EARIN Lab 1 Report

Variant 4 – Newton's Method algorithm

Lab group 105

Group 9 – Aditya Kandula, Martyna Wielgopolan

25.03.2025

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.

Formula with explanation:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \cdot \mathbf{H}^{-1}(\mathbf{x}_k) \cdot \nabla f(\mathbf{x}_k)$$

Where:

$\mathbf{x}_k$: the current point (vector of variables, e.g., $[x_k, y_k]$)

$\mathbf{x}_{k+1}$: the next point (after one iteration)

α: step size (also called the learning rate)

$\nabla f(\mathbf{x}_k)$: the **gradient vector** of the function at point $\mathbf{x}_k$

$\mathbf{H}(\mathbf{x}_k)$: the **Hessian matrix** (second derivatives) at point $\mathbf{x}_k$

$\mathbf{H}^{-1}$: the **inverse** of the Hessian

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: for numerical computations and array operations
- matplotlib.pyplot: for plotting
- matplotlib.cm: for color maps used in 3D surface plots

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.
This is the **function we want to minimize**:

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

It's periodic and non-convex — it has many minima and maxima.

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_with_tracking(initial_guess, alpha, tol=1e-6,
max_iter=1000):
    x = np.array(initial_guess, dtype=float)
    path = [x.copy()]
    num_iter = 0
```

- Start from the initial_guess.
- Convert it into a numpy array for calculations.
- x.copy() saves all the iterations points for later visualization

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(f"Hessian is not invertible at iteration {i} for initial
{initial_guess}")
            return None, num_iter, np.array(path)
```

- If det(hessian) = 0, you cannot solve the system.
- The loop breaks to avoid crashing the program.
- Returns with None to mark in which cases we are unable to find a convergence point using the Newton's method

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
        path.append(x_new.copy())
```

- Move towards the minimum.
- alpha is the step size (learning rate).
- path.append(x_new.copy()) creates a snapshot of the new position and adds it to the path list

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, np.array(path)
```

Return the approximate minimum, how many iterations it took and a list of points visited by the algorithm during optimization.

12. Visualize the 3D surface

```
def visualize(results):
    X = np.linspace(-5, 5, 150)
    Y = np.linspace(-5, 5, 150)
    X, Y = np.meshgrid(X, Y)
    Z = function(X, Y)


    fig = plt.figure(figsize=(18, 10))
    ax = fig.add_axes([0.05, 0.1, 0.65, 0.8], projection='3d')
    ax_text = fig.add_axes([0.72, 0.1, 0.25, 0.8])
    ax_text.axis('off')


    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, edgecolor='none',
alpha=0.3)


    zero_plane = np.zeros_like(Z)
    ax.plot_surface(X, Y, zero_plane, color='gray', alpha=0.2)


    path_colors = ['blue', 'green', 'orange', 'purple', 'crimson', 'magenta',
'darkcyan', 'darkred', 'darkblue']


    details_text = "Newton's Method Details:\n"
    details_text += "{:<10} {:<20} {:<25} {}\n".format("Test", "Iterations",
"Converged (x, y)", "Z-Value")

    for idx, result in enumerate(results):
        path = result['path']
        initial = path[0]
        final = path[-1]
        z_path = function(path[:, 0], path[:, 1])
```

```
        ax.plot(path[:, 0], path[:, 1], z_path,
                color=path_colors[idx % len(path_colors)],
                linewidth=2.5,
                label=f"Path {idx+1}")


        ax.scatter(initial[0], initial[1], function(initial[0], initial[1]),
                   color='black', s=80, marker='o', label=f"Start {idx+1}")


        ax.scatter(path[:, 0], path[:, 1], z_path,
                   color=path_colors[idx % len(path_colors)], s=40, alpha=0.8)


        ax.scatter(final[0], final[1], function(final[0], final[1]),
                   color='yellow', edgecolor='black', s=120, marker='X',
label=f"Converged {idx+1}")


        details_text += "{:<10} {:<20} ({: .3f}, {: .3f})       {:
.3f}\n".format(
            idx+1, result['iterations'], final[0], final[1],
function(final[0], final[1])
        )

    ax.set_title('Newton\'s Method Iteration Paths and Convergence',
fontsize=14)
    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
    ax.set_zlabel('Z-axis')
    fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)


    ax_text.text(0, 1, details_text, fontsize=10, verticalalignment='top',
family='monospace')

    plt.show()
```

This function generates a comprehensive 3D visualization of Newton's method applied to the function:

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

It displays the surface of the function and overlays the optimization paths taken from various starting points. It also includes a side text panel summarizing key information about each run.

In the final plot appear:

- Colored 3D surface of the function
- Optimization paths for each test case
- Black circles: starting points
- Yellow X markers: converged points (local or global minima)
- Side legend (text box):
  - Summarizes each test
  - Shows the number of iterations, final coordinates, and function value

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),
    ([4.5, -4.5], 0.1),
    ([-3, 3], 0.1),
    ([-3, 3], 0.5),
    ([-3, 3], 1)
]
```

- 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

```
results = []
for initial_guess, learning_rate in test_cases:
    minimum, iterations, path = newton_method_with_tracking(initial_guess,
learning_rate)
    results.append({
        'initial': initial_guess,
        'minimum': minimum,
        'iterations': iterations,
        'path': path
    })
```

16. Run Newton's method and collect results

```
results = []
for initial_guess, learning_rate in test_cases:
    minimum, iterations, path = newton_method_with_tracking(initial_guess,
learning_rate)
    results.append({
        'initial': initial_guess,
        'minimum': minimum,
        'iterations': iterations,
        'path': path
    })
```

17. Print test cases and results in the console and visualize everything.

```
print("\nNewton's Method Results Summary:\n")
print("{:<10} {:<25} {:<15} {:<30} {}".format("Test", "Initial Guess (x, y)",
"Iterations", "Converged (x, y)", "Z-Value"))

for idx, result in enumerate(results):
    initial = result['initial']
    final = result['minimum']
    iterations = result['iterations']

    if final is None:
        print("{:<10} ({: .4f}, {: .4f})      {:<15} {:<30} {}".format(
            idx + 1, initial[0], initial[1], iterations,
            "Hessian not invertible", "N/A"))
    else:
        z_value = function(final[0], final[1])
        print("{:<10} ({: .4f}, {: .4f})      {:<15} ({: .4f}, {:
.4f})      {: .4f}".format(
            idx + 1, initial[0], initial[1], iterations, final[0], final[1],
z_value))



# Visualize everything with side details
visualize(results)
```

III. Discussion

The Newton's method was implemented and tested using the function:

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

This function is **nonlinear and periodic** in both variables due to the sine and cosine components. As such, it contains **infinitely many critical points**, including both **local minima and maxima**, repeating across the domain.

To analyse the behaviour of Newton's method, we used 10 different test cases, each with a different **initial guess** and **step size (alpha)**. The program tracked all iterations and displayed the full trajectory of the optimization process on a 3D surface plot using the visualize() function.

Results:

```
Newton's Method Results Summary:

Test       Initial Guess (x, y)      Iterations      Converged (x, y)              Z-Value
1          ( 2.0000,  2.0000)      109             ( 1.5708,  3.1416)      -1.0000
2          (-4.0000, -4.0000)      211             (-4.7124, -3.1416)      -1.0000
3          ( 1.0000, -3.0000)      52              ( 1.5708, -3.1416)      -1.0000
4          (-2.0000,  4.0000)      73              (-1.5708,  3.1416)      -5.0000
5          ( 0.5000,  0.5000)      109             ( 1.5708,  0.0000)       5.0000
6          ( 0.0000,  5.0000)      0               Hessian not invertible         N/A
7          ( 4.5000, -4.5000)      108             ( 4.7124, -3.1416)      -5.0000
8          (-3.0000,  3.0000)      107             (-1.5708,  3.1416)      -5.0000
9          (-3.0000,  3.0000)      19              ( 1.5708,  3.1416)      -1.0000
10         (-3.0000,  3.0000)      4               ( 4.7124,  3.1416)      -5.0000
```

Summary of Results:

- Each test returned either a critical point or an error message if the Hessian matrix was non-invertible.
- The console output includes: initial point, number of iterations, final converged point, and the value of the function at that point.

Behaviour of Newton's Method in This Case

- **Newton's method does not guarantee a minimum**. It simply finds a **stationary point** where the gradient is close to zero.
- Because the function is periodic, there are many such points, including **minima, maxima, and saddle points**.

- Without analysing the **Hessian definiteness**, we cannot confirm whether the point is a minimum or maximum.

Observations from the Tests

- Starting from positive values like [2.0, 2.0] or [0.5, 0.5], we tend to converge near $x = \pi/2$ and $y=0$, leading to a function value of +5 (a **maximum**).
- Starting from negative values such as [-2.0, 4.0] or [-4.0, -4.0], we move toward local **minima** with values of -5 or -1.
- The **final result depends heavily on the initial guess** and on the **step size**.

Impact of Step Size (Alpha)

- We observed that even with the **same starting point**, changing only the step size led to **different final points**.
- For example, starting from [-3, 3]:
    - With $\alpha = 0.1$ → converged to a local minimum
    - With $\alpha = 0.5$ → moved to a nearby maximum
    - With $\alpha = 1.0$ → ended in a different region altogether

This proves that when **critical points are close together**, the step size directly affects **which one is reached**. This is especially common in **periodic functions**.

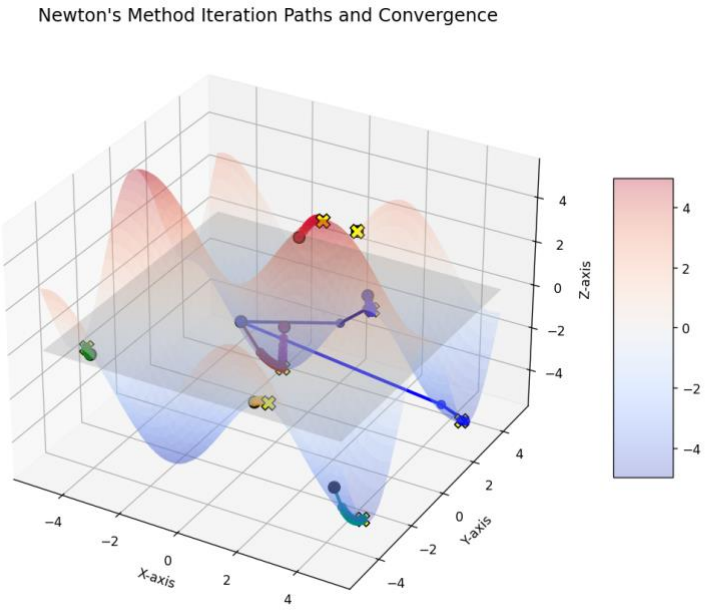
Handling of the Hessian

- Newton's method requires the **Hessian matrix to be invertible**.
- In one test case [0, 5], the Hessian was singular (determinant = 0), and the program stopped the optimization with a proper message:

  Hessian is not invertible at iteration 0 for initial [0, 5]

This condition was caught and handled cleanly in the code to prevent crashing.

Visualization




```
Newton's Method Details:
Test       Iterations          Converged (x, y)
1          109                 ( 1.571,  3.142)
2          211                 (-4.712, -3.142)
3          52                  ( 1.571, -3.142)
4          73                  (-1.571,  3.142)
5          109                 ( 1.571,  0.000)
6          0                   ( 0.000,  5.000)
7          108                 ( 4.712, -3.142)
8          107                 (-1.571,  3.142)
9          19                  ( 1.571,  3.142)
10         4                   ( 4.712,  3.142)
```

Using the visualize(results) function, we generated a 3D surface of the function and overlaid:

- **Colored paths** representing Newton's method steps
- **Black circles** for starting points
- **Yellow X markers** for final converged points
- A side panel listing iteration counts, coordinates, and final values for each test case

This visual feedback was essential to understanding how Newton's method behaves step-by-step — especially in cases where it converges to a **maximum instead of a minimum**, or when it lands in a completely different valley than expected.

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.

Key Learnings:

- Newton's method efficiently finds stationary points, but not always local minima.
- The method's result depends heavily on the starting point and step size.
- When multiple critical points are nearby (as in periodic functions), even small changes in alpha or starting position can lead to different outcomes.
- Visualizing the path of the algorithm significantly enhances understanding and analysis.

Challenges Faced:

- Making sure the Hessian matrix is invertible at each step.
- Choosing an appropriate step size that avoids overshooting or instability.
- Interpreting whether the result is a minimum or maximum, which Newton's method doesn't detect on its own.

What Can Be Improved:

- Automatically analyse the Hessian to classify each critical point (min/max/saddle).
- Implement an adaptive alpha strategy to dynamically adjust the step size.
- Add 2D contour plots with path overlays for better visual intuition.
- Explore global optimization methods to find the best possible minimum across multiple starting points.