python:**

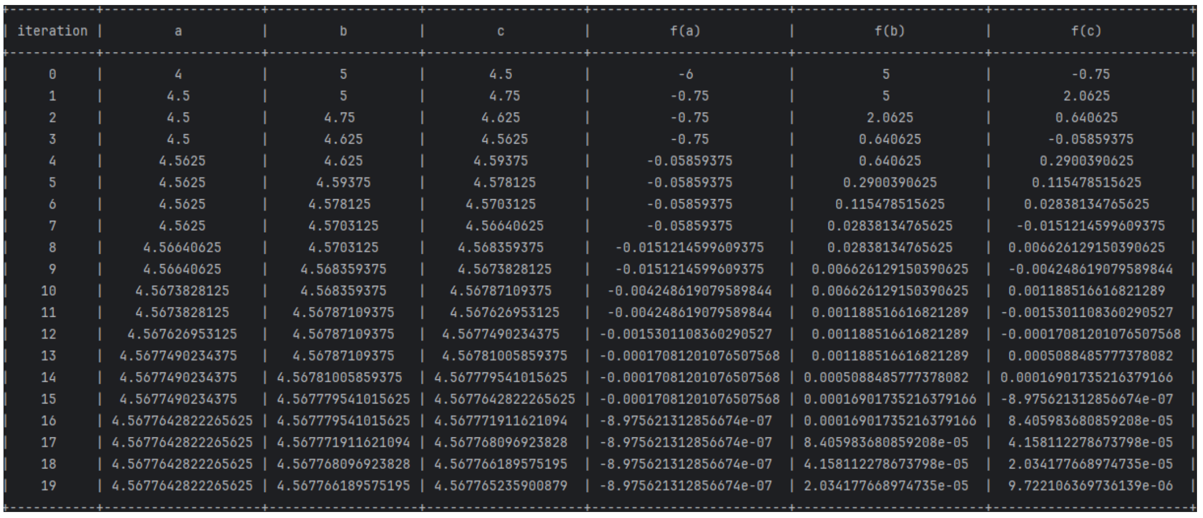
```
import matplotlib.pyplot as plt
from tabulate import tabulate
import math
def bisection(func,a,b,tol=1e-6,max_iter=100):
    iterations=[]
    a_values=[]
    b_values=[]
    c_values=[]
    f_a_values=[]
    f_b_values=[]
    f_c_values=[]

    if func(a)*func(b)>=0:
        raise ValueError("the value is not accurate")
    for i in range(max_iter):
        c=(a+b)/2
        iterations.append(i)
        a_values.append(a)
        b_values.append(b)
        c_values.append(c)
        f_a=func(a)
        f_b=func(b)
        f_c=func(c)
        f_a_values.append(f_a)
        f_b_values.append(f_b)
        f_c_values.append(f_c)

        if f_c==0 or abs(b-a)/2<tol:
            break
        elif f_a * f_c<0:
            b=c
        else:
            a=c
results=list(zip(iterations,a_values,b_values,c_values,f_a_values,f_b
_values,f_c_values))

table=tabulate(results,headers=["iteration","a","b","c","f(a)","f(b)"
,"f(c)"],tablefmt="pretty")
    print(table)

    x=[i/10 for i in range(int(10*min(a,b))-1,int(10*max(a,b))+2)]
    y=[func(xi) for xi in x]

    plt.plot(x,y,label='f(x)')
    plt.axhline(0,color='red',linestyle='--',label='y=0')
    plt.axvline(c,color='green',linestyle='--',label='Root
Approximation')

    plt.title('Bisection method')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.grid(True)
    plt.show()
    return c
def function(Q):
    return Q**2 + 2*Q - 30

a=4
b=5
root=bisection(function,a,b)
print(f"The approximate root is:{root:.6f}")
```

✪ **Result:**

➢ **Graph:**



➢ **Table:**



```
+-----------+--------------------+-------------------+--------------------+-------------------------+------------------------+-------------------------+
| iteration |         a          |         b         |         c          |          f(a)           |          f(b)          |          f(c)           |
+-----------+--------------------+-------------------+--------------------+-------------------------+------------------------+-------------------------+
|     0     |         4          |         5         |        4.5         |           -6            |           5            |          -0.75          |
|     1     |        4.5         |         5         |        4.75        |          -0.75          |           5            |         2.0625          |
|     2     |        4.5         |       4.75        |       4.625        |          -0.75          |         2.0625         |        0.640625         |
|     3     |        4.5         |       4.625       |       4.5625       |          -0.75          |        0.640625        |       -0.05859375       |
|     4     |       4.5625       |       4.625       |      4.59375       |       -0.05859375       |        0.640625        |      0.2900390625       |
|     5     |       4.5625       |      4.59375      |      4.578125      |       -0.05859375       |      0.2900390625      |     0.115478515625      |
|     6     |       4.5625       |     4.578125      |     4.5703125      |       -0.05859375       |     0.115478515625     |    0.02838134765625     |
|     7     |       4.5625       |     4.5703125     |     4.56640625     |       -0.05859375       |    0.02838134765625    |   -0.0151214599609375   |
|     8     |     4.56640625     |     4.5703125     |    4.568359375     |   -0.0151214599609375   |    0.02838134765625    |  0.006626129150390625   |
|     9     |     4.56640625     |    4.568359375    |    4.5673828125    |   -0.0151214599609375   |  0.006626129150390625  |  -0.004248619079589844  |
|    10     |    4.5673828125    |    4.568359375    |   4.56787109375    |  -0.004248619079589844  |  0.006626129150390625  |  0.0011885166168212 89  |
|    11     |    4.5673828125    |   4.56787109375   |   4.567626953125   |  -0.004248619079589844  |  0.001188516616821289  |  -0.001530110836029052 7|
|    12     |   4.567626953125   |   4.56787109375   |  4.5677490234375   |  -0.0015301108360290527 |  0.001188516616821289  | -0.00017081201076507568 |
|    13     |  4.5677490234375   |   4.56787109375   |  4.56781005859375  | -0.00017081201076507568 |  0.001188516616821289  |  0.0005088485777378082  |
|    14     |  4.5677490234375   | 4.56781005859375  | 4.567779541015625  | -0.00017081201076507568 | 0.0005088485777378082  | 0.00016901735216379166  |
|    15     |  4.5677490234375   | 4.567779541015625 | 4.5677642822265625 | -0.00017081201076507568 | 0.00016901735216379166 |  -8.975621312856674e-07 |
|    16     | 4.5677642822265625 | 4.567779541015625 |  4.567771911621094 |  -8.975621312856674e-07 | 0.00016901735216379166 |  8.405983680859208e-05  |
|    17     | 4.5677642822265625 | 4.567771911621094 |  4.567768096923828 |  -8.975621312856674e-07 | 8.405983680859208e-05  |  4.158112278673798e-05  |
|    18     | 4.5677642822265625 | 4.567768096923828 |  4.567766189575195 |  -8.975621312856674e-07 | 4.158112278673798e-05  |  2.034177668974735e-05  |
|    19     | 4.5677642822265625 | 4.567766189575195 |  4.567765235900879 |  -8.975621312856674e-07 | 2.034177668974735e-05  |  9.722106369736139e-06  |
+-----------+--------------------+-------------------+--------------------+-------------------------+------------------------+-------------------------+
```

3. **Determine the root of the given equation x^2-3 = 0 for x ∈ [1, 2] using Bisection method and Implement it by Python (Graphical and tabular approach).**

Given: x^2-3 = 0
Let f(x) = x^2-3
Now, find the value of f(x) at a= 1 and b=2. f(x=1) = 1^2-3 = 1 – 3 = -2 < 0 f(x=2) = 2^2-3 = 4 – 3 = 1 > 0
The given function is continuous, and the root lies in the interval [1, 2]. Let "t" be the midpoint of the interval. I.e., t = (1+2)/2 t =3 / 2 t = 1.5
Therefore, the value of the function at "t" is f(t) = f(1.5) = $(1.5)^2$-3 = 2.25 – 3 = -0.75 < 0
If f(t)<0, assume a = t. and
If f(t)>0, assume b = t.
f(t) is negative, so a is replaced with t = 1.5 for the next iterations. The iterations for the given functions are:

Iterations	**a**	**b**	c	**f(a)**	**f(b)**	**f(c)**
1	1	2	1.5	-2	1	-0.75
2	1.5	2	1.75	-0.75	1	0.062
3	1.5	1.75	1.625	-0.75	0.0625	-0.359
4	1.625	1.75	1.6875	-0.3594	0.0625	-0.1523

5	1.6875	1.75	1.7188	-01523	0.0625	-0.0457
6	1.7188	1.75	1.7344	-0.0457	0.0625	0.0081
7	1.7188	1.7344	1.7266	-0.0457	0.0081	-0.0189

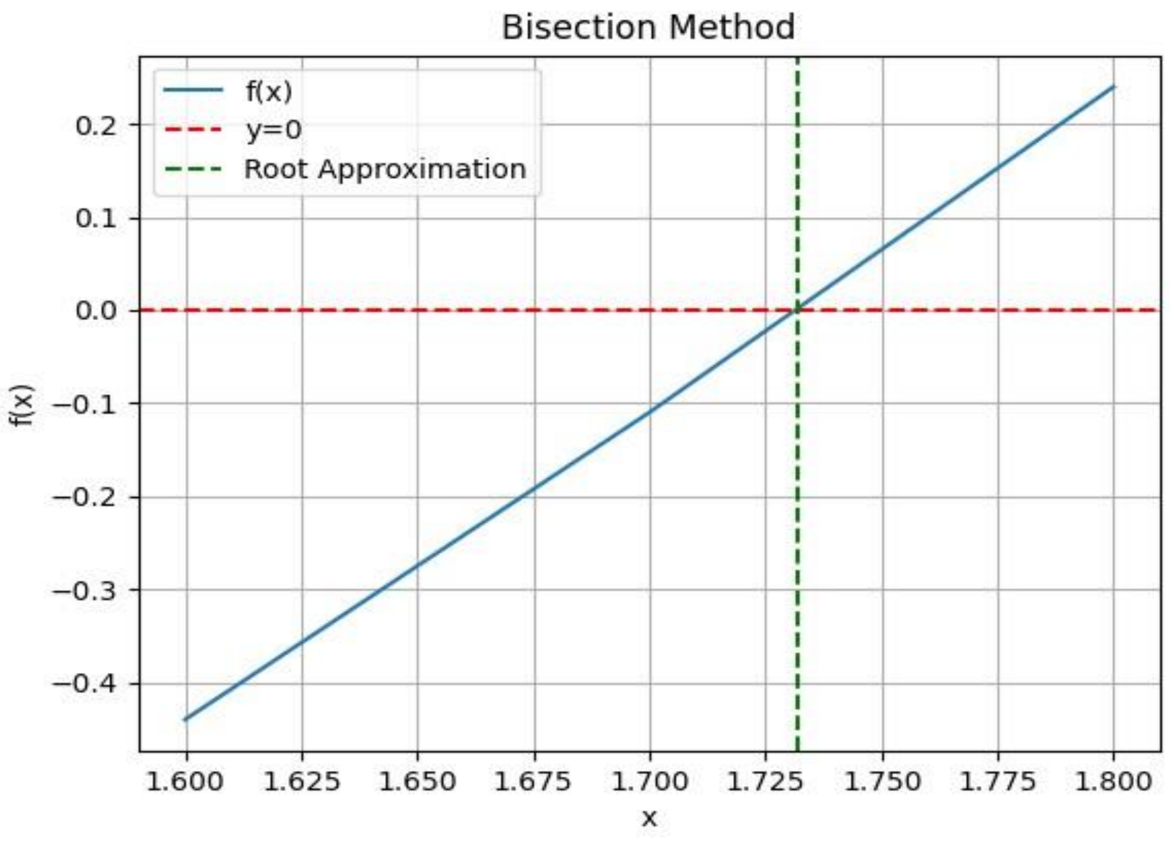
So, at the seventh iteration, we get the final interval [1.7266, 1.7344] Hence, 1.7344 is the approximated solution.

Implement with Python:

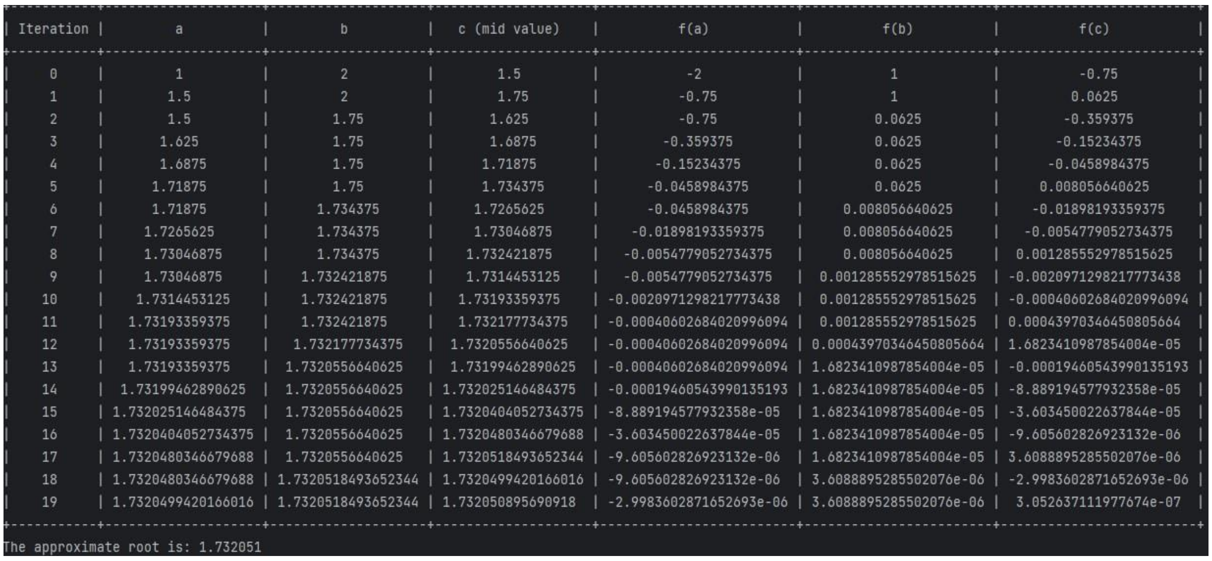
```
import matplotlib.pyplot as plt
from tabulate import tabulate
import math

def bisection(func, a, b, tol=1e-6, max_iter=100):
    iterations = []
    a_values = []
    b_values = []
    c_values = []
    f_a_values = []
    f_b_values = []
    f_c_values = []

    if func(a) * func(b) >= 0:
        raise ValueError("Function does not change sign over the
interval [a, b]")

    for i in range(max_iter):
        c = (a + b) / 2
        iterations.append(i)
        a_values.append(a)
        b_values.append(b)
        c_values.append(c)
        f_a = func(a)
        f_b = func(b)
        f_c = func(c)
        f_a_values.append(f_a)
        f_b_values.append(f_b)
        f_c_values.append(f_c)
        if f_c == 0 or abs(b - a) / 2 < tol:
            break
        elif f_a * f_c < 0:
            b = c
        else:
            a = c

    results = list(zip(iterations, a_values, b_values, c_values,
f_a_values, f_b_values, f_c_values))
    table = tabulate(results,
                     headers=["Iteration", "a", "b", "c (mid value)",
"f(a)", "f(b)", "f(c)"],
                     tablefmt="pretty")

    x = [i / 10 for i in range(int(10 * min(a, b)) - 1, int(10 *
max(a, b)) + 2)]
```

```
    y = [func(xi) for xi in x]

    plt.plot(x, y, label='f(x)')
    plt.axhline(0, color='red', linestyle='--', label='y=0')
    plt.axvline(c, color='green', linestyle='--', label='Root
Approximation')

    plt.title('Bisection Method')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.grid(True)
    plt.show()

    print(table)

    return c

def my_function(x):
    return x**2-3
a = 1
b = 2
root = bisection(my_function, a, b)
print(f"The approximate root is: {root:.6f}")
```

- **Result:**
- **Graph:**



➢ **Table:**



```
+-----------+--------------------+--------------------+--------------------+------------------------+------------------------+------------------------+
| Iteration |         a          |         b          |   c (mid value)    |          f(a)          |          f(b)          |          f(c)          |
+-----------+--------------------+--------------------+--------------------+------------------------+------------------------+------------------------+
|     0     |         1          |         2          |        1.5         |           -2           |           1            |         -0.75          |
|     1     |        1.5         |         2          |        1.75        |         -0.75          |           1            |         0.0625         |
|     2     |        1.5         |        1.75        |       1.625        |         -0.75          |         0.0625         |       -0.359375        |
|     3     |       1.625        |        1.75        |       1.6875       |       -0.359375        |         0.0625         |      -0.15234375       |
|     4     |       1.6875       |        1.75        |      1.71875       |      -0.15234375       |         0.0625         |     -0.0458984375      |
|     5     |      1.71875       |        1.75        |      1.734375      |     -0.0458984375      |         0.0625         |     0.008056640625     |
|     6     |      1.71875       |      1.734375      |     1.7265625      |     -0.0458984375      |     0.008056640625     |   -0.01898193359375    |
|     7     |     1.7265625      |      1.734375      |     1.73046875     |   -0.01898193359375    |     0.008056640625     |  -0.0054779052734375   |
|     8     |     1.73046875     |      1.734375      |    1.732421875     |  -0.0054779052734375   |     0.008056640625     |  0.001285552978515625  |
|     9     |     1.73046875     |    1.732421875     |    1.7314453125    |  -0.0054779052734375   |  0.001285552978515625  | -0.0020971298217773438 |
|    10     |    1.7314453125    |    1.732421875     |   1.73193359375    | -0.0020971298217773438 |  0.001285552978515625  | -0.0004060268402099609 |
|    11     |   1.73193359375    |    1.732421875     |   1.732177734375   | -0.0004060268402099609 |  0.001285552978515625  | 0.00043970346450805664 |
|    12     |   1.73193359375    |   1.732177734375   |  1.7320556640625   | -0.0004060268402099609 | 0.00043970346450805664 | 1.6823410987854004e-05 |
|    13     |   1.73193359375    |  1.7320556640625   |  1.7319946289062   | -0.0004060268402099609 | 1.6823410987854004e-05 | -0.00019460543990135193 |
|    14     |  1.7319946289062   |  1.7320556640625   |  1.732025146484375 | -0.00019460543990135193 | 1.6823410987854004e-05 | -8.889194577932358e-05 |
|    15     | 1.732025146484375  |  1.7320556640625   | 1.7320404052734375 | -8.889194577932358e-05 | 1.6823410987854004e-05 | -3.603450022637844e-05 |
|    16     | 1.7320404052734375 |  1.7320556640625   | 1.7320480346679688 | -3.603450022637844e-05 | 1.6823410987854004e-05 | -9.605602826923132e-06 |
|    17     | 1.7320480346679688 |  1.7320556640625   | 1.7320518493652344 | -9.605602826923132e-06 | 1.6823410987854004e-05 | 3.6088895285502076e-06 |
|    18     | 1.7320480346679688 | 1.7320518493652344 | 1.7320499420166016 | -9.605602826923132e-06 | 3.6088895285502076e-06 | -2.9983602871652693e-06 |
|    19     | 1.7320499420166016 | 1.7320518493652344 |  1.732050895690918 | -2.9983602871652693e-06 | 3.6088895285502076e-06 |  3.052637111977674e-07 |
+-----------+--------------------+--------------------+--------------------+------------------------+------------------------+------------------------+
The approximate root is: 1.732051
```

4. Determine the root of the given equation x^3-x-1 = 0 for x ∈ [1, 2] using Bisection method and Implement it by Python. (Graphical and tabular approach).

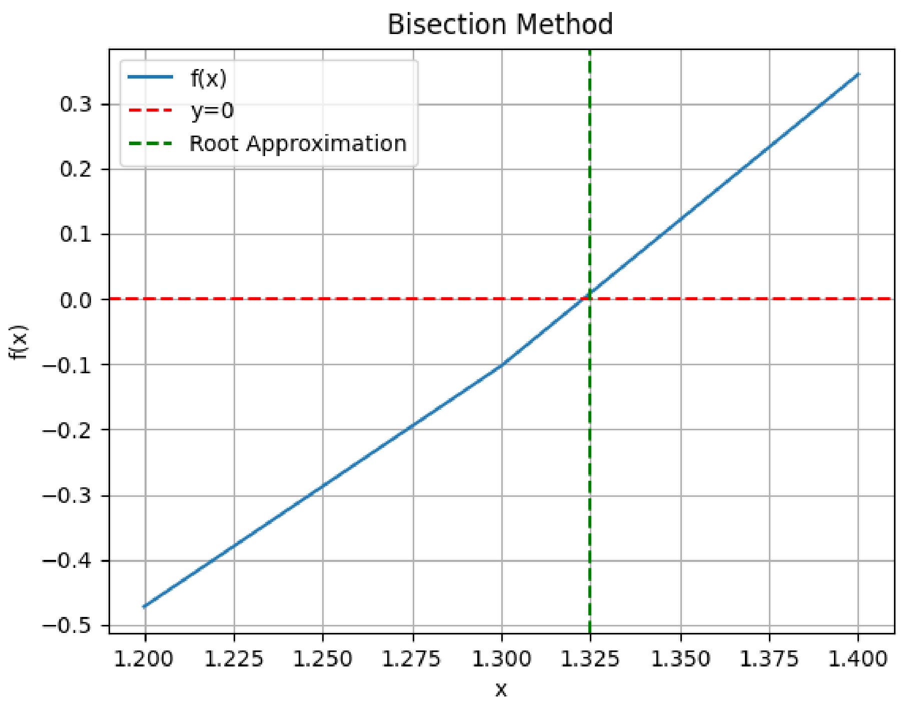
➢ **Implement with Python:**

```
import matplotlib.pyplot as plt
from tabulate import tabulate
import math

def bisection(func, a, b, tol=1e-6, max_iter=100):
    iterations = []
    a_values = []
    b_values = []
    c_values = []
    f_a_values = []
    f_b_values = []
    f_c_values = []

    if func(a) * func(b) >= 0:
        raise ValueError("Function does not change sign over the 
interval [a, b]")

    for i in range(max_iter):
        c = (a + b) / 2
        iterations.append(i)
        a_values.append(a)
        b_values.append(b)
        c_values.append(c)
        f_a = func(a)
        f_b = func(b)
        f_c = func(c)
        f_a_values.append(f_a)
        f_b_values.append(f_b)
        f_c_values.append(f_c)
        if f_c == 0 or abs(b - a) / 2 < tol:
            break
        elif f_a * f_c < 0:
            b = c
        else:
            a = c
    results = list(zip(iterations, a_values, b_values, c_values, 
f_a_values, f_b_values, f_c_values))
```

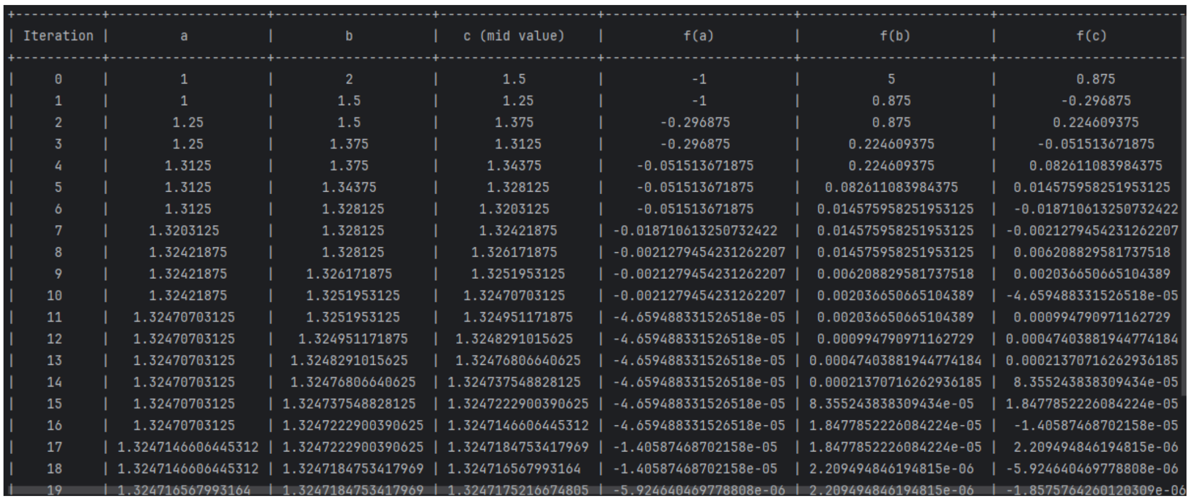
`

```
    table = tabulate(results,
                     headers=["Iteration", "a", "b", "c (mid value)",
"f(a)", "f(b)", "f(c)"],
                     tablefmt="pretty")

    x = [i / 10 for i in range(int(10 * min(a, b)) - 1, int(10 *
max(a, b)) + 2)]
    y = [func(xi) for xi in x]

    plt.plot(x, y, label='f(x)')
    plt.axhline(0, color='red', linestyle='--', label='y=0')
    plt.axvline(c, color='green', linestyle='--', label='Root
Approximation')

    plt.title('Bisection Method')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.grid(True)
    plt.show()

    print(table)

    return c

def my_function(x):
    return x ** 3 - x - 1
a = 1
b = 2
root = bisection(my_function, a, b)
print(f"The approximate root is: {root:.6f}")
```

- **result:**
  - **graph:**



- **table:**



```
+-----------+--------------------+--------------------+--------------------+-----------------------+-----------------------+-----------------------+
| Iteration |         a          |         b          |   c (mid value)    |         f(a)          |         f(b)          |         f(c)          |
+-----------+--------------------+--------------------+--------------------+-----------------------+-----------------------+-----------------------+
|     0     |         1          |         2          |        1.5         |          -1           |           5           |         0.875         |
|     1     |         1          |        1.5         |        1.25        |          -1           |         0.875         |       -0.296875       |
|     2     |        1.25        |        1.5         |       1.375        |       -0.296875       |         0.875         |      0.224609375      |
|     3     |        1.25        |       1.375        |       1.3125       |       -0.296875       |      0.224609375      |   -0.051513671875     |
|     4     |       1.3125       |       1.375        |      1.34375       |    -0.051513671875    |      0.224609375      |   0.082611083984375   |
|     5     |       1.3125       |      1.34375       |      1.328125      |    -0.051513671875    |   0.082611083984375   |  0.014575958251953125 |
|     6     |       1.3125       |      1.328125      |     1.3203125      |    -0.051513671875    |  0.014575958251953125 | -0.018710613250732422 |
|     7     |     1.3203125      |      1.328125      |     1.32421875     | -0.018710613250732422 |  0.014575958251953125 | -0.002127945423126207 |
|     8     |     1.32421875     |      1.328125      |    1.326171875     | -0.002127945423126207 |  0.014575958251953125 |  0.006208829581737518 |
|     9     |     1.32421875     |    1.326171875     |    1.3251953125    | -0.002127945423126207 |  0.006208829581737518 |  0.002036650665104389 |
|    10     |     1.32421875     |    1.3251953125    |   1.32470703125    | -0.002127945423126207 |  0.002036650665104389 | -4.659488331526518e-05 |
|    11     |   1.32470703125    |    1.3251953125    |   1.324951171875   | -4.659488331526518e-05 |  0.002036650665104389 |  0.000994790971162729 |
|    12     |   1.32470703125    |   1.324951171875   |  1.3248291015625   | -4.659488331526518e-05 |  0.000994790971162729 | 0.0004740388194474184 |
|    13     |   1.32470703125    |  1.3248291015625   |  1.32476806640625  | -4.659488331526518e-05 | 0.0004740388194474184 | 0.0002137071626293618 |
|    14     |   1.32470703125    |  1.32476806640625  | 1.324737548828125  | -4.659488331526518e-05 | 0.0002137071626293618 |  8.355243838309434e-05 |
|    15     |   1.32470703125    | 1.324737548828125  | 1.3247222900390625 | -4.659488331526518e-05 | 8.355243838309434e-05 | 1.8477852226084224e-05 |
|    16     |   1.32470703125    | 1.3247222900390625 | 1.3247146606445312 | -4.659488331526518e-05 | 1.8477852226084224e-05 |  -1.40587468702158e-05 |
|    17     | 1.3247146606445312 | 1.3247222900390625 | 1.3247184753417969 | -1.40587468702158e-05 | 1.8477852226084224e-05 |  2.209494846194815e-06 |
|    18     | 1.3247146606445312 | 1.3247184753417969 |  1.32471656799316  | -1.40587468702158e-05 | 2.209494846194815e-06 | -5.924640469778808e-06 |
|    19     | 1.324716567993164  | 1.3247184753417969 | 1.3247175216674805 | -5.924640469778808e-06 | 2.209494846194815e-06 | -1.8575764260120309e-06 |
```

5. **Find a root of an equation *f*(*x*)=2x^3-2*x*-5 using Bisection method and Implement it by Python. (Graphical and tabular approach).**

✪ **Python code:**

```
✪import matplotlib.pyplot as plt
 from tabulate import tabulate
 import math

 def bisection(func, a, b, tol=1e-6, max_iter=100):
     iterations = []
     a_values = []
     b_values = []
     c_values = []
     f_a_values = []
     f_b_values = []
     f_c_values = []

     if func(a) * func(b) >= 0:
         raise ValueError("Function does not change sign over the
 interval [a, b]")

     for i in range(max_iter):
         c = (a + b) / 2
         iterations.append(i)
         a_values.append(a)
         b_values.append(b)
         c_values.append(c)
         f_a = func(a)
         f_b = func(b)
         f_c = func(c)
         f_a_values.append(f_a)
         f_b_values.append(f_b)
         f_c_values.append(f_c)
         if f_c == 0 or abs(b - a) / 2 < tol:
             break
         elif f_a * f_c < 0:
             b = c
         else:
             a = c
     results = list(zip(iterations, a_values, b_values, c_values,
 f_a_values, f_b_values, f_c_values))
     table = tabulate(results,
                      headers=["Iteration", "a", "b", "c (mid
 value)", "f(a)", "f(b)", "f(c)"],
                      tablefmt="pretty")

     x = [i / 10 for i in range(int(10 * min(a, b)) - 1, int(10 *
 max(a, b)) + 2)]
     y = [func(xi) for xi in x]

     plt.plot(x, y, label='f(x)')
     plt.axhline(0, color='red', linestyle='--', label='y=0')
     plt.axvline(c, color='green', linestyle='--', label='Root
 Approximation')

     plt.title('Bisection Method')
     plt.xlabel('x')
     plt.ylabel('f(x)')
     plt.legend()
     plt.grid(True)
     plt.show()

     print(table)

     return c

 def my_function(x):
    return 2 * (x**3) - (2 * x) - 3
 a = 1
 b = 2
 root = bisection(my_function, a, b)
 print(f"The approximate root is: {root:.6f}")
```

- **We find result same as previous Question.**

6. **Find a root of an equation *f*(*x*)=√12 using Bisection method and Implement it by Python. (Graphical and tabular approach).**
   - **Python code:**

```
import matplotlib.pyplot as plt
from tabulate import tabulate
import math

def bisection(func, a, b, tol=1e-6, max_iter=100):
    iterations = []
    a_values = []
    b_values = []
    c_values = []
    f_a_values = []
    f_b_values = []
    f_c_values = []

    if func(a) * func(b) >= 0:
        raise ValueError("Function does not change sign over
the interval [a, b]")

    for i in range(max_iter):
        c = (a + b) / 2
        iterations.append(i)
        a_values.append(a)
        b_values.append(b)
        c_values.append(c)
        f_a = func(a)
        f_b = func(b)
        f_c = func(c)
        f_a_values.append(f_a)
        f_b_values.append(f_b)
        f_c_values.append(f_c)
        if f_c == 0 or abs(b - a) / 2 < tol:
            break
        elif f_a * f_c < 0:
            b = c
        else:
            a = c
    results = list(zip(iterations, a_values, b_values,
c_values, f_a_values, f_b_values, f_c_values))
    table = tabulate(results,
                     headers=["Iteration", "a", "b", "c (mid
value)", "f(a)", "f(b)", "f(c)"],
                     tablefmt="pretty")

    x = [i / 10 for i in range(int(10 * min(a, b)) - 1, int(10
* max(a, b)) + 2)]
    y = [func(xi) for xi in x]

    plt.plot(x, y, label='f(x)')
    plt.axhline(0, color='red', linestyle='--', label='y=0')
    plt.axvline(c, color='green', linestyle='--', label='Root
Approximation')

    plt.title('Bisection Method')
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend()
    plt.grid(True)
    plt.show()

    print(table)

    return c
def my_function(x):
    return x ** 2 - 12
a = 3
b = 4
root = bisection(my_function, a, b)
print(f"The approximate root is: {root:.6f}")
```

➢ **We find result same as previous Question (1,2).**

Cramer’s Rule.

1. **A total of 8,500 taka was invested in three interest earning accounts. The interest rates were 2%,3% and 6% if the total simple interest for one year was 380 taka and the amount invested at 6% was equal to the sum of the amounts in the other two accounts, then how much was invested in each account? (Use Cramer’s rule) and implement with python.**

   Let the amounts invested in the three accounts be Tk. x, Tk. y and Tk. z
   Interest for the three accounts are (2/100)x, (3/100)y and (6/100)z
   According to the problem,
   x + y + z = 8500 …….…………………………………………… (1)
   (2/100)x + (3/100)y + (6/100)z (or) multiplying by 100,
   2x + 3y + 6z = 38000 ……………………………………………. (2)
   z = x + y or x + y - z = 0.……………………………………… (3)

$$\text{Now,}\quad \Delta = \begin{vmatrix} 1 & 1 & 1 \\ 2 & 3 & 6 \\ 1 & 1 & -1 \end{vmatrix} \begin{aligned} &= 1(-3-6) - 1(-2-6) + 1(2-3) \\ &= -9+8-1 \\ &= -2 \neq 0 \end{aligned}$$

So there exists a unique solution to the system (1), (2) and (3)

$$\Delta_x = \begin{vmatrix} 8500 & 1 & 1 \\ 38000 & 3 & 6 \\ 0 & 1 & -1 \end{vmatrix} \begin{aligned} &= 8500(-3-6) - 1(-38000) + 1(38000) \\ &= -76500 + 76000 \\ &= -500 \end{aligned}$$

$$\Delta_y = \begin{vmatrix} 1 & 8500 & 1 \\ 2 & 38000 & 6 \\ 1 & 0 & -1 \end{vmatrix} \begin{aligned} &= 1(-38000) - 8500(-2-6) + 1(-38000) \\ &= -38000 + 68000 - 38000 \\ &= -8000 \end{aligned}$$

$$\Delta_z = \begin{vmatrix} 1 & 1 & 8500 \\ 2 & 3 & 38000 \\ 1 & 1 & 0 \end{vmatrix} \begin{aligned} &= 1(-38000) - 1(-38000) + 8500(2-3) \\ &= -38000 + 38000 - 8500 \\ &= -8500 \end{aligned}$$

So by Cramer's rule,

$$x = \frac{\Delta_x}{\Delta} = \frac{-500}{-2} = 250,\ y = \frac{\Delta_y}{\Delta} = \frac{-8000}{-2} = 4000,\ z = \frac{\Delta_z}{\Delta} = \frac{-8500}{-2} = 4250$$

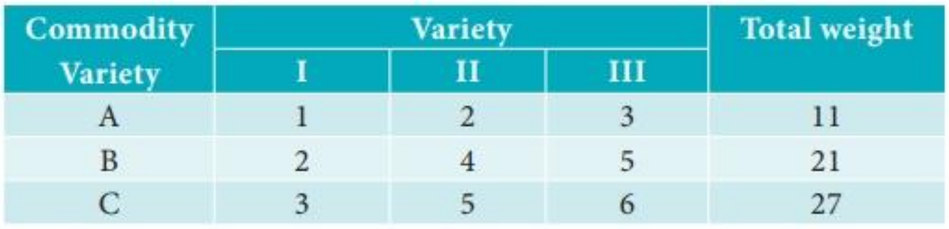
Thus the amount invested at 2% is Tk. 250, at 3% is Tk. 4000 and at 6% is Tk. 4250.

✪ **Implement using Python:**

```
import numpy as np

coefficients = np.array([[1, 1, 1],
                         [2, 3, 6],
                         [1, 1, -1]])

constants = np.array([8500, 38000, 0])

det_coefficients = np.linalg.det(coefficients)
coefficients_x = coefficients.copy()
coefficients_x[:, 0] = constants

coefficients_y = coefficients.copy()
coefficients_y[:, 1] = constants

coefficients_z = coefficients.copy()
coefficients_z[:, 2] = constants
det_x = np.linalg.det(coefficients_x)
det_y = np.linalg.det(coefficients_y)
det_z = np.linalg.det(coefficients_z)
solution_x = det_x / det_coefficients
solution_y = det_y / det_coefficients
```

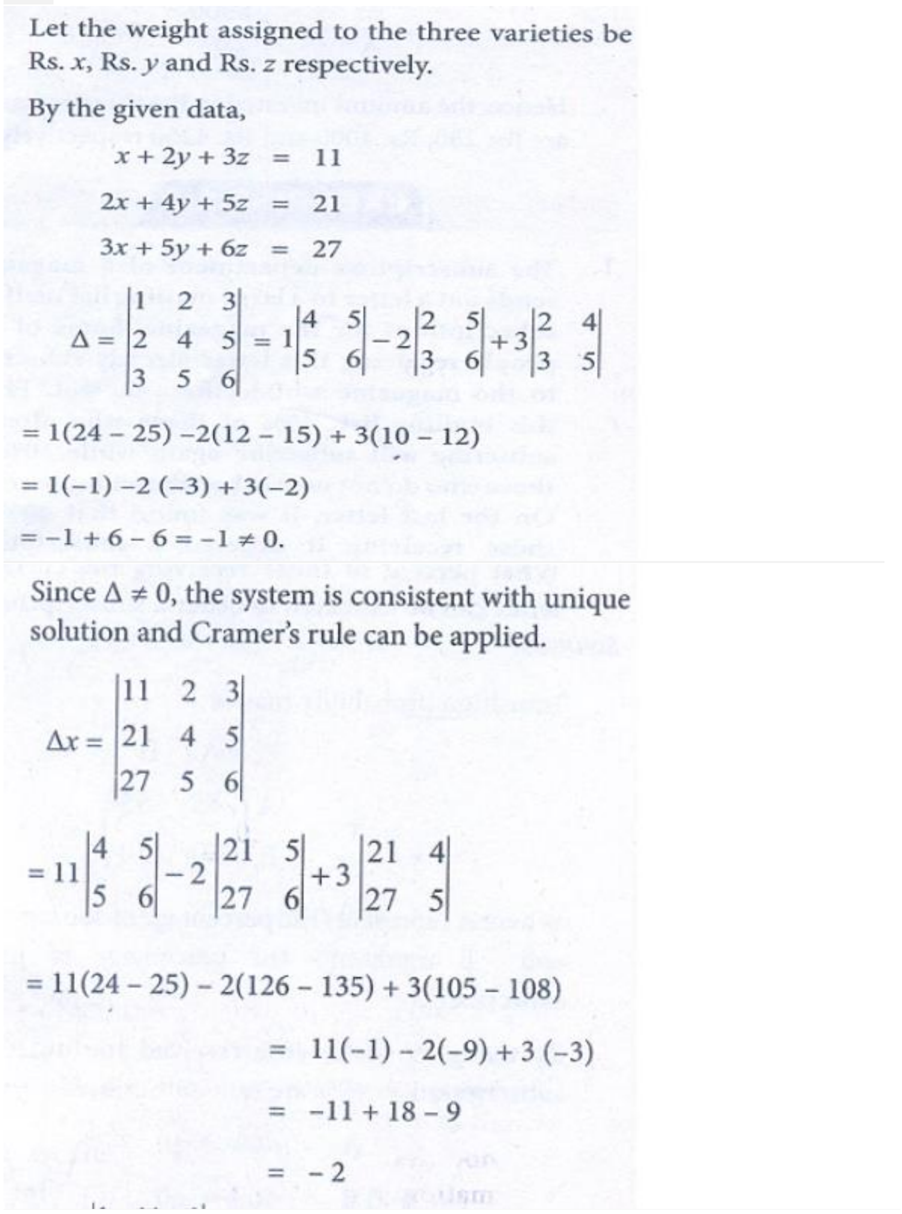
```
    solution_z = det_z / det_coefficients

    print("Solution:")
    print(f"x = {solution_x}")
    print(f"y = {solution_y}")
    print(f"z = {solution_z}")
```

2. **In a market survey three commodities A, B and C were considered. In finding out the index number some fixed weights were assigned to the three varieties in each of the commodities. The table below provides the information regarding the consumption of three commodities according to the three varieties and also the total weight received by the commodity.**



Commodity	Variety			Total weight
Variety	I	II	III	
A	1	2	3	11
B	2	4	5	21
C	3	5	6	27

Sol^n:



Let the weight assigned to the three varieties be Rs. x, Rs. y and Rs. z respectively.

By the given data,

$$x + 2y + 3z = 11$$
$$2x + 4y + 5z = 21$$
$$3x + 5y + 6z = 27$$

$$\Delta = \begin{vmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{vmatrix} = 1\begin{vmatrix} 4 & 5 \\ 5 & 6 \end{vmatrix} - 2\begin{vmatrix} 2 & 5 \\ 3 & 6 \end{vmatrix} + 3\begin{vmatrix} 2 & 4 \\ 3 & 5 \end{vmatrix}$$

$$= 1(24 - 25) - 2(12 - 15) + 3(10 - 12)$$
$$= 1(-1) - 2(-3) + 3(-2)$$
$$= -1 + 6 - 6 = -1 \neq 0.$$

Since $\Delta \neq 0$, the system is consistent with unique solution and Cramer's rule can be applied.

$$\Delta x = \begin{vmatrix} 11 & 2 & 3 \\ 21 & 4 & 5 \\ 27 & 5 & 6 \end{vmatrix}$$

$$= 11\begin{vmatrix} 4 & 5 \\ 5 & 6 \end{vmatrix} - 2\begin{vmatrix} 21 & 5 \\ 27 & 6 \end{vmatrix} + 3\begin{vmatrix} 21 & 4 \\ 27 & 5 \end{vmatrix}$$

$$= 11(24 - 25) - 2(126 - 135) + 3(105 - 108)$$
$$= 11(-1) - 2(-9) + 3(-3)$$
$$= -11 + 18 - 9$$
$$= -2$$

$$\Delta y = \begin{vmatrix} 1 & 11 & 3 \\ 2 & 21 & 5 \\ 3 & 27 & 6 \end{vmatrix}$$

$$= \begin{vmatrix} 21 & 5 \\ 27 & 6 \end{vmatrix} - 11 \begin{vmatrix} 2 & 5 \\ 3 & 6 \end{vmatrix} + 3 \begin{vmatrix} 2 & 21 \\ 3 & 27 \end{vmatrix}$$

$$= 1(126 - 135) - 11(12 - 15) + 3(54 - 63)$$

$$= -9 - 11(-3) + 3(-9)$$

$$= -9 + 33 - 27$$

$$= -3$$

$$\Delta z = \begin{vmatrix} 1 & 2 & 11 \\ 2 & 4 & 21 \\ 3 & 5 & 27 \end{vmatrix} = 1 \begin{vmatrix} 4 & 21 \\ 5 & 27 \end{vmatrix} - 2 \begin{vmatrix} 2 & 21 \\ 3 & 27 \end{vmatrix} + 11 \begin{vmatrix} 2 & 4 \\ 3 & 5 \end{vmatrix}$$

$$= 1(108 - 105) - 2(54 - 63) + 11(10 - 12)$$

$$= 1(3) - 2(-9) + 11(-2)$$

$$= 3 + 18 - 22$$

$$= -1$$

$$x = \frac{\Delta x}{\Delta} = \frac{-2}{-1} = 2$$

$$y = \frac{\Delta y}{\Delta} = \frac{-3}{-1} = 3$$

$$\text{and } z = \frac{\Delta z}{\Delta} = \frac{-1}{-1} = 1$$

Hence, the weights assigned to the three varieties are 2, 3 and 1 respectively.

✪ **Implement using Python:**

```
✪import numpy as np

  coefficients = np.array([[1, 2, 3],
                           [2, 4, 5],
                           [3, 5, 6]])

  constants = np.array([11, 21, 27])

  det_coefficients = np.linalg.det(coefficients)
```

```
    coefficients_x = coefficients.copy()
    coefficients_x[:, 0] = constants

    coefficients_y = coefficients.copy()
    coefficients_y[:, 1] = constants

    coefficients_z = coefficients.copy()
    coefficients_z[:, 2] = constants
    det_x = np.linalg.det(coefficients_x)
    det_y = np.linalg.det(coefficients_y)
    det_z = np.linalg.det(coefficients_z)
    solution_x = det_x / det_coefficients
    solution_y = det_y / det_coefficients
    solution_z = det_z / det_coefficients

    print("Solution:")
    print(f"x = {solution_x}")
    print(f"y = {solution_y}")
    print(f"z = {solution_z}")
```

3. **Using Cramer's rule solve the following after that implement it using Python.**

$.3x_1+.52x_2+x_3=0.01$

$.5x_1+.3x_2+.5x_3=.67$

$.1x_1+.3x_2+.5x_3=-.44$

Sol^n: Let us write these equations in the form AX=B

$$\begin{bmatrix} .3 & .52 & 1 \\ .5 & .3 & .5 \\ .1 & .3 & .5 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} .01 \\ .67 \\ -.44 \end{bmatrix}$$

D=|A|=$\begin{vmatrix} .3 & .52 & 1 \\ .5 & .3 & .5 \\ .1 & .3 & .5 \end{vmatrix}$=.3(.5x.3-.5x.3)-.52(.5x.5-.5x.1)+1(.5x.3-.3x.1)=0.016

Dx_1=$\begin{vmatrix} .01 & .52 & 1 \\ ..67 & .3 & .5 \\ -.44 & .3 & .5 \end{vmatrix}$=0.0444

Dx_2=$\begin{vmatrix} .3 & .01 & 1 \\ .5 & .67 & .5 \\ .1 & -.44 & .5 \end{vmatrix}$= -0.1255

$Dx_3 = \begin{vmatrix} .3 & .52 & .01 \\ .5 & .3 & .67 \\ .1 & .3 & -.44 \end{vmatrix}$=0.05054

X_1= Dx_1/D=0.0444/0.016=2.775

X_2= Dx_2/D=-0.1255/0.016=-7.84375

X_3= Dx_3/D=0.05054/0.016=3.158

✪ **Implement using python:**

```
✪import numpy as np
    coefficients = np.array([[0.3, 0.52, 1],
                             [0.5, 0.3, 0.5],
                             [0.1, 0.3, 0.5]])

    constants = np.array([0.01, 0.67, -0.44])

    det_coefficients = np.linalg.det(coefficients)

    coefficients_x = coefficients.copy()
    coefficients_x[:, 0] = constants

    coefficients_y = coefficients.copy()
    coefficients_y[:, 1] = constants

    coefficients_z = coefficients.copy()
    coefficients_z[:, 2] = constants

    det_x = np.linalg.det(coefficients_x)
    det_y = np.linalg.det(coefficients_y)
    det_z = np.linalg.det(coefficients_z)


    solution_x = det_x / det_coefficients
    solution_y = det_y / det_coefficients
    solution_z = det_z / det_coefficients

    print("Solution:")
    print(f"x = {solution_x}")
    print(f"y = {solution_y}")
    print(f"z = {solution_z}")
```

Gauss-elimination Method.

1. **Consider an online store selling T-shirts (T) and Hoodies (H). The prices are as follows:**
   - ✓ **A T-shirt costs \$10.**
   - ✓ **A Hoodie costs \$20.**

   A customer buys three items (either T-shirts or Hoodies) for a total cost of \$50. How many T-shirts (x) and Hoodies(y) did the customer buy?
   Solve Using Gauss-elimination method after that implement it by Python.

✪ Solve using Gauss-elimination method.

The objective is to find the values of x and y that satisfy the following system of linear equations:
10x + 20y=50 (Total cost)
x + y=3 (Total Number of items)
Converting given equations into matrix form

$$\left[\begin{array}{cc|c} 10 & 20 & 50 \\ 1 & 1 & 3 \end{array}\right]$$

$R_2 <- R_2 - 0.1 \times R_1$

$$\left[\begin{array}{cc|c} 10 & 20 & 50 \\ 0 & -1 & -2 \end{array}\right]$$

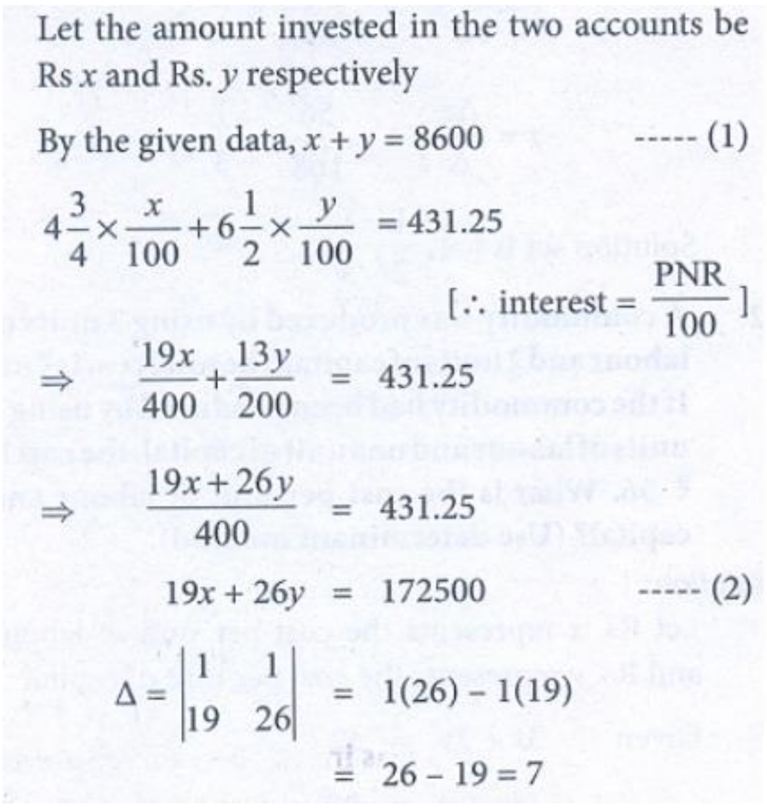
10x+20y=50 ---(i)
-y=-2---(ii)
Now, use back substitution method from (ii)
y=2
Using y=2 in (i) we find:
x=1
so, the customer bought x=1 Shirts and y=2 Hoodie

✪ **Implement using Python:**

```python
import numpy as np
coefficients = np.array([[10, 20], [1, 1]])

constants = np.array([50, 3])

augmented_matrix = np.column_stack((coefficients, constants))

n = len(constants)

for i in range(n):
    augmented_matrix[i, :] = augmented_matrix[i, :] /
augmented_matrix[i, i]

    for j in range(n):
        if i != j:
            augmented_matrix[j, :] -= augmented_matrix[j, i] *
augmented_matrix[i, :]

solutions = augmented_matrix[:, -1]

print("Number of T-shirts (x):", solutions[0])
print("Number of Hoodies (y):", solutions[1])
```

2. **A total of ₹8,600 was invested in two accounts. One account earned 4 (3/4) % annual1 interest and the other earned 6 (1/2) % annual interest. If the total interest for one year was ₹431.25, how much was invested in each account? (Use Gauss-elimination Method and Implement by Python).**

Sol^{n}:



Let the amount invested in the two accounts be Rs x and Rs. y respectively

By the given data, $x + y = 8600$ ----- (1)

$$4\frac{3}{4} \times \frac{x}{100} + 6\frac{1}{2} \times \frac{y}{100} = 431.25$$

$$\left[\therefore \text{ interest} = \frac{\text{PNR}}{100}\right]$$

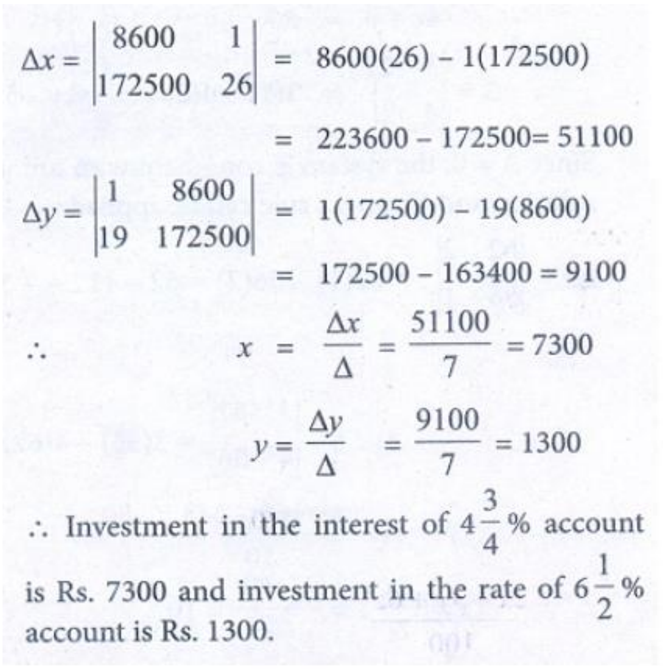
$$\Rightarrow \quad \frac{19x}{400} + \frac{13y}{200} = 431.25$$

$$\Rightarrow \quad \frac{19x+26y}{400} = 431.25$$

$$19x + 26y = 172500 \qquad \text{----- (2)}$$

$$\Delta = \begin{vmatrix} 1 & 1 \\ 19 & 26 \end{vmatrix} = 1(26) - 1(19)$$

$$= 26 - 19 = 7$$



$$\Delta x = \begin{vmatrix} 8600 & 1 \\ 172500 & 26 \end{vmatrix} = 8600(26) - 1(172500)$$

$$= 223600 - 172500 = 51100$$

$$\Delta y = \begin{vmatrix} 1 & 8600 \\ 19 & 172500 \end{vmatrix} = 1(172500) - 19(8600)$$

$$= 172500 - 163400 = 9100$$

$$\therefore \quad x = \frac{\Delta x}{\Delta} = \frac{51100}{7} = 7300$$

$$y = \frac{\Delta y}{\Delta} = \frac{9100}{7} = 1300$$

$\therefore$ Investment in the interest of $4\frac{3}{4}\%$ account is Rs. 7300 and investment in the rate of $6\frac{1}{2}\%$ account is Rs. 1300.

✪ **Implement using Python:**

```
✪ import numpy as np
  coefficients = np.array([[1, 1], [19, 26]])

  constants = np.array([8600, 172500])

  augmented_matrix = np.column_stack((coefficients, constants))

  n = len(constants)

  for i in range(n):
      augmented_matrix[i, :] = augmented_matrix[i, :] /
  augmented_matrix[i, i]

      for j in range(n):
          if i != j:
              augmented_matrix[j, :] -= augmented_matrix[j, i] *
  augmented_matrix[i, :]

  solutions = augmented_matrix[:, -1]

  print("Number of T-shirts (x):", solutions[0])
  print("Number of Hoodies (y):", solutions[1])
```

3. Solve the following system by the Gauss-Elimination method and Implement it using Python.

$3x_1+.1x_2-.2x_3 = 7.85$
$.1x_1+7x_2-.3x_3 = -19.3$
$.3x_1-2x_2+10x_3 = 71.4$

Sol^n: We'll create the augmented matrix and perform row operations:

$$\left[\begin{array}{ccc|c} 3 & 0.1 & -0.2 & 7.85 \\ 0.1 & 7 & -0.3 & -19.3 \\ 0.3 & -2 & 10 & 71.4 \end{array}\right]$$

Step 1: Perform row operations to create zeros below the leading coefficient in the first column.

✓ R2 = R2 - (0.1/3) * R1

- ✓ R3 = R3 - (0.3/3) * R1

The augmented matrix becomes:

$$\left[\begin{array}{ccc|c} 3 & 0.1 & -0.2 & 7.85 \\ 0 & 6.99967 & -0.2933 & -19.56167 \\ 0 & -2.01 & 10.02 & 70.615 \end{array}\right]$$

Step 2: Create zeros below the leading coefficient in the second column.

- ✓ R3 = R3 + (2.01/6.97) * R2

The augmented matrix becomes:

$$\left[\begin{array}{ccc|c} 3 & 0.1 & -0.2 & 7.85 \\ 0 & 6.99967 & -0.2933 & -19.56167 \\ 0 & 0 & 9.935 & 64.99774 \end{array}\right]$$

Step 3: Solve for x_3 using the last row:

$9.935x_3 = 64.99774$

$x_3 \approx 64.99774/ 9.935$

$x_3 \approx 6.54$

Step 4: Substitute the value of x_3 into the second row to solve for x_2:

$6.99967x_2 - 0.2933x_3 = -19.56167$

$6.99967x_2 - -0.2933(6.54) = -19.56167$

$6.99967x_2 - 1.98 \approx -19.56167$

$6.99967x_2 \approx -19.56167 + 1.98$

$6.99967x_2 \approx -17.5817$

$x_2 \approx -17.5817/ 6.99967$

$x_2 \approx -2.51$

Step 5: Substitute the values of x_2 and x_3 into the first row to solve for x_1:

$3x_1 + 0.1x_2 - 0.2x_3 = 7.85$

$3x_1 + 0.1(-2.51) - 0.2(6.54) \approx 7.85$

$3x_1 - 0.251 – 1.308 \approx 7.85$

$3x_1 \approx 7.85 + 1.559$

$3x_1 \approx 9.409$

$x_1 \approx 9.409 / 3$

$x_1 \approx 3.136$

Please check the calculation twice

- ✪ **Implement using python:**

```
import numpy as np
coefficients = np.array([[3, 0.1, -0.2],
                         [0.1, 7, -0.3],
                         [0.3, -2, 10]])
constants = np.array([7.85, -19.3, 71.4])
augmented_matrix = np.column_stack((coefficients, constants))

# Perform Gaussian elimination
n = len(constants)

for i in range(n):
    augmented_matrix[i, :] = augmented_matrix[i, :] /
augmented_matrix[i, i]

    for j in range(n):
        if i != j:
            augmented_matrix[j, :] -= augmented_matrix[j, i] *
augmented_matrix[i, :]

solutions = augmented_matrix[:, -1]
print("Solution:")
for i, sol in enumerate(solutions):
    print(f"x{i + 1} =", sol)
```

Gauss-Jordan Method.

1. **An investor has $10,000 to invest in two types of financial instruments: stocks and bonds. The expected return from stocks is 8%, and from bonds is 5%. The investor wants the total return to be $700. Find the amount invested in each type Using *Gauss-Jordan* method after that implement it using *Python.***

✪ Solve using Gauss-Jordan method.

from the Given problem we find two equations:-

0.08x+0.05y=700---(i)

x+y=10000---(ii)

here, x=stocks and y=bonds.

Converting given equations into matrix form

$$\left[\begin{array}{cc|c} 0.08 & 0.05 & 700 \\ 1 & 1 & 10000 \end{array}\right]$$

R_1<-R_1/0.08

$$\left[\begin{array}{cc|c} 1 & 00.625 & 8750 \\ 1 & 1 & 10000 \end{array}\right]$$

R_2<-R_2-R_1

$$\left[\begin{array}{cc|c} 1 & 00.625 & 8750 \\ 0 & 0.375 & 1250 \end{array}\right]$$

R_2<-R_2/.375

$$\left[\begin{array}{cc|c} 1 & 0.625 & 8750 \\ 0 & 1 & 3333.3333 \end{array}\right]$$

R_1<-R_1-.625xR_2

$$\left[\begin{array}{cc|c} 1 & 0 & 6666.667 \\ 0 & 1 & 3333.3333 \end{array}\right]$$

So, x=6666.667 and y=3333.3333.

Hence amount of stocks invested 6666 and amount of bonds invested 3333.

✪ **Implement using Python:**

```
import numpy as np

# Coefficients matrix
coefficients = np.array([[0.08, 0.05], [1, 1]])

# Constants vector
constants = np.array([700, 10000])

# Augmented matrix
augmented_matrix = np.column_stack((coefficients,
constants))

# Applying Gauss-Jordan elimination
```

```python
rows, cols = augmented_matrix.shape

for i in range(rows):
    # Normalize the pivot row
    augmented_matrix[i] = augmented_matrix[i] /
augmented_matrix[i, i]

    # Eliminate other rows
    for j in range(rows):
        if i != j:
            augmented_matrix[j] = augmented_matrix[j] -
augmented_matrix[j, i] * augmented_matrix[i]

# Extract the solution
solution = augmented_matrix[:, -1]

# Print the solution
print("Amount invested in stocks:", solution[0])
print("Amount invested in bonds:", solution[1])
```

2. **Solve the following system by the Gauss-Jordan method and Implement it using Python.**

$3x_1-0.1x_2-0.2x_3=7.85$

$0.1x_1+7x_2-0.3x_3=-19.3$

$0.3x_1-0.2x_2+10x_3=71.4$

- **Step-01:** First we have to express the coefficients and the right-hand side as an augmented matrix:

$$\left[\begin{array}{ccc|c} 3 & -0.1 & -0.2 & 7.85 \\ 0.1 & 7 & -0.3 & -19.3 \\ 0.3 & -0.2 & 10 & 71.4 \end{array}\right]$$

- **Step-02:** Divide Row 1 by 3 to make the leading coefficient 1 in the first row:

$$\left[\begin{array}{ccc|c} 1 & -0.0333333 & -0.066667 & 2.61667 \\ 0.1 & 7 & -0.3 & -19.3 \\ 0.3 & -0.2 & 10 & 71.4 \end{array}\right] r_1'=r_1/3$$

- **Step-03:** Subtract 0.1 times Row 1 from Row 2 and 0.3 times Row 1 from Row 3 to make the entries below the leading 1 in Row 1 equal to 0:

$$\left[\begin{array}{ccc|c} 1 & -0.0333333 & -0.066667 & 2.61667 \\ 0 & 7.00333 & -0.29333 & -19.878 \\ 0 & -0.19000 & 10.0200 & 70.6150 \end{array}\right] r_2'=r_2-r_1x0.1 \quad \text{and} \quad r_3'=r_3-r_1x0.3$$

- **Step-04:** Divide Row 2 by 7.0033 to make the leading coefficient 1 in the second row:

$$\left[\begin{array}{ccc|c} 1 & -0.0333333 & -0.066667 & 2.61667 \\ 0 & 1 & -0.0418848 & -2.79320 \\ 0 & -0.19000 & 10.0200 & 70.6150 \end{array}\right] r_2'=r_2/7.00333$$

- **Step-05:** Reduction of x_2 terms from first and third equation we can use,
$$\left[\begin{array}{ccc|c} 1 & 0 & -0.0680629 & 2.52356 \\ 0 & 1 & -0.0418848 & -2.79320 \\ 0 & 0 & 10.01200 & 70.0843 \end{array}\right]$$
$r_1' = r_1 + r_2 x0.0333$ and $r_3' = r_3 + r_2 x0.1900$

- **Step-06:** Divide Row 3 by 10.01200 to make the leading coefficient 1 in the third row:
$$\left[\begin{array}{ccc|c} 1 & 0 & -0.0680629 & 2.52356 \\ 0 & 1 & -0.0418848 & -2.79320 \\ 0 & 0 & 1 & 7 \end{array}\right]$$ $r_3' = r_3/10.01200$

- **Step-07:** Finally, reducing x_3 terms from equation 1 and 2 we need,
$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3.0 \\ 0 & 1 & 0 & -2.50 \\ 0 & 0 & 1 & 7.0 \end{array}\right]$$ $r_1' = r_1 + r_3 x0.0680629$ and $r_2' = r_2 + r_3 x0.0418848$

Now, we find the value of x_1, x_2 and x_3,

$x_1 = 3.0$

$x_2 = -2.50$ and

$x_3 = 7.0$

✪ **Implement with python:**

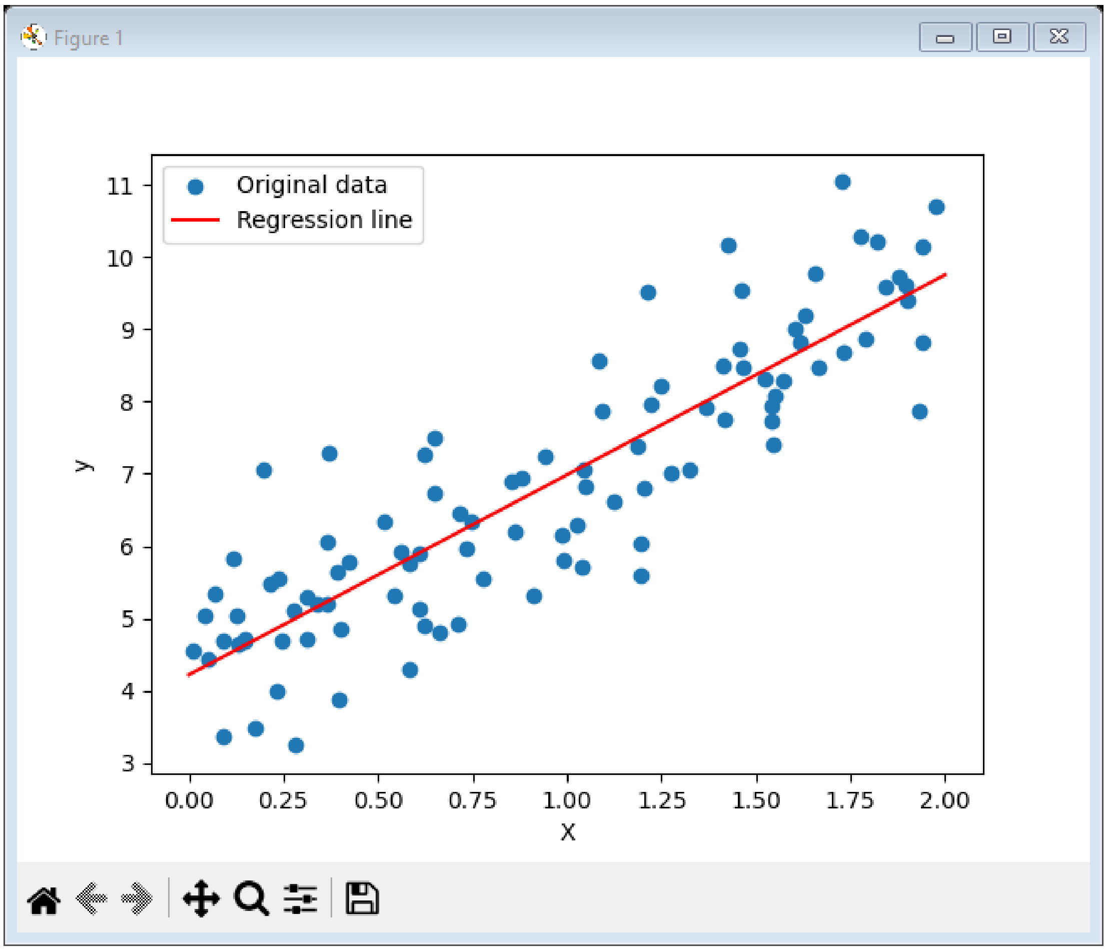
```
import numpy as np

# Coefficients matrix
coefficients = np.array([[3, -0.1, -0.2], [0.1, 7, -0.3],
[0.3, -0.2, 10]])
constants = np.array([7.85, -19.3, 71.4])
augmented_matrix = np.column_stack((coefficients,
constants))
rows, cols = augmented_matrix.shape

for i in range(rows):
    # Normalize the pivot row
    augmented_matrix[i] = augmented_matrix[i] /
augmented_matrix[i, i]

    # Eliminate other rows
    for j in range(rows):
        if i != j:
            augmented_matrix[j] = augmented_matrix[j] -
augmented_matrix[j, i] * augmented_matrix[i]

# Extract the solution
solution = augmented_matrix[:, -1]

print("Solution:")
for i, value in enumerate(solution):
    print(f"x{i+1} = {value}")
```

Linear Regression

1. Demonstration of Linear least regression.

```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

model = LinearRegression()

model.fit(X, y)

X_new = np.array([[0], [2]])
y_pred = model.predict(X_new)

plt.scatter(X, y, label='Original data')
plt.plot(X_new, y_pred, 'r-', label='Regression line')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()

print('Intercept (beta_0):', model.intercept_[0])
print('Slope (beta_1):', model.coef_[0][0])
```

✪ **Output: Represent Graphically.**



Polynomial Regression.

1. Polynomial regression demonstrations.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

np.random.seed(42)
size = np.random.uniform(1000, 5000, 100)
price = 100000 + 150 * size - 0.05 * (size ** 2) + np.random.normal(0,
5000, 100)

size = size.reshape(-1, 1)
price = price.reshape(-1, 1)

size_train, size_test, price_train, price_test = train_test_split(size,
price, test_size=0.2, random_state=42)

degree = 2
poly = PolynomialFeatures(degree=degree)
size_poly = poly.fit_transform(size_train)

model = LinearRegression()
model.fit(size_poly, price_train)

price_train_pred = model.predict(size_poly)

plt.scatter(size_train, price_train, color='blue')
plt.plot(size_train, price_train_pred, color='red')
plt.title('Polynomial Regression - House Prices')
plt.xlabel('Size')
plt.ylabel('Price')
plt.show()

size_test_poly = poly.transform(size_test)
price_test_pred = model.predict(size_test_poly)

mse = mean_squared_error(price_test, price_test_pred)
print(f'Mean Squared Error on Test Data: {mse}')

new_size = np.array([[3000]])
new_size_poly = poly.transform(new_size)
predicted_price = model.predict(new_size_poly)
print(f'Predicted Price for a new house: {predicted_price[0][0]}')
```

False-position method.

1. **Use the *False-position method* to determine the drag coefficients 'c' needed for a Parachutist of mass m=68.1 kg to have a velocity of 40 m/s after free-falling for timer t=10 s after that Implement it by *Python*. In Python, Result must be shown both *Graphical and Tabular format*. Note: The acceleration due to gravity is 9.81 m/s^2.**

The equation is f(c)=$\frac{gm}{c}\left(1-e^{-\left(\frac{c}{m}\right)t}\right)-v$

Solution. As in Example 5.3, initiate the computation with guesses of $x_l = 12$ and $x_u = 16$.

First iteration:

$$x_l = 12 \qquad f(x_l) = 6.0699$$
$$x_u = 16 \qquad f(x_u) = -2.2688$$
$$x_r = 16 - \frac{-2.2688(12-16)}{6.0669-(-2.2688)} = 14.9113$$

which has a true relative error of 0.89 percent.

Second iteration:

$$f(x_l)\,f(x_r) = -1.5426$$

Therefore, the root lies in the first subinterval, and x_r becomes the upper limit for the next iteration, $x_u = 14.9113$:

$$x_l = 12 \qquad f(x_l) = 6.0699$$
$$x_u = 14.9113 \qquad f(x_u) = -0.2543$$
$$x_r = 14.9113 - \frac{-0.2543(12-14.9113)}{6.0669-(-0.2543)} = 14.7942$$

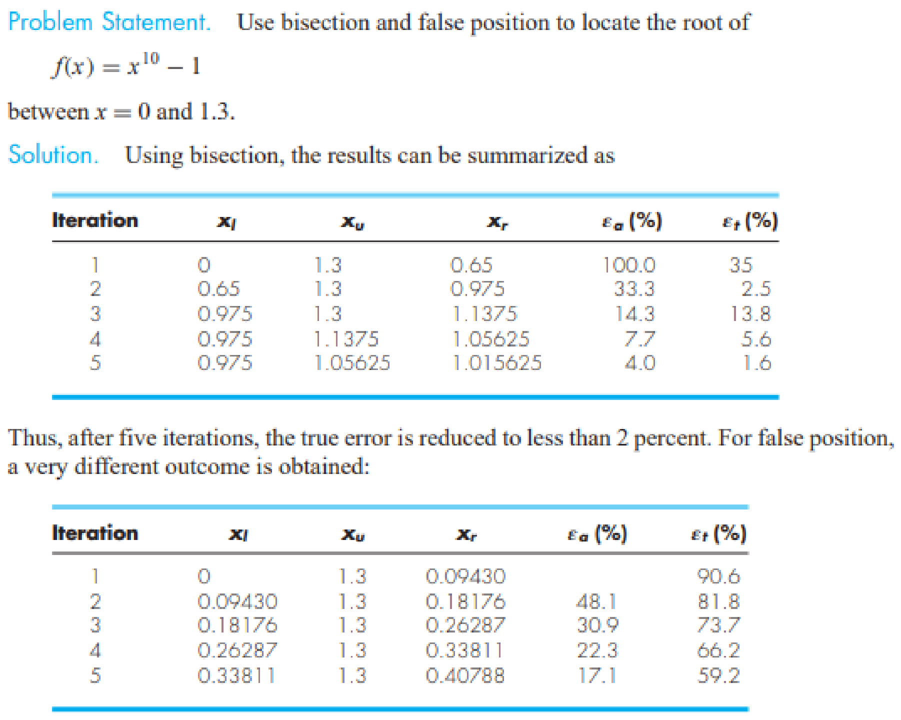
which has true and approximate relative errors of 0.09 and 0.79 percent. Additional iterations can be performed to refine the estimate of the roots.

✪ **Implement using python:**

```
import math


def function_to_find_root(c):
    return (667.38 / c) * (1 - math.exp(-0.146843 * c)) -
40


def false_position_method(func, a, b, tol=1e-6,
max_iter=100):

    if func(a) * func(b) > 0:
        raise ValueError("The function must have different
signs at the interval endpoints.")

    iterations = 0
    while iterations < max_iter:
        c = (a * func(b) - b * func(a)) / (func(b) -
func(a))
```

```
        if abs(func(c)) < tol:
            return c, iterations

        if func(c) * func(a) < 0:
            b = c
        else:
            a = c

        iterations += 1

    raise ValueError("False-position method did not
converge within the maximum number of iterations.")

x1 = 12
x2 = 16
tolerance = 1e-6

root, iterations =
false_position_method(function_to_find_root, x1, x2,
tol=tolerance)
print(f"Approximated root: {root:.6f}")
print(f"Iterations: {iterations}")
```

2.



Problem Statement. Use bisection and false position to locate the root of

$$f(x) = x^{10} - 1$$

between $x = 0$ and 1.3.

Solution. Using bisection, the results can be summarized as

Iteration	x_l	x_u	x_r	ε_a (%)	ε_t (%)
1	0	1.3	0.65	100.0	35
2	0.65	1.3	0.975	33.3	2.5
3	0.975	1.3	1.1375	14.3	13.8
4	0.975	1.1375	1.05625	7.7	5.6
5	0.975	1.05625	1.015625	4.0	1.6

Thus, after five iterations, the true error is reduced to less than 2 percent. For false position, a very different outcome is obtained:

Iteration	x_l	x_u	x_r	ε_a (%)	ε_t (%)
1	0	1.3	0.09430		90.6
2	0.09430	1.3	0.18176	48.1	81.8
3	0.18176	1.3	0.26287	30.9	73.7
4	0.26287	1.3	0.33811	22.3	66.2
5	0.33811	1.3	0.40788	17.1	59.2

.

✪ Implement by Python:

```
def false_position_method(func, a, b, tol=1e-6,
max_iter=100):
    if func(a) * func(b) > 0:
        raise ValueError("The function must have
different signs at the interval endpoints.")

    iterations = 0
    while iterations < max_iter:
        c = (a * func(b) - b * func(a)) / (func(b) -
func(a))

        if abs(func(c)) < tol:
            return c, iterations

        if func(c) * func(a) < 0:
            b = c
        else:
            a = c

        iterations += 1

    raise ValueError("False-position method did not
converge within the maximum number of iterations.")


# Example usage:
def example_function(x):
    return x ** 10 - 1


a = 0
b = 1.3
tolerance = 1e-6

root, iterations =
false_position_method(example_function, a, b,
tol=tolerance)
print(f"Approximated root: {root:.6f}")
print(f"Iterations: {iterations}")
```

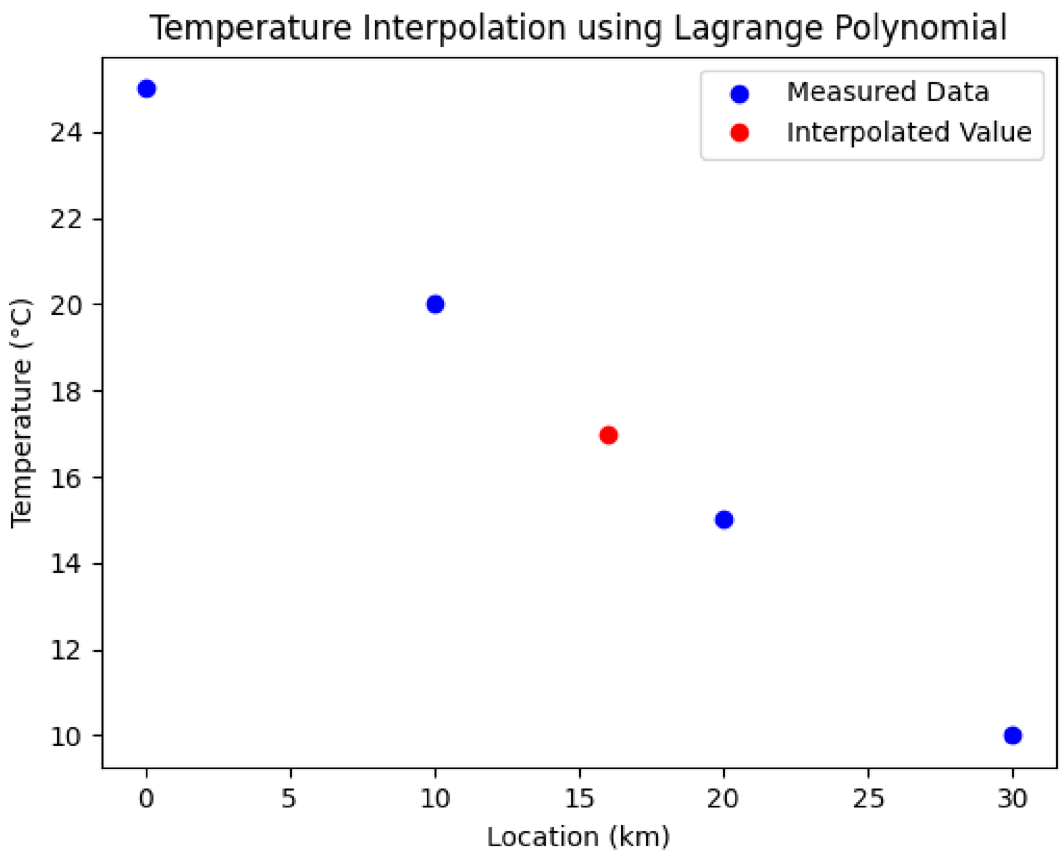
Lagrange Interpolation Formula

1. **Consider a problem involving temperature data at various locations and times. Suppose you have temperature measurements at different locations (represented by x-values) and times (represented by y-values). You want to interpolate the temperature at a specific location and time that falls between the measured data points.**
   Consider the locations: [0,10,20,30] and temperatures: [25,20,15,10] and the target location is 16.
   You need to solve it using Lagrange Interpolation formula and Implement by Python. Result must be shown in both console panel and Graphically.
   - **Python code:**

```
import matplotlib.pyplot as plt
def lagrange_interpolation(x_values, y_values, x):
    n = len(x_values)
    result = 0.0
    for i in range(n):
        term = y_values[i]
        for j in range(n):
            if i != j:
                term *= (x - x_values[j]) / (x_values[i] - x_values[j])
        result += term
    return result


locations = [0, 10, 20, 30]
temperatures = [25, 20, 15, 10]
target_location = 16

interpolated_temperature = lagrange_interpolation(locations,
temperatures, target_location)
print(f"Interpolated temperature at {target_location} km:
{interpolated_temperature}°C")

plt.scatter(locations, temperatures, label='Measured Data',
color='blue')
plt.plot(target_location, interpolated_temperature, 'ro',
label='Interpolated Value')
plt.xlabel('Location (km)')
plt.ylabel('Temperature (°C)')
plt.title('Temperature Interpolation using Lagrange Polynomial')
plt.legend()
plt.show()
```

output: Interpolated temperature at 16 km: 16.999999999999996°C

➢ **Graphically present:**



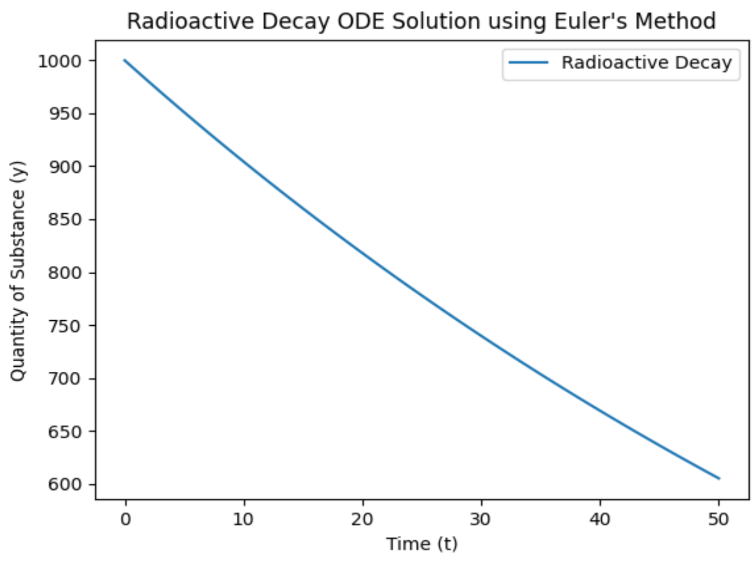
Euler’s Method/Euler-Cauchy method/Point slope method

1. **Consider a problem involving radioactive decay. The decay of a radioactive substance can be modeled by a first-order differential equation of the form dt/dy=−k·y. Where y is the quantity of the substance, t is time, and k is the decay constant**.

- **Python code:**

```
import matplotlib.pyplot as plt
def radioactive_decay(t, y):
    k = 0.01
    return -k * y
def euler_method(f, t0, y0, tn, h):
    t_values = [t0]
    y_values = [y0]

    num_steps = int((tn - t0) / h)

    for i in range(1, num_steps + 1):
        t_next = t_values[i - 1] + h
        y_next = y_values[i - 1] + h * f(t_values[i - 1], y_values[i
- 1])

        t_values.append(t_next)
        y_values.append(y_next)

    return t_values, y_values

t0 = 0
y0 = 1000  # Initial quantity of radioactive substance
tn = 50  # End time
h = 1    # Step size

t_values, y_values = euler_method(radioactive_decay, t0, y0, tn, h)

plt.plot(t_values, y_values, label='Radioactive Decay')
plt.xlabel('Time (t)')
plt.ylabel('Quantity of Substance (y)')
plt.title('Radioactive Decay ODE Solution using Euler\'s Method')
plt.legend()
plt.show()
```

- **Result:** Graphical Presentation.



Simpson’s 1/3 Rules.

1. Consider a problem where we want to calculate the probability of an event occurring within a certain range in a normal distribution. For instance, let's say we have a dataset representing the heights of individuals in a population, and we want to find the probability of selecting an individual with a height between -1 standard deviation and +1 standard deviation from the mean.

   In this case, the Gaussian distribution e^{-x^2} can represent the probability density function (PDF) of the heights in the population. The integral of this function over the range [−1,1] using Simpson's 1/3 rule can be interpreted as the probability of selecting an individual with a height within one standard deviation from the mean.[n=6].

   ✪ **Python code;**

```
import math

def simpsons_one_third_rule(func, a, b, n):
    h = (b - a) / n
    result = func(a) + func(b)

    for i in range(1, n, 2):
        result += 4 * func(a + i * h)

    for i in range(2, n-1, 2):
        result += 2 * func(a + i * h)

    result *= h / 3
    return result

def exponential_function(x):
    return math.exp(-x**2)

lower_limit = -1
upper_limit = 1
num_intervals = 6

approx_integral = simpsons_one_third_rule(exponential_function,
lower_limit, upper_limit, num_intervals)
print(f"Approximate integral: {approx_integral}")
```

   ✪ **output:** Approximate integral: 1.4938399174488888

Simpson’s 3/8 Rules.

1. **Imagine you are calculating the work done by a variable force F(x) to move an object along a straight path from x=1 to x=2. The force is given by $F(x)=x^3$, and you need to find the work done over this displacement. Implement the calculation using Python with Simpson’s 3/8 rule.[number of intervals,n=3]**

✪ Python code

```python
def simpsons_three_eighth_rule(func, a, b, n):
    h = (b - a) / n
    result = func(a) + func(b)

    for i in range(1, n, 3):
        result += 3 * (func(a + i * h) + func(a + (i + 1) * h))

    result *= 3 * h / 8
    return result
def cubic_function(x):
    return x**3

lower_limit = 1
upper_limit = 2
num_intervals = 3

approx_integral = simpsons_three_eighth_rule(cubic_function,
lower_limit, upper_limit, num_intervals)
print(f"Approximate integral: {approx_integral}")
```

✪ Output: Approximate integral: 3.7499999999999999

Milne’s Predictor-corrector Method.

1. **Consider a scenario where y represents the population of a substance, and x represents time. The negative sign indicates a decrease in the quantity over time. This could model the decay of a radioactive substance, the cooling of a hot object, or the decrease in the number of individuals in a population due to natural factors. It is expressed by equation as:**

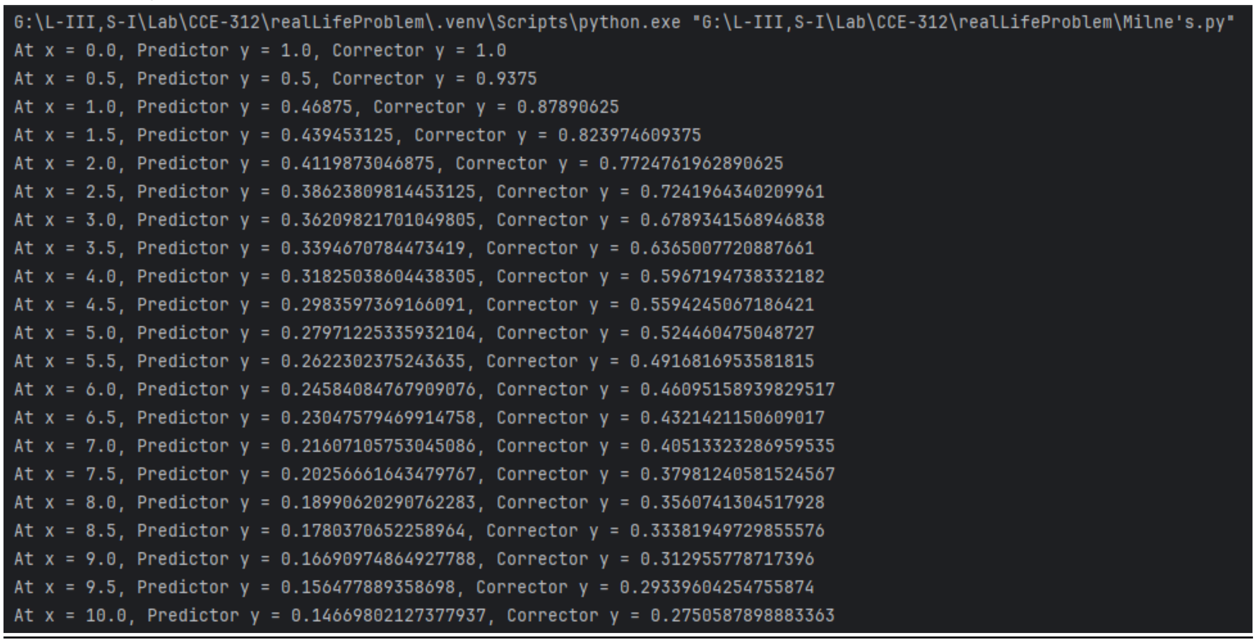
$$\frac{dy}{dx} = -y$$

Let's use this interpretation and plot the solution to the differential equation over time with implementing Milne’s Predictor-corrector method, assuming y(0)=1 and x ranging from 0 to 10 with a step size of 0.5. In Python result must be shown both console panel and Graphically.

Python code:

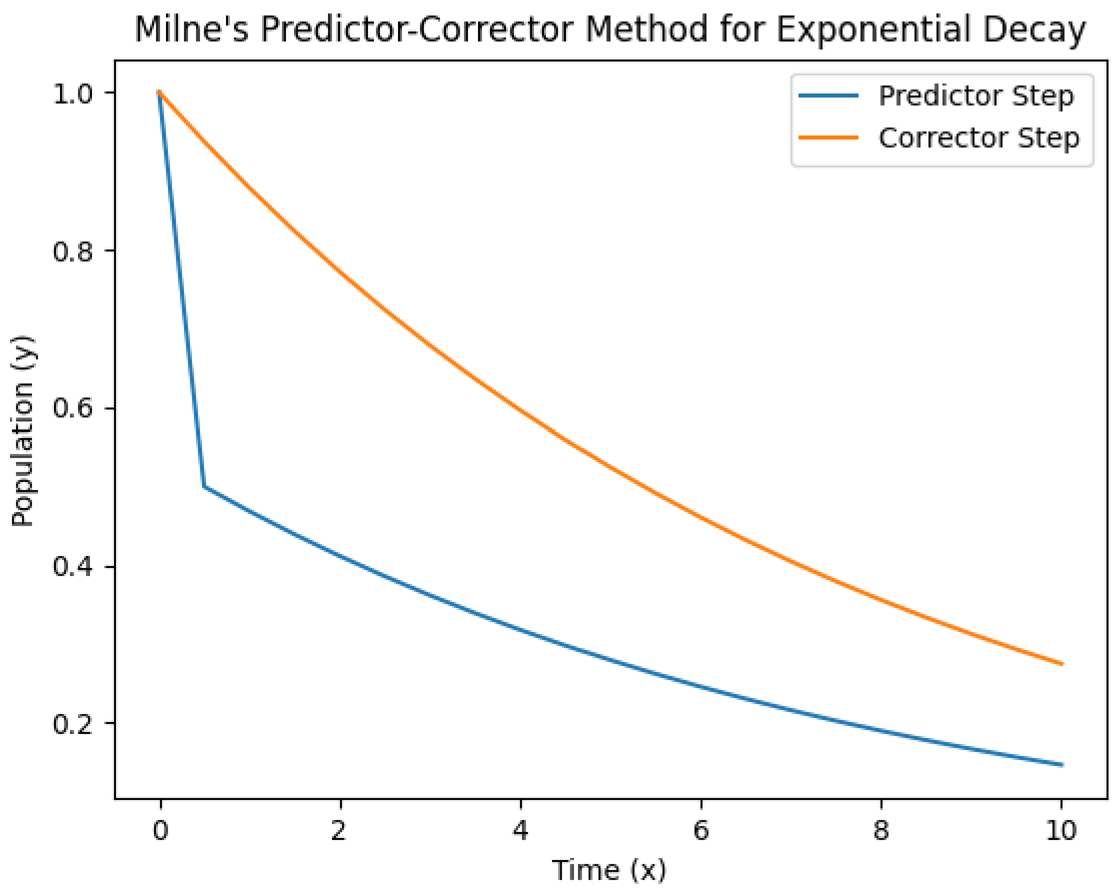
```
import numpy as np
import matplotlib.pyplot as plt
def milnes_method(f, y0, x0, x_end, h):
    x = np.arange(x0, x_end + h, h)
    y_predictor = np.zeros(len(x))
    y_corrector = np.zeros(len(x))
    y_predictor[0] = y0
    y_corrector[0] = y0

    for i in range(1, len(x)):
        y_pred = y_corrector[i - 1] + h * f(x[i - 1], y_corrector[i - 1])

        y_corrected = y_corrector[i - 1] + (h / 4) * (3 * f(x[i], y_pred) -
f(x[i - 1], y_corrector[i - 1]))

        y_predictor[i] = y_pred
        y_corrector[i] = y_corrected

    return x, y_predictor, y_corrector
def f(x, y):
    return -y


y0 = 1
x0 = 0
x_end = 10
h = 0.5

x, y_predictor, y_corrector = milnes_method(f, y0, x0, x_end, h)

for i in range(len(x)):
    print(f"At x = {x[i]}, Predictor y = {y_predictor[i]}, Corrector y =
{y_corrector[i]}")

plt.plot(x, y_predictor, label='Predictor Step')
plt.plot(x, y_corrector, label='Corrector Step')
plt.xlabel('Time (x)')
plt.ylabel('Population (y)')
plt.title('Milne\'s Predictor-Corrector Method for Exponential Decay')
plt.legend()
plt.show()
```

Console Output:



```
G:\L-III,S-I\Lab\CCE-312\realLifeProblem\.venv\Scripts\python.exe "G:\L-III,S-I\Lab\CCE-312\realLifeProblem\Milne's.py"
At x = 0.0, Predictor y = 1.0, Corrector y = 1.0
At x = 0.5, Predictor y = 0.5, Corrector y = 0.9375
At x = 1.0, Predictor y = 0.46875, Corrector y = 0.87890625
At x = 1.5, Predictor y = 0.439453125, Corrector y = 0.823974609375
At x = 2.0, Predictor y = 0.4119873046875, Corrector y = 0.7724761962890625
At x = 2.5, Predictor y = 0.38623809814453125, Corrector y = 0.7241964340209961
At x = 3.0, Predictor y = 0.36209821701049805, Corrector y = 0.6789341568946838
At x = 3.5, Predictor y = 0.3394670784473419, Corrector y = 0.6365007720887661
At x = 4.0, Predictor y = 0.31825038604438305, Corrector y = 0.5967194738332182
At x = 4.5, Predictor y = 0.2983597369166091, Corrector y = 0.5594245067186421
At x = 5.0, Predictor y = 0.27971225335932104, Corrector y = 0.524460475048727
At x = 5.5, Predictor y = 0.2622302375243635, Corrector y = 0.4916816953581815
At x = 6.0, Predictor y = 0.24584084767909076, Corrector y = 0.46095158939829517
At x = 6.5, Predictor y = 0.23047579469914758, Corrector y = 0.4321421150609017
At x = 7.0, Predictor y = 0.21607105753045086, Corrector y = 0.40513323286959535
At x = 7.5, Predictor y = 0.20256661643479767, Corrector y = 0.37981240581524567
At x = 8.0, Predictor y = 0.18990620290762283, Corrector y = 0.3560741304517928
At x = 8.5, Predictor y = 0.1780370652258964, Corrector y = 0.3338194972985576
At x = 9.0, Predictor y = 0.16690974864927788, Corrector y = 0.312955778717396
At x = 9.5, Predictor y = 0.156477889358698, Corrector y = 0.2933960425475587 4
At x = 10.0, Predictor y = 0.14669802127377937, Corrector y = 0.2750587898883363
```

Graphical Representation:

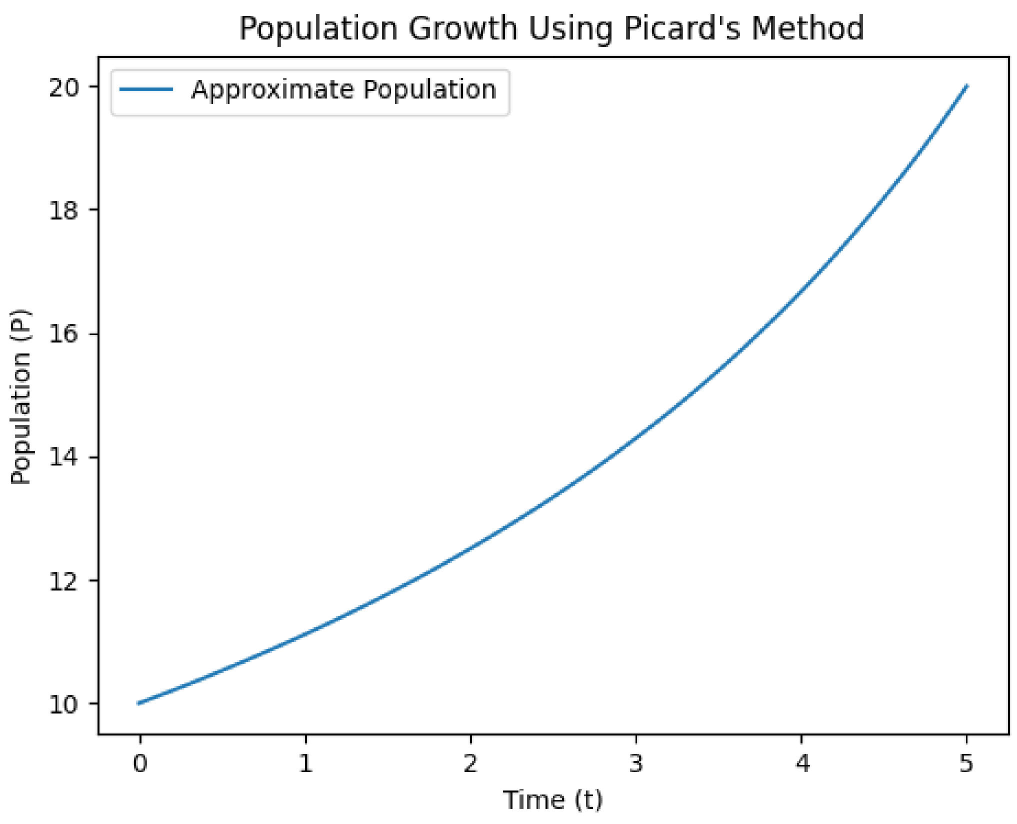


Picard’s Method.

1. **Imagine you are studying the population growth of a species in a controlled environment. The population (P) at any given time is influenced by the rate of growth (r) and the initial population (P0). The population growth can be modeled by the differential equation:**
   P(t)=P_0 + $\int_0^t r.P(s)\,ds$
   Implement it Using Picard's Method to find an approximate solution for P(t) with an initial guess P_0(t) = 10 and growth rate r=0.1. Growth rate must be shown in a graph.
   Python code:

```python
import numpy as np
from scipy import integrate
import matplotlib.pyplot as plt
def integral_equation(P, t, r):
    return 10 + integrate.quad(lambda s: r * P, 0, t)[0]
def picards_method(P0, t, r, iterations):
    for i in range(iterations):
        P1 = np.vectorize(integral_equation)(P0, t, r)

        if np.all(np.abs(P1 - P0) < 1e-8):
            break

        P0 = P1

    return P1

t_values = np.linspace(0, 5, 100)
initial_guess = 10  # Initial population
growth_rate = 0.1
iterations = 10

population_solution = picards_method(initial_guess, t_values,
growth_rate, iterations)

plt.plot(t_values, population_solution, label='Approximate
Population')
plt.xlabel('Time (t)')
plt.ylabel('Population (P)')
plt.title("Population Growth Using Picard's Method")
plt.legend()
plt.show()
```

Output:



Secant Method.

1. **Suppose you are developing a computer graphics application, and you need to find the root of a polynomial function that represents the position of a moving object at a specific time. The polynomial function is given by:**
   $F(x)=2x^3-4x^2-6x+3$
   You need to find a solution to the equation F(x)=0 to determine the time when the object reaches a certain position. Use the secant method to find the root of the equation by implementing it with Python. [N.B: Consider the Initial guess 1, 3 and iterations is 10]
   Python code:

```
def f(x):
    return 2 * x ** 3 - 4 * x ** 2 - 6 * x + 3
def secant_method(x0, x1, iterations):
    for i in range(iterations):
        fx0 = f(x0)
        fx1 = f(x1)

        x2 = x1 - (x1 - x0) / (fx1 - fx0) * fx1

        if abs(x2 - x1) < 1e-8:
            break

        x0, x1 = x1, x2

    return x2
```

```
x0 = 1
x1 = 3
iterations = 10

root = secant_method(x0, x1, iterations)

print("Root of the polynomial:", root)
```

Output: Root of the polynomial: 2.864496659191886

2. **Imagine you are designing a physics simulation, and you need to find the time(t) it takes for a projectile to hit the ground. The vertical position of the projectile is modeled by the quadratic equation:**
   $h(t)=-5t^2+10t+15$
   You need to find the time at which the projectile hits the ground ((h(t)=0) using the secant method by implementing Python. [N.B: Consider the Initial guess 1,2 and maximum iterations is 10]
   Python code:

```
def h(t):
    return -5 * t ** 2 + 10 * t + 15
def secant_method(t0, t1, iterations):
    for i in range(iterations):
        ht0 = h(t0)
        ht1 = h(t1)

        t2 = t1 - (t1 - t0) / (ht1 - ht0) * ht1
        if abs(t2 - t1) < 1e-8:
            break
        t0, t1 = t1, t2
    return t2

t0 = 1
t1 = 2
iterations = 10
time_of_impact = secant_method(t0, t1, iterations)
print("Time of impact:", time_of_impact, "seconds")
```

Output: Time of impact: 2.9999999999999996 seconds

Newton-Raphson (NR) Method

1. **Suppose you are involved in a manufacturing process, and you want to optimize the temperature setting for a chemical reaction to maximize the yield of a certain product. The rate of the chemical reaction (R) is temperature-dependent and is given by the Arrhenius equation:**

   R=A. $e^{-\frac{E}{RT}}$

   where: The pre-exponential factor is A = 2.5x10^8, the activation energy E = 50000, the ideal gas constant R = 8.314 j/mol/k, the initial temperature in Kelvin T_0 = 300. [iteration is 10] You need to find the optimal temperature (T) that maximizes the rate of the chemical reaction. Use the Newton-Raphson method to find the root of the equation by Implementing with Python.

   Python code:

```python
import math

def f(T, A, E, R, target_R):
    return A * math.exp(-E / (R * T)) - target_R
def df_dT(T, A, E, R):
    return (A * E / (R * T ** 2)) * math.exp(-E / (R * T))

def newton_raphson_method(T0, A, E, R, target_R, iterations):
    for i in range(iterations):
        f_T0 = f(T0, A, E, R, target_R)
        df_dT0 = df_dT(T0, A, E, R)
        T1 = T0 - f_T0 / df_dT0

        if abs(T1 - T0) < 1e-8:
            break
        T0 = T1
    return T1
A = 2.5e8
E = 50000
R = 8.314
target_R = 1
T0 = 300
iterations = 10

optimal_temperature = newton_raphson_method(T0, A, E, R, target_R,
iterations)

print("Optimal temperature:", optimal_temperature, "Kelvin")
```

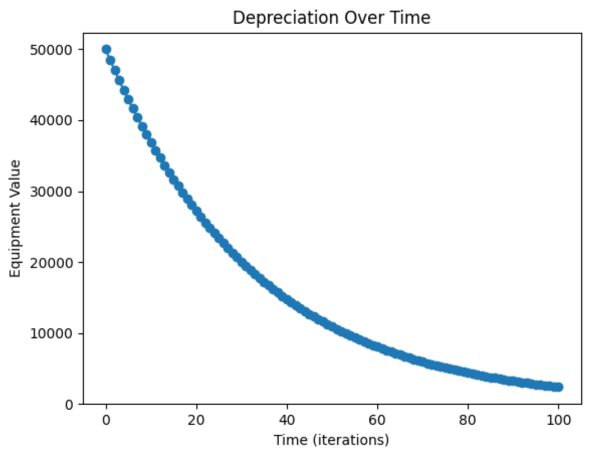
   output: Optimal temperature: 311.00797645667416 Kelvin

Simple Fixed-point Iterations.

1. **Suppose You are a financial analyst tasked with estimating the remaining value of a piece of equipment as it depreciates over time. The asset has an initial value (V_0) of $50,000, and its value decreases exponentially over time. The depreciation model is given by the formula: V(t+1) = V(t)·(1–d)**
   V(t) is the value of the equipment at time t, the depreciation rate is 3%. In Python result must be shown both Console panel and Graphically.
   [Maximum number of iterations: 100]
   Python code:

```python
import matplotlib.pyplot as plt
def f(V, d):
    return V * (1 - d)
def simple_fixed_point_iteration(V0, d, iterations):
    values = [V0]
    for i in range(iterations):
        V1 = f(V0, d)
        values.append(V1)

        if abs(V1 - V0) < 1e-8:
            break
        V0 = V1

    return values

V0 = 50000
d = 0.03
iterations = 100

estimated_values = simple_fixed_point_iteration(V0, d, iterations)

plt.plot(range(len(estimated_values)), estimated_values, marker='o')
plt.title('Depreciation Over Time')
plt.xlabel('Time (iterations)')
plt.ylabel('Equipment Value')
plt.show()

print("Estimated long-term value of the equipment:",
estimated_values[-1])
```

output: Estimated long-term value of the equipment: 2377.6253962702804
Graphical Representation:



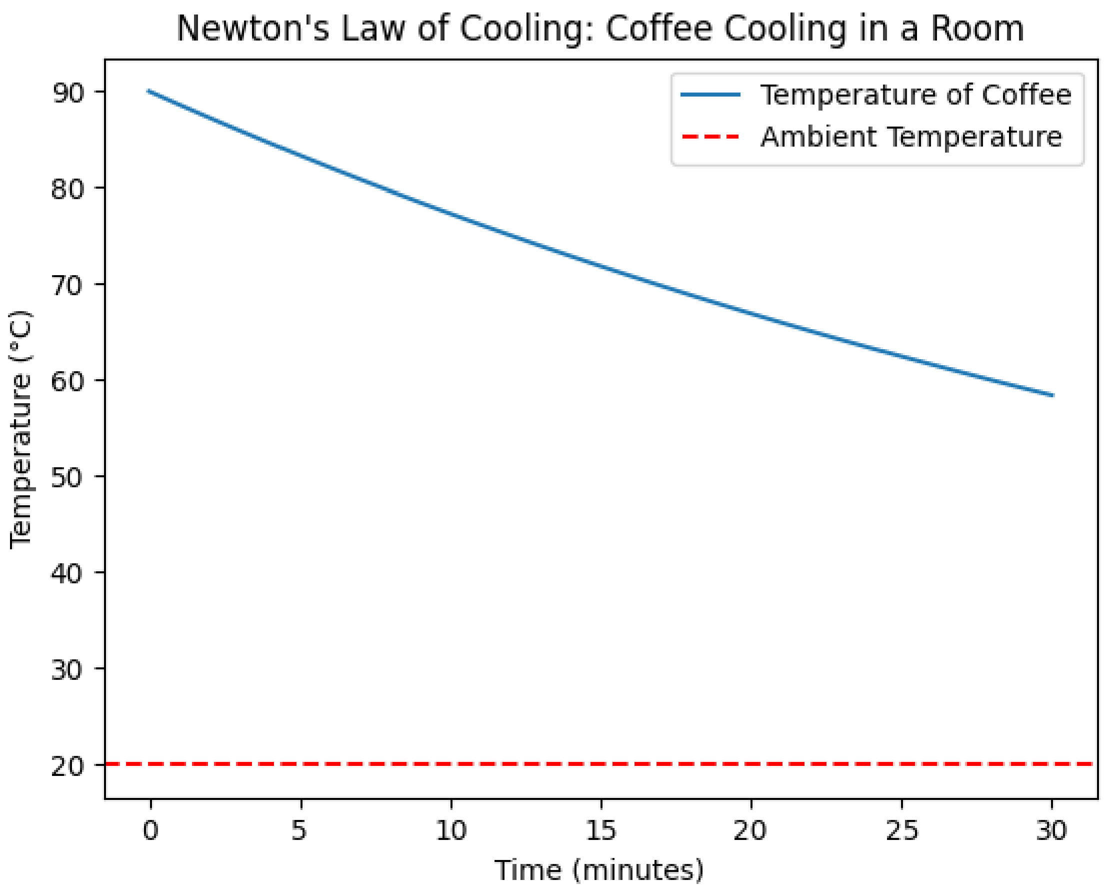
Heun’s Method.

1. **Newton's Law of Cooling characterizes the temporal evolution of an object's temperature as a function of its surrounding environment. The differential equation associated with this phenomenon is expressed as $\frac{dT}{dt} = -k(T - T_{ambient})$. Where T represents the temperature of the object, t is the time variable, k denotes the cooling constant, and $T_{ambient}$ is the ambient temperature. This equation signifies that the rate of temperature change $\frac{dT}{dt}$ if proportional to the temperature difference between the object and its surroundings. As an illustrative example, consider a hot cup of coffee (T_0=90°C) in a room with an ambient temperature of $T_{ambient}$=20^0 C. Employing Heun's method with a step size of h=1 min we can approximate the temperature evolution over a time span of 30 min (t_0= 0 min, t_{end}= 30 min) with a cooling constant k=0.02, This simulation provides insights into how the coffee cools down toward the room temperature, offering a practical application of Newton's Law of Cooling with specific initial conditions and parameters**.

- **Python code:**

```python
import matplotlib.pyplot as plt
def newtons_law_of_cooling(t, T, k, T_ambient):
    return -k * (T - T_ambient)
def heuns_method(f, t0, T0, tn, h, k, T_ambient):
    t_values = [t0]
    T_values = [T0]
    num_steps = int((tn - t0) / h)

    for i in range(1, num_steps + 1):
        t_n = t_values[i - 1]
        T_n = T_values[i - 1]
        T_pred = T_n + h * f(t_n, T_n, k, T_ambient)

        T_corrected = T_n + 0.5 * h * (f(t_n, T_n, k, T_ambient) +
f(t_n + h, T_pred, k, T_ambient))

        t_next = t_n + h
        t_values.append(t_next)
        T_values.append(T_corrected)
    return t_values, T_values
t0 = 0
T0 = 90
tn = 30
h = 1
k = 0.02
T_ambient = 20

t_values, T_values = heuns_method(newtons_law_of_cooling, t0, T0,
tn, h, k, T_ambient)

plt.plot(t_values, T_values, label="Temperature of Coffee")
plt.axhline(y=T_ambient, color='r', linestyle='--', label="Ambient
Temperature")
plt.xlabel('Time (minutes)')
plt.ylabel('Temperature (°C)')
plt.title("Newton's Law of Cooling: Coffee Cooling in a Room")
plt.legend()
plt.show()
```

➢ **Graphical approach:**


Newton's Law of Cooling: Coffee Cooling in a Room
Temperature of Coffee
Ambient Temperature
Temperature (°C)
Time (minutes)