Laboratory 1

Variant 4 – Newton’s Method algorithm

Lab group 105

Group 9 – Aditya Kandula, Martyna Wielgopolan

21.03.2025

I. Introduction

The goal of this assignment is to implement the **Newton's Method** algorithm to minimize a given two-variable mathematical function:

f(x,y) = 2sin(x)+3cos(y)

Newton's Method is a fast and useful algorithm for finding the minimum of a function. It is more powerful than simple methods because it uses information about how the function curves. However, it also needs more calculations and can sometimes fail if the function is too flat or the starting point is bad.

Advantages:

- **Fast Convergence:** It usually finds the minimum very quickly if the starting point is good.
- **Accurate:** It uses second-order information (curvature), so it gives a more precise direction compared to simpler methods like Gradient Descent.
- **Good for smooth functions:** Works well when the function is continuous and smooth.

Disadvantages:

- **Needs second derivatives:** Calculating the Hessian matrix can be hard or expensive for complex functions.
- **Can fail at saddle points:** If the Hessian is zero or not invertible, the method breaks or gives wrong results.
- **Depends on starting point:** If you start too far from the minimum, it might not work or go to the wrong place.

II. Implementation

1. Import libraries

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

- numpy is used for mathematical calculations (arrays, matrix operations).
- matplotlib is used for creating 3D plots to visualize the function.

2. Define the function to minimize

```
def function(x, y):
    return 2 * np.sin(x) + 3 * np.cos(y)
```

This is the function we want to minimize using Newton's Method.
The function takes two inputs x and y.

3. Define the gradient (first derivatives)

```
def gradient(x, y):
    df_dx = 2 * np.cos(x)
    df_dy = -3 * np.sin(y)
    return np.array([df_dx, df_dy])
```

The gradient shows the direction where the function is increasing.
Newton's method uses it to decide where to move next.

4. Define the Hessian matrix (second derivatives)
def hessian(x, y):

```
def hessian(x, y):

    d2f_dx2 = -2 * np.sin(x)
    d2f_dy2 = -3 * np.cos(y)
    return np.array([[d2f_dx2, 0],
                     [0, d2f_dy2]])
```

The Hessian tells us how the function curves.
It helps Newton's method compute a better direction and step size.

5. Define Newton's Method Algorithm

```
def newton_method(initial_guess, alpha, tol=1e-6, max_iter=1000):
    x = np.array(initial_guess, dtype=float)
    num_iter = 0
```

- Start from the initial_guess.
- Convert it into a numpy array for calculations.

6. Start Iteration Loop

```
for i in range(max_iter):
        grad = gradient(x[0], x[1])
        hess = hessian(x[0], x[1])
```

- Calculate the gradient and Hessian at the current point.

7. Check if Hessian is singular (non-invertible)

```
if np.linalg.det(hess) == 0:
            print("Cannot be solved as Hess inverse doen't exist", i)
            break
```

- If det(hessian) = 0, you cannot solve the system.
- The loop breaks to avoid crashing the program.

8. Calculate Newton's step (delta)

```
delta = np.linalg.solve(hess, grad)
```

- Solve the equation: Hessian * delta = gradient
- delta tells us how much to move to get closer to the minimum.
- This step can also be performed by finding the inverse of Hessian and multiplying it with gradient.

9. Update the current point

```
x_new = x - alpha * delta
```

- Move towards the minimum.
- alpha is the step size (learning rate).

10. Check for convergence (stopping condition)

```
if np.linalg.norm(x_new - x) < tol:

        break
```

If the movement is very small, the algorithm stops.

11. Update, count iterations and return results

```
 x = x_new

     num_iter += 1
  return x, num_iter
```

Return the approximate minimum and how many iterations it took.

13. Visualize the 3D surface

```
def visualize():
   X = np.linspace(-5, 5, 100)
   Y = np.linspace(-5, 5, 100)
   X, Y = np.meshgrid(X, Y)
   Z = function(X, Y)
   fig = plt.figure(figsize=(10, 6))
   ax = fig.add_subplot(111, projection='3d')
   surf = ax.plot_surface(X, Y, Z, cmap=cm.viridis, edgecolor='none')
   ax.set_title('Function Surface: 2sin(x) + 3cos(y)')
   ax.set_xlabel('X')
   ax.set_ylabel('Y')
   ax.set_zlabel('Z')
   fig.colorbar(surf, shrink=0.5, aspect=5)
```

```
    plt.show()

visualize()
```

- Creates a 3D plot of the function 2sin(x) + 3cos(y)
- Helps to see the function's shape and understand where the minimum might be.

14. Test different initial guesses

```
test_cases = [
    ([2.0, 2.0], 0.1),
    ([-4.0, -4.0], 0.05),
    ([1.0, -3.0], 0.2),
    ([-2.0, 4.0], 0.15),
    ([0.5, 0.5], 0.1),     ([0, 5], 0.1) #hessian inverse doesn't exist

]
```

- Different starting points help check if the algorithm finds different minima.
- Also shows how step size affects the result.

15. Run the tests and print the results

```
for initial_guess, learning_rate in test_cases:
    minimum, iterations = newton_method(initial_guess, learning_rate)
    print(f"Initial guess: {initial_guess}, Learning rate: {learning_rate}")
    print(f"Minimum approximation: {minimum}, Iterations: {iterations}\n")
```

- Runs Newton’s method for each case.
- Prints the minimum found and number of iterations.

FINAL OUTPUT EXAMPLE:
Initial guess: [2.0, 2.0], Learning rate: 0.1
Minimum approximation: [1.57079633 3.14159265], Iterations: 7

Summary:

This program uses Newton's Method to find the minimum of the function 2sin(x) + 3cos(y). The algorithm calculates the slope and the curve of the function to decide where to move next. It repeats this process until it gets very close to the minimum. The program also creates a 3D plot of the function to help us see the shape of the surface.

III. Discussion

We tested the code with six different starting points and step sizes. Here's a summary:

Initial guess	Step size	Result (approx.)	Nr of iterations
[2.0, 2.0]	0.1	[1.57, 3.141]	109
[-4.0, -4.0]	0.05	[-4.71, -3.141]	211
[1.0, -3.0]	0.2	[1.57, -3.141]	53
[-2.0, 4.0]	0.15	[-1.57, 3.141]	73
[0.5, 0.5]	0.1	[1.57e+00, 4.89e-06]	109
[0,0.5] # hessian det is 0	0.1	0	0

*** The above test cases are chosen to make sure all 4 quadrants in the XY-plain are accounted, with varying step sizes.**

Impact of Initial Vectors

The initial guess has a strong effect on the result. The function has several local minima due to the sine and cosine, so depending on where we start, we might end up at different places. For example:

- Starting near 1 or 2 gives us a solution near $x = \pi/2 \approx 1.57$
- Starting at negative values like -4 leads to a solution near $x = -\pi/2 \approx -1.57$

So, the algorithm does not always give the same result – it depends on where we start.

Impact of Step Size (Alpha)

Step size affects how fast or slow the algorithm moves toward the minimum. We found:

- **Smaller step sizes** (like 0.05) are safer but need more iterations.
- **Larger step sizes** (like 0.2) can converge faster but might overshoot or behave unpredictably if too large.

Choosing the right step size is a balance between speed and stability.

As shown in the last test case if the hessian matrix is non-inversible the programme throws an exception.

About the Function and Results

Our function is not difficult to minimize, but it has many waves (since it uses sine and cosine), so it has many local minima. Newton's method can find a local minimum, but not always the global one, especially without checking multiple points.

We only visualized the function surface so shown in fig 3.1, but the next step could be plotting the iteration path to see how the algorithm moves step by step.

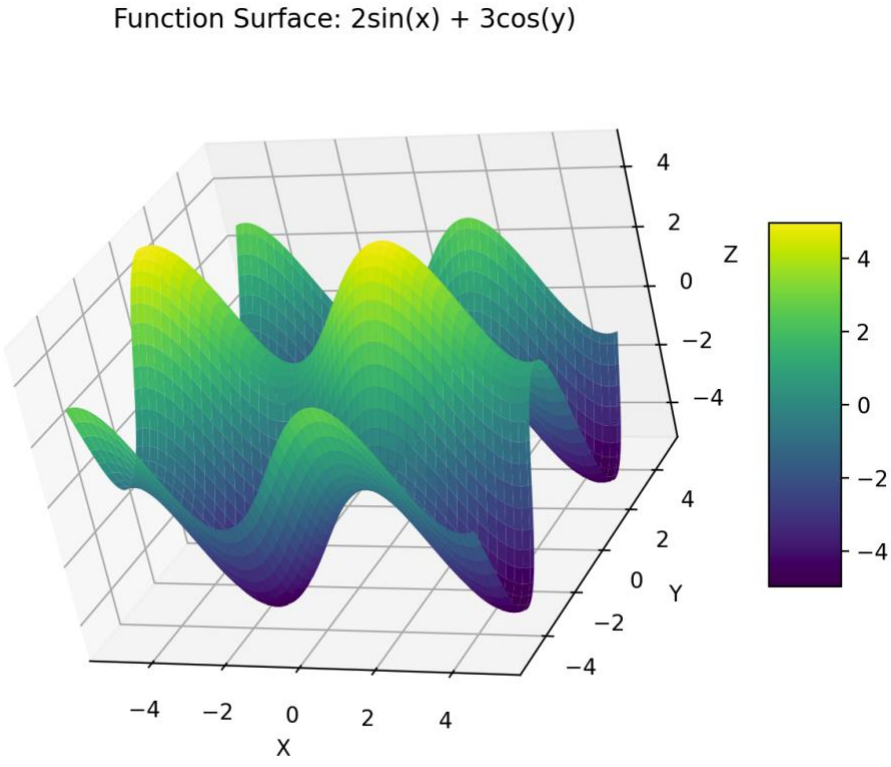

Fig 3.1

IV. Conclusions

We learned how Newton’s Method works for minimizing a function with two variables. It is powerful and fast when used correctly, but depends heavily on the starting point and step size.

What we learned:

- Initial values affect the result.
- Step size affects speed and accuracy.
- Visualization helps understand the shape of the function.

What was difficult:

- Making sure the Hessian isn’t zero (or the method will crash).
- Choosing the best step size.

What can be improved:

- Plotting the path of each iteration for better understanding.
- Adding checks to handle when the Hessian is singular.