Function Library

Assignment 1

1. Checks for numbers, natural numbers.

2. Also Sum of natural number, odd numbers.

3. Sum of AP, GP, HP

4. factorial

5. sine and exponential function

In [1]:

```
import math
import matplotlib.pyplot as plt


# checking for validity of input - whether it is numeric value and greater than 0 or not
# also checks for 0 separately
# returns after conversion to int datatype

def check_natural_number(n):
    try :
        int(n)
        res = True
    except :
        res = False

    if res==True and int(n)>0:
        return(int(n))
    else:
        # printing error message
        print("Invalid input. Please enter a natural number.")
        return 'F'
        # returning 'F' for False value. Could have used False expression itself,
        # but that will be taken as 0 in binary which will confuse with the actual number 0



# checking for validity of input - whether it is float number or not
# returns after conversion to float datatype

def check_number(n):
    try :
        float(n)
        res = True
    except :
        res = False

    if res==True:
        return(float(n))
    else:
        # printing error message
        print("Invalid input. Please enter a number.")
```

```
        return 'F'
        # returning 'F' for False value. Could have used False expression itself,
        # but that will be taken as 0 in binary which will confuse with the actual number 0
```

In [2]:

```
# function for sum of first n natural numbers

def sum_natural_numbers(n):
    sum=0
    for i in range(n+1):
        sum=sum+i
    print("The sum of first " + str(n) + " odd numbers is " + str(sum))



# function for sum of first n odd numbers

def sum_odd_numbers(n):
    sum=0
    for i in range(n):
        sum=sum + 2*i+1
    print("The sum of first " + str(n) + " odd numbers is " + str(sum))



# function for sum of n terms of an AP
# with first term and number of terms taken as input

def sum_AP(a,n,d=1.5):
    sum=0
    for i in range(n):
        sum=sum+a
        a=a+d
    print("\nThe sum of first " + str(n) + " terms of an AP is " + str(sum))



# function for sum of n terms of a GP
# with first term and number of terms taken as input

def sum_GP(a,n,r=0.5):
    sum=0
    for i in range(n):
        sum=sum+a
        a=a*r
    print("\nThe sum of first " + str(n) + " terms of an GP is " + str(sum))
```

```python
# function for sum of n terms of a HP
# with first term and number of terms taken as input

def sum_HP(a,n,d=1.5):
    sum=0
    for i in range(n):
        sum=sum+1/a
        a=a+d
    print("\nThe sum of first " + str(n) + " terms of an HP is " + str(sum))
```

In [3]:

```
# function for finding factorial of a number

def FACTORIAL(n):
    fact=1
    while n>0:
        fact=fact*n
        n-=1
    return fact




# sine function
# with argument of sine, and number of terms in its taylor expansion taken as input

def SINE(x,n):
    sum=0
    for i in range(n):  # starting the index with i=1 because factorial of -1 is not defined
        d=(-1)**(i) * x**(2*i+1)/FACTORIAL(2*i+1)  # taylor expansion terms
        sum=sum+d
    return sum




# exponential function
# with argument of sine and number of terms in its taylor expansion taken as input

def EXP(x,n):
    sum=0
    for i in range(0,n):
        d=(-1)**i * x**i/FACTORIAL(i)  # taylor expansion terms
        sum=sum+d
    return sum
```

Assignment 2

1. **printing a matrix, multiplying two matrices, transpose of a matrix**

2. **sum, difference, product, division, conjugate, modulud, phase of complex numbers - Using class**

3. **avg disatnce of n points in a 1D array**

4. **HANGMAN game**

In [4]:

```
# Matrix algebra

import numpy as np
def print_matrix(A,r,c):
    print()
    for i in range(r):
        for j in range(c):
            # prints the matrix with appropriate spaces for easy understanding
            print(A[i][j], end='    ')
        print("\n")
    print()



def matrix_multiply(A,r1,c1,B,r2,c2):
    if c1==r2: # checking compatibility
        C=[[0 for i in range(c2)] for j in range(r1)] # initializing matrix C
        for i in range(r1):
            for j in range(c2):
                for k in range(c2):
                    C[i][j]+=float(A[i][k])*float(B[k][j]) # multiplication algorithm
        return C,r1,c2
    else:
        print("matrices incompatible for multiplication")



def transpose_matrix(A,r,c):
    B = [[0 for x in range(r)] for y in range(c)]
    for i in range(r):
        for j in range(c):
            B[j][i]=A[i][j]
    return B,c,r
```

In [5]:

```
# Class for complex number algebra

import math
class myComplex:
    def __init__(self, a1, b1, a2, b2): # initializing class variables
        self.a1=a1
        self.b1=b1
        self.a2=a2
        self.b2=b2

    # Sum of two complex numbers
    def sum_complex(self,a1,b1,a2,b2):
        return self.a1+self.a2, self.b1+self.b2

    # Difference of two complex numbers
    def difference_complex(self,a1,b1,a2,b2):
        return self.a1-self.a2, self.b1-self.b2

    # Product of two complex numbers
    def product_complex(self,a1,b1,a2,b2):
        return self.a1*self.a2-self.b1*self.b2, self.a1*self.b2+self.b1*self.a2

    # complex conjugate of a complex number
    def conjugate_complex(self,a3,b3):
        self.a3=a3
        self.b3=b3
        return self.a3, -1*self.b3

    # Modulus of a complex number
    def modulus_complex(self,a4,b4):
        self.a4=a4
        self.b4=b4
        return math.sqrt(self.a4**2 + self.b4**2)

    # Division of two complex numbers
    def divide_complex(self,a1,b1,a2,b2):
        if a2==0 and b2==0:
            print("Division by zero is invalid")
        else:
            a,b=mc.conjugate_complex(self.a2,self.b2)
            p,q=mc.product_complex(self.a1,self.b1,a,b)
```

```
            return p/(mc.modulus_complex(a,b))**2, q/(mc.modulus_complex(a,b))**2

    # Phase angle of a complex number in degrees
    def phase_complex(self,a5,b5):
        self.a5=a5
        self.b5=b5
        return 180*math.atan(self.b5/self.a5)/(math.pi)
```

In [6]:

```
# To calculate average distance for n-point 1D grid

def calculate_avg_dist(k):
    dist=0
    for i in range(k):
        for j in range(k):
            dist+=abs(i-j)
    return dist/k**2
```

In [7]:

```
# To generate list of country and captital for hangman game

import random

def generate_word():
    countries=['Argentina', 'Australia', 'Brazil', 'Cameroon', 'Canada', 'Chile', 'China', 'England',
          'France', 'Germany', 'Italy', 'Jamaica', 'Japan', 'Netherlands', 'New Zealand', 'Nigeria',
          'Norway', 'Scotland', 'South Africa', 'South Korea', 'Spain', 'Sweden', 'Thailand', 'United States']
    capitals=['buenosaires', 'canberra', 'brasilia', 'yaounde', 'ottawa', 'santiago', 'beijing', 'london',
          'paris', 'berlin', 'rome', 'kingston', 'tokyo', 'amsterdam', 'wellington', 'abuja',
          'oslo', 'edinburgh', 'capetown', 'seoul', 'madrid', 'stockholm', 'bangkok', 'washington']
    val=random.randrange(0,24)
    return countries[val], capitals[val]
```

Assignment 3

1. Read matrix from a file, Round a number, round elements of a matrix

2. Swap rows, partial pivot, Gauss jordan, inverse using gauss-jordan. (gauss jordan also gives the determinant)

In [8]:

```
#function for reading the matrix

def read_matrix(txt):
    with open(txt, 'r') as a:
        matrix=[[float(num) for num in row.split(' ')] for row in a ]
    row=len(matrix)
    column=len(matrix[0])
    return matrix, row, column



# Round function

def round_half_up(n, decimals=0):
    multiplier = 10 ** decimals
    return math.floor(n*multiplier + 0.5) / multiplier

def ROUND(n, decimals=10):
    rounded_abs = round_half_up(abs(n), decimals)
    if n>0:
        return rounded_abs
    elif n<0:
        return(-1)*rounded_abs
    else:
        return 0



# Function to round off all elements of a matrix

def round_matrix(M):
    for i in range(len(M)):
        for j in range(len(M[0])):
            M[i][j]=ROUND(M[i][j],2)
    return M
```

In [9]:

```
# function to swap row1 and row2

def swap_rows(Ab,row1,row2):
    temp = Ab[row1]
    Ab[row1] = Ab[row2]
    Ab[row2] = temp
    return Ab



# Function for partial pivoting

def partial_pivot(Ab,m,nrows):
    pivot = Ab[m][m]    # declaring the pivot
    if (Ab[m][m] != 0):
        return Ab    # return if partial pivot is not required
    else:
        for r in range(m+1,nrows):
            pivot=Ab[r][m]
            # check for non-zero pivot and swap rows with it
            for k in range(m+1,nrows):
                if abs(Ab[k][m])>pivot:
                    pivot=Ab[k][m]
                    r=k
            if Ab[r][m] != 0:
                pivot = Ab[r][m]
                Ab=swap_rows(Ab,m,r)
                return Ab
            else:
                r+=1
    if (pivot==0):    # no unique solution case
        return None



# Gauss Jordan Elimiination method

def gauss_jordan(Ab,nrows,ncols):
    det=1
    r=0
    # does partial pivoting
```

```
    Ab = partial_pivot(Ab,r,nrows)
    for r in range(0,nrows):
        # no solution case
        if Ab==None:
            return Ab
        else:
            # Changes the diagonal elements to unity
            fact=Ab[r][r]
            if fact==0:
                # does partial pivoting
                Ab = partial_pivot(Ab,r,nrows)
            fact=Ab[r][r]
            det=det*fact # calculates the determinant
            for c in range(r,ncols):
                Ab[r][c]*=1/fact
            # Changes the off-diagonal elements to zero
            for r1 in range(0,nrows):
                # does not change if it is already done
                if (r1==r or Ab[r1][r]==0):
                    r1+=1
                else:
                    factor = Ab[r1][r]
                    for c in range(r,ncols):
                        Ab[r1][c]-= factor * Ab[r][c]
    return Ab, det



# Function to extract inverse from augmented matrix

def get_inv(A,n):
    r=len(A)
    c=len(A[0])
    M=[[0 for j in range(n)] for i in range(n)]
    for i in range(r):
        for j in range(n,c):
            M[i][j-n]=A[i][j]
    return M
```

Supplementary - assignment 3

Gauss jordan with details on what is happening at which step, by printing matrix after each operation

In [10]:

```
# Function for reference to know what happens at each step

def gauss_jordan_steps(Ab,nrows,ncols):
    # does partial pivoting
    det=1
    r=0
    Ab = partial_pivot(Ab,r,nrows)
    for r in range(0,nrows):
        # no solution case
        if Ab==None:
            return Ab
        else:
            # Changes the diagonal elements to unity
            print("value of  r  =  "+str(r))
            print_matrix(Ab,nrows,ncols)
            fact=Ab[r][r]
            if fact==0:
                # does partial pivoting
                Ab = partial_pivot(Ab,r,nrows)
            fact=Ab[r][r]
            print("changing values of diagonal")
            det=det*fact # calculates the determinant
            for c in range(r,ncols):
                print("fact value  =  "+str(fact))
                Ab[r][c]*=1/fact
                print_matrix(Ab,nrows,ncols)
                print("loop -> value of  c  =  "+str(c))
            # Changes the off-diagonal elements to zero
            print("Now changing values other than diagonal")
            for r1 in range(0,nrows):
                # does not change if it is already done
                print("loop -> value of  r1  =  "+str(r1)+"  when  r  =  "+str(r))
                if (r1==r or Ab[r1][r]==0):
                    r1+=1
                else:
                    factor = Ab[r1][r]
                    for c in range(r,ncols):
                        Ab[r1][c]-= factor * Ab[r][c]
                print_matrix(Ab,nrows,ncols)
    return Ab, det
```

Assignment 4

1. identity matrix, partial pivot for LU decomp, determinant using LU and check positive definite

2. LU doolittle, forward backward substitution doolittle

3. LU crout, forward backward substitution crout

4. inverse using LU doolittle decomposition

5. LU cholesky, forward backward substitution cholesky

In [11]:

```
import copy

# Function for partial pivot for LU decomposition

def partial_pivot_LU (mat, vec, n):
    for i in range (n-1):
        if mat[i][i] ==0:
            for j in range (i+1,n):
                # checks for max absolute value and swaps rows
                # of both the input matrix and the vector as well
                if abs(mat[j][i]) > abs(mat[i][i]):
                    mat[i], mat[j] = mat[j], mat[i]
                    vec[i], vec[j] = vec[j], vec[i]
    return mat, vec



# Function to calculate the determinant of a matrix
# via product of transformed L or U matrix

def determinant(mat,n):
    det=1
    for i in range(n):
        det*=-1*mat[i][i]
    return det



# Function to produce n x n identity matrix

def get_identity(n):
    I=[[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        I[i][i]=1
    return I



# Function for checking hermitian matrix for cholesky decomposition

def check_positive_definite(mat):
```

```
    l=0
    n=len(mat)
    for i in range(n):
        for j in range(n):
            if mat[i][j]==mat[j][i]:
                l+=1
    if l==n**2:
        return(True)
    else:
        return(False)
```

In [12]:

```
# LU decomposition using Doolittle's condition L[i][i]=1
# without making separate L and U matrices

def LU_doolittle(mat,n):
    for i in range(n):
        for j in range(n):
            if i>0 and i<=j: # changing values of upper triangular matrix
                sum=0
                for k in range(i):
                    sum+=mat[i][k]*mat[k][j]
                mat[i][j]=mat[i][j]-sum
            if i>j: # changing values of lower triangular matrix
                sum=0
                for k in range(j):
                    sum+=mat[i][k]*mat[k][j]
                mat[i][j]=(mat[i][j]-sum)/mat[j][j]
    return mat



# Function to find the solution matrix provided a vector using
# forward and backward substitution respectively

def for_back_subs_doolittle(mat,n,vect):
    # initialization
    y=[0 for i in range(n)]

    # forward substitution
    y[0]=vect[0]
    for i in range(n):
        sum=0
        for j in range(i):
            sum+=mat[i][j]*y[j]
        y[i]=vect[i]-sum

    # backward substitution
    x[n-1]=y[n-1]/mat[n-1][n-1]
    for i in range(n-1,-1,-1):
        sum=0
        for j in range(i+1,n):
            sum+=mat[i][j]*x[j]
```

```
        x[i]=(y[i]-sum)/mat[i][i]
    del(y)
    return x
```

In [13]:

```
# LU decomposition using Crout's condition U[i][i]=1
# without making separate L and U matrices

def LU_crout(mat,n):
    for i in range(n):
        for j in range(n):
            if i>=j: # changing values of lower triangular matrix
                sum=0
                for k in range(j):
                    sum+=mat[i][k]*mat[k][j]
                mat[i][j]=mat[i][j]-sum
            if i<j: # changing values of uppr triangular matrix
                sum=0
                for k in range(i):
                    sum+=mat[i][k]*mat[k][j]
                mat[i][j]=(mat[i][j]-sum)/mat[i][i]
    return mat



# Function to find the solution matrix provided a vector using
# forward and backward substitution respectively

def for_back_subs_crout(mat,n,vect):
    y=[0 for i in range(n)]

    # forward substitution
    y[0]=vect[0]/mat[0][0]
    for i in range(n):
        sum=0
        for j in range(i):
            sum+=mat[i][j]*y[j]
        y[i]=(vect[i]-sum)/mat[i][i]

    # backward substitution
    x[n-1]=y[n-1]
    for i in range(n-1,-1,-1):
        sum=0
        for j in range(i+1,n):
            sum+=mat[i][j]*x[j]
        x[i]=y[i]-sum
```

```
    del(y)
    return x
```

In [14]:

```
def inverse_by_lu_decomposition (matrix, n):

    identity=get_identity(ro)
    x=[]

    '''
    The inverse finding process could have been done using
    a loop for the four columns. But while partial pivoting,
    the rows of final inverse matrix and the vector both are
    also interchanged. So it is done manually for each row and vector.

    deepcopy() is used so that the original matrix doesn't change on
    changing the copied entities. We reuire the original multiple times here

    1. First the matrix is deepcopied.
    2. Then partial pivoting is done for both matrix and vector.
    3. Then the decomposition algorithm is applied.
    4. Then solution is obtained.
    5. And finally it is appended to a separate matrix to get the inverse.
    Note: The final answer is also deepcopied because there is some error
        due to which all x0, x1, x2 and x3 are also getting falsely appended.
    '''

    matrix_0 = copy.deepcopy(matrix)
    partial_pivot_LU(matrix_0, identity[0], n)
    matrix_0 = LU_doolittle(matrix_0, n)
    x0 = for_back_subs_doolittle(matrix_0, n, identity[0])
    x.append(copy.deepcopy(x0))


    matrix_1 = copy.deepcopy(matrix)
    partial_pivot_LU(matrix_1, identity[1], n)
    matrix_1 = LU_doolittle(matrix_1, n)
    x1 = for_back_subs_doolittle(matrix_1, n, identity[1])
    x.append(copy.deepcopy(x1))

    matrix_2 = copy.deepcopy(matrix)
    partial_pivot_LU(matrix_2, identity[2], n)
    matrix_2 = LU_doolittle(matrix_2, n)
    x2 = for_back_subs_doolittle(matrix_2, n, identity[2])
    x.append(copy.deepcopy(x2))
```

```
    matrix_3 = copy.deepcopy(matrix)
    partial_pivot_LU(matrix_3, identity[3], n)
    matrix_3 = LU_doolittle(matrix_3, n)
    x3 = for_back_subs_doolittle(matrix_3, n, identity[3])
    x.append(copy.deepcopy(x3))

    # The x matrix to be transposed to get the inverse in desired form
    inverse,r,c=transpose_matrix(x,n,n)
    return (inverse)
```

In [15]:

```
# Function for Cholesky decomposition
# Only works for Hermitian and positive definite matrices
# In this case, we use real matrices only

def LU_cho(mat,n):
    if check_positive_definite(mat)==True:
        for i in range(n):
            for j in range(i,n):
                if i==j: # changing diagonal elements
                    sum=0
                    for k in range(i):
                        sum+=mat[i][k]**2
                    mat[i][i]=math.sqrt(mat[i][i]-sum)
                if i<j: # changing upper traiangular matrix
                    sum=0
                    for k in range(i):
                        sum+=mat[i][k]*mat[k][j]
                    mat[i][j]=(mat[i][j]-sum)/mat[i][i]

                    # setting the lower triangular elements same as elements at the transposition
                    mat[j][i]=mat[i][j]
        return mat
    else:
        print("Given matrix is not hermitian, cholesky method cannot be applied.")
        return False



# Function to find the solution matrix provided a vector using
# forward and backward substitution respectively

def for_back_subs_cho(mat,n,vect):
    y=[0 for i in range(n)]

    # forward substitution
    y[0]=vect[0]/mat[0][0]
    for i in range(n):
        sum=0
        for j in range(i):
            sum+=mat[i][j]*y[j]
        y[i]=(vect[i]-sum)/mat[i][i]
```

```
    # forward substitution
    x[n-1]=y[n-1]
    for i in range(n-1,-1,-1):
        sum=0
        for j in range(i+1,n):
            sum+=mat[i][j]*x[j]
        x[i]=(y[i]-sum)/mat[i][i]
    del(y)
    return x
```

Supplementary - assignment 4

LU decop with separate Lower and Upper matrix

In [16]:

```
# LU decomposition using Doolittle's condition L[i][i]=1
# by making separate L and U matrices

def LU_do2(M,n):
    # initialization
    L=[[0 for j in range(n)] for i in range(n)]
    U=[[0 for j in range(n)] for i in range(n)]

    for i in range(n):
        L[i][i]=1
        for j in range(n):
            if i>j:
                U[i][j]=0
            elif i<j:
                L[i][j]=0
            U[0][j]=M[0][j]
            L[i][0]=M[i][0]/U[0][0]
            if i>0 and i<=j: # changing values for upper traiangular matrix
                sum=0
                for k in range(i):
                    sum+=L[i][k]*U[k][j]
                U[i][j]=M[i][j]-sum
            if i>j: # changing values for lower traiangular matrix
                sum=0
                for k in range(j):
                    sum+=L[i][k]*U[k][j]
                L[i][j]=(M[i][j]-sum)/U[j][j]
    print_matrix(L,n,n)
    print_matrix(U,n,n)

    # To check if the L and U matrices are correct, use this for verification
    m,r,c=matrix_multiply(L,ro,ro,U,ro,ro)
    print_matrix(m,r,c)

    return M
```

Assignment 5

1. Derivative and Double derivative

2. Bracketting, bisection, regula falsi - with separate plotting fumctions

3. Newton raphson - with separate plotting fumctions

4. Polynomial function, its derivative and double derivative

5. Synthetic deivision - deflate

6. Laqueere method for finding all roots of polynomial function

In [17]:

```
# Function for finding derivative of a function at given x

def derivative(f,x):
    h=10**-8
    fd=(f(x+h)-f(x))/h # Derivative algorithm
    return fd



# Function for finding double derivative of a function at given x

def double_derivative(f,x):
    h=10**-8
    fdd=(f(x+h)+f(x-h)-2*f(x))/(2*h) # Double derivative algorithm
    return fdd
```

In [1]:

```
# Function for bracketing the root
# the algorithm changes the intervals towards lower value among f(a) and f(b)

def bracketing(a,b,f):
    scale=0.1 # defining scaling factor for changing the interval

    while f(a)*f(b)>0:
        if abs(f(a)) <= abs(f(b)):
            a = a - scale*(b-a)
        else:
            b = b + scale*(b-a)
    return a,b



# Function for finding root using bisection method i.e. c=(a+b)/2

def bisection(a,b,f):
    # Checking if root is landed by default - really lucky
    if f(a)*f(b)==0.0:
        if f(a)==0.0:
            return a
        else:
            return b

    c=(a+b)/2
    while (b-a)/2>eps: # checking if the accuracy is achieved
        c=(a+b)/2
        if (f(a)*f(c))<=0.0: # Check if the root is properly bracketted
            b=c
        else:
            a=c
    return (a+b)/2



# Same bisection function but this gives arrays instead of roots for plotting purpose

def bisection_for_plotting(a,b,f):
    loop_count=[]
    lc=0
```

```
    loop_value=[]
    root_conv=[]

    # Checking if root is landed by default - really lucky
    if f(a)*f(b)==0:
        lc+=1
        loop_count.append(lc)
        loop_value.append(eps)
        root_conv.append(eps)
        if f(a)==0:
            return a
        else:
            return b

    c=(a+b)/2
    while (b-a)/2>eps: # checking if the accuracy is achieved
        lc+=1
        c=(a+b)/2
        if (f(a)*f(c))<=0: # Check if the root is properly bracketted
            b=c
        else:
            a=c
        loop_count.append(lc)
        root_conv.append((b+a)/2)
        loop_value.append(f((b+a)/2))
    return loop_count, loop_value, root_conv



# Function for finding root using regula-falsi method i.e. c=b-(b-a)*f(b)/(f(b)-f(a))

def regula_falsi(a,b,f):
    # Checking if root is landed by default - really lucky
    if f(a)*f(b)==0:
        if f(a)==0:
            return a
        else:
            return b

    c=(b-a)/2
    cn=b-a
    while abs(c-cn)>eps: # checking if the accuracy is achieved
        cn=c
```

```python
        c=b-(b-a)*f(b)/(f(b)-f(a))
        if (f(a)*f(c))<=0: # Check if the root is properly bracketted
            b=c
        else:
            a=c
    return c



# Same regula falsi function but this gives arrays instead of roots for plotting purpose

def regula_falsi_for_plotting(a,b,f):
    loop_count=[]
    lc=0
    loop_value=[]
    root_conv=[]

    # Checking if root is landed by default - really lucky
    if f(a)*f(b)==0:
        lc+=1
        loop_count.append(lc)
        loop_value.append(eps)
        root_conv.append(eps)
        if f(a)==0:
            return a
        else:
            return b

    c=(b-a)/2
    cn=b-a
    while abs(c-cn)>eps: # checking if the accuracy is achieved
        lc+=1
        cn=c
        c=b-(b-a)*f(b)/(f(b)-f(a))
        if (f(a)*f(c))<=0: # Check if the root is properly bracketted
            b=c
        else:
            a=c
        loop_count.append(lc)
        root_conv.append(c)
        loop_value.append(f(c))
    return loop_count, loop_value, root_conv
```

```python
# Function for finding root using newton-raphson method i.e. x=x-f(x)/deriv(f,x)
# when given a guess solution x far from extrema

def newton_raphson(x,f, max_it=100):
    xn=x
    k=0
    x=x-f(x)/derivative(f,x)
    while abs(x-xn)>eps and k<max_it: # checking if the accuracy is achieved
        xn=x
        x=x-f(x)/derivative(f,x)
        k+=1
    return x



# Same newton-raphson function but this gives arrays instead of roots for plotting purpose

def newton_raphson_for_plotting(x,f, max_it=100):
    loop_count=[]
    lc=0
    k=0
    loop_value=[]
    root_conv=[]
    xn=x
    x=x-f(x)/derivative(f,x)
    while abs(x-xn)>eps and k<max_it: # checking if the accuracy is achieved
        lc+=1
        xn=x
        k+=1
        x=x-f(x)/derivative(f,x)
        loop_count.append(lc)
        root_conv.append(x)
        loop_value.append(f(x))
    return loop_count, loop_value, root_conv
```

In [19]:

```
# Functions for laguerre method


# Function to give the polynomial given coefficient array

def poly_function(A):
    def p(x):
        n=len(A)
        s=0
        for i in range(n):
            s+=A[i]*x**(n-1-i)
        return s
    return p



# Function for synthetic division - deflation
# it works simply the sythetic division way, the ouptput coefficients are stored in array C

def deflate(A, sol):
    n=len(A)
    B=[0 for i in range(n)]
    C=[0 for i in range(n-1)]
    C[0]=A[0]
    for i in range(n-1):
        B[i+1]=C[i]*sol
        if i!=n-2:
            C[i+1]=A[i+1]+B[i+1]
    return C



# Function for laguerre method of finding roots for polynomial function
# this functions works only when all roots are real.
# may give garbage values if polynomials with complex roots are taken

def laguerre(A, guess, max_iter=100):
    n = len(A)
    #define the polynomial function
    p = poly_function(A)
    #check if guess was correct
```

```
    x = guess
    if p(x) == 0:
        return x

    # defining a range for max iterations, so that it does not run into infinite loops
    # the functions here must converge in this limit, else it is not a good guess
    for i in range(max_iter):
        xn = x

        G = derivative(p,x)/p(x)
        H = G**2 - double_derivative(p,x)/p(x)
        denom1 = G+((n-2)*((n-1)*H - G**2))**0.5
        denom2 = G-((n-2)*((n-1)*H - G**2))**0.5

        #compare denominators
        if abs(denom2)>abs(denom1):
            a = (n-1)/denom2
        else:
            a = (n-1)/denom1

        x = x-a
        #check if convergence criteria satisfied
        if abs(x-xn) < eps:
            return x
    return x # Change it to return False since it would not converge



# Function to collect all the roots and deflate the polynomial

def poly_solution(A, x):
    n = len(A)
    p=poly_function(A)
    roots = []

    for i in range(n-1):
        root = laguerre(A, x)

        # newton raphson for polishing the roots
        root=newton_raphson(root,p)

        # appending the root into list
        roots.append(root)
```

```
        # deflating the polynomial by synthetic division
        A = deflate(A, root)
    return roots
```

Assignment 6

Numerical Integration

1. Mid-point method

2. Trapezoidal method

3. Simpson's method

4. Monte Carlo Integration

In [3]:

```
# Function to calculate the number of iterations which will give
# correct integration value upto eps number of decimal places

def calculate_N(fn_mp, fn_t, fn_s, eps=10**-6):
    # Calculation of N from error calculation formula
    N_mp=((b-a)**3/24/eps*fn_mp)**0.5
    N_t=((b-a)**3/12/eps*fn_t)**0.5
    N_s=((b-a)**5/180/eps*fn_s)**0.25

    # Using integral value, also handling the case where eps=0
    if N_mp==0:
        N_mp=1
    else:
        N_mp=int(N_mp)

    if N_t==0:
        N_t=1
    else:
        N_t=int(N_t)

    if N_s==0:
        N_s=1
    else:
        N_s=int(N_s)

    # Special case with simpson's rule
    # It is observed for simpson rule for even N_s, it uses same value
    # but for odd N_s, it should be 1 more else the value is coming wrong
    if N_s%2!=0:
        N_s+=1

    return N_mp, N_t, N_s



# numerical integration by mid-point method
def int_mid_point(f, a, b, n):
    s=0
    h=(b-a)/n # step size

    # integration algorithm
```

```python
    for i in range(1,n+1):
        x=a+(2*i-1)*h/2
        s+=f(x)

    return s*h



# numerical integration by Trapezoidal method
def int_trapezoidal(f, a, b, n):
    s=0
    h=(b-a)/n # step size

    # integration algorithm
    for i in range(1,n+1):
        s+=f(a+i*h)+f(a+(i-1)*h)

    return s*h/2



# numerical integration by Simpson method
def int_simpson(f, a, b, n):
    s=f(a)+f(b)
    h=(b-a)/n

    # integration algorithm
    for i in range(1,n):
        if i%2!=0:
            s+=4*f(a+i*h)
        else:
            s+=2*f(a+i*h)

    return s*h/3
```

In [57]:

```
def pdf(x):
    return 1/(b-a)


def Average(x):
    s=0
    for i in range(len(x)):
        s+=x[i]
    return s/len(x)



def int_monte_carlo(f, pdf, a, b, n):
    I=[]
    for i in range(100):
        x = np.random.uniform(low=a, high=b, size=n)
        F=0
        for i in range(len(x)):
            F+=f(x[i])/pdf(x[i])
        I.append(F/n)
    return Average(I)
```

Supplementary - Assignment 6

1. Standard deviation function

2. Monte Carlo square function (to calculate the deviations)

In [ ]:

```
def stdev_s(a):
    sig=0
    mean=0
    for i in range(len(a)):
        mean+=a[i]
    mean=mean/len(a)
    for i in range(len(a)):
        sig+=(a[i]-mean)**2
    sig=math.sqrt(sig/(len(a)-1))
    return sig



def int_monte_carlo_square(f, p, a, b, n):
    x = np.random.uniform(low=a, high=b, size=(n))
    F=0
    for i in range(len(x)):
        F+=(f(x[i]))**2/p(x[i])
    F=F/n
    return F
```

Assignment 7

Ordinary differential equations

1. Euler forward / explicit

2. RK-1 - Predictor Corrector / Trapezoidal method

3. RK-2 - Mid-point method

4. RK-4 - Runge-Kutta method - for N coupled equations

5. Runge_kutta_for_shooting, Lagrange interpolation, Shooting method - for boundary value problems

In [14]:

```
def exp_euler(x,y,h, lim, dydx):
    # Constructing solution arrays
    X = [x]
    Y = [y]
    while x <= lim:
        k1 = h* dydx(x, y) # k1 calculation
        y = y + k1
        x = x + h
        X.append(x)
        Y.append(y)
    return X, Y



def predictor_corrector(x,y,h, lim, dydx):
    # Constructing solution arrays
    X = [x]
    Y = [y]
    while x <= lim:
        k1 = h* dydx(x, y) # k1 calculation
        k = h* dydx(x+h, y+k1) # k' calculation
        y = y + (k1+k)/2
        x = x + h
        X.append(x)
        Y.append(y)
    return X, Y



def RK2(x,y,h, lim, dydx):
    # Constructing solution arrays
    X = [x]
    Y = [y]
    while x <= lim:
        k1 = h* dydx(x, y) # k1 calculation
        k2 = h* dydx(x+h/2, y+k1/2) # k2 calculation
        y = y + k2
        x = x + h
        X.append(x)
        Y.append(y)
    return X, Y
```

In [3]:

```
def RK4(x,y,p, h, l_bound, u_bound, dydx, d2ydx2):
    # p = dy/dx
    x1=x
    y1=y
    p1=p

    X=[x]
    Y=[y]
    P=[p]
    while x <= u_bound:

        # Calculation for each stepsize h
        k1 = h* dydx(x,y,p)
        l1 = h* d2ydx2(x,y,p)

        k2 = h* dydx(x+h/2, y+k1/2, p+l1/2)
        l2 = h* d2ydx2(x+h/2, y+k1/2, p+l1/2)

        k3 = h* dydx(x+h/2, y+k2/2, p+l2/2)
        l3 = h* d2ydx2(x+h/2, y+k2/2, p+l2/2)

        k4 = h* dydx(x+h, y+k3, p+l3)
        l4 = h* d2ydx2(x+h, y+k3, p+l3)

        y = y + 1/6* (k1 +2*k2 +2*k3 +k4)
        p = p + 1/6* (l1 +2*l2 +2*l3 +l4)
        x = x + h

        # Appending to arrays
        X.append(ROUND(x,8))
        Y.append(ROUND(y,8))
        P.append(ROUND(p,8))

    while x1 >= l_bound:

        # Calculation for each stepsize h
        k1 = h* dydx(x,y,p1)
        l1 = h* d2ydx2(x1,y1,p1)

        k2 = h* dydx(x1-h/2, y1-k1/2, p1-l1/2)
        l2 = h* d2ydx2(x1-h/2, y1-k1/2, p1-l1/2)
```

```
        k3 = h* dydx(x1-h/2, y1-k2/2, p1-l2/2)
        l3 = h* d2ydx2(x1-h/2, y1-k2/2, p1-l2/2)

        k4 = h* dydx(x1-h, y1-k3, p1-l3)
        l4 = h* d2ydx2(x1-h, y1-k3, p1-l3)

        y1 = y1 - 1/6* (k1 +2*k2 +2*k3 +k4)
        p1 = p1 - 1/6* (l1 +2*l2 +2*l3 +l4)
        x1 = x1-h

        # Appending to arrays
        X.append(ROUND(x1,8))
        Y.append(ROUND(y1,8))
        P.append(ROUND(p1,8))
    return X,Y,P
```

In [2]:

```
# Solves differential equation using Runge-Kutta method
def runge_kutta_for_shooting(d2ydx2, dydx, x0, y0, z0, xf, h):
    # Yields solution from x=x0 to x=xf
    # y(x0) = y0 & y'(x0) = z0

    # Creating and initialising arrays
    x = []
    x.append(x0)
    y = []
    y.append(y0)
    z = []
    z.append(z0)

    n = int((xf-x0)/h)      # no. of steps
    for i in range(n):

        x.append(x[i] + h)

        # Calculation for each stepsize h

        k1 = h * dydx(x[i], y[i], z[i])
        l1 = h * d2ydx2(x[i], y[i], z[i])

        k2 = h * dydx(x[i] + h/2, y[i] + k1/2, z[i] + l1/2)
        l2 = h * d2ydx2(x[i] + h/2, y[i] + k1/2, z[i] + l1/2)

        k3 = h * dydx(x[i] + h/2, y[i] + k2/2, z[i] + l2/2)
        l3 = h * d2ydx2(x[i] + h/2, y[i] + k2/2, z[i] + l2/2)

        k4 = h * dydx(x[i] + h, y[i] + k3, z[i] + l3)
        l4 = h * d2ydx2(x[i] + h, y[i] + k3, z[i] + l3)

        y.append(y[i] + (k1 + 2*k2 + 2*k3 + k4)/6)
        z.append(z[i] + (l1 + 2*l2 + 2*l3 + l4)/6)

    return x, y, z



# Function for Lagrange's interpolation formula
def lagrange_interpolation(chi_h, chi_l, yh, yl, y):
```

```
    chi = chi_l + (chi_h - chi_l) * (y - yl)/(yh - yl)
    return chi



# Solves 2nd order ODE with the given boundary conditions
def shooting_method(d2ydx2, dydx, x_init, y_init, x_fin, y_fin, z_guess1, z_guess2, step_size, tol=1e-6):

    x, y, z = runge_kutta_for_shooting(d2ydx2, dydx, x_init, y_init, z_guess1, x_fin, step_size)
    yn = y[-1]

    if abs(yn - y_fin) > tol:
        if yn < y_fin:
            chi_l = z_guess1
            yl = yn

            x, y, z = runge_kutta_for_shooting(d2ydx2, dydx, x_init, y_init, z_guess2, x_fin, step_size)
            yn = y[-1]

            if yn > y_fin:
                chi_h = z_guess2
                yh = yn

                # calculate chi using Lagrange interpolation
                chi = lagrange_interpolation(chi_h, chi_l, yh, yl, y_fin)

                # using this chi to solve using RK4
                x, y, z = runge_kutta_for_shooting(d2ydx2, dydx, x_init, y_init, chi, x_fin, step_size)
                return x, y, z

            else:
                print("Bracketing FAIL! Try another set of guesses.")


        elif yn > y_fin:
            chi_h = z_guess1
            yh = yn

            x, y, z = runge_kutta_for_shooting(d2ydx2, dydx, x_init, y_init, z_guess2, x_fin, step_size)
            yn = y[-1]

            if yn < y_fin:
                chi_l = z_guess2
```

```
                yl = yn

                # calculate chi using Lagrange interpolation
                chi = lagrange_interpolation(chi_h, chi_l, yh, yl, y_fin)

                x, y, z = runge_kutta_for_shooting(d2ydx2, dydx, x_init, y_init, chi, x_fin, step_size)
                return x, y, z

            else:
                print("GUESSES FAILED! Try another set.")

    else:
        return x, y, z
```

Curve Fitting (least square)

1. Line fitting

2. Polynomial fitting

In [5]:

```
# Function to import data from csv file and append to array

def get_from_csv(file):
    C=np.genfromtxt(file, delimiter=',')
    X=[]
    Y=[]
    for i in range(len(C)):
        X.append(C[i][0])
        Y.append(C[i][1])
    return X,Y



# All find statistics

def find_stats(X, Y):
    n=len(X)
    Sx=sum(X)   # Sun of all x
    Sy=sum(Y)   # Sun of all y

    x_mean=sum(X)/n    # Mean x
    y_mean=sum(Y)/n    # Mean y

    Sxx=0
    Sxy=0
    Syy=0
    for i in range(len(X)):
        Sxx += (X[i] - x_mean)**2
        Sxy += (X[i] - x_mean) * (Y[i] - y_mean)
        Syy += (Y[i] - y_mean)**2
    return n, x_mean, y_mean, Sx, Sy, Sxx, Syy, Sxy



# Function to calculate Pearson Coefficient

def Pearson_coeff(X, Y):
    S=find_stats(X,Y)
    r2 = S[7]**2 / (S[5] * S[6])
    r = r2**(0.5)
```

```
    return r
```

In [6]:

```
# solve for m and c

def Line_fit(X, Y):
    n = len(X) # or len(Y)
    xbar = sum(X)/n
    ybar = sum(Y)/n

    # Calculating numerator and denominator
    numer = sum([xi*yi for xi,yi in zip(X, Y)]) - n * xbar * ybar
    denum = sum([xi**2 for xi in X]) - n * xbar**2

    # calculation of slope and intercept
    m = numer / denum
    c = ybar - m * xbar
    return c, m



# Plotting the graph

def plot_graph_linear(X, Y, c, m):
    plt.figure(figsize=(7,5))
    # plot points and fit line
    plt.scatter(X, Y, s=50, color='blue')
    yfit = [c + m * xi for xi in X]
    plt.plot(X, yfit, 'r-', label="Best fit line")
```

In [7]:

```
# Ploynomial fit with given degree

def polynomial_fitting(X,Y, order):
    X1=copy.deepcopy(X)
    Y1=copy.deepcopy(Y)
    order+=1

    # Finding the coefficient matrix - refer notes
    A=[[0 for j in range(order)] for i in range(order)]
    vector=[0 for i in range(order)]

    for i in range(order):
        for j in range(order):
            for k in range(len(X)):
                A[i][j] += X[k]**(i+j)

    Det=determinant(A,order)
    print("Determinant is = "+ str(Det))
    if Det==0:
        print("Determinant is zero. Inverse does not exist")
    print("Determinant is not zero. Inverse exists.\n")
    # Finding the coefficient vector - refer notes
    for i in range(order):
        for k in range(len(X)):
            vector[i] += X[k]**i * Y[k]

    # Solution finding using LU decomposition using Doolittle's condition L[i][i]=1
    # partial pivoting to avoid division by zero at pivot place
    A, vector = partial_pivot_LU(A, vector, order)
    A = LU_doolittle(A,order)

    # Finding coefficient vector
    solution = for_back_subs_doolittle(A,order,vector)

    return solution[0:order]



# Plotting the graph

def plot_graph_poly(X, Y, sol, order):
```

```
    yfit=[0 for i in range(len(X))]
    # finding yfit
    for k in range(len(X)):
        for l in range(order):
            yfit[k]+=sol[l]*X[k]**l

    # plotting X and y_fit
    plt.plot(X, yfit, 'r-', label="Curve fit with polynomial of degree = "+ str(order-1))
```

In [ ]:

```

```