Problem Statement:

Implement Steepest Descent Method and Conjugate Gradient Method for finding the minimum of the function $f(x_1, x_2) = 2x_1^2 + 4x_1x_2 + 4x_2^2 + 2x_2 - 4x_1 + 16$

Steps for Steepest Descent Algorithm:

1. Choose Starting Point x = $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ and evalute the function at $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$. Also define the stopping criterias δ_1, δ_2 and δ_3.
2. Find $f(x^{(k)})$ and $\nabla f(x^{(k)})$
3. Find $d_k = -\nabla f(x^{(k)})$
4. Find out $-\nabla^2 f(x^{(k)})$
5. Find λ_k = $\dfrac{-\nabla f(x^{(k)})^T d_k}{d_k^T \nabla^2 f(x^{(k)}) d_k}$
6. Find $x^{(k+1)} = x^{(k)} + \lambda_k d_k$
7. Evaluate $\triangle f = |f(x^{(k+1)}) - f(x^{(k)})|$
8. If $\triangle f < \delta_1$ then stop, If $\triangle x^T \triangle x < \delta_2$ then stop, If $\nabla f(x^{(k+1)})^T \nabla f(x^{(k+1)}) < \delta_3$ then stop, else go to step 2.

In Congugate Gradient Method:

We just update the direction vector d_k in each iteration as:

$$d_k = -\nabla f(x^{(k)}) + \beta_k d_{k-1}$$

where β_k is calculated as:

$$\beta_k = \frac{\nabla f(x^{(k)})^T \nabla f(x^{(k)})}{\nabla f(x^{(k-1)})^T \nabla f(x^{(k-1)})}$$

Some Necessary Calculations

$$f(x) = f(x_1, x_2) = 2x_1^2 + 4x_1x_2 + 4x_2^2 + 2x_2 - 4x_1 + 16$$

$$\nabla f(x) = \nabla f(x_1, x_2) = \begin{bmatrix} 4x_1 + 4x_2 - 4 \\ 4x_1 + 8x_2 + 2 \end{bmatrix}$$

$$\nabla f(x)^T = \nabla f(x_1, x_2)^T = \begin{bmatrix} 4x_1 + 4x_2 - 4 & 4x_1 + 8x_2 + 2 \end{bmatrix}$$

$$\nabla f(x)^2 = \nabla^2 f(x_1, x_2) = \begin{bmatrix} 4 & 4 \\ 4 & 8 \end{bmatrix}$$

Code

Importing the necessary libraries

In [219…

```
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from sympy import *
```

In [220…

```
# Define the function
def function(x):
    return 2*x[0]**2 + 4*x[0]*x[1] + 4*x[1]**2 + 2*x[1] - 4*x[0] + 16
```

In [221…

```
x1, x2 = symbols('x1 x2')

f = 2*x1**2 + 4*x1*x2 + 4*x2**2 + 2*x2 - 4*x1 + 16

f
```

Out[221]: $2x_1^2 + 4x_1x_2 - 4x_1 + 4x_2^2 + 2x_2 + 16$

In [222…

```
# Define the gradient of the function
def gradient(f, x1, x2):
    return np.array([f.diff(x1), f.diff(x2)])

gradient(f, x1, x2)
```

Out[222]:

```
array([4*x1 + 4*x2 - 4, 4*x1 + 8*x2 + 2], dtype=object)
```

In [223…

```
# Define the hessian of the function
def hessian(f, x1, x2):
    return np.array([[f.diff(x1, x1), f.diff(x1, x2)], [f.diff(x2, x1), f.di

hessian(f, x1, x2)
```

Out[223]:

```
array([[4, 4],
       [4, 8]], dtype=object)
```

Plotting the Function

In [224…

```
# Plotting the Function in 3D Space using Plotly

# Define the range of x and y
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)

# Create a meshgrid
X, Y = np.meshgrid(x, y)
```

```
# Calculate the function values
z = 2*X**2 + 4*X*Y + 4*Y**2 + 2*Y - 4*X + 16

fig = go.Figure(data=[go.Surface(z=z, x=x, y=y, colorscale='turbo')])
fig.update_layout(title='Example Curve', autosize=False,
                  width=800, height=800,
                  margin=dict(l=65, r=50, b=65, t=90))

fig.show()
```

Steepest Descent Method

In [225…

```
# Implementing the Steepest Descent Method
def steepest_descent(x_0, delta_1, delta_2, delta_3, max_iter=1000):

    x = x_0

    # Value of the function at x
    f_x = f.evalf(subs={x1: x[0], x2: x[1]})

    # Gradient of the function
    grad_f = gradient(f, x1, x2)

    # Gradient of the function at x
    grad_f_x = np.array([grad_f[0].evalf(subs={x1: x[0], x2: x[1]}), grad_f[

    # Hessian of the function
    hess_f = hessian(f, x1, x2)

    # Hessian of the function at x
    hess_f_x = np.array([[hess_f[0][0].evalf(subs={x1: x[0], x2: x[1]}), hes

    while max_iter > 0:

        # Value of the function at x
        f_x = f.evalf(subs={x1: x[0], x2: x[1]})

        # Gradient of the function
        grad_f = gradient(f, x1, x2)

        # Gradient of the function at x
        grad_f_x = np.array([grad_f[0].evalf(subs={x1: x[0], x2: x[1]}), gra

        # Hessian of the function
        hess_f = hessian(f, x1, x2)

        # Hessian of the function at x
        hess_f_x = np.array([[hess_f[0][0].evalf(subs={x1: x[0], x2: x[1]}),

        # Calculate the direction of descent
        d_k = -grad_f_x

        # Calculate the step size
```

```
        lambda_k = -np.dot(grad_f_x.T, d_k)/np.dot(np.dot(d_k.T, hess_f_x),

        # Update the value of x
        x = x + (lambda_k*d_k)

        # Value of the function at x
        new_f_x = f.evalf(subs={x1: x[0], x2: x[1]})

        # Find the difference between the function values
        triangle_f = abs(f_x - new_f_x)

        # Check if the stopping criteria 1 is met
        if triangle_f < delta_1:
            break

        # Check if the stopping criteria 2 is met
        elif np.dot(x.T, x) < delta_2:
            break

        # Check if the stopping criteria 3 is met
        elif np.dot(grad_f_x.T, grad_f_x) < delta_3:
            break

        # If none of the stopping criteria is met, then update the value of
        else:
            max_iter -= 1

    # Return the value of minima point and the minimum value of the function
    return x, f.evalf(subs={x1: x[0], x2: x[1]})
```

In [226…

```
# Define the initial point to infinity
x = np.array([10 ** 100, 10 ** 100])

# Define the tolerance
delta_1 = 1e-6
delta_2 = 1e-6
delta_3 = 1e-6

# Run the algorithm
x, min_f_x = steepest_descent(x, delta_1, delta_2, delta_3)
```

Print the values of x and min(f(x))

In [227…

```
print("The minimum point is: ", x)
print("The minimum value is: ", min_f_x)
```

```
The minimum point is:  [2.50001475369140 -1.49998524630860]
The minimum value is:  9.50000000217671
```

Conjugate Gradient Method

In [228…

```
# Implementing the Conjugate Descent Method
def conjugate_descent(x_0, delta_1, delta_2, delta_3, max_iter=1000):

    x = x_0
```

```
    # Value of the function at x
    f_x = f.evalf(subs={x1: x[0], x2: x[1]})

    # Gradient of the function
    grad_f = gradient(f, x1, x2)

    # Gradient of the function at x
    grad_f_x = np.array([grad_f[0].evalf(subs={x1: x[0], x2: x[1]}), grad_f[

    # Hessian of the function
    hess_f = hessian(f, x1, x2)

    # Hessian of the function at x
    hess_f_x = np.array([[hess_f[0][0].evalf(subs={x1: x[0], x2: x[1]}), hes

    # Calculate the direction of descent
    d_k = -grad_f_x

    while max_iter > 0:

        # Value of the function at x
        f_x = f.evalf(subs={x1: x[0], x2: x[1]})

        # Store the value of the gradient at x
        grad_f_x_old = grad_f_x

        # Gradient of the function
        grad_f = gradient(f, x1, x2)

        # Gradient of the function at x
        grad_f_x = np.array([grad_f[0].evalf(subs={x1: x[0], x2: x[1]}), gra

        # Hessian of the function
        hess_f = hessian(f, x1, x2)

        # Hessian of the function at x
        hess_f_x = np.array([[hess_f[0][0].evalf(subs={x1: x[0], x2: x[1]}),

        # Beta value
        beta_k = np.dot(grad_f_x.T, grad_f_x)/np.dot(grad_f_x_old.T, grad_f_

        # Calculate the direction of descent
        d_k = -grad_f_x + beta_k*d_k

        # Calculate the step size
        lambda_k = -np.dot(grad_f_x.T, d_k)/np.dot(np.dot(d_k.T, hess_f_x),

        # Update the value of x
        x = x + lambda_k*d_k

        # Difference between the function values
        triangle_f = abs(f_x - f.evalf(subs={x1: x[0], x2: x[1]}))

        # Check if the stopping criteria 1 is met
        if triangle_f < delta_1:
```

```
            break

        # Check if the stopping criteria 2 is met
        elif np.dot(x.T, x) < delta_2:
            break

        # Check if the stopping criteria 3 is met
        elif np.dot(grad_f_x.T, grad_f_x) < delta_3:
            break

        # If none of the stopping criteria is met, then update the value of
        else:
            max_iter -= 1

    # Return the value of minima point and the minimum value of the function
    return x, f.evalf(subs={x1: x[0], x2: x[1]})
```

In [229…

```
# Define the initial point
x = np.array([10 ** 100, 10 ** 100])

# Define the tolerance
delta_1 = 1e-6
delta_2 = 1e-6
delta_3 = 1e-6

# Run the algorithm
x, f_x = conjugate_descent(x, delta_1, delta_2, delta_3)
```

Print the values of x and min(f(x))

In [230…

```
print("The minimum point is: ", x)
print("The minimum value is: ", f_x)
```

```
The minimum point is:  [2.49981132307248 -1.49987579504955]
The minimum value is:  9.50000003916701
```

Plotting the Function along with the Minima Point

In [231…

```
# Plotting the Function in 3D Space using Plotly

# Define the range of x and y
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)

# Create a meshgrid
X, Y = np.meshgrid(x, y)

# Calculate the function values
z = function([X, Y])

fig = go.Figure(data=[go.Surface(z=z, x=x, y=y, colorscale='turbo_r')])

fig.update_layout(title='Example Curve', autosize=False,
                  width=800, height=800,
```

```
                  margin=dict(l=65, r=50, b=65, t=90))

# Plot the minimum point
fig.add_trace(go.Scatter3d(x=[2.5], y=[-1.5], z=[function([2.5, -1.5])], moc

fig.show()
```