TP1 Roots of Equations

Name: Ny Chantharith

ID: e20220472

Group: I3-AMS-TP-B

1. Explain and develop a computer program to find roots of the equation,

$$bx + c = 0$$

In [12]:
```
def solve_linear_eq1(b, c):
    if b == 0:
        if c == 0:
            return "Infinitive"
        else:
            return "No solution"
    else:
        return -c / b

print("Enter the coefficients of the linear equation ax + b = 0")
b = float(input("Enter b: "))
c = float(input("Enter c: "))
print("Enter b:", b, "\nEnter c:", c)
print("Result is", solve_linear_eq1(b, c))
```

```
Enter the coefficients of the linear equation ax + b = 0
Enter b: 2.0
Enter c: 3.0
Result is -1.5
```

2. Explain and develop a computer program to find roots of the equation,

$$ax^2 + bx + c = 0$$

In [13]:
```
def solve_quadratic_eq(a, b, c):
    if a == 0:
        return solve_linear_eq1(b, c)
    else:
        d = b**2 - 4*a*c
        if d < 0:
            return "No solution"
        elif d == 0:
            return -b / (2*a), -b / (2*a)
        else:
            return (-b + d**0.5) / (2*a), (-b - d**0.5) / (2*a)

print("Enter the coefficients of the quadratic equation ax^2 + bx + c = 0
a = float(input("Enter a: "))
b = float(input("Enter b: "))
c = float(input("Enter c: "))
```

```
print("Enter a:", a, "\nEnter b:", b, "\nEnter c:", c)
print("Result is", solve_quadratic_eq(a, b, c))
```

```
Enter the coefficients of the quadratic equation ax^2 + bx + c = 0
Enter a: 1.0
Enter b: 4.0
Enter c: 3.0
Result is (-1.0, -3.0)
```

3. Develop computer programs to compute the following sums with loops and then with mathematical formula for $n = 10, 20, 50$.
   a. $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$
   b. $\frac{1}{1.2} + \frac{1}{2.3} + \ldots + \frac{1}{n(n+1)} = 1 - \frac{1}{n+1}$

In [14]:
```
def compute_sum_a(n):
    return n * (n + 1) / 2

def compute_sum_b(n):
    return 1-(1/n+1)

print("compute the following sums with loops and then with mathematical f
print("\na. 1+2+3+...+n=n(n+1)/2")
print(f"for n=10, the result is {compute_sum_a(10):.2f}")
print(f"for n=20, the result is {compute_sum_a(20):.2f}")
print(f"for n=50, the result is {compute_sum_a(50):.2f}")
print("\nb. 1-1/2+1/3-1/4+...+1/n=1-(1/n+1)")
print(f"for n=10, the result is {compute_sum_b(10):.2f}")
print(f"for n=20, the result is {compute_sum_b(20):.2f}")
print(f"for n=50, the result is {compute_sum_b(50):.2f}")
```

```
compute the following sums with loops and then with mathematical formula f
or n=10,20,50.

a. 1+2+3+...+n=n(n+1)/2
for n=10, the result is 55.00
for n=20, the result is 210.00
for n=50, the result is 1275.00

b. 1-1/2+1/3-1/4+...+1/n=1-(1/n+1)
for n=10, the result is -0.10
for n=20, the result is -0.05
for n=50, the result is -0.02
```

4. Develop computer programs to compute the sum $\sum_{k=1}^{n} \frac{1}{k!}$.

In [15]:
```
def compute_sum(n):
    sum = 0
    factorial = 1
    for i in range(1, n):
        factorial *= i
        sum += 1 / factorial
    return sum

print(f"Compute the sum above")
n = int(input("Enter n: "))
```

```
print("Enter n:", n)
print(f"Result is {compute_sum(n):.4f}")
```

```
Compute the sum above
Enter n: 6
Result is 1.7167
```

5. Develop a computer program to determine the exponential function which can be written in infinite series form

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

then compute the value of e^x for $x = 0.1, 0.5, 1$ with polynomial approximation of degree 5.

In [16]:
```
import math

def exp_taylor(x, degree=5):
    result = 0.0
    for n in range(degree + 1):
        result += (x ** n) / math.factorial(n)
    return result

print("Compute the exponential value of x")
x = [0.1, 0.5, 1]
for i in x:
    print(f"for x={i}, the result is {exp_taylor(i):.4f}")
```

```
Compute the exponential value of x
for x=0.1, the result is 1.1052
for x=0.5, the result is 1.6487
for x=1, the result is 2.7167
```

6. Develop a computer program to compute the value of polnomial $\sum_{k=0}^{n} a_k x^k$ at $x = x_0$. Use your program to compute $p(0.1)$ if $p(x) = 1 - 3x^2 + 2x^3 - x^5 + 4x^6$.

In [17]:
```
def evaluate_polynomial(coefficients, x0):
    result = 0
    for i in range(len(coefficients)):
        result += coefficients[i] * (x0 ** i)
    return result

print("Evaluate the polynomial")
print("p(x)=1 - 3x^2 + 2x^3 - x^5 + 4x^6")
coefficients = [1, 0, -3, 2, 0, -1, 4]
print(f"Since x0 = 0.1, the result is {evaluate_polynomial(coefficients,
```

```
Evaluate the polynomial
p(x)=1 - 3x^2 + 2x^3 - x^5 + 4x^6
Since x0 = 0.1, the result is 0.9720
```

7. Let $f(x) = (x-1)^{10}, p = 1$ and $p_n = 1 + \frac{1}{n}$. Show that $|f(p_n) < 10^{-3}$ whenever $n > 1$ but that $|p - p_n| < 10^{-3}$ requires that $n > 1000$.

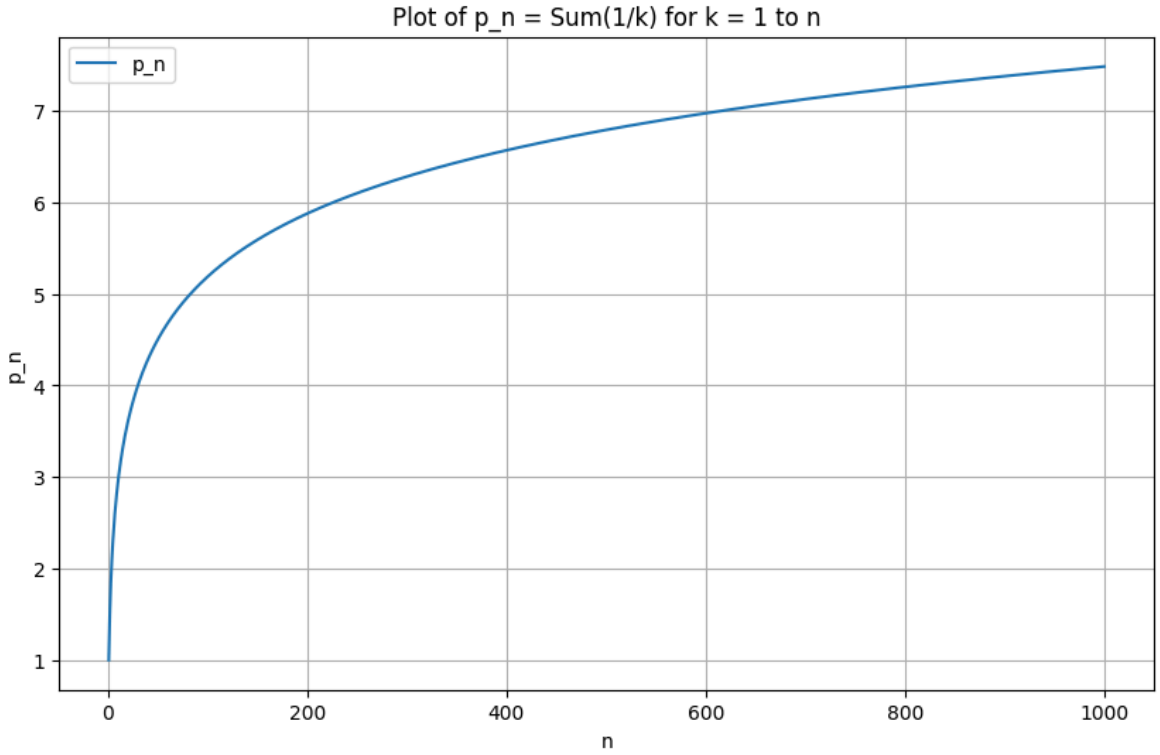
In [18]:
```
def f(p_n):
    """Compute f(p_n) = (p_n - 1)^10."""
    return (p_n - 1) ** 10

# Condition 1: Check |f(p_n)| < 10^(-3) for n > 1
print("Check |f(p_n)| < 10^(-3) for n > 1")
for n in range(2, 11):
    p_n = 1 + 1/n
    f_value = f(p_n)
    print(f"n = {n}, p_n = {p_n:.6f}, |f(p_n)| = {f_value:.6e}, Condition

print("\n")

# Condition 2: Find the smallest n such that |p - p_n| < 10^(-3)
print("Check |p - p_n| < 10^(-3) requires that n>1000")
required_n = 1001
p_n_required = 1 + 1/required_n
abs_diff = abs(1 - p_n_required)

print(f"n = {required_n}, p_n = {p_n_required:.6f}, |p - p_n| = {abs_diff
```

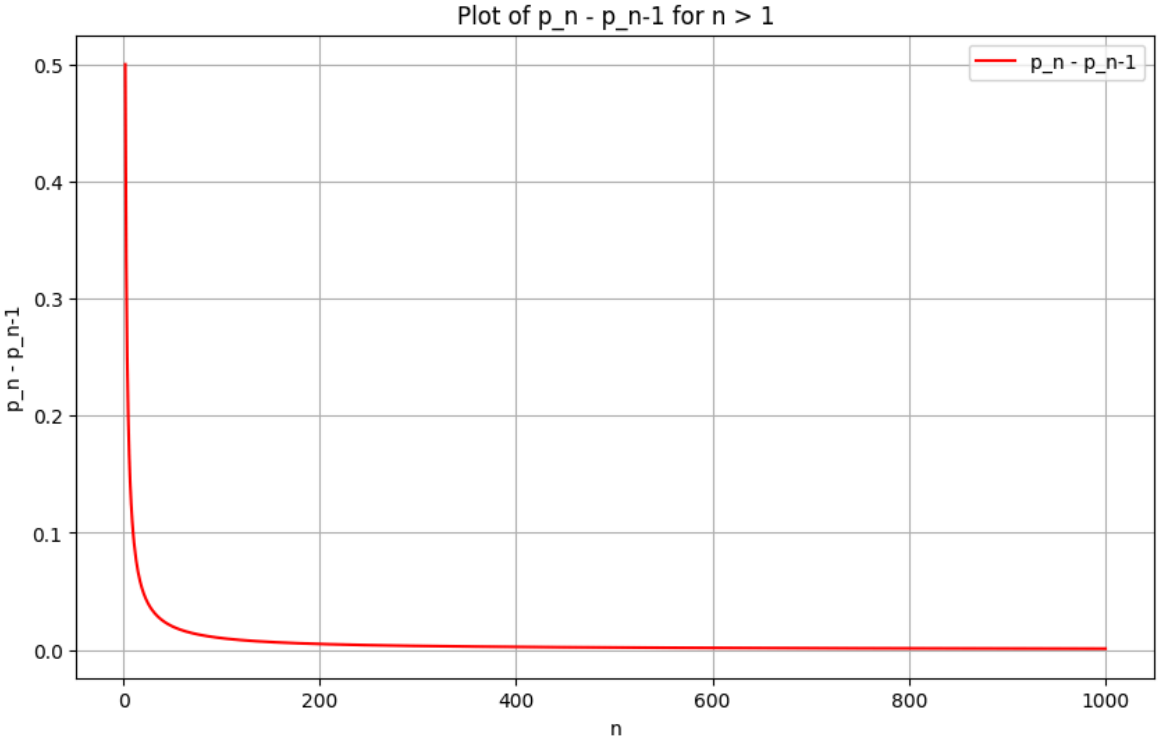
```
Check |f(p_n)| < 10^(-3) for n > 1
n = 2, p_n = 1.500000, |f(p_n)| = 9.765625e-04, Condition Met: True
n = 3, p_n = 1.333333, |f(p_n)| = 1.693509e-05, Condition Met: True
n = 4, p_n = 1.250000, |f(p_n)| = 9.536743e-07, Condition Met: True
n = 5, p_n = 1.200000, |f(p_n)| = 1.024000e-07, Condition Met: True
n = 6, p_n = 1.166667, |f(p_n)| = 1.653817e-08, Condition Met: True
n = 7, p_n = 1.142857, |f(p_n)| = 3.540133e-09, Condition Met: True
n = 8, p_n = 1.125000, |f(p_n)| = 9.313226e-10, Condition Met: True
n = 9, p_n = 1.111111, |f(p_n)| = 2.867972e-10, Condition Met: True
n = 10, p_n = 1.100000, |f(p_n)| = 1.000000e-10, Condition Met: True


Check |p - p_n| < 10^(-3) requires that n>1000
n = 1001, p_n = 1.000999, |p - p_n| = 9.990010e-04, Condition Met: True
```

8. Let (p_n) be the sequence defined by $p_n = \sum_{k=1}^{n} \frac{1}{k}$. Show that (p_n) diverge even though $\lim_{n\to\infty} (p_n - p_{n-1}) = 0$.

In [19]:
```
import numpy as np
import matplotlib.pyplot as plt

# function to compute p_n
def p_n(n):
    return np.sum(1 / np.arange(1, n + 1))

# Value of n
n_values = np.arange(1, 1001)

# compute p_n for each n
p_n_values = np.array([p_n(n) for n in n_values])

# compute p_n - p_n-1 for each n
p_diff_values = np.array([p_n(n) - p_n(n - 1) for n in n_values])

# plot p_n to show divergence
plt.figure(figsize=(10, 6))
plt.plot(n_values, p_n_values, label="p_n")
```

```
plt.xlabel("n")
plt.ylabel("p_n")
plt.title("Plot of p_n = Sum(1/k) for k = 1 to n")
plt.grid(True)
plt.legend()
plt.show()

# plot p_n - p_n-1 to show it approaches 0
plt.figure(figsize=(10, 6))
plt.plot(n_values[1:], p_diff_values[1:], label="p_n - p_n-1", color="red
plt.xlabel("n")
plt.ylabel("p_n - p_n-1")
plt.title("Plot of p_n - p_n-1 for n > 1")
plt.grid(True)
plt.legend()
plt.show()

# Print some values for illustration
for n in [10, 50, 100, 1000]:
    print(f"p_{n} = {p_n(n):.6f}, p_{n} - p_{n-1} = {p_n(n) - p_n(n-1):.6
```






```
p_10 = 2.928968, p_10 - p_9 = 0.100000
p_50 = 4.499205, p_50 - p_49 = 0.020000
p_100 = 5.187378, p_100 - p_99 = 0.010000
p_1000 = 7.485471, p_1000 - p_999 = 0.001000
```

9. Use the Bisection method to find p_3 for $f(x) = \sqrt{x} - \cos x$ on $[0, 1]$.

In [20]:
```python
import math

def f(x):
    return math.sqrt(x) - math.cos(x)

def bisection_method(a, b, tol= 1e-6, max_iter= 50):
    iter_count = 0
    while iter_count < max_iter:
        m = (a + b) / 2
        fm = f(m)

        if abs(fm) < tol:
            return m

        if f(a) * fm < 0:
            b = m
        else:
            a = m

        iter_count += 1
    return (a+b)/2

# Define the interval [a, b]
a = 0
b = 1

# Find p3, which is the third approximation
p3 = bisection_method(a, b, tol=1e-6, max_iter=3)
```

```
print("Use the Bisection method to find the third approximation for f(x)
print(f"p3 = {p3:.6f}")
```

```
Use the Bisection method to find the third approximation for f(x) = sqrt
(x) - cos(x) on [0, 1]
p3 = 0.687500
```

10. Use the Bisection method to find solutions accurate to within 10^{-2} for $x^3 - 7x^2 + 14x - 6 = 0$ on each interval.
    a. $[0, 1]$
    b. $[1, 3.2]$
    c. $[3.2, 4]$

In [21]:
```
# compute f(x)
def f(x):
    return x**3 - 7*x**2 + 14*x - 6

def bisection_method(a, b, tol= 1e-6, max_iter= 50):
    iter_count = 0
    while iter_count < max_iter:
        m = (a + b) / 2
        fm = f(m)

        if abs(fm) < tol:
            return m

        if f(a) * fm < 0:
            b = m
        else:
            a = m

        iter_count += 1
    return (a+b)/2

intervals = [[0, 1], [1, 3.2], [3.2, 4]]
print("Use the Bisection method to find the solution accurate to within 1
for i, (a, b) in enumerate(intervals):
    p = bisection_method(a, b, tol=1e-6)
    print(f"for {a, b}, p{i+1} = {p:.6f}")
```

```
Use the Bisection method to find the solution accurate to within 10^-2 for
f(x) = x^3 - 7x^2 + 14x - 6 = 0
for (0, 1), p1 = 0.585786
for (1, 3.2), p2 = 3.000000
for (3.2, 4), p3 = 3.414214
```

11. Use the Bisection method to find solutions accurate to within 10^{-5} for the following problems.
    a. $x - 2^{-x} = 0$ for $0 \leq x \leq 1$
    b. $e^x - x^2 + 3x - 2 = 0$ for $0 \leq x \leq 1$
    c. $2x\cos(2x) - (x+1)^2 = 0$ for $-3 \leq x \leq -2$ and $-1 \leq x \leq 0$
    d. $x\cos(x) - 2x^2 + 3x - 1 = 0$ for $0.2 \leq x \leq 0.3$ and $1.2 \leq x \leq 1.3$

In [22]:
```
import math

# compute f(x) in a
```

```
def f_a(x):
    return x**3 - 7*x**2 + 14*x - 6

# compute f(x) in b
def f_b(x):
    return math.exp(x) - x**2 +3*x - 2

#compute f(x) in c
def f_c(x):
    return 2*x*math.cos(2*x) - pow((x+1), 2)

# compute f(x) in d
def f_d(x):
    return x*math.cos(x) -2*x**2 + 3*x - 1

# bisection method for f(x) in a
def bisection_method_a(a, b, tol= 1e-6, max_iter= 50):
    iter_count = 0
    while iter_count < max_iter:
        m = (a + b) / 2
        fm = f_a(m)

        if abs(fm) < tol:
            return m

        if f_a(a) * fm < 0:
            b = m
        else:
            a = m

        iter_count += 1
    return (a+b)/2

# bisection method for f(x) in b
def bisection_method_b(a, b, tol= 1e-6, max_iter= 50):
    iter_count = 0
    while iter_count < max_iter:
        m = (a + b) / 2
        fm = f_b(m)

        if abs(fm) < tol:
            return m

        if f_b(a) * fm < 0:
            b = m
        else:
            a = m

        iter_count += 1
    return (a+b)/2

# bisection method for f(x) in c
def bisection_method_c(a, b, tol= 1e-6, max_iter= 50):
    iter_count = 0
    while iter_count < max_iter:
        m = (a + b) / 2
        fm = f_c(m)

        if abs(fm) < tol:
            return m
```

```
        if f_c(a) * fm < 0:
            b = m
        else:
            a = m

        iter_count += 1
    return (a+b)/2

# bisection method for f(x) in d
def bisection_method_d(a, b, tol= 1e-6, max_iter= 50):
    iter_count = 0
    while iter_count < max_iter:
        m = (a + b) / 2
        fm = f_d(m)

        if abs(fm) < tol:
            return m

        if f_d(a) * fm < 0:
            b = m
        else:
            a = m

        iter_count += 1
    return (a+b)/2

print("Use the Bisection method to find the solution accurate to within 1
print("a. f(x)=x-2^(-x)=0 for 0 <= x <= 1")
print(f"The root in [0, 1] is: {bisection_method_a(0, 1, tol=1e-5):.4f}")
print("\nb. f(x)=e^x-x^2+3x-2=0 for 0 <= x <= 1")
print(f"The root in [0, 1] is: {bisection_method_b(0, 1, tol=1e-5):.4f}")
print("\nc. f(x)=2xcos(2x)-(x+1)^2=0 for -3 <= x <= -2 and -1 <= x <= 0")
print(f"The root in [-3, -2] is: {bisection_method_c(-3, -2, tol=1e-5):.4
print(f"The root in [-1, 0] is: {bisection_method_c(-1, 0, tol=1e-5):.4f}
print("\nd. f(x)=xcos(x)-2x^2+3x-1=0 for 0.2 <= x <= 0.3 and 1.2 <= x <=
print(f"The root in [0.2, 0.3] is: {bisection_method_d(0.2, 0.3, tol=1e-5
print(f"The root in [1.2, 1.3] is: {bisection_method_d(1.2, 1.3, tol=1e-5
```

```
Use the Bisection method to find the solution accurate to within 10^-5 for
the following problems
a. f(x)=x-2^(-x)=0 for 0 <= x <= 1
The root in [0, 1] is: 0.5858

b. f(x)=e^x-x^2+3x-2=0 for 0 <= x <= 1
The root in [0, 1] is: 0.2575

c. f(x)=2xcos(2x)-(x+1)^2=0 for -3 <= x <= -2 and -1 <= x <= 0
The root in [-3, -2] is: -2.1913
The root in [-1, 0] is: -0.7982

d. f(x)=xcos(x)-2x^2+3x-1=0 for 0.2 <= x <= 0.3 and 1.2 <= x <= 1.3
The root in [0.2, 0.3] is: 0.2975
The root in [1.2, 1.3] is: 1.2566
```

12. Use a fixed-point iteration method to determine a solution accurate to within 10^{-2} for $x^4 - 3x^2 - 3 = 0$ on $[1, 2]$. Use $p_0 = 1$.

Convert into Fixed-form

$$x^4 - 3x^2 - 3 = 0$$

$$\Rightarrow x = (3x^2 + 3)^{\frac{1}{4}}$$

In [23]:
```
def g(x):
    return (3*x**2+3)**(1/4)

def fixed_point_iteration(p0, tol=1e-2, max_iter=50):
    iter_count = 0
    while iter_count < max_iter:
        p1 = g(p0)
        if abs(p1 - p0) < tol:
            return p1
        p0 = p1
        iter_count += 1
    raise ValueError("Method did not converge")

p0 = 1
solution = fixed_point_iteration(p0)
print(f"Use the fixed-point iteration method to find the solution accurat
print(f"The solution is: {solution:.4f}")
```

```
Use the fixed-point iteration method to find the solution accurate to with
in 10^-2 for x^4-3x^2-3=0
The solution is: 1.9433
```

13. Use a fixed-point iteration method to determine a solution accurate to within 10^{-2} for $x^3 - x - 1 = 0$ on $[1, 2]$. Use $p_0 = 1$.

Convert into Fixed-form

$$x^3 - x - 1 = 0$$

$$\Rightarrow x = (x + 1)^{\frac{1}{3}}$$

In [24]:
```
def g(x):
    return (x+1)**(1/3)

def fixed_point_iteration(p0, tol=1e-2, max_iter=50):
    iter_count = 0
    while iter_count < max_iter:
        p1 = g(p0)
        if abs(p1 - p0) < tol:
            return p1
        p0 = p1
        iter_count += 1
    raise ValueError("Method did not converge")

p0 = 1
solution = fixed_point_iteration(p0)
print(f"Use the fixed-point iteration method to find the solution accurat
print(f"The solution is: {solution:.4f}")
```

```
Use the fixed-point iteration method to find the solution accurate to with
in 10^-2 for x^3-x-1=0
The solution is: 1.3243
```

14. For each of the following equations, use the given interval or determine an interval $[a, b]$ on which fixed-point iteration will coverge. Estimate the number of iterations necessary to obtain approximations accurate to within 10^{-5}, and perform the calculations.
    a. $2 + \sin(x) - x = 0$ use $[2, 3]$
    b. $x^3 - 2x - 5 = 0$ use $[2, 3]$
    c. $3x^2 - e^x = 0$
    d. $x - \cos(x) = 0$

In [25]:
```
import math

def g_a(x):
    return 2+math.sin(x)

def g_b(x):
    return math.pow((2*x+5), 1/3)

def g_c(x):
    return math.log(3*x**2)

def g_d(x):
    return math.cos(x)

def fixed_point_a(g, p0, tol=1e-5, max_iter=100):
    for _ in range(max_iter):
        p1 = g(p0)
        if abs(p1 - p0) < tol:
            return p1
        p0 = p1
    return None

print("Use the fixed-point iteration method to find the solution accurate
print("\na. 2+sin(x)-x=0 use [2,3]")
print(f"The solution is: {fixed_point_a(g_a, 2, tol=1e-5):.4f}")
print("\nb. x^3-2x-5=0 use [2,3]")
print(f"The solution is: {fixed_point_a(g_b, 2, tol=1e-5):.4f}")
print("\nc. 3x^2-e^x=0")
print(f"The solution is: {fixed_point_a(g_c, 1, tol=1e-5):.4f}")
print("\nd. x-cos(x)=0")
print(f"The solution is: {fixed_point_a(g_d, 0.5, tol=1e-5):.4f}")
```

```
Use the fixed-point iteration method to find the solution accurate to with
in 10^-5 for the following problems

a. 2+sin(x)-x=0 use [2,3]
The solution is: 2.5542

b. x^3-2x-5=0 use [2,3]
The solution is: 2.0946

c. 3x^2-e^x=0
The solution is: 3.7331

d. x-cos(x)=0
The solution is: 0.7391
```

15. Let $f(x) = x^2 - 6$ and $p_0 = 1$. Use Newton's method to find p_2.

In [26]:
```
def f(x):
    return x**2 - 6

def f_prime(x):
    return 2*x

def newton_method(p0, tol=1e-5, iteration = 2):
    p = p0
    for _ in range(iteration):
        p = p - f(p) / f_prime(p)
    return p

p2 = newton_method(1, 2)
print("Use Newton's method to find p_2:")
print(f"p_2 = {p2:.4f}")
```

```
Use Newton's method to find p_2:
p_2 = 2.6071
```

16. Let $f(x) = -x^3 - \cos(x)$ and $p_0 = -1$. Use Newton's method to find p_2.

In [27]:
```
import math

def f(x):
    return -1*x**3 - math.cos(x)

def f_prime(x):
    return -3*x**2 + math.sin(x)

def newton_method(p0, iteration = 2):
    p = p0
    for _ in range(iteration):
        p = p - f(p) / f_prime(p)
    return p

p2 = newton_method(-1, 2)
print("Use Newton's method to find p_2:")
print(f"p_2 = {p2:.4f}")
```

```
Use Newton's method to find p_2:
p_2 = -0.8657
```

17. Write a general purpose alogorithm for the method of False Position.
    a. Describe your algorithm objective?
    b. Describe required input parameter(s)?
    c. Describe output value(s)?
    d. Write your algorithm body here.
    e. Implement your developed algorithm with Python, or any other programming language.

False Position Method Algorithm

a. Algorithm objective

The False Position Method (Regula Falsi) is a root-finding algorithm that iteratively refines an approximation of the root of a function $f(x) = 0$. It is similar to the Bisection Method, but instead of using the midpoint, it selects a new approximation using a secant line between two points. The method guarantees convergence if $f(x)$ is continuous in the interval and there is a sign change, i.e., $f(a)f(b) < 0$.

b. Required Input Parameter(s)

- $f(x)$: The function for which we are finding the root.
- a: The lower bound of the interval where $f(a)$ is defined.
- b: The upper bound of the interval where $f(b)$ is defined.
- tol: The tolerance value (stopping criterion).
- max_iter: Maximum number of iterations to avoid infinite loops.

c. Output Value(s)

- p: The approximate root of $f(x) = 0$
- $f(p)$: The function value at p (should be close to zero)
- **Number of iterations** taken to reach the required tolerance.

d. Algorithm Body

- Check that $f(a)f(b) < 0$
- Compute the first approximation using the False Position formula:

$$p = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

- Evaluate $f(p)$:
  - If $f(p)$ is close enough to zero (within tolerance), stop.
  - Otherwise, update either a or b depending on the sign of $f(p)$.
- Repeat the process until convergences or max iterations.

e. Python Implementation

In [28]:
```
def false_position(f, a, b, tol=1e-5, max_iter=100):
    if f(a) * f(b) >= 0:
        raise ValueError("Function values at a and b must have opposite s

    for i in range(max_iter):
        p = (a * f(b) - b * f(a)) / (f(b) - f(a))

        if abs(f(p)) < tol:
            return p, i + 1

        if f(a) * f(p) < 0:
            b = p
        else:
            a = p
```

```
    return p, max_iter

def f(x):
    return x**3 - 4*x - 9

root, iterations = false_position(f, 2, 3)
print(f"Approximate root: {root:.5f} (found in {iterations} iterations)")
```

```
Approximate root: 2.70653 (found in 7 iterations)
```

18. Let $f(x) = x^2 - 6$. With $p_0 = 3$ and $p_1 = 2$, find p_3.
    a. Use the secant Method.
    b. Use the method of False Position.
    c. Which of (a) or (b) is closer to $\sqrt{6}$

In [29]:
```
def f(x):
    return x**2 - 6

# Secant Method
def secant_method(p0, p1, iterations=2):
    for _ in range(iterations):
        p_next = p1 - f(p1) * (p1 - p0) / (f(p1) - f(p0))
        p0, p1 = p1, p_next
    return p1

# False Position Method
def false_position_method(a, b, iterations=2):
    for _ in range(iterations):
        p = (a * f(b) - b * f(a)) / (f(b) - f(a))
        if f(p) * f(a) < 0:
            b = p
        else:
            a = p
    return p

# Compute values
p3_secant = secant_method(3, 2, 2)
p3_false_position = false_position_method(2, 3, 2)

print("p_3:")
print(f"a. Secant Method p3: {p3_secant:.5f}")
print(f"b. False Position Method p3: {p3_false_position:.5f}")
print(f"c. Actual sqrt(6): {6**0.5:.5f}. So, (a) Secant Method p3: {p3_se
```

```
p_3:
a. Secant Method p3: 2.45455
b. False Position Method p3: 2.44444
c. Actual sqrt(6): 2.44949. So, (a) Secant Method p3: 2.45455 is close to
sqrt(6): 2.44949
```

19. Let $f(x) = -x^3 - \cos x$. With $p_0 = 0$ and $p_1 = 0$, find p_3.
    a. Use the Secant method.
    b. Use the method of False Position.

In [30]:
```
import math

def f(x):
```

```
    return -1*x**3 - math.cos(x)

# Secant Method
def secant_method(p0, p1, iterations=2):
    for _ in range(iterations):
        p_next = p1 - f(p1) * (p1 - p0) / (f(p1) - f(p0))
        p0, p1 = p1, p_next
    return p1

# False Position Method
def false_position_method(a, b, iterations=2):
    for _ in range(iterations):
        p = (a * f(b) - b * f(a)) / (f(b) - f(a))
        if f(p) * f(a) < 0:
            b = p
        else:
            a = p
    return p

# Compute values
p3_secant = secant_method(3, 2, 2)
p3_false_position = false_position_method(2, 3, 2)

print("p_3:")
print(f"Secant Method p3: {p3_secant:.5f}")
print(f"False Position Method p3: {p3_false_position:.5f}")
```

```
p_3:
Secant Method p3: 1.13145
False Position Method p3: 1.33264
```

20. Find solutions accurate to within 10^{-5} for the problem $e^x - 3x^2 = 0$ for $0 \leq x \leq 1$ and $3 \leq x \leq 5$ using

    a. Newton's method.

    b. Secant method.

    c. The method of False Position.

In [31]:

```
import math

def f(x):
    return math.exp(x) - 3*x**2

def df(x):
    return math.exp(x) - 6*x

# Newton's Method
def newton_method(p0, tol=1e-5, max_iter=100):
    for _ in range(max_iter):
        p1 = p0 - f(p0) / df(p0)
        if abs(p1 - p0) < tol:
            return p1
        p0 = p1
    return None  # Did not converge

# Secant Method
def secant_method(p0, p1, tol=1e-5, max_iter=100):
    for _ in range(max_iter):
        p2 = p1 - f(p1) * (p1 - p0) / (f(p1) - f(p0))
```

```
        if abs(p2 - p1) < tol:
            return p2
        p0, p1 = p1, p2
    return None  # Did not converge

# False Position Method
def false_position_method(a, b, tol=1e-5, max_iter=100):
    for _ in range(max_iter):
        p = (a * f(b) - b * f(a)) / (f(b) - f(a))
        if abs(f(p)) < tol:
            return p
        if f(a) * f(p) < 0:
            b = p
        else:
            a = p
    return None  # Did not converge

# Compute solutions
x1_newton = newton_method(0.5)
x2_newton = newton_method(4)

x1_secant = secant_method(0, 1)
x2_secant = secant_method(3, 5)

x1_false_pos = false_position_method(0, 1)
x2_false_pos = false_position_method(3, 5)

# Print results
print(f"Newton's Method:\n \tx in [0,1] = {x1_newton:.5f},\n \tx in [3,5]
print(f"Secant Method:\n \tx in [0,1] = {x1_secant:.5f},\n \tx in [3,5] =
print(f"False Position Method:\n \tx in [0,1] = {x1_false_pos:.5f},\n \tx
```

```
Newton's Method:
        x in [0,1] = 0.91001,
        x in [3,5] = 3.73308

Secant Method:
        x in [0,1] = 0.91001,
        x in [3,5] = 3.73308

False Position Method:
        x in [0,1] = 0.91001,
        x in [3,5] = 3.73308
```

21. The four-degree polynomial $f(x) = 230x^4 + 18x^3 + 9x^2 - 221x - 9$ has two real zeros, one in $[-1, 0]$ and the other in $[0, 1]$. Attempt to approximate these zeros to within 10^{-6} using the
    a. Method of False Position
    b. Secant method
    c. Newton's method

In [32]:
```
import math

# Define the function
def f(x):
    return 230*x**4 + 18*x**3 + 9*x**2 - 221*x - 9

# Define its derivative (for Newton's method)
```

```python
def df(x):
    return 920*x**3 + 54*x**2 + 18*x - 221

# False Position Method
def false_position(a, b, tol=1e-6, max_iter=100):
    for _ in range(max_iter):
        p = (a * f(b) - b * f(a)) / (f(b) - f(a))
        if abs(f(p)) < tol:
            return p
        if f(a) * f(p) < 0:
            b = p
        else:
            a = p
    return None  # Did not converge

# Secant Method
def secant_method(p0, p1, tol=1e-6, max_iter=100):
    for _ in range(max_iter):
        if abs(f(p1) - f(p0)) < 1e-12:  # Avoid division by zero
            return None
        p2 = p1 - f(p1) * (p1 - p0) / (f(p1) - f(p0))
        if abs(p2 - p1) < tol:
            return p2
        p0, p1 = p1, p2
    return None  # Did not converge

# Newton's Method
def newton_method(p0, tol=1e-6, max_iter=100):
    for _ in range(max_iter):
        if abs(df(p0)) < 1e-12:  # Avoid division by zero
            return None
        p1 = p0 - f(p0) / df(p0)
        if abs(p1 - p0) < tol:
            return p1
        p0 = p1
    return None  # Did not converge

# Compute roots in [-1,0] and [0,1]
x1_false_pos = false_position(-1, 0)
x2_false_pos = false_position(0, 1)

x1_secant = secant_method(-1, 0)
x2_secant = secant_method(0, 1)

x1_newton = newton_method(-0.5)  # Start near the root
x2_newton = newton_method(0.5)   # Start near the root

# Print results
print(f"False Position Method:\n \tx in [-1,0] = {x1_false_pos:.6f},\n \t
print(f"Secant Method:\n \tx in [-1,0] = {x1_secant:.6f},\n \tx in [0,1]
print(f"Newton's Method:\n \tx in [-1,0] = {x1_newton:.6f},\n \tx in [0,1
```

```
False Position Method:
        x in [-1,0] = -0.040659,
        x in [0,1] = 0.962398

Secant Method:
        x in [-1,0] = -0.040659,
        x in [0,1] = -0.040659

Newton's Method:
        x in [-1,0] = -0.040659,
        x in [0,1] = -0.040659
```

22. The accumulated value of a savings account based on regular periodic payments can be determined from the annuity due equation,

$$A = \frac{P}{i}[(1+i)^n - 1].$$

In this equation, A is the amount in the account, P is the amount regularly deposited, and i is the rate of interest per period for the n deposit periods. An engineer would like to have a savings account valued at $750,000 upon retirement is 20 year and can afford to put 1,500 per month toward this goal. What is the minimal interest rate at which this amount can be invested, assuming that the interest is compounded monthly?

In [33]:
```python
import math

# Given data
A = 750000   # Target amount
P = 1500     # Monthly deposit
n = 20 * 12  # Total deposit periods

# Define function f(i) = (1+i)^n - 1 - (A*i)/P
def f(i):
    return (1 + i)**n - 1 - (A * i) / P

# Define derivative f'(i) for Newton's method
def df(i):
    return n * (1 + i)**(n - 1) - (A / P)

# Newton's method to find i
def newtons_method(i0, tol=1e-6, max_iter=100):
    i = i0
    for _ in range(max_iter):
        fi = f(i)
        dfi = df(i)
        if abs(fi) < tol:
            return i
        i -= fi / dfi  # Newton update
    return None  # If not converged

# Initial guess (small interest rate)
i_min = newtons_method(0.005)  # Initial guess: 0.5% per month

# Convert to annual interest rate
```

```
r_min = i_min * 12 * 100

# Print results
print(f"We have:\n\t- Target amount: {A}\n\t- Monthly deposit: {P}\n\t- T
print(f"Minimum required annual interest rate: {r_min:.6f}%")
```

```
We have:
        - Target amount: 750000
        - Monthly deposit: 1500
        - Total deposit periods: 240

Minimum required annual interest rate: 6.660941%
```

23. Write a general purpose alogorithm for Müller's method.
    a. Describe your alogorithm objective?
    b. Describe required input parameter(s)?
    c. Describe output value(s)?
    d. Write your algorithm body here.
    e. Implement your developed algorithm with Python, or any other programming language.

Müller's Method: General-Purpose Algorithm

a. Algorithm Objective

Müller's method is used to find the roots of a function $f(x)$. It extends the secant method by using a quadratic interpolation to approximate the root. It is particularly useful when the function is non-linear or when other methods converge slowly. b. Required Input Parameters

- Function $f(x)$ to find the root.
- Initial three points x_0, x_1, x_2 (must be reasonably close to the root).
- Tolerance ϵ (stopping criteria).
- Maximum iterations to prevent infinite loops.

c. Output Values

- The approximated root p of $f(x)$ within the given tolerance.
- The number of iterations taken to reach the root.
- If the method fails to converge, an appropriate message.

d. Algorithm Steps

- Initial three points x_0, x_1, x_2.
- Compute function values $f(x_0), f(x_1), f(x_2)$.
- Iterate until convergence:
  - Compute divided differences:

$$h_1 = x_1 - x_0, h_2 = x_2 - x_1$$

$$\delta_1 = \frac{f(x_1) - f(x_0)}{h_1}, \delta_2 = \frac{f(x_2) - f(x_1)}{h_2}$$

$$a = \frac{\delta_2 - \delta_1}{h_2 + h_1}$$

$$b = ah_2 + \delta_2, c = f(x_2)$$

- Solve for the root of the quadratic equation:

$$x = x_2 + \frac{-2c}{b \backslash \text{plusmn} \sqrt{b^2 - 4ac}}$$

  Choose the sign in the denominator to avoid cancellation errors.
- Check convergence $|x - x_2| < \epsilon$
- Update x_0, x_1, x_2 for the next iteration.

- Return the computed root if the method converges.

e. Python Implementation of Müller's Method

In [34]:
```python
import cmath  # Supports complex numbers

def muller(f, x0, x1, x2, tol=1e-6, max_iter=100):
    for _ in range(max_iter):
        h1, h2 = x1 - x0, x2 - x1
        delta1 = (f(x1) - f(x0)) / h1
        delta2 = (f(x2) - f(x1)) / h2
        a = (delta2 - delta1) / (h2 + h1)
        b = a * h2 + delta2
        c = f(x2)

        # Discriminant of quadratic equation
        disc = cmath.sqrt(b**2 - 4*a*c)

        # Choose the denominator to avoid cancellation error
        if abs(b + disc) > abs(b - disc):
            denom = b + disc
        else:
            denom = b - disc

        # Compute new root
        x_new = x2 - (2 * c) / denom

        # Check for convergence
        if abs(x_new - x2) < tol:
            return x_new

        # Update points
        x0, x1, x2 = x1, x2, x_new

    return None  # If method does not converge

# Example Usage:
f = lambda x: x**3 - 5*x**2 + 7*x - 3  # Example function
root = muller(f, 0.5, 1.0, 1.5)
print(f"Root found: {root:.6f}")
```

```
Root found: 1.000000+0.000000j
```

24. Use each of the following methods to find a solution in $[0.1, 1]$ accurate to within 10^{-4} for

$$600x^4 - 550x^3 + 200x^2 - 20x - 1 = 0.$$

a. Bisection method
b. Newton's method
c. Secant method
d. method of False Position
e. Müller's method.

In [35]:
```python
import numpy as np

def f(x):
    return 600*x**4 - 550*x**3 + 200*x**2 - 20*x - 1

def df(x):
    return 2400*x**3 - 1650*x**2 + 400*x - 20  # Derivative of f(x)

# a. Bisection method
def bisection_method(a, b, tol=1e-4, max_iter=100):
    if f(a) * f(b) >= 0:
        print("Bisection method fails. No root found in the given interva
        return None

    for _ in range(max_iter):
        c = (a + b) / 2
        if abs(f(c)) < tol or abs(b - a) < tol:
            return c
        elif f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return c

#b. Newton's method
def newton_method(p0, tol=1e-4, max_iter=100):
    p = p0
    for _ in range(max_iter):
        p_next = p - f(p) / df(p)
        if abs(p_next - p) < tol:
            return p_next
        p = p_next
    return p

#c. Secant method
def secant_method(p0, p1, tol=1e-4, max_iter=100):
    for _ in range(max_iter):
        if abs(f(p1) - f(p0)) < 1e-10:  # Prevent division by zero
            return None
        p2 = p1 - f(p1) * (p1 - p0) / (f(p1) - f(p0))
        if abs(p2 - p1) < tol:
            return p2
        p0, p1 = p1, p2
    return p1
```

```
#d. False Position method
def false_position_method(a, b, tol=1e-4, max_iter=100):
    if f(a) * f(b) >= 0:
        print("False Position method fails. No root found in the given in
        return None

    for _ in range(max_iter):
        c = (a * f(b) - b * f(a)) / (f(b) - f(a))
        if abs(f(c)) < tol or abs(b - a) < tol:
            return c
        elif f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return c

#e. Müller Method
def mullers_method(f, x0, x1, x2, tol=1e-4, max_iter=100):
    for _ in range(max_iter):
        h1, h2 = x1 - x0, x2 - x1
        d1, d2 = (f(x1) - f(x0)) / h1, (f(x2) - f(x1)) / h2
        a = (d2 - d1) / (h2 + h1)
        b = a * h2 + d2
        c = f(x2)

        # Prevent square root of negative number
        discriminant = b**2 - 4*a*c
        if discriminant < 0:
            rad = np.sqrt(complex(discriminant, 0))  # Handle complex roo
        else:
            rad = np.sqrt(discriminant)

        if abs(b + rad) > abs(b - rad):
            den = b + rad
        else:
            den = b - rad

        if den == 0:
            print("Müller's method failed: Division by zero")
            return None

        x3 = x2 - (2 * c) / den

        if abs(x3 - x2) < tol:
            return x3.real  # Ensure returning only real part if complex

        x0, x1, x2 = x1, x2, x3

    return x3.real  # Ensure returning real root

root_bisection = bisection_method(0.1, 1)
print("a. Bisection method:")
print(f"\tBisection Method Root: {root_bisection}\n")
root_newton = newton_method(0.5)
print("b. Newton's Method:")
print(f"\tNewton's Method Root: {root_newton}\n")
root_secant = secant_method(0.1, 1)
print("c. Secant Method:")
print(f"\tSecant Method Root: {root_secant}\n")
root_false_position = false_position_method(0.1, 1)
```

```
print("d. False Position Method:")
print(f"\tFalse Position Method Root: {root_false_position}\n")
root_muller = mullers_method(f, 0.1, 0.5, 1)
print("e. Müller's Method:")
print(f"\tMüller's Method Root: {root_muller}")
```

```
a. Bisection method:
        Bisection Method Root: 0.23235778808593754

b. Newton's Method:
        Newton's Method Root: 0.23235296476876366

c. Secant Method:
        Secant Method Root: 0.23235295673399128

d. False Position Method:
        False Position Method Root: 0.23127760311637524

e. Müller's Method:
        Müller's Method Root: 0.3600766969201697
```

25. Write a general purpose algorithm for a method that can find all real and complex zeros of a given polynomial.
    a. Describe your algorithm objective?
    b. Describe required input parameter(s)?
    c. Describe output value(s)?
    d. Write your algorithm body here.
    e. Implement your developed algorithm with Python, or any other programming language.

General Purpose Algorithm to Find All Real and Complex Zeros of a Polynomial

a. Algorithm Objective

The objective of this algorithm is to find all the real and complex zeros (roots) of a given polynomial equation. This can be achieved using numerical methods like Newton's method, synthetic division, or leveraging libraries for symbolic computation. The algorithm should be able to handle any degree of polynomial, and return both real and complex roots.

b. Required Input Parameter(s)

- Polynomial Coefficients: A list or array of coefficients of the polynomial.
- Tolerances/Accuracy: Precision required for approximating roots
- Maximum Iterations (optional): The maximum number of iterations to prevent infinite loops in methods like Newton's method.

c. Output Value(s)

The output should be a list of all real and complex roots of the polynomial. Complex roots should be represented as complex numbers.

d. Algorithm Body

- Step 1: Begin by expressing the polynomial in terms of its coefficients.
- Step 2: Use a numerical method like Newton’s Method, Bisection, or a polynomial root-finding algorithm such as Durand-Kerner or Laguerre's method.

- Step 3: Ensure that both real and complex roots are found by solving the polynomial equation iteratively.
- Step 4: Return the roots as a list of numbers (real and complex).

e. Python Implementation:

In [36]:
```
import numpy as np

def find_all_zeros(coefficients):
    # Use numpy's built-in roots function to find all roots of the polyno
    roots = np.roots(coefficients)

    # Filter out any small imaginary parts if you want purely real roots
    # This step is optional, depending on whether you want to keep comple
    roots = [root.real if abs(root.imag) < 1e-5 else root for root in roo

    return roots

# Example usage
coefficients = [1, -6, 11, -6]  # Polynomial: x^3 - 6x^2 + 11x - 6
roots = find_all_zeros(coefficients)
print("Polynomial Equation: x^3 - 6x^2 + 11x - 6 = 0")
print("All Roots (Real and Complex):", roots)
```

```
Polynomial Equation: x^3 - 6x^2 + 11x - 6 = 0
All Roots (Real and Complex): [3.0000000000000018, 1.999999999999998, 1.00
00000000000002]
```

26. Find approximations to within 10^{-5} to all the zeros of each of the following polynoials

    a. $-20 + 16x + x^2 - 4x^3 + x^4$

    b. $x^4 + 5x^3 - 9x^2 - 85x - 136$

    c. $x^5 + 11x^4 - 21x^3 - 10x^2 - 21x - 5$

In [37]:
```
import numpy as np

def find_zeros_of_polynomial(coefficients, tol=1e-5):
    # Find the roots of the polynomial using numpy
    roots = np.roots(coefficients)

    # Filter roots to have no more than the specified tolerance
    roots = [root.real if abs(root.imag) < tol else root for root in root

    return roots

# Define polynomials and their coefficients
polynomial_a = [-20, 16, 1, -4, 1]  # -20 + 16x + x^2 - 4x^3 + x^4
polynomial_b = [1, 5, -9, -85, -136]  # x^4 + 5x^3 - 9x^2 - 85x - 136
polynomial_c = [1, 11, -21, -10, -21, -5]  # x^5 + 11x^4 - 21x^3 - 10x^2

# Find and print the roots for each polynomial
roots_a = find_zeros_of_polynomial(polynomial_a)
roots_b = find_zeros_of_polynomial(polynomial_b)
roots_c = find_zeros_of_polynomial(polynomial_c)

print("a. -20 + 16x + x^2 - 4x^3 + x^4")
```

```
print("a. -20 + 16x + x^2 - 4x^3 + x^4")
print(f"\tRoots of polynomial a: {[f'{root:.2f}' for root in roots_a]}\n"
print("b. x^4 + 5x^3 - 9x^2 - 85x - 136")
print(f"\tRoots of polynomial b: {[f'{root:.2f}' for root in roots_b]}\n"
print("c. x^5 + 11x^4 - 21x^3 - 10x^2 - 21x - 5")
print(f"\tRoots of polynomial c: {[f'{root:.2f}' for root in roots_c]}\n"
```

```
a. -20 + 16x + x^2 - 4x^3 + x^4
a. -20 + 16x + x^2 - 4x^3 + x^4
        Roots of polynomial a: ['-0.50', '0.50', '0.40+0.20j', '0.40-0.20
j']

b. x^4 + 5x^3 - 9x^2 - 85x - 136
        Roots of polynomial b: ['4.12', '-4.12', '-2.50+1.32j', '-2.50-1.3
2j']

c. x^5 + 11x^4 - 21x^3 - 10x^2 - 21x - 5
        Roots of polynomial c: ['-12.61', '2.26', '-0.20+0.81j', '-0.20-0.
81j', '-0.25']
```

27. Find approximations to within 10^{-10} to all zeros of each of the following Lagurre polynomials using Müller's and Lagurre's method.

n	$L_n(x)$
0	1
1	$-x+1$
2	$\frac{1}{2}(x^2-4x+2)$
3	$\frac{1}{6}(-x^3+9x^2-18x+6)$
4	$\frac{1}{24}(x^4-16x^3+72x^2-96x+6)$
5	$\frac{1}{720}(x^6-36x^5+450x^4-2400x^3+5400x^2-4320x+720)$
n	$\frac{1}{n!}((-x)^{-n}+n^2x^{n-1}+\ldots+n(n!)(-x)+n!)$

In [38]:
```
import numpy as np
from scipy.optimize import root_scalar

# Define Laguerre polynomials as lambda functions
L_2 = lambda x: (1/2) * (x**2 - 4*x + 2)
L_3 = lambda x: (1/6) * (-x**3 + 9*x**2 - 18*x + 6)
L_4 = lambda x: (1/24) * (x**4 - 16*x**3 + 72*x**2 - 96*x + 6)
L_5 = lambda x: (1/120) * (x**5 - 25*x**4 + 200*x**3 - 600*x**2 + 600*x -

# Define their derivatives
L_2_deriv = lambda x: (1/2) * (2*x - 4)
L_3_deriv = lambda x: (1/6) * (-3*x**2 + 18*x - 18)
L_4_deriv = lambda x: (1/24) * (4*x**3 - 48*x**2 + 144*x - 96)
L_5_deriv = lambda x: (1/120) * (5*x**4 - 100*x**3 + 600*x**2 - 1200*x +

# List of polynomials and their derivatives
polynomials = [(L_2, L_2_deriv), (L_3, L_3_deriv), (L_4, L_4_deriv), (L_5
```

```
# Finding roots using SciPy's root_scalar method
roots = {}
for i, (L, L_deriv) in enumerate(polynomials, start=2):
    roots[i] = []
    # Use multiple starting points to find all roots
    for x0 in np.linspace(0, i, i+1):
        sol = root_scalar(L, fprime=L_deriv, x0=x0, x1=x0+0.5, method='ne
        if sol.converged:
            root = np.round(sol.root, 10)  # Round to 10 decimal places
            if root not in roots[i]:  # Avoid duplicates
                roots[i].append(root)

# Display results
for n, r in roots.items():
    print(f"Roots of L_{n}(x): {sorted(r)}")
```

```
Roots of L_2(x): [0.5857864376]
Roots of L_3(x): [0.4157745568, 2.2942803603]
Roots of L_4(x): [0.0656892608, 2.320392921, 4.1666859655, 9.4472318527]
Roots of L_5(x): [0.2635603197, 1.4134030591, 3.596425771]
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pac
kages/scipy/optimize/_root_scalar.py:315: RuntimeWarning: Derivative was z
ero.
  r, sol = methodc(f, x0, args=args, fprime=fprime, fprime2=None,
```

28. Find approximations to within 10^{-10} to all zeros of each of the following Legendre polynomials using Müller's and Lagurre's method.

n	$L_n(x)$
0	1
1	x
2	$-\frac{1}{3} + x^2$
3	$-\frac{3}{5}x + x^3$
4	$\frac{3}{35} - \frac{6}{7}x^2 + x^4$
5	$\frac{5}{21}x - \frac{10}{9}x^3 + x^5$
6	$-\frac{5}{231} + \frac{5}{11}x^2 - \frac{15}{11}x^4 + x^6$
7	$-\frac{35}{429}x + \frac{105}{143}x^3 - \frac{21}{13}x^5 + x^7$
8	$\frac{7}{1287} - \frac{28}{143}x^2 + \frac{14}{13}x^4 - \frac{28}{15}x^6 + x^8$
9	$\frac{63}{2431}x - \frac{84}{221}x^3 + \frac{126}{85}x^5 - \frac{36}{17}x^7 + x^9$
10	$-\frac{63}{46189} + \frac{315}{4199}x^2 - \frac{210}{323}x^4 + \frac{630}{323}x^6 - \frac{45}{19}x^8 + x^{10}$

In [39]:
```
import numpy as np
import scipy.special as sp
```

```python
def legendre_polynomial(n):
    coeffs = {
        0: [1],
        1: [1, 0],
        2: [-1/3, 0, 1],
        3: [0, -3/5, 0, 1],
        4: [3/35, 0, -6/7, 0, 1],
        5: [0, 5/21, 0, -10/9, 0, 1],
        6: [-5/231, 0, 5/11, 0, -15/11, 0, 1],
        7: [0, -35/429, 0, 105/143, 0, -21/13, 0, 1],
        8: [7/1287, 0, -28/143, 0, 14/13, 0, -28/15, 0, 1],
        9: [0, 63/2431, 0, -84/221, 0, 126/85, 0, -36/17, 0, 1],
        10: [-63/46189, 0, 315/4199, 0, -210/323, 0, 630/323, 0, -45/19,
    }

def find_legendre_roots(n, tol=1e-10):
    """Find roots of Legendre polynomial L_n(x) with given tolerance."""
    # Get the Legendre polynomial coefficients
    coeffs = sp.legendre(n).coef[::-1]  # Reverse order for NumPy

    # Find the roots using NumPy's root-finding algorithm
    roots = np.roots(coeffs)

    # Filter and sort real roots within the tolerance
    real_roots = np.sort(roots[np.abs(roots.imag) < tol].real)

    return real_roots

# Compute and print roots for L_n(x), 2 ≤ n ≤ 10
for n in range(2, 11):
    roots = find_legendre_roots(n)
    print(f"Roots of L_{n}(x): {roots}")
```

```
Roots of L_2(x): [-1.73205081  1.73205081]
Roots of L_3(x): [-1.29099445  1.29099445]
Roots of L_4(x): [-2.94134046 -1.16125634  1.16125634  2.94134046]
Roots of L_5(x): [-1.85711605 -1.1035337   1.1035337   1.85711605]
Roots of L_6(x): [-4.19077785 -1.51238022 -1.07242112  1.07242112  1.51238
022  4.19077785]
Roots of L_7(x): [-2.46399396 -1.34856095 -1.05362097  1.05362097  1.34856
095  2.46399396]
Roots of L_8(x): [-5.45153296 -1.90283222 -1.25523042 -1.04135225  1.04135
225  1.25523042
  1.90283222  5.45153296]
Roots of L_9(x): [-3.08400753 -1.63033351 -1.19612774 -1.03288687  1.03288
687  1.19612774
  1.63033351  3.08400753]
Roots of L_10(x): [-6.71707432 -2.30736185 -1.47186623 -1.15598468 -1.0267
9258  1.02679258
  1.15598468  1.47186623  2.30736185  6.71707432]
```

29. Find approximations to within 10^{-10} to all the zeros of each of the following probabilist's Hermite polynomials using Müller's and Lagurre's method.

n	$H_{e_n}(x)$
0	1
1	x

n	$H_{e_n}(x)$
2	$x^2 - 1$
3	$x^3 - 3x$
4	$x^4 - 6x^2 - 3$
5	$x^5 - 10x^3 + 15x$
6	$x^6 - 15x^4 + 45x^2 - 15$
7	$x^7 - 21x^5 + 105x^3 - 105x$
8	$x^8 - 28x^6 + 210x^4 - 420x^2 + 105$
9	$x^9 - 36x^7 + 378x^5 - 1260x^3 + 945x$
10	$x^{10} - 45x^8 + 630x^6 - 3150x^4 + 4725x^2 - 945$

In [40]:
```python
import numpy as np
import scipy.optimize as opt
import scipy.special as sp

def hermite_polynomial(n):
    """Returns the coefficients of the probabilist's Hermite polynomial H
    coeffs = {
        0: [1],
        1: [1, 0],
        2: [1, 0, -1],
        3: [1, 0, -3, 0],
        4: [1, 0, -6, 0, -3],
        5: [1, 0, -10, 0, 15, 0],
        6: [1, 0, -15, 0, 45, 0, -15],
        7: [1, 0, -21, 0, 105, 0, -105, 0],
        8: [1, 0, -28, 0, 210, 0, -420, 0, 105],
        9: [1, 0, -36, 0, 378, 0, -1260, 0, 945, 0],
        10: [1, 0, -45, 0, 630, 0, -3150, 0, 4725, 0, -945]
    }
    return np.poly1d(coeffs[n])

def find_roots(n):
    """Find roots of the probabilist's Hermite polynomial H_n(x)."""
    # Get the coefficients of the probabilist's Hermite polynomial
    coeffs = sp.hermite(n, monic=False).coef[::-1]  # Reverse order for N

    # Construct the polynomial and its derivative
    poly = np.poly1d(coeffs)
    poly_deriv = poly.deriv()

    # Use eigenvalues of the companion matrix (more robust than Newton's
    roots = np.roots(coeffs)

    # Sort and return real roots
    return np.sort(roots.real)

# Compute and print roots for H_n(x), 2 ≤ n ≤ 10
for n in range(2, 11):
    roots = find_roots(n)
    print(f"Roots of H_{n}(x): {roots}")
```

```
Roots of H_2(x): [-1.41421356  1.41421356]
Roots of H_3(x): [-0.81649658  0.81649658]
Roots of H_4(x): [-1.90604123 -0.60581089  0.60581089  1.90604123]
Roots of H_5(x): [-1.04321795 -0.49500469  0.49500469  1.04321795]
Roots of H_6(x): [-2.29317083 -0.74858756 -0.4254224   0.4254224   0.74858
756  2.29317083]
Roots of H_7(x): [-1.225058   -0.59753161 -0.3770794   0.3770794   0.59753
161  1.225058  ]
Roots of H_8(x): [-2.62338439 -0.86415955 -0.50462826 -0.3412227   0.34122
27   0.50462826
  0.86415955  2.62338439]
Roots of H_9(x): [-1.38207255 -0.68094226 -0.44119323 -0.31338205  0.31338
205  0.44119323
  0.68094226  1.38207255]
Roots of H_10(x): [-2.91629084 -0.96468218 -0.56925446 -0.39483061 -0.2910
2261  0.29102261
  0.39483061  0.56925446  0.96468218  2.91629084]
```

24. Use each of the following methods to find a solution in $[0.1, 1]$ accurate to within 10^{-4} for

$$600x^4 - 550x^3 + 200x^2 - 20x - 1 = 0.$$

a. Bisection method

b. Newton's method

c. Secant method

d. method of False Position

e. Müller's method.

In [41]:
```python
import numpy as np

# Define the function
def f(x):
    return 600*x**4 - 550*x**3 + 200*x**2 - 20*x - 1

# Define the derivative of the function for Newton's method
def df(x):
    return 2400*x**3 - 1650*x**2 + 400*x - 20

# Bisection Method
def bisection(f, a, b, tol=1e-4, max_iter=100):
    if f(a) * f(b) > 0:
        raise ValueError("Function values at the endpoints must have oppo
    for _ in range(max_iter):
        c = (a + b) / 2
        if abs(f(c)) < tol:
            return c
        elif f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return (a + b) / 2

# Newton's Method
def newton(f, df, x0, tol=1e-4, max_iter=100):
    for _ in range(max_iter):
        x1 = x0 - f(x0) / df(x0)
```

```
        if abs(x1 - x0) < tol:
            return x1
        x0 = x1
    return x0

# Secant Method
def secant(f, x0, x1, tol=1e-4, max_iter=100):
    for _ in range(max_iter):
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        if abs(x2 - x1) < tol:
            return x2
        x0, x1 = x1, x2
    return x1

# Method of False Position (Regula Falsi)
def false_position(f, a, b, tol=1e-4, max_iter=100):
    if f(a) * f(b) > 0:
        raise ValueError("Function values at the endpoints must have oppo
    for _ in range(max_iter):
        c = (a * f(b) - b * f(a)) / (f(b) - f(a))
        if abs(f(c)) < tol:
            return c
        elif f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return (a + b) / 2

# Müller's Method
def mullers_method(f, x0, x1, x2, tol=1e-4, max_iter=100):
    for _ in range(max_iter):
        f0, f1, f2 = f(x0), f(x1), f(x2)
        h0, h1 = x1 - x0, x2 - x1
        delta0 = (f1 - f0) / h0
        delta1 = (f2 - f1) / h1
        d = (delta1 - delta0) / (h1 + h0)

        # Calculate the discriminant and check if it is positive
        discriminant = delta1**2 - 4*f2*d

        if discriminant < 0:
            print("Discriminant is negative. Skipping iteration.")
            break  # Skip this iteration or handle it differently

        # Two possible roots
        discriminant_sqrt = np.sqrt(discriminant)

        root1 = x2 - (2*f2 / (delta1 + discriminant_sqrt))
        root2 = x2 - (2*f2 / (delta1 - discriminant_sqrt))

        if abs(f(root1)) < abs(f(root2)):
            x2 = root1
        else:
            x2 = root2

        if abs(f(x2)) < tol:
            return x2
    return x2

# Initial guesses
```

```
a, b = 0.1, 1
x0, x1 = 0.5, 0.7
x2 = 0.8

# Solving with different methods
bisection_root = bisection(f, a, b)
newton_root = newton(f, df, 0.5)
secant_root = secant(f, 0.5, 0.6)
false_position_root = false_position(f, a, b)
muller_root = mullers_method(f, 0.1, 0.5, 0.9)

# Print results
print("a. Bisection Method")
print(f"\tBisection Method Root: {bisection_root}\n")
print("b. Newton's Method")
print(f"\tNewton's Method Root: {newton_root}\n")
print("c. Secant Method")
print(f"\tSecant Method Root: {secant_root}\n")
print("d. False Position Method")
print(f"\tFalse Position Method Root: {false_position_root}\n")
print("e. Muller's Method")
print(f"\tMuller's Method Root: {muller_root}\n")
```

```
Discriminant is negative. Skipping iteration.
a. Bisection Method
        Bisection Method Root: 0.23235778808593754

b. Newton's Method
        Newton's Method Root: 0.23235296476876366

c. Secant Method
        Secant Method Root: 0.23235296984022955

d. False Position Method
        False Position Method Root: 0.6156388015581876

e. Muller's Method
        Muller's Method Root: 0.9
```

30. Find approximations to within 10^{-10} to all the zeros of each of the following probabilist's Hermite polynomials using Müller's and Lagurre's method.

n	$H_{e_n}(x)$
0	1
1	$2x$
2	$4x^2 - 2$
3	$8x^3 - 12x$
4	$16x^4 - 48x^2 + 12$
5	$32x^5 - 160x^3 + 120x$
6	$64x^6 - 480x^4 + 720x^2 - 120$
7	$128x^7 - 1344x^5 + 3360x^3 - 1680x$

n	$H_{e_n}(x)$
8	$256x^8 - 3584x^6 + 13440x^4 - 13440x^2 + 1680$
9	$512x^9 - 9216x^7 + 48384x^5 - 80640x^3 + 30240x$
10	$1024x^{10} - 23040x^8 + 161280x^6 - 403200x^4 + 302400x^2 - 30240$

In [42]:
```python
import numpy as np
import scipy.optimize as opt
from numpy.polynomial.hermite import Hermite

# Define the Hermite polynomials based on the given expressions
def hermite_poly(n):
    coefficients = {
        0: [1],
        1: [2, 0],
        2: [4, 0, -2],
        3: [8, 0, -12, 0],
        4: [16, 0, -48, 0, 12],
        5: [32, 0, -160, 0, 120, 0],
        6: [64, 0, -480, 0, 720, 0, -120],
        7: [128, 0, -1344, 0, 3360, 0, -1680, 0],
        8: [256, 0, -3584, 0, 13440, 0, -13440, 0, 1680],
        9: [512, 0, -9216, 0, 48384, 0, -80640, 0, 30240, 0],
        10: [1024, 0, -23040, 0, 161280, 0, -403200, 0, 302400, 0, -30240
    }
    return np.poly1d(coefficients[n][::-1])  # Convert to numpy poly1d

# Function to find roots of a given polynomial
def find_roots(n):
    poly = hermite_poly(n)
    initial_guesses = np.linspace(-np.sqrt(n) * 2, np.sqrt(n) * 2, n)  #
    roots = []

    for i in range(n - 1):
        try:
            root = opt.root_scalar(poly, bracket=[initial_guesses[i], ini
            if not any(np.isclose(root, r, atol=1e-10) for r in roots):
                roots.append(root)
        except ValueError:
            continue  # Skip if the bracket does not contain a root

    roots.sort()
    return np.array(roots)

# Compute and print the roots for n = 2 to 10
for n in range(2, 11):
    roots = find_roots(n)
    print(f"Roots of H_{n}(x): {roots}")
```

```
Roots of H_2(x): []
Roots of H_3(x): [-0.81649658  0.81649658]
Roots of H_4(x): [-1.90604123  1.90604123]
Roots of H_5(x): []
Roots of H_6(x): [-2.29317083  2.29317083]
Roots of H_7(x): [-1.225058  1.225058]
Roots of H_8(x): [-2.62338439 -0.86415955  0.86415955  2.62338439]
Roots of H_9(x): []
Roots of H_10(x): [-2.91629084 -0.96468218  0.96468218  2.91629084]
```