COMPENG 3SK3 – Project 2: Newton's Method in Optimization

Professor: Dr. Xiaolin Wu

Khaled Hassan, hassak9, 400203796

Due: February 27, 2023

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is my own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario. Submitted by Khaled Hassan (hassak9, 400203796).

Pseudocode

The pseudocode used to implement each of the 3 major components of this project can be found below:

1. Implementing the Newton Optimization Algorithm:
    a. Read the 4 input matrices described below:
        i. *pts_o:* the matrix containing LiDAR sensor position data. This matrix is also referred to as *q* and contains position data for *K* evenly spaced sensors. Since *K* = 21, it is a 21×3 matrix.
        ii. *pts_markers:* the matrix containing noisy tunnel marker position data taken by each sensor. This matrix is also referred to as $\hat{p}$, and contains position data of *N* tunnel markers taken by *K* sensors. Since *N* = 100, it is a $21 \times 100 \times 3$ matrix.
        iii. *dist:* the matrix containing distance readings from each of the sensors to each of the markers using noisy position data. This matrix is also referred to as *d* and contains position data for *N* tunnel markers relative to the *K* sensors positioned on the ground. It is a 100×21 matrix.
        iv. *pts_markers_gt:* the matrix containing the ground-truth tunnel marker positions. This matrix is also referred to as *p* and is mainly used when determining the Root Mean Square Error (RMSE). It is a 100×3 matrix.
    b. Call the newton optimization function to determine the optimized coordinates.
    c. Display results to show RMSE and the number of iterations needed to optimize.

The Newton Optimization Algorithm implemented followed the theory described in lecture, and the process outlined in the project document. The implementation follows the procedure described below:

- Determine relevant matrix dimensions *N* and *K* and instantiate an empty 100×3 matrix to hold the approximated tunnel marker positions determined by applying the optimization algorithm, *p_tilda*. This matrix is referred to as $\tilde{p}$ in the project specifications.
- For each of the *N* markers, determine the appropriate *p_hat_i* matrix and *d_i* array, which are relevant to the current i^{th} marker. Also, the function defines an initialization $p^{(0)}$ to serve as the initial guess to begin the optimization process.
- The matrix of residuals, the Jacobian matrix, the gradient array and the Hessian matrix are calculated according to the equations outlined in the project specification document.
- The descent direction, *delta_p*, is calculated to determine whether to continue the optimization, and the next iteration's coordinates are calculated.
- These calculations are repeated until the norm of *delta_p* is below the defined break condition, or we hit the defined max number of iterations.

2. Fine tuning λ, the constant that controls the impact of the error term when calculating the coordinate estimates, $\tilde{p}_i$, found using the implementation of the Newton Optimization Algorithm. To implement this step, the *newton_optimization()* function defined above is

called across a range of λ values between 0 and 1. The RMSE corresponding to each λ value is stored, and the λ corresponding to the lowest RMSE is returned. This particular λ value corresponds to the optimal λ, λ_{op}.

3. Fine tuning $p^{(0)}$, the initial guess used when implementing the Optimization Algorithm. To perform this step, three different initializations were used when calling the *newton_optimization()* function: the average of the *K* measured coordinates, one of the measured coordinates (the first one) and a random vector. The number of iterations is tracked for each of the 3 different initialization approaches.

Code

The script can also be found as a file titled "3SK3_P2_hassak9_main.py" submitted alongside this report.

```
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt

import scipy.io

PATH_TO_DATA = Path(__file__).parent

ALPHA = 0.1
MAX_ITER = 200
BREAK_TERM = 1e-6

# N = 100, K = 21
# q is the positions of the LiDAR detectors.

# p is the actual locations of the tunnel markers.
# p_hat is the noisy measurement
# p_~ is the current estimate of p

# d is the distance between p_hat and q


def main():
    # read .mat files as <numpy.ndarray>'s
    pts_o = scipy.io.loadmat(f"{PATH_TO_DATA}/observation_R5_L40_N100_K21")[
        "pts_o"
    ]  # 21 x 3 q matrix. Contains the positions of each of the 21 sensors.
```

```
    pts_markers = scipy.io.loadmat(f"{PATH_TO_DATA}/pts_R5_L40_N100_K21")[
        "pts_markers"
    ]  # 21 x 100 x 3 p_hat matrix.
    # Contains noisy measurements of the tunnel marker positions by each sensor

    dist = scipy.io.loadmat(f"{PATH_TO_DATA}/dist_R5_L40_N100_K21.mat")[
        "dist"
    ]  # 100 x 21 d matrix
    # Contains distance measurements between q and p_hat

    pts_marks_gt = scipy.io.loadmat(f"{PATH_TO_DATA}/gt_R5_L40_N100_K21")[
        "pts_marks_gt"
    ]  # 100 x 3 ground truth tunnel marker position matrix. Used for error
calculation

    ##### Part 1: Newton Optimization Algorithm Implementation #####

    # lambda_val = 0.0160603015
    # lambda_val = 0.1
    # p_tilda, num_iters = newton_optimization(
    #     pts_o, pts_markers, dist, lambda_val, 1, MAX_ITER
    # )
    # error = RMSE(p_tilda, pts_marks_gt)
    # print(f"RMSE = {error: .10f} for Lambda = {lambda_val} in {num_iters}
iterations")

    ##### Part 2: Fine Tuning Lambda #####

    # optimal_LAMBDA(pts_o, pts_markers, dist, pts_marks_gt)

    ##### Part 3: Fine Tuning Initialization #####

    optimal_initialization(pts_o, pts_markers, dist, pts_marks_gt)

    return 0


def newton_optimization(
    sensor_positions,
    marker_positions,
    distance_readings,
    lambda_val,
    initialization,
    max_iterations,
):
```

```
    # pts_o, pts_markers, dist

    N_num_markers = marker_positions.shape[1]  # 100
    K_num_sensors = sensor_positions.shape[0]  # 21

    p_tilda = np.zeros(shape=(N_num_markers, 3))  # optimized sensor positions
matrix

    for i in range(N_num_markers):
        # for the ith marker
        p_hat_i = marker_positions[:, i, :]
        d_i = distance_readings[i, :]

        # initial marker position
        if initialization == 1:
            p = np.mean(
                p_hat_i, axis=0
            )  # initial marker guess is the mean of measured coords
        elif initialization == 2:
            p = p_hat_i[0, :]  # initial marker guess is the first measured
coordinates

        elif initialization == 3:
            p = np.random.rand(1, 3)  # initial marker guess is a random vector

        num_iter = 0

        # Perform optimization
        while num_iter < max_iterations:
            residuals = np.linalg.norm(p - sensor_positions, axis=1) - d_i

            Jacobian = (p - sensor_positions) / np.linalg.norm(
                p - sensor_positions, axis=1
            )[:, None]

            gradient = Jacobian.T.dot(residuals) + lambda_val * np.sum(
                2 * (p - p_hat_i), axis=0
            )
            Hessian = Jacobian.T.dot(
                Jacobian
            ) + 2 * lambda_val * K_num_sensors * np.eye(3)

            delta_p = -np.linalg.inv(Hessian).dot(gradient)

            # p for next iteration
```

```
            p = p + ALPHA * delta_p

            # Check convergence
            p_tilda[i, :] = p
            num_iter += 1

            if np.linalg.norm(delta_p) < BREAK_TERM:
                break

    return p_tilda, num_iter


def RMSE(p_tilda, p):
    return np.sqrt(np.sum((p_tilda - p) ** 2) / p.size)


def optimal_LAMBDA(sensor_positions, marker_positions, distance_readings,
gnd_truth):
    # Create array of equally spaced lambda values
    lambda_values = np.linspace(1e-3, 1, 200)

    # Initialize array to store RMSE values
    rmse_values = np.zeros_like(lambda_values)

    # Iterate over lambda values
    for j, lambda_val in enumerate(lambda_values):
        # Optimize points and calculate RMSE
        p_tilda, _ = newton_optimization(
            sensor_positions,
            marker_positions,
            distance_readings,
            lambda_val,
            1,
            MAX_ITER,
        )
        rmse_values[j] = RMSE(p_tilda, gnd_truth)

    # Identify lambda value with minimum RMSE
    lambda_min = lambda_values[np.argmin(rmse_values)]

    # Print minimum lambda value
    print(f"Minimum lambda, lambda_op = {lambda_min:.10f}")

    # Plot RMSE values against lambda values
    plt.plot(lambda_values, rmse_values)
```

```
    plt.title("RMSE vs Lambda")
    plt.xlabel("Lambda")
    plt.ylabel("RMSE")
    plt.show()


def optimal_initialization(pts_o, pts_markers, dist, pts_markers_gt):
    # plotting RMSE vs number of iterations for each of the 3 initializations
    lambda_val = 0.0160603015  # optimized from step 2

    # 1. Initialization = Average of K coordinates
    # num_ters_1 = []
    # rmse_1 = []

    # for i in range(1, MAX_ITER):
    #     p_tilda, num_iters = newton_optimization(
    #         pts_o, pts_markers, dist, lambda_val, 1, i
    #     )
    #     error = RMSE(p_tilda, pts_markers_gt)

    #     num_ters_1.append(i)
    #     rmse_1.append(error)

    # plt.plot(num_ters_1, rmse_1)
    # plt.title("RMSE vs Number of Iterations with Average Initialization")
    # plt.xlim([1, MAX_ITER])
    # plt.xlabel("Number of Iterations")
    # plt.ylabel("RMSE")
    # print(f"Mimimum RMSE reached in {num_iters} iterations")
    # plt.show()

    # 2. Initialization = First Measured Coordinate
    # num_ters_2 = []
    # rmse_2 = []

    # for i in range(1, MAX_ITER):
    #     p_tilda, num_iters = newton_optimization(
    #         pts_o, pts_markers, dist, lambda_val, 2, i
    #     )
    #     error = RMSE(p_tilda, pts_markers_gt)

    #     num_ters_2.append(i)
    #     rmse_2.append(error)

    # plt.plot(num_ters_2, rmse_2)
```

```
    # plt.title("RMSE vs Number of Iterations with Single-Point
Initialization")
    # plt.xlim([1, MAX_ITER])
    # plt.xlabel("Number of Iterations")
    # plt.ylabel("RMSE")
    # print(f"Mimimum RMSE reached in {num_iters} iterations")
    # plt.show()

    # 3. Initialization = Random Vector
    num_ters_3 = []
    rmse_3 = []

    for i in range(1, MAX_ITER):
        p_tilda, num_iters = newton_optimization(
            pts_o, pts_markers, dist, lambda_val, 3, i
        )
        error = RMSE(p_tilda, pts_markers_gt)

        num_ters_3.append(i)
        rmse_3.append(error)
    plt.plot(num_ters_3, rmse_3)
    plt.title("RMSE vs Number of Iterations with Random Initialization")
    plt.xlim([1, MAX_ITER])
    plt.xlabel("Number of Iterations")
    plt.ylabel("RMSE")
    print(f"Mimimum RMSE reached in {num_iters} iterations")
    plt.show()


if __name__ == "__main__":
    main()
```

The function called *newton_optimization()* requires to be passed input parameters *sensor_positions, marker_positions, distance_readings and lambda_val*which corresponds to the descriptions of the matrices above. The function is also passed the *initialization* and *max_iterations* parameters, which control the initialization method and the maximum number of iterations before breaking, respectively. It performs Newton Optimization by iterating over each marker's data and applying the optimization defined within the while loop, as described in the pseudocode above.

In addition to this function, the *optimal_LAMBDA()* function is used to determine λ_{op}. The function requires to the 4 matrices defined above to be passed as parameters. An array of 200 evenly spaced lambda values is defined, between 10^{-3} and 1. For each of these lambda values, the *newton_optimization()* function is called and the resulting RMSE is found and stored.

Finally, the λ value corresponding to the lowest RMSE is returned and the graph demonstrating the resulting relationship between λ and RMSE is displayed using matplotlib.

The last major function defined, *optimal_initialization()* also requires the 4 matrices to perform its required function. The function is made up of 3 main parts, each corresponding to one of the 3 defined methods of setting the initialization coordinates. Each of the 3 parts returns the expected output graph showing the relation between RMSE and the number of iterations after calling the *newton_optimization()* for different values of the maximum number of iterations, between 1 and the *MAX_ITER* constant (200).

Design Decisions

Throughout the implementation of this project, the major design decisions that were considered are outlined below:

- Python was used instead of MATLAB due to my familiarity with using it, and due to the various open source libraries that I could use to implement the objectives of this project.
- Numpy was used as it contains all the necessary methods to implement the various matrix operations that were necessary throughout this project. Matplotlib was used to display graphical results and scipy.io was used to read the given matrices, which were given in a ".mat" format.
- λ and α were initially both chosen to be 0.1. α is defined as the step size along the descent direction and is used to determine how big of a change we make from one coordinate optimization to the next. While λ was later optimized, and the optimization algorithm was implemented again using λ_{op}, α was kept the same.

Discussion

1. Algorithm Implementation

The algorithm applying Newton Optimization was implemented according to the steps outlined in the pseudocode section above. The defined break condition for the inner while loop (that controls the number of optimization iterations performed on a single sensor's data) constitutes that the descent direction, or the difference between the current and next loop optimization iteration for this marker is less than 10^{-6}, indicating that we reached a reasonably high precision. Further iterations of the optimization algorithm provide very little improvement in precision for the computational resources used.

2. Fine tuning λ

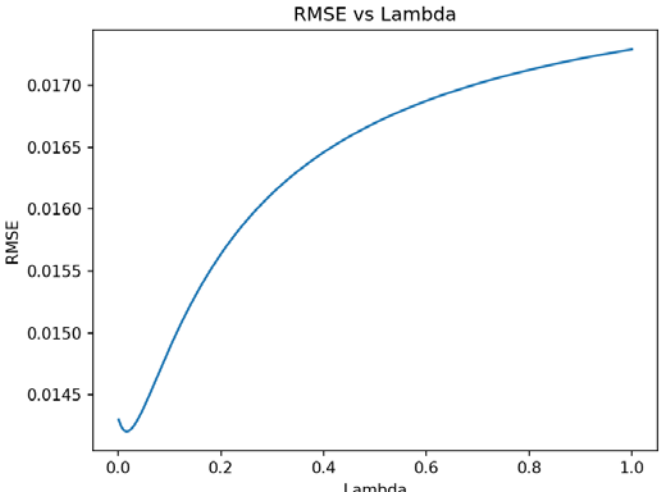

Figure 1: RMSE for Different λ Values

As can be seen in Figure 1, RMSE initially decreases for very small values of λ. As λ increases beyond approximately 0.016, RMSE increases in an almost exponential fashion. The optimal λ, $\lambda_{op} = 0.0160603015$, and that is within the expected range. This is in line with the expectation based on the definition of λ: as the influence of the error terms increases, the overall error increases beyond the optimal amount.

3. Fine tuning $p^{(0)}$

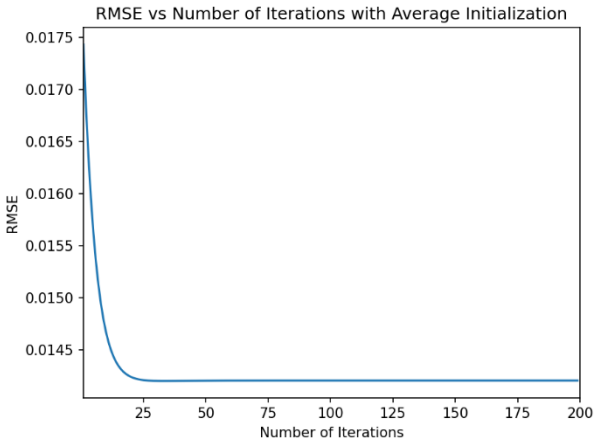

Figure 2: RMSE vs The Number of Iterations with Average Initialization

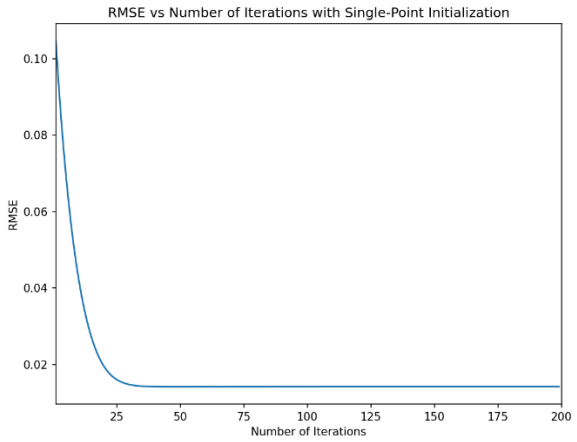

Figure 3: RMSE vs the Number of Iterations with Single-Point Initialization

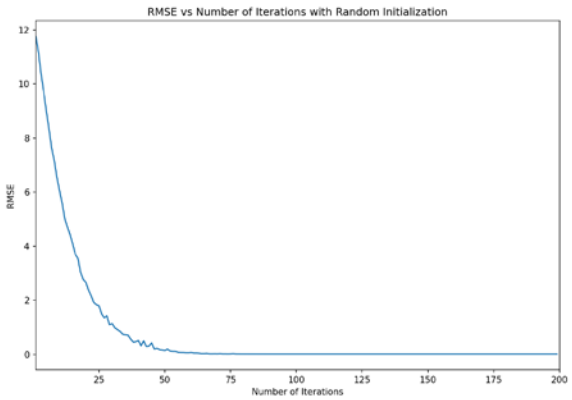

Figure 4: RMSE vs the Number of Iterations with Random Initialization

Figures 2 to 4 demonstrate the relation between RMSE and the number of iterations for each of the aforementioned 3 methods of determining the initialization point. For each of the 3 methods, the RMSE decreases exponentially as the maximum number of iterations increases until that number reaches a plateau point, which is 79, 89 and 150 iterations respectively. The first two methods, which rely on non-random starting points, demonstrate a smooth relationship, whereas using random points for the initialization introduces non-continuity to the graph. This trend is also in line with intuition: optimization starting from an average of points is better (faster) than that starting from a single point. Randomizing the initial guess yields the worst performance.