Course Objective: To understand the PYTHON environment and make numerical computations and analysis.

Course Outcomes:

At the end of the course, student will be able to

CO1 Solve the different methods for linear, non-linear and differential equations

CO2 Learn the PYTHON Programming language

CO3 Familiar with the strings and matrices in PYTHON

CO4 Write the Program scripts and functions in PYTHON to solve the methods

CONTENTS

Write Programs in PYTHON Programming for the following:

1. To find the roots of non-linear equation using Bisection method
2. To find the roots of non-linear equation using Newton Raphson's method.
3. Curve fitting by least – square approximations
4. To solve the system of linear equations using Gauss - elimination method
5. To solve the system of linear equations using Gauss - Siedal method
6. To solve the system of linear equations using Gauss - Jordan method
7. To integrate numerically using Trapezoidal rule
8. To integrate numerically using Simpsons rule
9. To find the largest eigen value of a matrix by Power – method
10. To find numerical solution of ordinary differential equations by Euler's method
11. To find numerical solution of ordinary differential equations by Runge-Kutta method
12. To find numerical solution of ordinary differential equations by Milne's method
13. To find the numerical solution of Laplace equation
14. To find the numerical solution of Wave equation
15. To find the solution of a tri-diagonal matrix using Thomas algorithm
16. To fit a straight using least square technique

Software Details

Language : python 3.6.3
IDE : Anaconda3-2021.05-Windows-x86_64

Note:
Comment must be written with pencil
The comments starts with #
Ex: # Pseudocode For Bisection Method

1. To find the roots of non-linear equation using Bisection method

Aim: To find the roots of non-linear equation using Bisection method

Algorithm:

```
# Pseudocode For Bisection Method
if f(a)*f(b) > 0
  print("No root found") # Both f(a) and f(b) are the same sign
else
  while abs(b - a) > tolerance
     c = (b + a) /2 # c is like a midpoint
     if f(c) == 0
       return(midpt) # The midpt is the root such that f(midpt) = 0
     else if f(c) < 0
       a = c # Shrink interval from right.
     else
       b = c
return c
```

Program:

```
import sys
import numpy as np
import matplotlib.pyplot as plt

def  f(x):
   return x**2-4

def bisection(a,b,tol):
   i=1
   while abs(b-a)>tol:
      c= (a+b)/2
      print("interations of i = ",i,"x = ",c,"f(x) = ",f(c))
      if f(c)==0:
         print("root is found at",c)
         return c
      elif  f(c)<0:
         a=c
      else:
         b=c
      i= i+1
   return c

a= float(input("Enter the input a:"))
b= float(input("Enter the input b:"))
tol = float(input("Enter the tolerance:"))
```

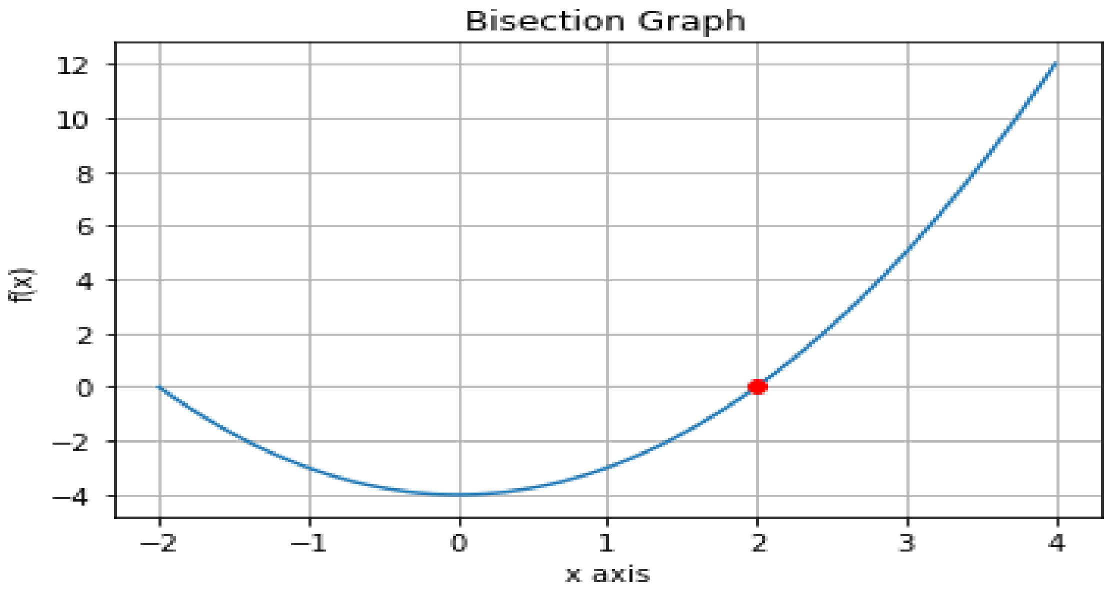
```
if f(a)*f(b)>0:
    print("No roots existed in the equation")
    sys.exit()
else:
    s = bisection(a, b, tol)

x= np.linspace(a,b,100)
plt.plot(x,f(x))
plt.xlabel("x axis")
plt.ylabel("f(x)")
plt.title("Bisection Graph")
plt.grid()
plt.plot(s,0,marker='o',color='red')
plt.show()
```

Output:

```
Enter the input a:-2
Enter the input b:4
Enter the tolerance:1e-100
interations of i = 1 x = 1.0 f(x) = -3.0
interations of i = 2 x = 2.5 f(x) = 2.25
interations of i = 3 x = 1.75 f(x) = -0.9375
interations of i = 4 x = 2.125 f(x) = 0.515625
interations of i =  5 x =  1.9375 f(x) =  -0.24609375
interations of i = 6 x = 2.03125 f(x) = 0.1259765625
interations of i = 7 x = 1.984375 f(x) = -0.062255859375
interations of i = 8 x = 2.0078125 f(x) = 0.03131103515625
interations of i =  9 x =  1.99609375 f(x) =  -0.0156097412109375
interations of i = 10 x = 2.001953125 f(x) = 0.007816314697265625
interations of i = 11 x = 1.9990234375 f(x) = -0.0039052963256835938
interations of i = 12 x = 2.00048828125 f(x) = 0.0019533634185791016
interations of i = 13 x = 1.999755859375 f(x) = -0.0009765028953552246
interations of i = 14 x = 2.0001220703125 f(x) =  0.0004882961511611938
interations of i = 15 x = 1.99993896484375 f(x) = -0.00024413689970970154
interations of i = 16 x = 2.000030517578125 f(x) = 0.00012207124382257462
interations of i = 17 x = 1.9999847412109375 f(x) = -6.1034923419356346e-05
interations of i = 18 x = 2.0000076293945312 f(x) =  3.0517636332660913e-05
interations of i = 19 x = 1.9999961853027344 f(x) = -1.5258774510584772e-05
interations of i = 20 x = 2.000001907348633 f(x) = 7.629398169228807e-06
interations of i = 21 x = 1.9999990463256836 f(x) = -3.8146963561302982e-06
interations of i = 22 x = 2.000000476837158 f(x) = 1.9073488601861754e-06
interations of i = 23 x = 1.999999761581421 f(x) = -9.536742595628311e-07
interations of i = 24 x = 2.0000001192092896 f(x) =  4.768371724139797e-07
```

interations of i = 25 x = 1.9999999403953552 f(x) = -2.3841857554884882e-07
interations of i = 26 x = 2.0000000298023224 f(x) = 1.1920929043895967e-07
interations of i = 27 x = 1.9999999850988388 f(x) = -5.960464477539063e-08
interations of i = 28 x = 2.0000000074505806 f(x) = 2.9802322387695312e-08
interations of i = 29 x = 1.9999999962747097 f(x) = -1.4901161193847656e-08
interations of i = 30 x = 2.000000001862645 f(x) = 7.450580596923828e-09
interations of i = 31 x = 1.9999999990686774 f(x) = -3.725290298461914e-09
interations of i = 32 x = 2.0000000004656613 f(x) = 1.862645149230957e-09
interations of i = 33 x = 1.9999999997671694 f(x) = -9.313225746154785e-10
interations of i = 34 x = 2.0000000001164153 f(x) = 4.656612873077393e-10
interations of i = 35 x = 1.9999999999417923 f(x) = -2.3283064365386963e-10
interations of i = 36 x = 2.000000000029104 f(x) = 1.1641532182693481e-10
interations of i = 37 x = 1.999999999985448 f(x) = -5.820766091346741e-11
interations of i = 38 x = 2.000000000007276 f(x) = 2.9103830456733704e-11
interations of i = 39 x = 1.999999999996362 f(x) = -1.4551915228366852e-11
interations of i = 40 x = 2.000000000001819 f(x) = 7.275957614183426e-12
interations of i = 41 x = 1.9999999999990905 f(x) = -3.637978807091713e-12
interations of i = 42 x = 2.0000000000004547 f(x) = 1.8189894035458565e-12
interations of i = 43 x = 1.9999999999997726 f(x) = -9.094947017729282e-13
interations of i = 44 x = 2.0000000000001137 f(x) = 4.547473508864641e-13
interations of i = 45 x = 1.9999999999999432 f(x) = -2.2737367544323206e-13
interations of i = 46 x = 2.0000000000000284 f(x) = 1.1368683772161603e-13
interations of i = 47 x = 1.9999999999999858 f(x) = -5.684341886080802e-14
interations of i = 48 x = 2.000000000000007 f(x) = 2.842170943040401e-14
interations of i = 49 x = 1.9999999999999964 f(x) = -1.4210854715202004e-14
interations of i = 50 x = 2.0000000000000018 f(x) = 7.105427357601002e-15
interations of i = 51 x = 1.9999999999999991 f(x) = -3.552713678800501e-15
interations of i = 52 x = 2.0000000000000004 f(x) = 1.7763568394002505e-15
interations of i = 53 x = 1.9999999999999998 f(x) = -8.881784197001252e-16
interations of i = 54 x = 2.0 f(x) = 0.0
root is found at 2.0

Graph:



2. To find the roots of non-linear equation using Newton Raphson’s method.

Aim: To find the roots of non-linear equation using Newton Raphson’s method.

Algorithm:

step1: Start

step2: Define function as f(x)

step3: Define first derivative of f(x) as g(x)

step4: Input initial guess (x0) and required decimal values (N)

step5: Initialize iteration counter i = 1

step6: If g(x0) = 0 then print "Mathematical Error" and goto (12) otherwise goto (7)

step7: m=x0

step8: Calculate x1 = x0 - f(x0) / g(x0)

step9: k=x1

step10: if m==k goto (13) otherwise goto (11)

step11: x0 = x1

step12: Increment iteration counter i = i + 1

step13: Print root as x1

step14: Stop

Program:

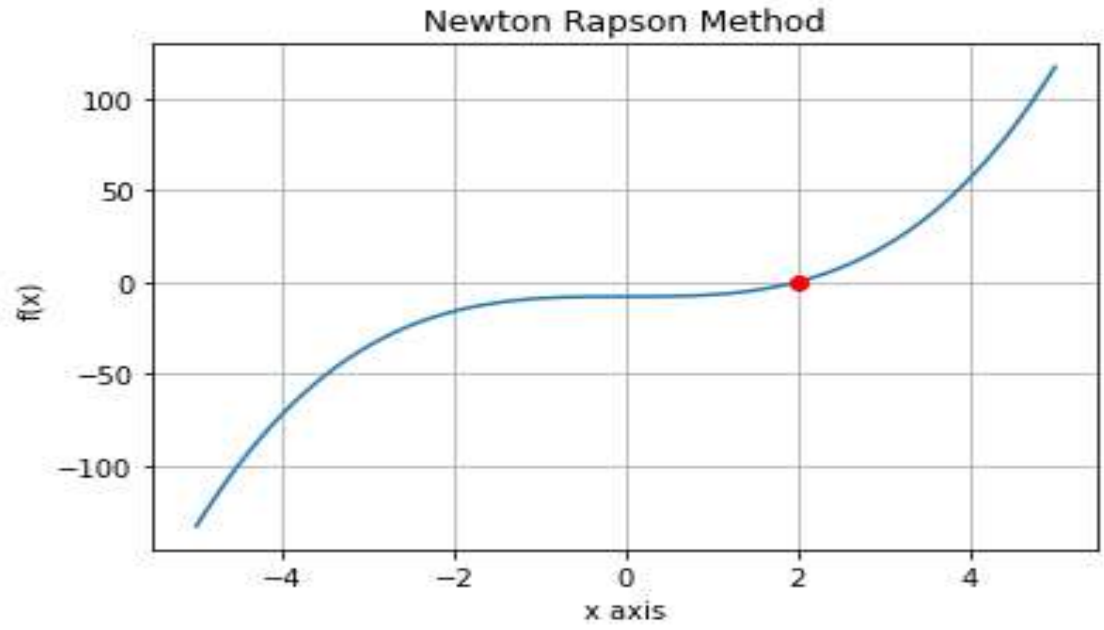
```
import sys
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

x=sp.Symbol('x')
f= input("Enter equation:")
f_print =f
f = sp.sympify(f)                    # Converting string to sympy Expresion
f_prime= f.diff(x)                   # creating differential equation from original equation
f_prime= sp.lambdify(x,f_prime)      # Converting differential equation to function
f = sp.lambdify(x,f)                 # Converting differential equation to function

x= float(input("Enter the value of x:"))
n=int(input("enter the required correct decimal values:"))
print("...............................")
if f_prime(x)==0:
   print("Mathmatical error")
   sys.exit(0)
else:
   i=1
   condition = True
   while condition:
      g= str(x)
```

```
        x_n = x- (f(x)/f_prime(x))
        print("iteration i:",i,"x= ",x_n,"f(x)=",f(x))
        m=str(x_n)
        """
        print(m)
        print("..................................")
        print(m[0:n+2])
        """
        if m[0:n+2]==g[0:n+2]:
            condition =False
        else:
            condition=True
            x=x_n
            i=i+1

x=str(x)
s= x[0:n+2]
print("The Root of equation f(x) ={} is {}".format(f_print,x))
x= np.linspace(-5,5,100)
plt.plot(x,f(x))
plt.xlabel("x axis")
plt.ylabel("f(x)")
plt.title("Newton Rapson Method")
plt.grid()
s=float(s)
plt.plot(s,0,marker='o',color='red')
plt.show()
```

Output:

```
Enter equation:x**3-8

Enter the value of x:8

enter the required correct decimal values:7
------------------------
iteration i: 1 x=  5.375 f(x)= 504.0
iteration i: 2 x=  3.6756354786371013 f(x)= 147.287109375
iteration i: 3 x=  2.6478039787146694 f(x)= 41.65892393602114
iteration i: 4 x=  2.1455646079059756 f(x)= 10.563398649931376
iteration i: 5 x=  2.0096524096938087 f(x)= 1.8769940018144933
iteration i: 6 x=  2.00004628653597 f(x)= 0.11638882970856912
iteration i: 7 x=  2.000000001071189 f(x)= 0.0005554512863970018
iteration i: 8 x= 2.0 f(x)= 1.2854266984163587e-08
iteration i: 9 x= 2.0 f(x)= 0.0
The Root of equation f(x) =x**3-8 is 2.0
```

Graph:



3. Curve fitting by least – square approximations

Aim: To draw Curve fitting by least – square approximations.

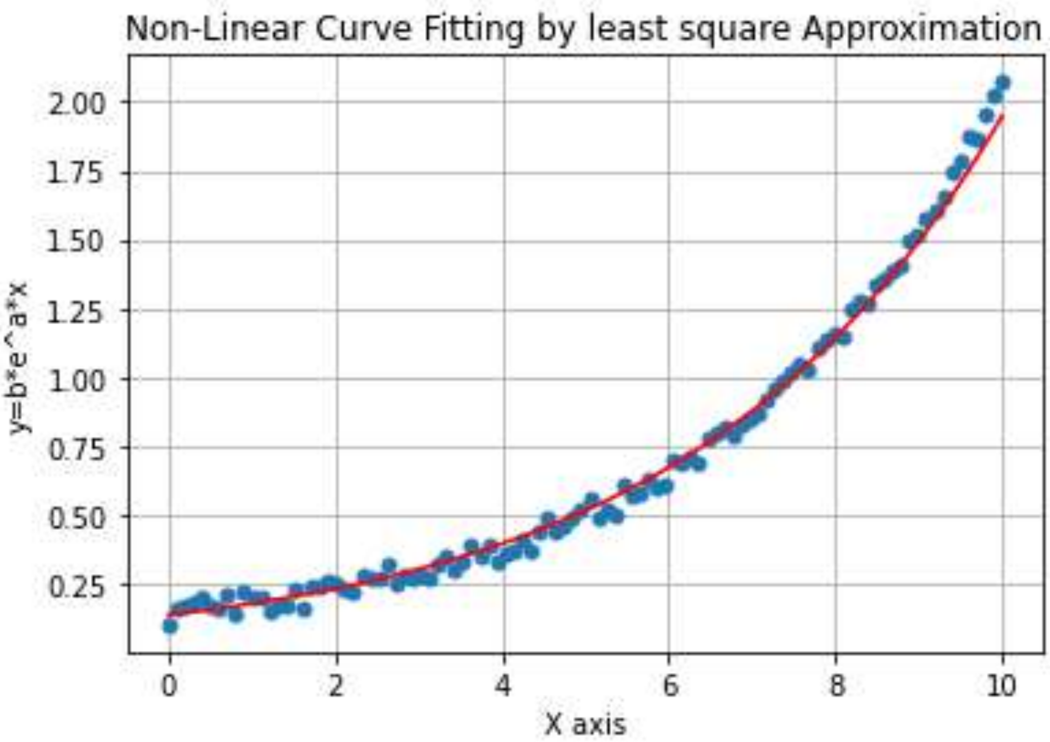
Program:

```
#Curve fitting by least – square approximations non-Linear approach
# y = b*exp(a*x) where b=0.1 and a =0.3
# log y =log b + log(ax)--> a=exp(x) becomes linear
import numpy as np
import matplotlib.pyplot as plt

x= np.linspace(0,10,100)    # create 100 values from 0 to 1

y= 0.1 *np.exp(0.3*x)+0.1*np.random.random(len(x)) #np.random.random creates noise reduction in the equation

m = np.vstack([x,np.ones(len(x))]).T    # np.ones creates one dimentional array of length x
                                        # vstack combines collection of one dimentional array to single array of ndimentional
                                        # .T convert matrix to its transpose

alpha = np.linalg.lstsq(m,np.log(y),rcond=None)[0] # getting first row from array of data

b= np.exp(alpha[1])      # converting y=logb to b=e^y

print("The coefficient of a = ",alpha[0],"and b=",b)

plt.plot(x,y,'o',markersize=5)
plt.plot(x,b*np.exp(alpha[0]*x),'r')
plt.xlabel("X axis")
plt.ylabel("y=b*e^a*x")
plt.title("Non-Linear Curve Fitting by least square Approximation")
plt.grid()
plt.show()
```

Output:

The coefficient of a = 0.2657667044851251 and b= 0.13658721956743416

Graph:


Non-Linear Curve Fitting by least square Approximation
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
y=b*e^a*x
0
2
4
6
8
10
X axis

4. To solve the system of linear equations using Gauss - elimination method

Aim: To solve the system of linear equations using Gauss - elimination method.

Program:

```
#system of linear equations using Gauss - elimination method
import numpy as np

A= np.array(
    [
     [4.0,-2.0,1.0],
     [-2.0,4.0,-2.0],
     [1.0,-2.0,4.0]
     ])
B= np.array(
     [11.0,-16.0,17.0]
     )

n = len(A)

print("\nThe matrix A :\n",A)
print("\nThe matrix B :\n",B)

x = np.zeros((n,n+1))
print("\nX matrix with zeros filling is:\n",x)

#Combining Marix A and Matrix B
m=0
for i in range(n):
    for j in range(n+1):
        if j<=2:
            x[i][j] = A[i][j]
        else:
            x[i][j] = B[m]
            m = m+1
print("\nThe Combined Matrix A and B:\n",x)

#Eliminating Matrix
for i in range(n):
    if x[i][i]==0:
        print("divdide error")
        break
    for j in range(n):
        if i!=j:
            r = x[j][i]/x[i][i]
            for k in range(n+1):
```

```
            x[j][k] = x[j][k]- r*x[i][k]
        print("\nAfter Iteration i = ",i,"\n",x)

# Substituting values
for i in range(n):
    B[i] = x[i][n]/x[i][i]

#The values variables
print("\nThe Values of varibles in matrix form \n",B)
print("\nx1= ",B[0],"\nx2= ",B[1],"\nx3= ",B[2])
```

Output:

```
The matrix A :
 [[ 4. -2. 1.]
 [-2.  4. -2.]
 [ 1. -2.  4.]]

The matrix B :
 [ 11. -16.  17.]

X matrix with zeros filling is:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

The Combined Matrix A and B:
 [[ 4. -2. 1. 11.]
 [ -2.  4.  -2. -16.]
 [ 1.  -2.   4.  17.]]

After Iteration i = 0
 [[  4.   -2.    1.   11. ]
 [  0.    3.   -1.5 -10.5]
 [  1.   -2.    4.  17. ]]

After Iteration i =  0
 [[  4.    -2.     1.    11. ]
  [ 0.     3.    -1.5  -10.5 ]
  [ 0.    -1.5    3.75  14.25]]

After Iteration i =  1
 [[  4.     0.     0.     4.  ]
 [  0.     3.    -1.5  -10.5 ]
```

```
 [  0.    -1.5    3.75  14.25]]

After Iteration i =  1
 [[  4.    0.    0.    4. ]
  [ 0.    3.   -1.5 -10.5]
  [ 0.    0.    3.    9. ]]

After Iteration i =  2
 [[  4.    0.    0.    4. ]
  [ 0.    3.   -1.5 -10.5]
  [ 0.    0.    3.    9. ]]

After Iteration i = 2
 [[ 4. 0. 0. 4.]
 [ 0.  3.  0. -6.]
 [ 0.  0.  3.  9.]]

The Values of varibles in matrix form
 [ 1. -2. 3.]

x1= 1.0
x2= -2.0
x3= 3.0
```

5. To solve the system of linear equations using Gauss - Siedal method

Aim: To solve the system of linear equations using Gauss - Siedal method.

Program:

```
#system of linear equations using Gauss - Siedal method
# input the equation in python
f1 = lambda x,y,z:(5-y-z)/2
f2 = lambda x,y,z:(15-3*x-2*z)/5
f3 = lambda x,y,z: (8-2*x-y)/4

e=1e-20 # tolerance for output
iteration = 0
#initialization of x,y,z values
x0=0
y0=0
z0=0

condition =True

while condition:
    x1 = f1(x0,y0,z0)
    y1 = f2(x1,y0,z0)
    z1 = f3(x1,y1,z0)
    #checking the tolerance of e1 ,e2 and e3
    e1= abs(x0-x1)
    e2= abs(y0-y1)
    e3= abs(z0-z1)

    iteration = iteration+1
    #Reassigning the x1, y1,z1 to x0,y0 and z0
    x0=x1
    y0=y1
    z0=z1

    condition = e1>e and e2>e and e3>e

print(f"The value of x,y and z are {x1}, {y1}, {z1} of {iteration} iteration")
```

Output:

The value of x,y and z are 1.0000000000000002, 2.0, 1.0 of 33 iteration

6. To solve the system of linear equations using Gauss - Jordan method

Aim: To solve the system of linear equations using Gauss - Jordan method.

Program:

```
#system of linear equations using Gauss - Jordan method
import numpy as np
def showMatrix():
    for i in sd:
        for j in i:
            print(" ",j, end="\t\t")
        print("\n")

#converting diagonal to 1's
def getone(pp):
    for i in range(len(sd[0])):
        if sd[pp][pp] != 1:
            q00 = sd[pp][pp]
            for j in range(len(sd[0])):
                sd[pp][j] = sd[pp][j] / q00



def getzero(r, c):
    for i in range(len(sd[0])):
        if sd[r][c] != 0:
            q04 = sd[r][c]
            for j in range(len(sd[0])):
                sd[r][j] = sd[r][j] - ((q04) * sd[c][j])



# defined matrix
sd = [
    [1, 1, 2, 9],
    [2, 4, -3, 1],
    [3, 6, -5, 0]
]

print("\nThe original Matrix:")
showMatrix()

for i in range(len(sd)):
    getone(i)
    for j in range(len(sd)):
        if i != j:
            getzero(j, i)
```

```
print("The matrix after gauss siedal method")
showMatrix()
n = len(sd)
print(n)
x = np.zeros(len(sd))

for i in range(len(sd)):
    x[i] = sd[i][n]/sd[i][i]

print("\nx= ",x[0],"\ny= ",x[1],"\nz= ",x[2])
```

Output:

```
The original Matrix:
   1          1          2          9

   2          4          -3         1

   3          6          -5         0

The matrix after gauss siedal method
   1.0        0.0        0.0        1.0

   0.0        1.0        0.0        2.0

   -0.0       -0.0       1.0        3.0



x=  1.0
y=  2.0
z=  3.0
```

7. To integrate numerically using Trapezoidal rule

Aim: To integrate numerically using Trapezoidal rule.

Program:

#Solve using Trapezoidal Rule

$$\int_0^{\pi/2} x * sin(x)$$

#Formula

$$f = h[\frac{1}{2}\{f(x_a) + f(x_b)\} + f(x_1) + f(x_2) + \cdots \ldots \ldots \ldots \ldots .. + f(x_{n-1})]$$

Where $x_1 = a+h$

$x_2 = a + 2h$

$x_3 = a + 3h$.. $x_{n-1} = a+(n-1)h$

```
import numpy as np
f= lambda x: x*np.sin(x)        #defining Equation in numpy
a=0.0
b=np.pi/2
n=10                            #the number of intervals
h=(b-a)/n
s= 0.5*(f(a)+f(b))
for i in range(1,n):
    s= s+f(a+i*h)
sol = h*s
print("Integral of equation f(x) = x*sin(x) from  0 to pi/2 is" , sol)
```

Output:

Integral of equation f(x) = x*sin(x) from 0 to pi/2 is 1.0020587067645337

8. To integrate numerically using Simpsons rule

Aim: To integrate numerically using Simpsons rule.

Program:

#Solve using Simpsons one third Rule

$$\int_{0}^{\pi/2} x * sin(x)$$

#Formula

$$f = \frac{h}{3}[\{ f(xa) + f(xb) \} + \sum_{i=1,3,5..}^{n-1} 4f(x_i) + \sum_{i=2,4,6..}^{n-2} 2f(x_i)]$$

```
import numpy as np


a= float(input("Enter the first point:"))
b= float(input("Enter the Second point:"))
n= int(input("Number of Panels:"))
h = (b-a)/n

# Defining Equation
def f(x):
    return x*np.sin(x)

def simpson(a,b):
    s=0
    t=0
    for i in range(1,n):
        if i%2==1:
            x = a+i*h
            s = s+f(x)
        else:
            x = a+i*h
            t = t+f(x)
    Integral = (f(a)+f(b)+4*s+2*t)*h/3
    print("The Integral is %.9f" %Integral)

# Calling The Equation
simpson(a,b)
```

Output:

Enter the first point:0

Enter the Second point:1.57075

Number of Panels:100

The Integral is 0.999927230

9. To find the largest eigen value of a matrix by Power – method

Aim: To find the largest eigen value of a matrix by Power – method.

Program:

```
import numpy as np

def normalize(x):
    fac = abs(x).max()
    x_n = x / x.max()
    return fac, x_n

x = np.array([1, 1,1])
a = np.array([[0, 2,5],
              [2, 3,6],
              [5, 3,6]
              ])
n= int(input("Enter the no of iterations:"))

for i in range(n):
    x = np.dot(a, x)
    lambda_1, x = normalize(x)
    print("iteration i= ",i,"eigen value=",lambda_1,"Eigen Vector = ",x)

print('Eigenvalue:', lambda_1)
print('Eigenvector:', x)
```

Output:

```
Enter the no of iterations:15
iteration i=  0 eigen value= 14 Eigen Vector =  [0.5        0.78571429 1.        ]
iteration i= 1 eigen value= 10.857142857142858 Eigen Vector =  [0.60526316 0.86184211
1.        ]
iteration i= 2 eigen value= 11.611842105263158 Eigen Vector =  [0.57903683 0.84362606
1.        ]
iteration i= 3 eigen value= 11.426062322946176 Eigen Vector =  [0.58526305 0.84796946
1.        ]
iteration i= 4 eigen value= 11.470223632667228 Eigen Vector =  [0.58376708 0.84692634
1.        ]
iteration i= 5 eigen value= 11.45961438699637 Eigen Vector =  [0.58412547 0.84717625 1.
]
iteration i= 6 eigen value= 11.462156118178477 Eigen Vector =  [0.58403955 0.84711634
1.        ]
iteration i= 7 eigen value= 11.461546760353588 Eigen Vector =  [0.58406015 0.8471307  1.
]
iteration i= 8 eigen value= 11.461692824210132 Eigen Vector =  [0.58405521 0.84712726
1.        ]
iteration i= 9 eigen value= 11.461657811107015 Eigen Vector =  [0.58405639 0.84712808
1.        ]
```

```
iteration i= 10 eigen value= 11.461666204049433 Eigen Vector =  [0.58405611 0.84712788
1.        ]
iteration i= 11 eigen value= 11.46166419218417 Eigen Vector =  [0.58405618 0.84712793
1.        ]
iteration i=  12 eigen value= 11.461664674446455 Eigen Vector =  [0.58405616 0.84712792
1.        ]
iteration i=  13 eigen value= 11.461664558843811 Eigen Vector =  [0.58405616 0.84712792
1.        ]
iteration i=  14 eigen value= 11.461664586554814 Eigen Vector =  [0.58405616 0.84712792
1.        ]
Eigenvalue: 11.461664586554814
Eigenvector: [0.58405616 0.84712792 1.        ]
```

10. To find numerical solution of ordinary differential equations by Euler's method

Aim: To find numerical solution of ordinary differential equations by Euler's method.

Algorithm:

Step1: define f(x,y)

Step2: input x0 and y0.

Step3: input step size, h and the number of steps, n.

Step4: for j from 1 to n do

a) m=f(t0,y0)
b) y1=y0+h∗m
c) t1=t0+h
d) Print t1 and y1
e) t0=t1
f) y0=y1

Step5:end

Program:

```
# function to be solved
def f(x,y):
    return x+y

# Euler method
def euler(x0,y0,xn,n):
    # Calculating step size
    h = (xn-x0)/n

    print('\n-----------SOLUTION ----------')
    print('..........................................')
    print('x0\t\ty0\t\tslope\t\tyn')
    print('..........................................')
    for i in range(n):
        slope = f(x0, y0)
        yn = y0 + h * slope
        print('%.4f\t%.4f\t%0.4f\t%.4f'% (x0,y0,slope,yn) )
        print('..........................................')
        y0 = yn
        x0 = x0+h

    print('\nAt x=%.4f, y=%.4f' %(xn,yn))

# Inputs
print('Enter initial conditions:')
x0 = float(input('x0 = '))
y0 = float(input('y0 = '))

print('Enter calculation point: ')
xn = float(input('xn = '))

print('Enter number of steps:')
```

```
step = int(input('Number of steps = '))

# Euler method call
euler(x0,y0,xn,step)
```

Output:

```
x0 = 0
y0 = 1
Enter calculation point:
xn = 1
Enter number of steps:

Number of steps = 20
-----------SOLUTION-----------
----------------------------------
x0          y0      slope    yn
----------------------------------
0.0000 1.0000 1.0000 1.0500
----------------------------------
0.0500 1.0500 1.1000 1.1050
----------------------------------
0.1000 1.1050 1.2050 1.1652
----------------------------------
0.1500 1.1652 1.3152 1.2310
----------------------------------
0.2000 1.2310 1.4310 1.3026
----------------------------------
0.2500 1.3026 1.5526 1.3802
----------------------------------
0.3000 1.3802 1.6802 1.4642
----------------------------------
0.3500 1.4642 1.8142 1.5549
----------------------------------
0.4000 1.5549 1.9549 1.6527
----------------------------------
0.4500 1.6527 2.1027 1.7578
----------------------------------
0.5000 1.7578 2.2578 1.8707
----------------------------------
0.5500 1.8707 2.4207 1.9917
----------------------------------
0.6000 1.9917 2.5917 2.1213
----------------------------------
0.6500 2.1213 2.7713 2.2599
----------------------------------
0.7000 2.2599 2.9599 2.4079
----------------------------------
0.7500 2.4079 3.1579 2.5657
----------------------------------
0.8000 2.5657 3.3657 2.7340
```

```
----------------------------------
0.8500 2.7340 3.5840 2.9132
----------------------------------
0.9000 2.9132 3.8132 3.1039
----------------------------------
0.9500 3.1039 4.0539 3.3066
----------------------------------

At x=1.0000, y=3.3066
```

11. To find numerical solution of ordinary differential equations by Runge-Kutta method

Aim: To find numerical solution of ordinary differential equations by Runge-Kutta method.

Program:

```
#Solution of ordinary differential equations by Runge-Kutta method
def f(x,y):
    return x+y

# RK-4 method
def rk4(x0,y0,xn,n):

    # Calculating step size
    h = (xn-x0)/n

    print('\n--------SOLUTION -------')
    print('....................................')
    print('x0\t\t\t\ty0\t\t\tyn')
    print('....................................')
    for i in range(n):
        k1 = h * (f(x0, y0))
        k2 = h * (f((x0+h/2), (y0+k1/2)))
        k3 = h * (f((x0+h/2), (y0+k2/2)))
        k4 = h * (f((x0+h), (y0+k3)))
        k = (k1+2*k2+2*k3+k4)/6
        yn = y0 + k
        print('%.4f\t\t%.4f\t\t%.4f'% (x0,y0,yn) )
        print('....................................')
        y0 = yn
        x0 = x0+h
    print('\nAt x=%.4f, y=%.4f' %(xn,yn))

# Inputs
print('Enter initial conditions:')
x0 = float(input('x0 = '))
y0 = float(input('y0 = '))

print('\nEnter calculation point: ')
xn = float(input('xn = '))

print('\nEnter number of steps:')
step = int(input('Number of steps = '))

# RK4 method call
rk4(x0,y0,xn,step)
```

Output:

```
Enter initial conditions:

x0 = 0

y0 = 1

Enter calculation point:

xn = 6

Enter number of steps:

Number of steps = 15

...........SOLUTION...........
------------------------------
x0          y0          yn
------------------------------
0.0000      1.0000      1.5835
------------------------------
0.4000      1.5835      2.6505
------------------------------
0.8000      2.6505      4.4390
------------------------------
1.2000      4.4390      7.3036
------------------------------
1.6000      7.3036      11.7736
------------------------------
2.0000      11.7736     18.6383
------------------------------
2.4000      18.6383     29.0752
------------------------------
2.8000      29.0752     44.8410
------------------------------
3.2000      44.8410     68.5561
------------------------------
3.6000      68.5561     104.1294
------------------------------
4.0000      104.1294    157.3920
------------------------------
4.4000      157.3920    237.0423
------------------------------
4.8000      237.0423    356.0559
```

```
--------------------------------
5.2000          356.0559              533.7893
--------------------------------
5.6000          533.7893              799.1167
-----------------------------

At x=6.0000, y=799.1167
```

12. To find numerical solution of ordinary differential equations by Milne’s method

Aim: To find numerical solution of ordinary differential equations by Milne’s method.

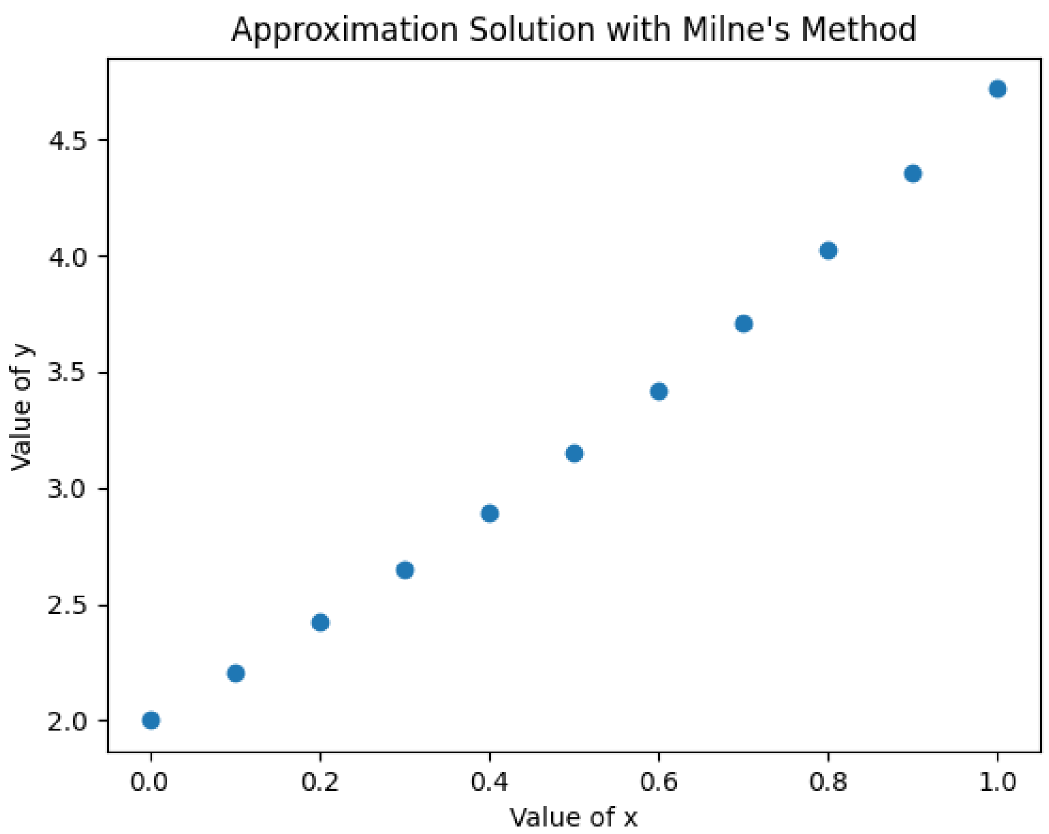
Program:

```
import numpy as np
from matplotlib import pyplot as plt

x0 = 0
y0 = 2
xf = 1
n = 11
deltax = (xf-x0)/(n-1)
x = np.linspace(x0,xf,n)
def f(x,y):
        return y-x

y = np.zeros([n])
y[0] = y0
py = np.zeros([n])
for i in range(0,4):
        py[i] = None

for i in range(1,4):
        k1 = deltax*f(x[i-1],y0)
        k2 = deltax*f(x[i-1]+deltax/2,y0+k1/2)
        k3 = deltax*f(x[i-1]+deltax/2,y0+k2/2)
        k4 = deltax*f(x[i-1]+deltax,y0+k3)
        y[i] = y0 + (k1 + 2*k2 + 2*k3 + k4)/6
        y0 = y[i]

for i in range(4,n):
        py[i] = 4*deltax/3*(2*f(x[i-1],y[i-1]) - f(x[i-2],y[i-2]) + 2*f(x[i-3],y[i-3]) ) + y[i-4]
        y[i] = deltax/3*( f(x[i],py[i]) + 4*f(x[i-1],y[i-1]) + f(x[i-2],y[i-2]) ) + y[i-2]

print("x_n\t\t   py_n\t            y_n")
for i in range(n):
        print (format(x[i],'.1f'),"\t",format(py[i],'6f'),"\t",format(y[i],'6f'))

plt.plot(x,y,'o')
plt.xlabel("Value of x")
plt.ylabel("Value of y")
plt.title("Approximation Solution with Milne's Method")
plt.show()
```

Output:

x_n	py_n	y_n
0.0	nan	2.000000
0.1	nan	2.205171
0.2	nan	2.421403
0.3	nan	2.649858
0.4	2.891821	2.891824
0.5	3.148717	3.148721
0.6	3.422114	3.422119
0.7	3.713747	3.713752
0.8	4.025535	4.025541
0.9	4.359596	4.359603
1.0	4.718274	4.718282

Graph:



13. To find the numerical solution of Laplace equation
Aim: To find the numerical solution of Laplace equation.
Program:
Output:

14. To find the numerical solution of Wave equation
Aim: To find the numerical solution of Wave equation.
Program:
Output:

15. To find the solution of a tri-diagonal matrix using Thomas algorithm
Aim: To find the solution of a tri-diagonal matrix using Thomas algorithm.
Program:
Output:

16. To fit a straight using least square technique

Aim: To fit a straight using least square technique.

Program:

```
#Curve Fitting by least – square approximations Linear approach
# linear Equation: y = ax+b  where b=1 and a=1
import numpy as np
import matplotlib.pyplot as plt
x= np.linspace(0,1,100)                    # create 100 values from 0 to 1

y= 1+x+x*np.random.random(len(x)) #np.random.random creates noise reduction in the
equation

m = np.vstack([x,np.ones(len(x))]).T    # np.ones creates one dimentional array of length x
                                        # vstack combines collection of one dimentional array
                                        to single array of ndimentional
                                        # .T convert matrix to its transpose
y = y[:,np.newaxis]         # converting one dimentional array to 2 dimentional array

alpha = np.linalg.lstsq(m,y,rcond=None)[0]              # getting first row from array of data
print("The coefficient of a = ",alpha[0],"and b=",alpha[1])
plt.plot(x,y,'o',markersize=5)
plt.plot(x,(alpha[0]*x)+alpha[1],'r')
plt.xlabel("X axis")
plt.ylabel("y=ax+b")
plt.title("Linear Curve Fitting by least square Approximation")
plt.grid()
plt.show()
```

Output:

The coefficient of a = [1.41041107] and b= [1.0411543]

Graph: