2

RANK NULLITY THEROEM

```
import numpy as np
from scipy.linalg import null_space
A = np. array ([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
rank = np.linalg.matrix_rank (A)
print ("Rank of the matrix", rank)
ns = null_space (A)
print ("Null space of the matrix" ,ns)
nullity = ns. shape [1]
print ("Null space of the matrix" ,nullity)
if rank + nullity == A. shape [1] :
    print ("Rank-nullity theorem holds.")
else:
    print ("Rank-nullity theorem does not hold.")
```

NEWTON RAPSHON

```
3*x-cos(x)-1
1
5
```

```
from sympy import *
x=Symbol('x')
g =input ('Enter the function')
f=lambdify(x,g)
dg=diff(g);
df=lambdify (x,dg)
x0=float(input('Enter the intial approximation'));
n=int(input ('Enter the number of iterations'));
for i in range (1,n+1) :
      x1 = (x0-(f(x0)/df(x0)))
      print ('iteration %d \t the root %0.3f \t function value %0.3f \n'%(i, x1, f(x1)));
     x0=x1
```

SOLVE y’’-5y’+6y=cos(4x)

```
from sympy import*
x=Symbol('x')
y=Function("y")(x)
C1,C2=symbols('C1,C2')
y1=Derivative(y,x)
y2=Derivative(y1,x)
print("Differentil Equation:\n")
diff1=Eq(y2-5*y1+6*y-cos(4*x),0)
display(diff1)
print("\n\n General solution : \n")
z=dsolve(diff1)
display(z)

PS=z.subs({C1:1,C2:2})
print("\n\n Particular Solution :\n")
display(PS)
```

REGULA FALSI METHOD

```
X**3-2*x-5
2
3
5
```

```
from sympy import *
x=Symbol('x')
g=input ('Enter the function')
f= lambdify(x,g)
a=float(input('Enter a value : '))
b=float(input('Enter a value : '))
N=int (input ('Enter number of iteration :'))

for i in range (1, N+1) :
     c = (a*f (b) -b*f (a)) / (f (b) - f(a))
     if ((f (a) *f (c) <0)) :
          b=c
     else:
          a=c
     print ('itration %d \t the root %0.3f \t function value %0.3f \n' %(i, c, f (c))) ;
```

TAYLOR SERIES

```
from numpy import array
def taylor(deriv,x,y, xStop,h) :
    X =[]
    Y= []
    X.append(x)
    Y.append(y)
    while X<xStop :
       D = deriv(x,y)
       H = 1.0
       for j in range (3):
          H = H*h/(j + 1)
          y = y + D[j]*H
     x = x + h
     X.append(x)
     Y.append(y)

    return array(X), array(Y)
def deriv(x,y):
    D=zeros((4,1))
    D[0] = [2*[0] + 3*exp (x)]
    D[1] = [4*y [0] + 9*exp (x)]
    D[2] = [8*y [0] + 21 *exp (x) ]
    D[3] = [16 *y[0] + 45 *exp (x)]
    return D
x = 0.0
xStop = 0.3
y = array ([0.0])
h = 0.1
X,Y= taylor (deriv,x, y,xStop,h)
print ("The required values are : at x=%0.2f,y=%0.5f,x=%0.2f,y=%0.5f,x=%0.2f,y=%0.5f,x=%0.2f,y=%0.5f" %(X[0],Y[0],X[1],Y[1],X [2],Y[2],X[3],Y[3]))
```

TO FIND DIVERGENCE

```
from sympy.physics.vector import*
from sympy import var
var('x,y,z')
v=ReferenceFrame('v')
F=v[0]**2*v[1]*v.x+v[1]*v[2]**2*v.y+v[0]**2*v[2]*v.z
G=divergence(F,v)
F=F.subs([(v[0],x),(v[1],y),(v[2],z)])
print("Given vector point function is ")
display(F)
G=G.subs([(v[0],x),(v[1],y),(v[2],z)])
print("Divergence of F=")
display(G)
```

TO FIND CURL

```
from sympy.physics.vector import*
from sympy import var
var('x,y,z')
v=ReferenceFrame('v')
F=v[0]*v[1]**2*v.x+2*v[0]**2*v[1]*v[2]*v.y-3*v[1]*v[2]**2*v.z
G=curl(F,v)
F=F.subs([(v[0],x),(v[1],y),(v[2],z)])
print("Given vector point function is")
display(F)
G=G.subs([(v[0],x),(v[1],y),(v[2],z)])
print("curl of F=")
display(G)
```

RUNGE KUTTA AT Y(2) TAKING H=0.2.GIVEN THAT Y(1)=2

```
from sympy import*
import numpy as np
def RungeKutta (g, x0,h, y0, xn) :

   x,y=symbols ('x,y')
   f= lambdify ([x,y] ,g)
   xt =x0+h
   Y= [y0]
   while xt<=xn :
        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)
        y1=y0+ (1/6) *(k1+2*k2+2*k3+k4)
        Y.append (y1)
        x0=xt
        y0 =y1
        xt=xt+h
    return np. round (Y, 2)
RungeKutta ('1+(y/x) ',1, 0.2,2,2)
```

APPLY MILNES PREDICTOR CORRECTOR METHOD

```
from sympy import*
def Milne (g, x0,h, y0,y1, y2, y3) :
      x,y=symbols ('x,y')
      f=lambdify ([x, y],g)
      x1=x0+h
      x2=x1+h
      x3=x2+h
      x4=x3+h

     y10=f (x0, y0)
     y11=f (x1, y1)
     y12=f (x2, y2)
     y13=f (x3, y3)
     y4p=y0+ (4*h/3) *(2*y11-y12+2*y13)
     print ('predicted value of y4', y4p)
     y14=f (x4, y4p)
     for i in range (1,4) :
           y4=y2+(h/3) *(y14+4*y13+y12)
           print ('corrected value of y4 , iteration %d '%i,y4)
            y14=f(x4,y4)
Milne('x**2+y/2',1,0.1,2,2.2156,2.4649,2.7514)
```

<u>SIMPSONS 1/3</u>

```
def my_func (x) :
      return 1 / (1 + x ** 2)
def simpson13 (x0, xn,n) :
   h = (xn - x0) / n
   integration = (my_func (x0) + my_func(xn))
   k = x0
   for i in range (1,n) :
   if i%2== 0:
     integration = integration + 4 * my_func(k)
   else:
     integration = integration + 2 * my_func (k)
   k += h
  integration = integration*h*(1/3)
  return integration
```

```
lower_limit = float (input ("Enter lower limit of integration: "))
upper_limit= float (input ("Enter upper limit of integration: "))
sub_interval = int (input ("Enter number of sub intervals: "))

result = simpson13 (lower_limit, upper_limit, sub_interval)
print ("Integration result by Simpson's 1/3 method is: %0.6f" %
(result))
```