In [1]:

```
#Evaluate the integral
from sympy import *
x= Symbol ('x')
y= Symbol ('y')
z= Symbol ('z')
w2= integrate (( x*y*z ) ,(z ,0 , 3-x-y ) ,(y ,0 , 3-x ) ,(x ,0 , 3 ) )
print ( w2 )
```

```
81/80
```

In [2]:

```
#Find the area of an ellipse by double integration. A=4
from sympy import *
x= Symbol ('x')
y= Symbol ('y')
#a= Symbol ('a ')
#b= Symbol ('b ')
a=4
b=6
w3=4* integrate (1 ,( y ,0 ,( b/a )* sqrt ( a ** 2-x ** 2 ) ) ,(x ,0 , a ) )
print ( w3 )
```

```
24.0*pi
```

In [5]:

```
#Find the area of the cardioid r = a (1 + cosϑ) by double integration
from sympy import *
r= Symbol ('r')
t= Symbol ('t')
a= Symbol ('a')
#a=4
w3=2* integrate (r ,( r ,0 , a*( 1+cos ( t ) ) ) ,(t ,0 ,pi) )
pprint ( w3 )
```

```
     2
3⋅π⋅a
──────
  2
```

In [9]:

```
#Find Beta (3, 5), Gamma (5)
from sympy import beta , gamma
m= input ('m :') ;
n= input ('n :') ;
m= float ( m ) ;
n= float ( n ) ;
s= beta (m , n ) ;
t= gamma ( n )
print ('gamma (',n,') is %3.3f '%t )
print ('Beta (',m ,n ,') is %3.3f '%s )
```

```
m :3
n :5
gamma ( 5.0 ) is 24.000
Beta ( 3.0 5.0 ) is 0.010
```

In [10]:

```
#Calculate Beta (5/2, 7/2) and Gamma (5/2)
from sympy import beta , gamma
m= float ( input ('m : ') ) ;
n= float ( input ('n :') ) ;
s= beta (m , n ) ;
t= gamma ( n )
print ('gamma (',n ,') is %3.3f '%t )
print ('Beta (',m ,n ,') is %3.3f '%s )
```

```
m : 2.5
n :3.5
gamma ( 3.5 ) is 3.323
Beta ( 2.5 3.5 ) is 0.037
```

In [18]:

```
#Verify that Beta (m, n) = Gamma (m) Gamma (n)/Gamma (m + n) for m=5 and n=7
from sympy import beta , gamma
m=5 ;
n=7 ;
m= float ( m ) ;
n= float ( n ) ;
s= beta (m , n ) ;
t=( gamma ( m )* gamma ( n ) )/ gamma ( m+n ) ;
print (s , t )
if abs(s - t) <= 0.00001:
    print ('beta and gamma are related ')
else :
    print ('given values are wrong ')
```

```
0.000432900432900433 0.000432900432900433
beta and gamma are related
```

In [20]:

```
#To find gradient of Ø = x2 y + 2xz − 4.
from sympy . vector import *
from sympy import symbols
N= CoordSys3D ('N')
x ,y , z= symbols ('x y z')
A=N . x ** 2*N . y+2*N . x*N . z-4
delop =Del ()
display ( delop ( A ) )
gradA = gradient ( A )
print ( f"\n Gradient of {A} is \n")
display ( gradA )
```

$$\left(\frac{\partial}{\partial \mathbf{x_N}}\left(\mathbf{x_N}^2\mathbf{y_N} + 2\mathbf{x_N}\mathbf{z_N} - 4\right)\right)\hat{\mathbf{i}}_\mathbf{N} + \left(\frac{\partial}{\partial \mathbf{y_N}}\left(\mathbf{x_N}^2\mathbf{y_N} + 2\mathbf{x_N}\mathbf{z_N} - 4\right)\right)\hat{\mathbf{j}}_\mathbf{N} + \left(\frac{\partial}{\partial \mathbf{z_N}}\left(\mathbf{x_N}^2\mathbf{y_N} + 2\mathbf{x_N}\mathbf{z_N} - 4\right)\right)\hat{\mathbf{k}}_\mathbf{N}$$

```
 Gradient of N.x**2*N.y + 2*N.x*N.z - 4 is
```

$$\left(2\mathbf{x_N}\mathbf{y_N} + 2\mathbf{z_N}\right)\hat{\mathbf{i}}_\mathbf{N} + \left(\mathbf{x_N}^2\right)\hat{\mathbf{j}}_\mathbf{N} + \left(2\mathbf{x_N}\right)\hat{\mathbf{k}}_\mathbf{N}$$

In [21]:

```
#To find divergence of = x2 yz + y 2 zx  + z2 xy
from sympy . vector import *
from sympy import symbols
N= CoordSys3D ('N')
x ,y , z= symbols ('x y z')
A=N . x ** 2*N . y*N . z*N . i+N . y ** 2*N . z*N . x*N . j+N . z ** 2*N . x*N . y*N . k
delop =Del ()
divA = delop .dot ( A )
display ( divA )
print ( f"\n Divergence of {A} is \n")
display ( divergence ( A ) )
```

$$\frac{\partial}{\partial \mathbf{z_N}} \mathbf{x_N} \mathbf{y_N} \mathbf{z_N}^2 + \frac{\partial}{\partial \mathbf{y_N}} \mathbf{x_N} \mathbf{y_N}^2 \mathbf{z_N} + \frac{\partial}{\partial \mathbf{x_N}} \mathbf{x_N}^2 \mathbf{y_N} \mathbf{z_N}$$

```
 Divergence of N.x**2*N.y*N.z*N.i + N.x*N.y**2*N.z*N.j + N.x*N.y*N.z**2*N.k is
```

$$6\mathbf{x_N}\mathbf{y_N}\mathbf{z_N}$$

In [23]:

```
#To find curl of = x2 yz + y2 zx  + z2 xy Find the inner product of the vectors (2, 1, 5, 4) and (3, 4, 7, 8)
import numpy as np
from sympy . vector import *
from sympy import symbols
N= CoordSys3D ('N')
x ,y , z= symbols ('x y z')
A=N . x ** 2*N . y*N . z*N . i+N . y ** 2*N . z*N . x*N . j+N . z ** 2*N . x*N . y*N . k
delop =Del ()
curlA = delop . cross ( A )
display ( curlA )
print ( f"\n Curl of {A} is \n")
display ( curl ( A ) )
A = np . array ([2 , 1 , 5 , 4])
B = np . array ([3 , 4 , 7 , 8])
output = np . dot(A , B )
print ( output )
```

$$\left(\frac{\partial}{\partial \mathbf{y_N}} \mathbf{x_N} \mathbf{y_N} \mathbf{z_N}^2 - \frac{\partial}{\partial \mathbf{z_N}} \mathbf{x_N} \mathbf{y_N}^2 \mathbf{z_N}\right) \hat{\mathbf{i}}_{\mathbf{N}} + \left(-\frac{\partial}{\partial \mathbf{x_N}} \mathbf{x_N} \mathbf{y_N} \mathbf{z_N}^2 + \frac{\partial}{\partial \mathbf{z_N}} \mathbf{x_N}^2 \mathbf{y_N} \mathbf{z_N}\right) \hat{\mathbf{j}}_{\mathbf{N}} + \left(\frac{\partial}{\partial \mathbf{x_N}} \mathbf{x_N} \mathbf{y_N}^2 \mathbf{z_N} - \frac{\partial}{\partial \mathbf{y_N}} \mathbf{x_N}^2 \mathbf{y_N} \mathbf{z_N}\right) \hat{\mathbf{k}}_{\mathbf{N}}$$

```
 Curl of N.x**2*N.y*N.z*N.i + N.x*N.y**2*N.z*N.j + N.x*N.y*N.z**2*N.k is
```

$$\left(-\mathbf{x_N}\mathbf{y_N}^2 + \mathbf{x_N}\mathbf{z_N}^2\right) \hat{\mathbf{i}}_{\mathbf{N}} + \left(\mathbf{x_N}^2\mathbf{y_N} - \mathbf{y_N}\mathbf{z_N}^2\right) \hat{\mathbf{j}}_{\mathbf{N}} + \left(-\mathbf{x_N}^2\mathbf{z_N} + \mathbf{y_N}^2\mathbf{z_N}\right) \hat{\mathbf{k}}_{\mathbf{N}}$$

```
77
```

In [24]:

```python
#Verify whether the following vectors (2, 1, 5, 4) and (3, 4, 7, 8) are orthogonal
import numpy as np
A = np . array ([2 , 1 , 5 , 4])
B = np . array ([3 , 4 , 7 , 8])
output = np . dot(A , B )
print ('Inner product is :', output )
if output ==0:
    print ('given vectors are orthognal ')
else :
    print ('given vectors are not orthognal ')
```

```
Inner product is : 77
given vectors are not orthognal
```

In [1]:

```python
#Obtain a root of the equation x3 - 2x - 5 = 0 between 2 and 3 by regula-falsi method. Perform 5 iterations
from sympy import *
x= Symbol ('x')
g = input ('Enter the function ')
f= lambdify (x , g )
a= float ( input ('Enter a valus :') )
b= float ( input ('Enter b valus :') )
N=int( input ('Enter number of iterations :') )
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 ) ) ) ;
```

```
Enter the function x**3-2*x-5
Enter a valus :2
Enter b valus :3
Enter number of iterations :5
itration 1       the root 2.059          function value -0.391

itration 2       the root 2.081          function value -0.147

itration 3       the root 2.090          function value -0.055

itration 4       the root 2.093          function value -0.020

itration 5       the root 2.094          function value -0.007
```

In [2]:

```python
""""Using tolerance value we can write the same program as follows: Obtain a root of the equation x3 − 2x − 5 = 0
between 2 and 3 by regula-falsi method. Correct to 3 decimal places"""
from sympy import *
x= Symbol ('x')
g = input ('Enter the function ')
f= lambdify (x , g )
a= float ( input ('Enter a valus :') )
b= float ( input ('Enter b valus :') )
N= float ( input ('Enter tolarence :') )
x=a ;
c=b ;
i=0
while (abs( x-c )>=N ):
    x=c
    c=(( a*f ( b )-b*f ( a ) )/( f ( b )-f ( a ) ) ) ;
    if (( f ( a )*f ( c )<0 ) ):
        b=c
    else :
        a=c
        i=i+1
    print ('itration %d \t the root %0.3f \t function value %0.3f \n'%(i ,c , f ( c ) ) ) ;
print ('final value of the root is %0.5f '%c )
```

```
Enter the function x**3-2*x-5
Enter a valus :2
Enter b valus :3
Enter tolarence :0.001
itration 1       the root 2.059          function value -0.391

itration 2       the root 2.081          function value -0.147

itration 3       the root 2.090          function value -0.055

itration 4       the root 2.093          function value -0.020

itration 5       the root 2.094          function value -0.007

itration 6       the root 2.094          function value -0.003

final value of the root is 2.09431
```

In [3]:

```python
#Find a root of the equation 3x = cos x+ 1, near 1, by Newton Raphson method. Perform 5 iterations
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 ('itration %d \t the root %0.3f \t function value %0.3f \n'%(i , x1 , f ( x1 ) ) )

    x0 = x1
```

```
Enter the function 3*x-cos(x)-1
Enter the intial approximation 1
Enter the number of iterations 5
itration 1       the root 0.620          function value 0.046

itration 2       the root 0.607          function value 0.000

itration 3       the root 0.607          function value 0.000

itration 4       the root 0.607          function value 0.000

itration 5       the root 0.607          function value 0.000
```

In [8]:

```python
#Evaluate   - Trapezoidal Rule
def my_func ( x ):
    return 1 / ( 1 + x ** 2 )
def trapezoidal ( x0 , xn , n ):
    h = ( xn - x0 ) / n
    integration = my_func ( x0 ) + my_func ( xn )
    for i in range (1 , n ):
        k = x0 + i * h

        integration = integration + 2 * my_func ( k )
    integration = integration * h / 2
    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 = trapezoidal ( lower_limit , upper_limit , sub_interval )
print (" Integration result by Trapezoidal method is: " , result )
```

```
 Enter lower limit of integration : 0
 Enter upper limit of integration : 5
 Enter number of sub intervals : 10
 Integration result by Trapezoidal method is:  1.3731040812301099
```

In [6]:

```python
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 += 4 * my_func(k)
        else:
            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)

```

```
Enter lower limit of integration: 0
Enter upper limit of integration: 5
Enter number of sub intervals: 100
Integration result by Simpson's 1/3 method is: 1.404120
```

In [9]:

```python
#Evaluate using Simpson’s 3/8th rule, taking 6 sub intervals
def simpsons_3_8_rule (f , a , b , n ):
    h = ( b - a ) / n
    s = f ( a ) + f ( b )
    for i in range (1 , n , 3 ):
        s += 3 * f ( a + i * h )
    for i in range (3 , n-1 , 3 ):
        s += 3 * f ( a + i * h )
    for i in range (2 , n-2 , 3 ):
        s += 2 * f ( a + i * h )
    return s * 3 * h / 8
def f ( x ):
    return 1/( 1+x ** 2 )
a = 0
b = 6
n = 6
result = simpsons_3_8_rule (f , a , b , n )
print ('%3.5f '% result )
```

```
1.27631
```

In [12]:

```python
#Solve:  with y (0) = 0 using Taylor series method at x = 0.1(0.1)0.3.
from numpy import array, exp, zeros
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 * y[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]))
```

```
The required values are: at x=0.00, y=0.00000, x=0.10, y=0.34850, x=0.20, y=0.81079, x=0.30, y=1.41590

C:\Users\kshit\AppData\Local\Temp\ipykernel_11596\2265566907.py:30: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)
  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]))
```

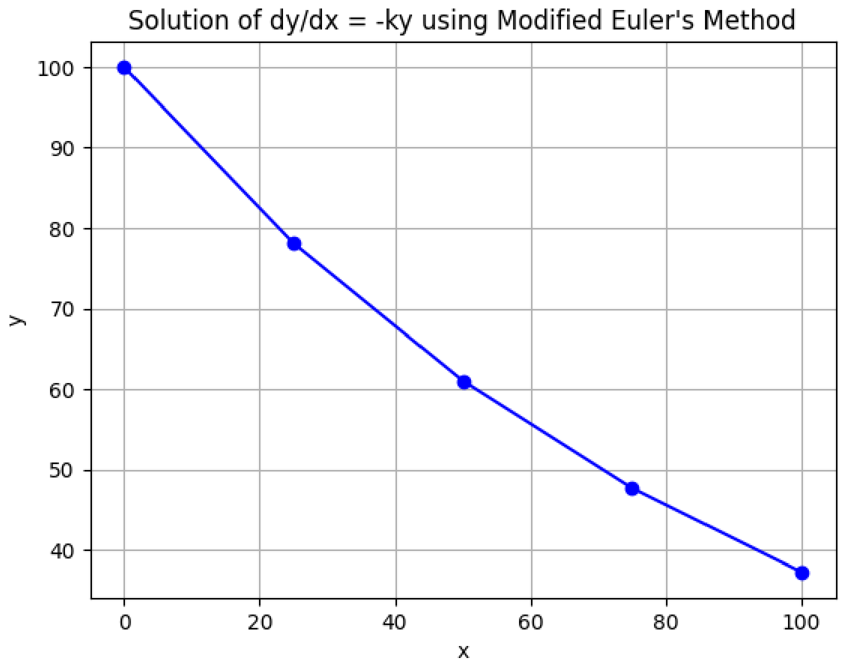
In [ ]:

```
#Solve: y ’ = e-x with y (0) = -1 using Euler’s method at x = 0.2(0.2)0.6.
import numpy as np
import matplotlib.pyplot as plt
f = lambda x, y: np.exp(-x)
h = 0.2
y0 = -1
n = 3
x = [0] * (n + 1)
y = [0] * (n + 1)
x[0] = 0
y[0] = y0
for i in range(n):
    x[i+1] = x[i] + h
    y[i+1] = y[i] + h * f(x[i], y[i])
print("The required values are:")
for i in range(n + 1):
    print("at x=%.2f, y=%.5f" % (x[i], y[i]))
plt.plot(x, y, 'bo- -', label='Approximate')
plt.plot(x, -np.exp(-x), 'g*-', label='Exact')
plt.title("Approximate and Exact Solution")
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid()
plt.legend(loc='best')
plt.show()
```

In [19]:

```python
#Solve y ’ = -ky with y (0) = 100 using modified Euler’s method at x = 100, by taking h = 25.
import numpy as np
import matplotlib.pyplot as plt
def modified_euler(f, x0, y0, h, n):
    x = np.zeros(n+1)
    y = np.zeros(n+1)
    x[0] = x0
    y[0] = y0
    for i in range(n):
        x[i+1] = x[i] + h
        k1 = h * f(x[i], y[i])
        k2 = h * f(x[i+1], y[i] + k1)
        y[i+1] = y[i] + 0.5 * (k1 + k2)
    return x, y
def f(x, y):
    return -0.01 * y
x0 = 0.0
y0 = 100.0
h = 25
n = 4
x, y = modified_euler(f, x0, y0, h, n)
print("The required value at x=%0.2f, y=%0.5f" % (x[4], y[4]))
print("\n\n")
plt.plot(x, y, 'bo-')
plt.xlabel('x')
plt.ylabel('y')
plt.title("Solution of dy/dx = -ky using Modified Euler's Method")
plt.grid(True)
plt.show()
```

```
The required value at x=100.00, y=37.25290
```



In [20]:

```
#Apply the Runge Kutta method to find the solution of 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)
```

Out[20]:

```
array([2.  , 2.62, 3.27, 3.95, 4.66, 5.39])
```

In [21]:

```python
#Given that y (1) = 2, y (1.1) =2.2156, y (1.2) = 2.4649, y (1.3) = 2.7514. Use corrector formula thrice
x0 = 1
y0 = 2
y1 = 2.2156
y2 = 2.4649
y3 = 2.7514
h = 0.1
x1 = x0 + h
x2 = x1 + h
x3 = x2 + h
x4 = x3 + h
def f(x, y):
    return x**2 + (y / 2)
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 is %3.3f' % y4p)
y14 = f(x4, y4p)
for i in range(1, 4):
    y4 = y2 + (h / 3) * (y14 + 4 * y13 + y12)
    print('corrected value of y4 after iteration %d is %3.5f' % (i, y4))
    y14 = f(x4, y4)
```

```
predicted value of y4 is 3.079
corrected value of y4 after iteration 1 is 3.07940
corrected value of y4 after iteration 2 is 3.07940
corrected value of y4 after iteration 3 is 3.07940
```

In [24]:

```python
#Apply Milne’s predictor and corrector method to solve   , y(0)=2 obtain y(0.8). Take h=0.2. Use Runge-Kutta method to calculate required initial values
def f(x, y):
    return x - y**2

# Initial values using Runge-Kutta method
h = 0.2
x0 = 0
y0 = 2
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 + (k1 + 2*k2 + 2*k3 + k4) / 6

x1 = x0 + h
k1 = h * f(x1, y1)
k2 = h * f(x1 + h/2, y1 + k1/2)
k3 = h * f(x1 + h/2, y1 + k2/2)
k4 = h * f(x1 + h, y1 + k3)
y2 = y1 + (k1 + 2*k2 + 2*k3 + k4) / 6

x2 = x1 + h
k1 = h * f(x2, y2)
k2 = h * f(x2 + h/2, y2 + k1/2)
k3 = h * f(x2 + h/2, y2 + k2/2)
k4 = h * f(x2 + h, y2 + k3)
y3 = y2 + (k1 + 2*k2 + 2*k3 + k4) / 6

# Predictor
y_pred = y1 + (4*h/3) * (2*f(x1, y1) - f(x2, y2) + 2*f(x0, y0))

# Corrector
y_corrected = y1 + (h/3) * (f(x0, y0) + 4*f(x2, y_pred) + f(x2, y2))

print("Predicted y(0.8):", y_pred)
print("Corrected y(0.8):", y_corrected)
```

```
Predicted y(0.8): -1.4370334083894307
Corrected y(0.8): 0.6698637229296666
```