NAVARCH 568 001 WN 2023 Homework SLAM PDF

Vaishnavi Harikumar

TOTAL POINTS

100 / 100

QUESTION 1

2D Graph SLAM 50 pts

1.1 1A **10 / 10**

✓ **- 0 pts** *Correct*

- 2 pts minor mistakes

- 5 pts major mistaks

- 10 pts no submission

1.2 1B **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots

1.3 1C **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots

QUESTION 2

3D Graph SLAM 50 pts

2.1 2A **10 / 10**

✓ **- 0 pts** *Correct*

- 2 pts minor mistakes

- 5 pts major mistaks

- 10 pts no submission

2.2 2B **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots

2.3 2C **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots

Harikumar, Vaishnavi

NA 568 Mobile Robotics: Methods & Algorithms Winter 2023 – Homework – SLAM

Remark: For this assignment, all code was written in Google Colab. To read the G2o files, I added the 2D Intel dataset file and the 3D Garage dataset file into the same folder as the code files and added the path to the directory where I had saved the data files (my personal Google Drive folder) in the code for both questions. While grading, the grader will need to change (in the code) the path to the directory where the data files are saved depending on where the datafiles are stored on their computer, to run the code I have provided.

Problem 1: 2D Graph SLAM

A. *Constructed a function to read the 2D Intel dataset.*
A Python function called read_g2o_2d_intel2 was constructed which reads data from a file in the G2O format for 2D Intel dataset 2. It returns two lists, poses, and edges.

- **poses** contains tuples that represent poses of the robot, where each tuple has the form (id, x, y, theta). 'id' is an integer identifying the pose, and x, y, and theta are the position and orientation of the pose in 2D space.
- **edges** contains tuples that represent edges between pairs of poses. Each tuple has the form (id1, id2, x, y, theta, cov_mat).
- **id1** and **id2** are the ids of the poses that are connected by the edge.
- **x**, **y**, and **theta** are the position and orientation of the edge in 2D space.
- **cov_mat** is a 3x3 covariance matrix that describes the uncertainty in the edge

The function opens the specified file and reads each line, parsing the data according to the type of line. If the line starts with **VERTEX_SE2**, the function extracts the pose information and adds it to the **poses** list. If the line starts with **EDGE_SE2**, the function extracts the edge information and calculates the covariance matrix and adds it to the **edges** list. Finally, the function returns the **poses** and **edges** lists as numpy arrays.

B. *Batch Solution*:

- Creating a NonlinearFactorGraph and Values objects: This initializes an empty graph object and a values object to hold the initial estimate of the poses.
- Inserting initial pose values: This loop iterates over the set of poses and inserts them into the **init** object with their corresponding keys. **Pose2** is a GTSAM object representing a 2D pose.
- Adding a prior factor: This adds a prior factor to the graph, which is essentially a constraint that forces the first pose to be close to its initial estimate. The **prior_model** object represents the noise model of the prior, which in this case is a diagonal covariance matrix with variances of 0.3, 0.3, and 0.1 for the x, y, and theta components of the pose. **atPose2(0)** retrieves the initial estimate of the first pose.

```
!pip install gtsam
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gtsam in /usr/local/lib/python3.8/dist-packages (4.1.1)
Requirement already satisfied: numpy>=1.11.0 in /usr/local/lib/python3.8/dist-packages (from gtsam) (1.
Requirement already satisfied: pyparsing>=2.4.2 in /usr/local/lib/python3.8/dist-packages (from gtsam)
```

```
import gtsam
import numpy as np
import matplotlib.pyplot as plt
```

```
def read_g2o_2d_intel2(filename):
    """
    Reads a 2D Intel dataset 2 from G2O format and outputs poses and edges.
    Returns two lists: poses and edges.
    Each pose is a tuple (id, x, y, theta).
    Each edge is a tuple (id1, id2, x, y, theta, info).
    """
    poses = []
    edges = []
    with open(filename, 'r') as f:
        for line in f:
            data = line.strip().split()
            if data[0] == 'VERTEX_SE2':
                id = int(data[1])
                x = float(data[2])
                y = float(data[3])
                theta = float(data[4])
                poses.append((id, x, y, theta))
            elif data[0] == 'EDGE_SE2':
                id1 = int(data[1])
                id2 = int(data[2])
                x = float(data[3])
                y = float(data[4])
                theta = float(data[5])
                info = np.array([float(x) for x in data[6:]])
                info_mat = np.array([[info[0], info[1], info[2]],
                                     [info[1], info[3], info[4]],
                                     [info[2], info[4], info[5]]])
                cov_mat = np.linalg.inv(info_mat)
                edges.append((id1, id2, x, y, theta, cov_mat))
    return np.asarray(poses), np.asarray(edges)
```

```
poses, edges = read_g2o_2d_intel2(f'/content/drive/MyDrive/SLAM_HW4/input_INTEL_g2o.g2o')
```

```
<ipython-input-3-a65c1f50e049>:31: VisibleDeprecationWarning: Creating an ndarray from ragged nested se
  return np.asarray(poses), np.asarray(edges)
```

```
graph = gtsam.NonlinearFactorGraph()
init = gtsam.Values()

for pose in poses:
```

1.1 1A **10 / 10**

✓ **- 0 pts** *Correct*

- 2 pts minor mistakes

- 5 pts major mistaks

- 10 pts no submission

Harikumar, Vaishnavi

NA 568 Mobile Robotics: Methods & Algorithms Winter 2023 – Homework – SLAM

Remark: For this assignment, all code was written in Google Colab. To read the G2o files, I added the 2D Intel dataset file and the 3D Garage dataset file into the same folder as the code files and added the path to the directory where I had saved the data files (my personal Google Drive folder) in the code for both questions. While grading, the grader will need to change (in the code) the path to the directory where the data files are saved depending on where the datafiles are stored on their computer, to run the code I have provided.

Problem 1: 2D Graph SLAM

A. *Constructed a function to read the 2D Intel dataset.*
A Python function called read_g2o_2d_intel2 was constructed which reads data from a file in the G2O format for 2D Intel dataset 2. It returns two lists, poses, and edges.

- **poses** contains tuples that represent poses of the robot, where each tuple has the form (id, x, y, theta). 'id' is an integer identifying the pose, and x, y, and theta are the position and orientation of the pose in 2D space.
- **edges** contains tuples that represent edges between pairs of poses. Each tuple has the form (id1, id2, x, y, theta, cov_mat).
- **id1** and **id2** are the ids of the poses that are connected by the edge.
- **x**, **y**, and **theta** are the position and orientation of the edge in 2D space.
- **cov_mat** is a 3x3 covariance matrix that describes the uncertainty in the edge

The function opens the specified file and reads each line, parsing the data according to the type of line. If the line starts with **VERTEX_SE2**, the function extracts the pose information and adds it to the **poses** list. If the line starts with **EDGE_SE2**, the function extracts the edge information and calculates the covariance matrix and adds it to the **edges** list. Finally, the function returns the **poses** and **edges** lists as numpy arrays.

B. *Batch Solution*:

- Creating a NonlinearFactorGraph and Values objects: This initializes an empty graph object and a values object to hold the initial estimate of the poses.
- Inserting initial pose values: This loop iterates over the set of poses and inserts them into the **init** object with their corresponding keys. **Pose2** is a GTSAM object representing a 2D pose.
- Adding a prior factor: This adds a prior factor to the graph, which is essentially a constraint that forces the first pose to be close to its initial estimate. The **prior_model** object represents the noise model of the prior, which in this case is a diagonal covariance matrix with variances of 0.3, 0.3, and 0.1 for the x, y, and theta components of the pose. **atPose2(0)** retrieves the initial estimate of the first pose.

- Adding between factors: This loop iterates over the set of edges between poses and adds a between factor for each edge to the graph. **BetweenFactorPose2** represents a constraint between two poses and takes as input the keys of the two poses, the relative pose between them, and the noise model of the measurement. **Covariance** creates a noise model object with the given covariance matrix.
- Setting optimization parameters and optimizing: This sets up the Gauss-Newton optimizer and optimizes the graph with the given initial estimate and optimization parameters. The resulting **result** object contains the optimized estimate of the poses.
- Extracting optimized poses and plotting: This extracts the optimized poses from the result object using the **extractPose2** function, and plots both the original trajectory and the optimized trajectory on the same plot. The resulting plot shows the improvement in pose estimation after optimization. Figure 1 shows the plot with the optimized and unoptimized trajectories.

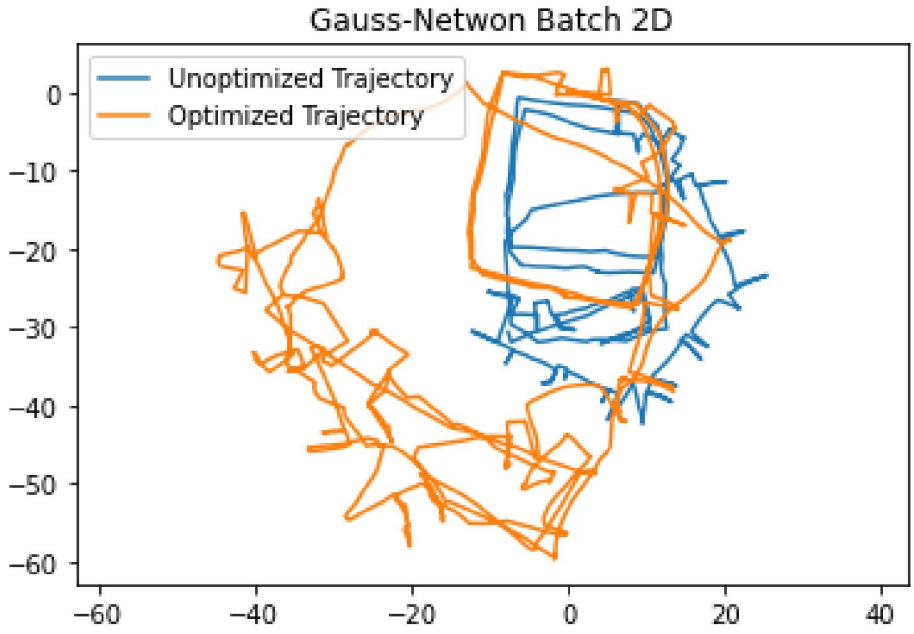
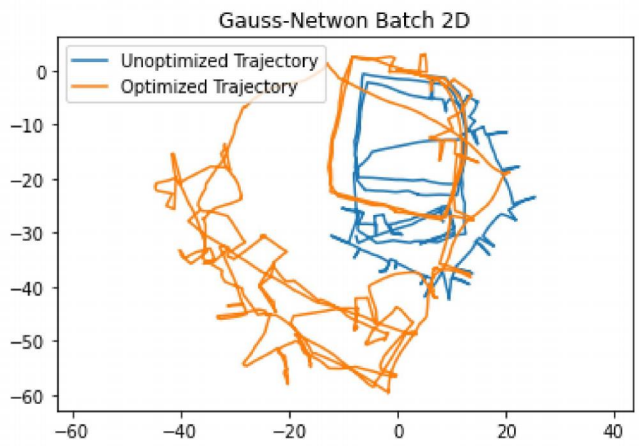


Figure 1: Gauss-Newton Optimized and Unoptimized Trajectory in 2D

C. *Incremental Solution:* To find an incremental solution, I used ISAM2 to optimize a 2D trajectory represented by a set of poses and edges, with constraints added to the NonlinearFactorGraph. ISAM2 is a factor graph-based algorithm for smoothing and mapping that is designed to work well with large, sparse graphs. The resulting optimized trajectory is plotted alongside the original trajectory in Figure 2. A description of the steps is as follows:

- Create an empty NonlinearFactorGraph to hold constraints between variables and an empty set of Values to hold variable values
- Define parameters for the ISAM2 (Incremental Smoothing and Mapping) algorithm and create an ISAM2 object with these parameters

```
        init.insert(int(pose[0]),gtsam.Pose2(pose[1],pose[2],pose[3]))

    prior_model = gtsam.noiseModel.Diagonal.Variances(np.array([0.3, 0.3, 0.1]))
    graph.add(gtsam.PriorFactorPose2(0, init.atPose2(0), prior_model))

    for edge in edges:
        model = gtsam.noiseModel.Gaussian.Covariance(edge[5])
        graph.add(gtsam.BetweenFactorPose2(edge[0],edge[1],gtsam.Pose2(edge[2],edge[3],edge[4]),model))

    params = gtsam.GaussNewtonParams()
    optimizer = gtsam.GaussNewtonOptimizer(graph, init, params)
    result = optimizer.optimize()

    opt_poses = gtsam.utilities.extractPose2(result)

    plt.plot(poses[:,1],poses[:,2])
    plt.plot(opt_poses[:,0],opt_poses[:,1])
    plt.title('Gauss-Netwon Batch 2D')
    plt.legend(['Unoptimized Trajectory', 'Optimized Trajectory'])
    plt.axis('equal')
    plt.show()
```




```
    graph = gtsam.NonlinearFactorGraph()
    init = gtsam.Values()

    params = gtsam.ISAM2Params()
    isam = gtsam.ISAM2(params)

    for ii, pose in enumerate(poses):

        if(ii==0):
            init.insert(int(pose[0]),gtsam.Pose2(pose[1],pose[2],pose[3]))
            prior_model = gtsam.noiseModel.Diagonal.Variances(np.array([0.5, 0.5, 0.1]))
            graph.add(gtsam.PriorFactorPose2(0, init.atPose2(0), prior_model))
        else:
            init.insert(int(pose[0]),result.atPose2(ii-1))

        for edge in edges:
          if(edge[1]==ii):
            model = gtsam.noiseModel.Gaussian.Covariance(edge[5])
            graph.add(gtsam.BetweenFactorPose2(edge[0],edge[1],gtsam.Pose2(edge[2],edge[3],edge[4]),model))

        isam.update(graph,init)
        result = isam.calculateEstimate()
```

1.2 1B **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots

- Adding between factors: This loop iterates over the set of edges between poses and adds a between factor for each edge to the graph. **BetweenFactorPose2** represents a constraint between two poses and takes as input the keys of the two poses, the relative pose between them, and the noise model of the measurement. **Covariance** creates a noise model object with the given covariance matrix.
- Setting optimization parameters and optimizing: This sets up the Gauss-Newton optimizer and optimizes the graph with the given initial estimate and optimization parameters. The resulting **result** object contains the optimized estimate of the poses.
- Extracting optimized poses and plotting: This extracts the optimized poses from the result object using the **extractPose2** function, and plots both the original trajectory and the optimized trajectory on the same plot. The resulting plot shows the improvement in pose estimation after optimization. Figure 1 shows the plot with the optimized and unoptimized trajectories.

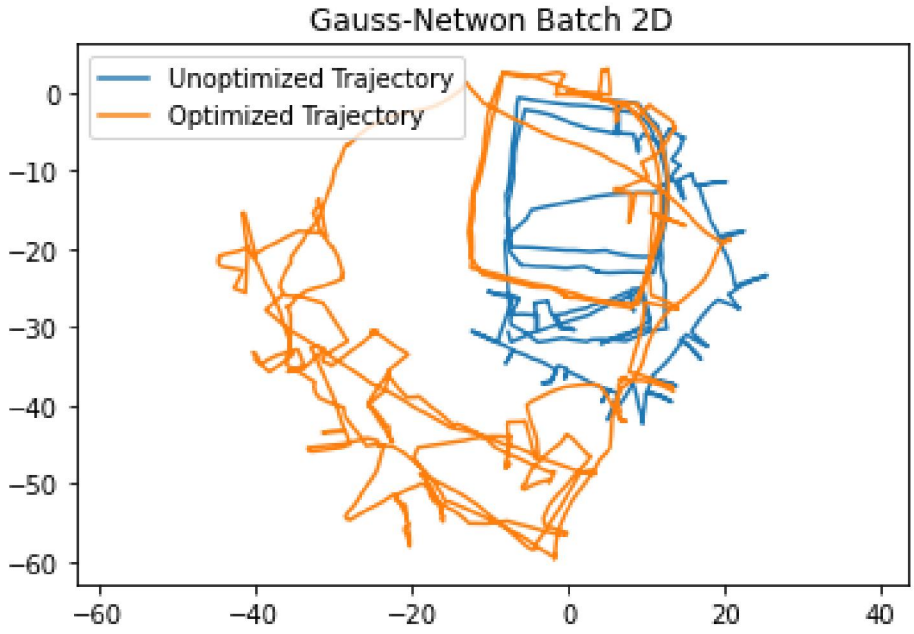


Figure 1: Gauss-Newton Optimized and Unoptimized Trajectory in 2D

C. *Incremental Solution:* To find an incremental solution, I used ISAM2 to optimize a 2D trajectory represented by a set of poses and edges, with constraints added to the NonlinearFactorGraph. ISAM2 is a factor graph-based algorithm for smoothing and mapping that is designed to work well with large, sparse graphs. The resulting optimized trajectory is plotted alongside the original trajectory in Figure 2. A description of the steps is as follows:

- Create an empty NonlinearFactorGraph to hold constraints between variables and an empty set of Values to hold variable values
- Define parameters for the ISAM2 (Incremental Smoothing and Mapping) algorithm and create an ISAM2 object with these parameters

- Loop over each pose in the trajectory and add it to the Values set, along with a prior factor if it is the first pose
- Loop over each edge in the graph (representing measurements or constraints between poses) and add a factor to the graph if the edge connects to the current pose
- Call the ISAM2 update method with the current graph and initial values, and get the updated values from the ISAM2 object
- Reset the graph and initial values for the next iteration
- Extract the optimized poses from the result Values set and plot the original and optimized trajectories

Figure 2 shows the optimized and unoptimized trajectories from the ISAM2 algorithm for the 2D case.

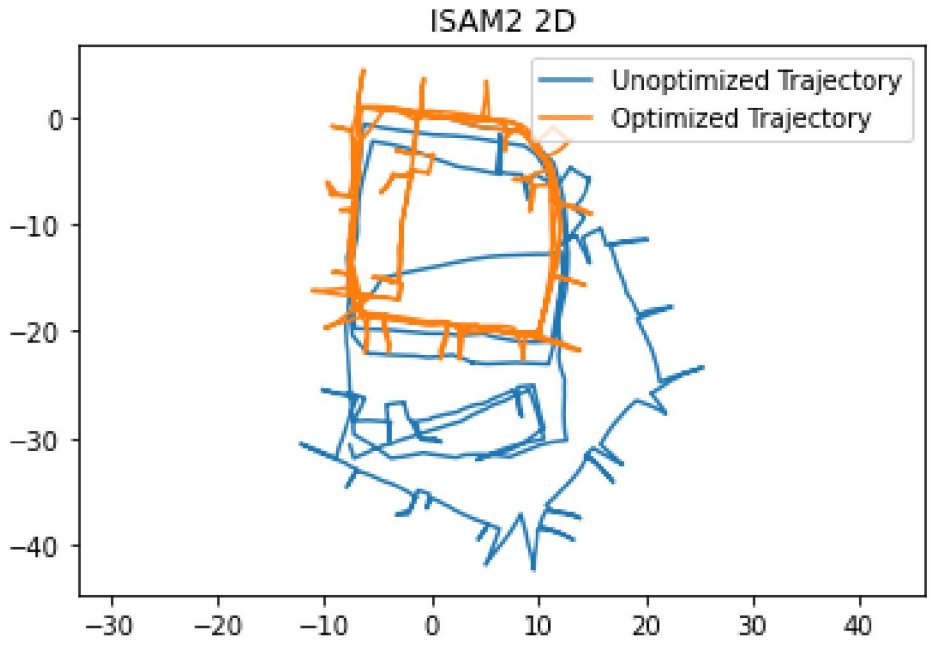


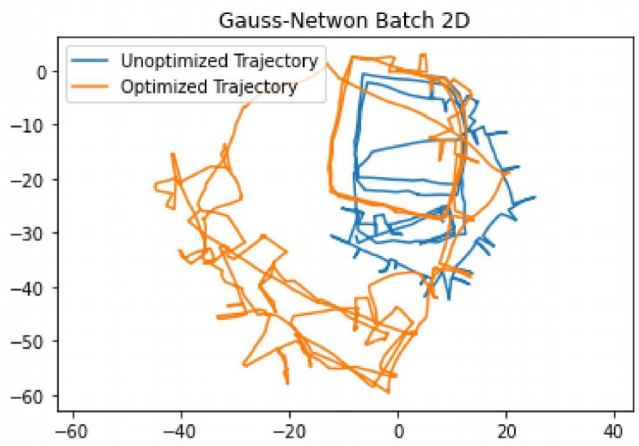
Figure 2: ISAM2 Optimized and Unoptimized Trajectory in 2D

Problem 2: 3D Graph SLAM

A. *Constructed a function to read the 3D Garage G2o dataset:*
This code defines a function **read_g2o_3d** that takes a single argument **file_path**, which is a string representing the path to a file in the **g2o** format. The function reads this file and extracts the poses and edges specified in the file.

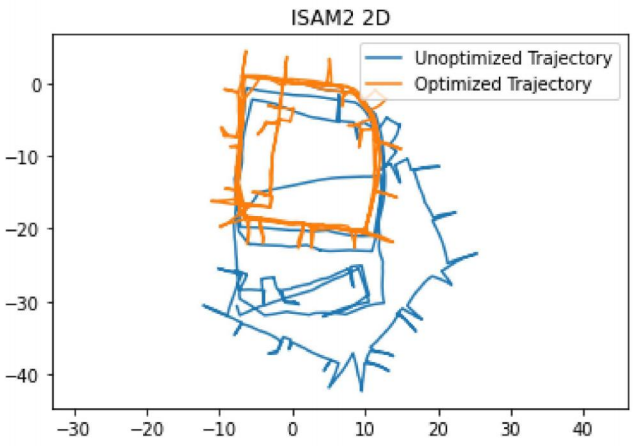
- The function starts by creating two empty lists called **poses** and **edges**. These will be used to store the parsed pose and edge information from the **g2o** file.
- The file is read line by line using the **readlines()** method. Each line is stripped of leading and trailing white space using the **strip()** method, and then split into a list of individual words using the **split()** method.

```
        init.insert(int(pose[0]),gtsam.Pose2(pose[1],pose[2],pose[3]))

    prior_model = gtsam.noiseModel.Diagonal.Variances(np.array([0.3, 0.3, 0.1]))
    graph.add(gtsam.PriorFactorPose2(0, init.atPose2(0), prior_model))

    for edge in edges:
        model = gtsam.noiseModel.Gaussian.Covariance(edge[5])
        graph.add(gtsam.BetweenFactorPose2(edge[0],edge[1],gtsam.Pose2(edge[2],edge[3],edge[4]),model))

    params = gtsam.GaussNewtonParams()
    optimizer = gtsam.GaussNewtonOptimizer(graph, init, params)
    result = optimizer.optimize()

    opt_poses = gtsam.utilities.extractPose2(result)

    plt.plot(poses[:,1],poses[:,2])
    plt.plot(opt_poses[:,0],opt_poses[:,1])
    plt.title('Gauss-Netwon Batch 2D')
    plt.legend(['Unoptimized Trajectory', 'Optimized Trajectory'])
    plt.axis('equal')
    plt.show()
```




```
    graph = gtsam.NonlinearFactorGraph()
    init = gtsam.Values()

    params = gtsam.ISAM2Params()
    isam = gtsam.ISAM2(params)

    for ii, pose in enumerate(poses):

        if(ii==0):
            init.insert(int(pose[0]),gtsam.Pose2(pose[1],pose[2],pose[3]))
            prior_model = gtsam.noiseModel.Diagonal.Variances(np.array([0.5, 0.5, 0.1]))
            graph.add(gtsam.PriorFactorPose2(0, init.atPose2(0), prior_model))
        else:
            init.insert(int(pose[0]),result.atPose2(ii-1))

        for edge in edges:
          if(edge[1]==ii):
            model = gtsam.noiseModel.Gaussian.Covariance(edge[5])
            graph.add(gtsam.BetweenFactorPose2(edge[0],edge[1],gtsam.Pose2(edge[2],edge[3],edge[4]),model))

        isam.update(graph,init)
        result = isam.calculateEstimate()
```

```
        graph.resize(0)
        init.clear()

    opt_poses = gtsam.utilities.extractPose2(result)

    plt.plot(poses[:,1],poses[:,2])
    plt.plot(opt_poses[:,0],opt_poses[:,1])
    plt.title('ISAM2 2D')
    plt.legend(['Unoptimized Trajectory', 'Optimized Trajectory'])
    plt.axis('equal')
    plt.show()
```



1.3 1C **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots

- Loop over each pose in the trajectory and add it to the Values set, along with a prior factor if it is the first pose
- Loop over each edge in the graph (representing measurements or constraints between poses) and add a factor to the graph if the edge connects to the current pose
- Call the ISAM2 update method with the current graph and initial values, and get the updated values from the ISAM2 object
- Reset the graph and initial values for the next iteration
- Extract the optimized poses from the result Values set and plot the original and optimized trajectories

Figure 2 shows the optimized and unoptimized trajectories from the ISAM2 algorithm for the 2D case.

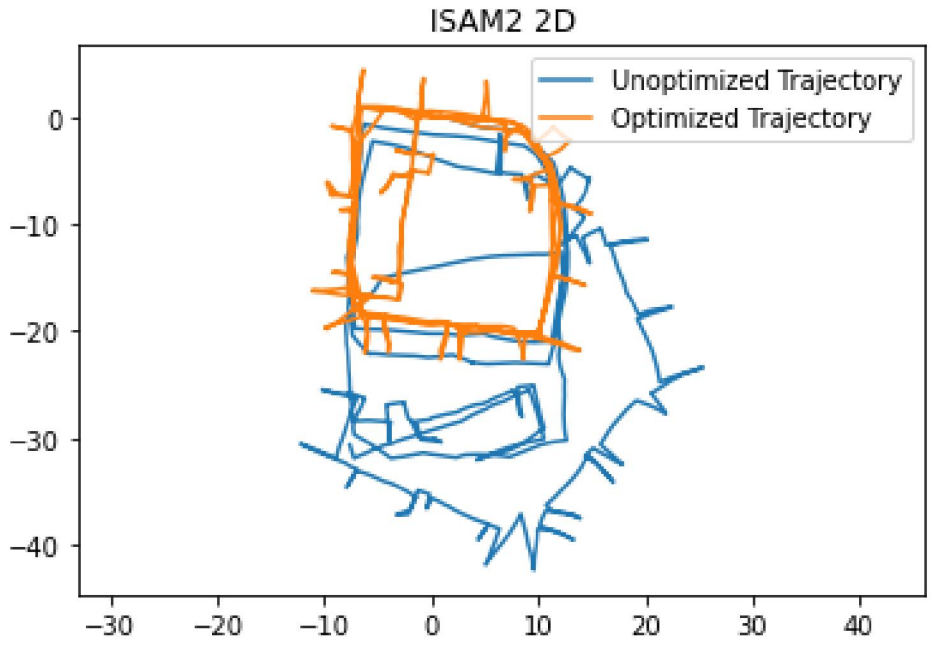


Figure 2: ISAM2 Optimized and Unoptimized Trajectory in 2D

Problem 2: 3D Graph SLAM

A. *Constructed a function to read the 3D Garage G2o dataset:*
This code defines a function **read_g2o_3d** that takes a single argument **file_path**, which is a string representing the path to a file in the **g2o** format. The function reads this file and extracts the poses and edges specified in the file.

- The function starts by creating two empty lists called **poses** and **edges**. These will be used to store the parsed pose and edge information from the **g2o** file.
- The file is read line by line using the **readlines()** method. Each line is stripped of leading and trailing white space using the **strip()** method, and then split into a list of individual words using the **split()** method.

```
!pip install gtsam
!pip install transforms3d
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gtsam in /usr/local/lib/python3.8/dist-packages (4.1.1)
Requirement already satisfied: numpy>=1.11.0 in /usr/local/lib/python3.8/dist-packages (from gtsam) (1.
Requirement already satisfied: pyparsing>=2.4.2 in /usr/local/lib/python3.8/dist-packages (from gtsam)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: transforms3d in /usr/local/lib/python3.8/dist-packages (0.4.1)
```

```
import gtsam
import numpy as np
from transforms3d.quaternions import quat2mat
import matplotlib.pyplot as plt


def read_g2o_3d(file_path):
    poses = []
    edges = []

    with open(file_path, 'r') as f:
        lines = f.readlines()

        for line in lines:
            data = line.strip().split()

            if data[0] == 'VERTEX_SE3:QUAT':
                # Parse vertex
                idx = int(data[1])
                x = float(data[2])
                y = float(data[3])
                z = float(data[4])

                R = (float(data[8]), float(data[5]),float(data[6]),float(data[7]))
                # Append pose to list
                poses.append((idx, x, y, z, R))

            elif data[0] == 'EDGE_SE3:QUAT':
                # Parse edge
                i = int(data[1])
                j = int(data[2])
                x = float(data[3])
                y = float(data[4])
                z = float(data[5])
                R = (float(data[9]), float(data[6]),float(data[7]),float(data[8]))

                info = np.zeros((6,6))
                c=10
                for ii in range(6):
                  for jj in range(6):
                    if ii>jj: continue
                    info[ii,jj] = data[c]
                    c+=1
                info += info.T - np.eye(6)*np.diag(info)

                # Compute covariance matrix
                cov = np.linalg.inv(info)
```

2.1 2A **10 / 10**

✓ **- 0 pts** *Correct*

- 2 pts minor mistakes

- 5 pts major mistaks

- 10 pts no submission

- The function then checks if the first word of the line is either **'VERTEX_SE3:QUAT'** or **'EDGE_SE3:QUAT'** using an **if-elif** statement. These correspond to the two types of nodes that can be present in a **g2o** file.
- If the line corresponds to a **'VERTEX_SE3:QUAT'**, then the code extracts the index of the node, its x, y, and z position, and a quaternion representing its orientation. This information is stored as a tuple **(idx, x, y, z, R)** and appended to the **poses** list.
- If the line corresponds to an **'EDGE_SE3:QUAT'**, then the code extracts the indices of the two nodes that the edge connects, the x, y, and z position of the edge, a quaternion representing its relative orientation, and a 6x6 information matrix. The information matrix is used to compute the covariance matrix of the edge, which is then stored along with the other edge information as a tuple **(i, j, x, y, z, R, cov)** and appended to the **edges** list.
- After all lines in the file have been read, the function returns the **poses** and **edges** lists as numpy arrays.

B. *Batch Solution*: The graph is constructed by iterating over the poses and edges and adding factors to the graph using the **add** method. A PriorFactorPose3 is added to fix the first pose to the origin of the coordinate system, and a BetweenFactorPose3 is added for each edge to constrain the relative pose between two adjacent poses.
The initial estimate of the robot poses is stored in a Values object, which associates each pose variable with a value. In this code, the initial estimate is constructed by iterating over the poses and adding a Pose3 object to the Values object using the **insert** method. Next, the code sets up a Gauss-Newton optimizer and optimizes the graph to obtain a refined estimate of the robot poses. The **GaussNewtonParams** object specifies the parameters for the optimizer. The **GaussNewtonOptimizer** object takes the Nonlinear Factor Graph, the initial estimate, and the optimization parameters as input, and returns the optimized estimate. The steps are described as follows:

- Create an empty nonlinear factor graph and an empty set of values using the gtsam library.
- Loop through a set of poses and insert each pose into the set of values
- Define a prior factor and add it to the factor graph
- Loop through a set of edges and add a between factor for each pair of poses to the factor graph
- Define optimization parameters, create an optimizer, and run optimization on the factor graph using the set of values as an initial estimate
- Extract optimized poses from the result and plot the unoptimized and optimized trajectories

Figure 3 shows the optimized and unoptimized trajectories using the Gauss-Newton Optimizer for the 3D case.

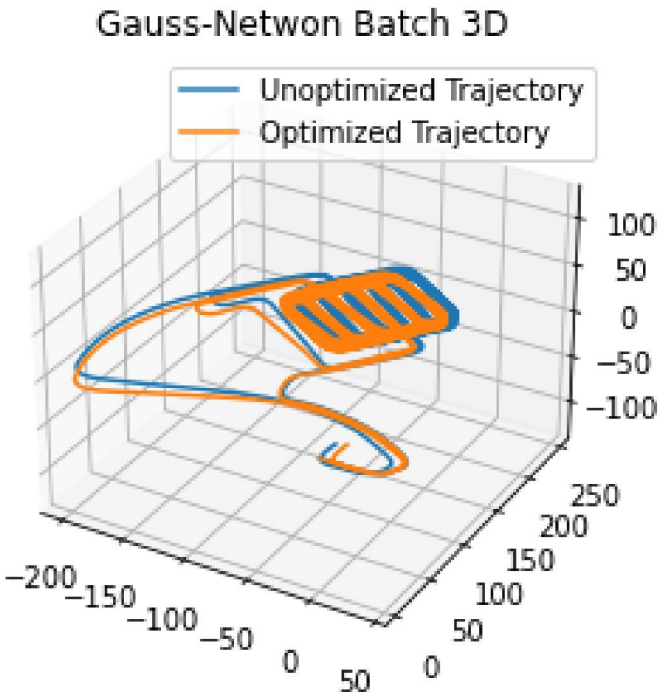


Figure 3: Gauss-Newton Optimized and Unoptimized Trajectory in 3D

C. Incremental Solution:

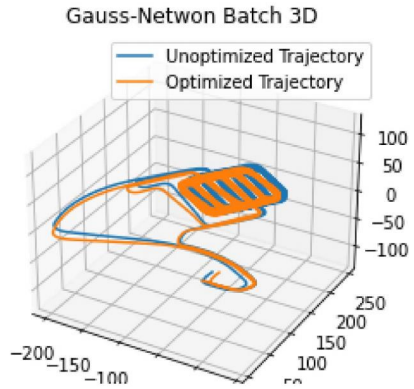
- Create an empty NonlinearFactorGraph and Values object. The graph contains PriorFactorPose3 and BetweenFactorPose3 factors that relate Pose3 variables.
- Create an ISAM2 object with default parameters. It incrementally updates the posterior distribution of the variables given the new data, making it more computationally efficient than batch optimization methods.
- Loop over each pose in the input trajectory and add a PriorFactorPose3 factor for the first pose (index 0) and a BetweenFactorPose3 factor for all subsequent poses. The PriorFactorPose3 factor sets the initial value and uncertainty for the first pose in the graph, and the BetweenFactorPose3 factors represent the constraints between consecutive poses.
- Update the ISAM2 algorithm with the current graph and initial values and calculate the optimized values. Then clear the graph and initial values for the next iteration.
- Extract the optimized poses from the Values object.

Figure 4 shows the optimized and unoptimized trajectories from the ISAM2 algorithm for the 3D case.

```
                # Append edge to list
                edges.append((i, j, x, y, z, R, cov))

    return np.asarray(poses), np.asarray(edges)



poses, edges = read_g2o_3d(f'/content/drive/MyDrive/SLAM_HW4/parking-garage.g2o')
```

```
<ipython-input-3-0c0fcc5e4820>:46: VisibleDeprecationWarning: Creating an ndarray from ragged nested se
  return np.asarray(poses), np.asarray(edges)
```

```
graph = gtsam.NonlinearFactorGraph()
init = gtsam.Values()

for pose in poses:
    r = gtsam.Rot3.Quaternion(pose[4][0], pose[4][1], pose[4][2], pose[4][3])
    t = gtsam.Point3(pose[1],pose[2],pose[3])
    init.insert(int(pose[0]),gtsam.Pose3(r,t))


prior_model = gtsam.noiseModel.Diagonal.Variances(np.array([0.3, 0.3, 0.3,0.1,0.1, 0.1]))
graph.add(gtsam.PriorFactorPose3(0, init.atPose3(0), prior_model))

for edge in edges:
    r = gtsam.Rot3.Quaternion(edge[5][0], edge[5][1], edge[5][2], edge[5][3])
    t = gtsam.Point3(edge[2],edge[3],edge[4])
    model = gtsam.noiseModel.Gaussian.Covariance(edge[6])
    graph.add(gtsam.BetweenFactorPose3(edge[0],edge[1],gtsam.Pose3(r,t),model))

params = gtsam.GaussNewtonParams()
optimizer = gtsam.GaussNewtonOptimizer(graph, init, params)
result = optimizer.optimize()

opt_poses = gtsam.utilities.extractPose3(result)

ax = plt.axes(projection='3d')
plt.plot(poses[:,1],poses[:,2],poses[:,3])
plt.plot(opt_poses[:,9],opt_poses[:,10],opt_poses[:,11])
plt.title('Gauss-Netwon Batch 3D')
plt.legend(['Unoptimized Trajectory', 'Optimized Trajectory'])

X = opt_poses[:,9]
Y = opt_poses[:,10]
Z = opt_poses[:,11]
max_range = np.array([X.max()-X.min(), Y.max()-Y.min(), Z.max()-Z.min()]).max() / 2.0

mid_x = (X.max()+X.min()) * 0.5
mid_y = (Y.max()+Y.min()) * 0.5
mid_z = (Z.max()+Z.min()) * 0.5
ax.set_xlim(mid_x - max_range, mid_x + max_range)
ax.set_ylim(mid_y - max_range, mid_y + max_range)
ax.set_zlim(mid_z - max_range, mid_z + max_range)

plt.show()
```




```
graph = gtsam.NonlinearFactorGraph()
init = gtsam.Values()

params = gtsam.ISAM2Params()
isam = gtsam.ISAM2(params)

for ii, pose in enumerate(poses):
  if ii==0:
    r = gtsam.Rot3.Quaternion(pose[4][0], pose[4][1], pose[4][2], pose[4][3])
    t = gtsam.Point3(pose[1],pose[2],pose[3])
    init.insert(int(pose[0]),gtsam.Pose3(r,t))
    prior_model = gtsam.noiseModel.Diagonal.Variances(np.array([0.3, 0.3, 0.3,0.1,0.1, 0.1]))
    graph.add(gtsam.PriorFactorPose3(0, init.atPose3(0), prior_model))
  else:
    init.insert(int(pose[0]),result.atPose3(ii-1))

  for edge in edges:
    if edge[1]==ii:
      r = gtsam.Rot3.Quaternion(edge[5][0], edge[5][1], edge[5][2], edge[5][3])
      t = gtsam.Point3(edge[2],edge[3],edge[4])
      model = gtsam.noiseModel.Gaussian.Covariance(edge[6])
      graph.add(gtsam.BetweenFactorPose3(edge[0],edge[1],gtsam.Pose3(r,t),model))

  isam.update(graph,init)
  result = isam.calculateEstimate()
  graph.resize(0)
  init.clear()

opt_poses = gtsam.utilities.extractPose3(result)

ax = plt.axes(projection='3d')
plt.plot(poses[:,1],poses[:,2],poses[:,3])
plt.plot(opt_poses[:,9],opt_poses[:,10],opt_poses[:,11])
plt.title('ISAM2 3D')
plt.legend(['Unoptimized Trajectory', 'Optimized Trajectory'])

X = opt_poses[:,9]
Y = opt_poses[:,10]
Z = opt_poses[:,11]
max_range = np.array([X.max()-X.min(), Y.max()-Y.min(), Z.max()-Z.min()]).max() / 2.0

mid_x = (X.max()+X.min()) * 0.5
mid_y = (Y.max()+Y.min()) * 0.5
mid_z = (Z.max()+Z.min()) * 0.5
ax.set_xlim(mid_x - max_range, mid_x + max_range)
ax.set_ylim(mid_y - max_range, mid_y + max_range)
ax.set_zlim(mid_z - max_range, mid_z + max_range)
```

2.2 2B **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots

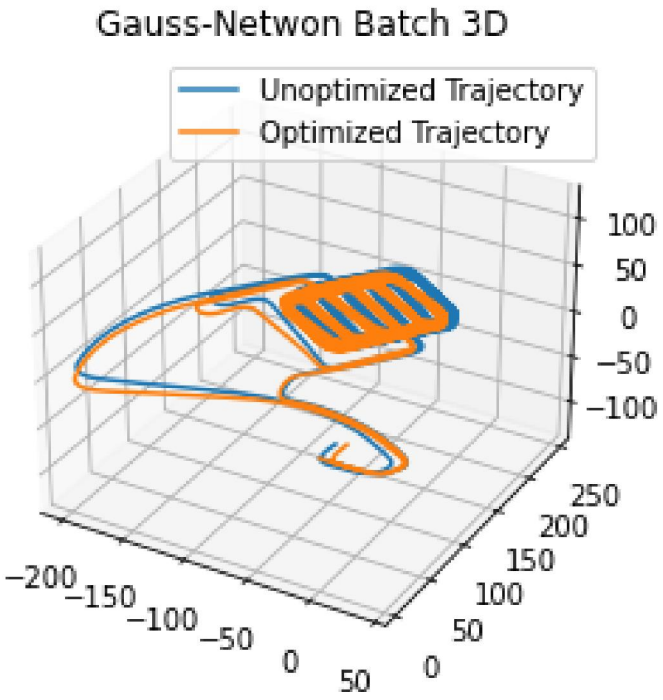


Figure 3: Gauss-Newton Optimized and Unoptimized Trajectory in 3D

C. Incremental Solution:

- Create an empty NonlinearFactorGraph and Values object. The graph contains PriorFactorPose3 and BetweenFactorPose3 factors that relate Pose3 variables.
- Create an ISAM2 object with default parameters. It incrementally updates the posterior distribution of the variables given the new data, making it more computationally efficient than batch optimization methods.
- Loop over each pose in the input trajectory and add a PriorFactorPose3 factor for the first pose (index 0) and a BetweenFactorPose3 factor for all subsequent poses. The PriorFactorPose3 factor sets the initial value and uncertainty for the first pose in the graph, and the BetweenFactorPose3 factors represent the constraints between consecutive poses.
- Update the ISAM2 algorithm with the current graph and initial values and calculate the optimized values. Then clear the graph and initial values for the next iteration.
- Extract the optimized poses from the Values object.

Figure 4 shows the optimized and unoptimized trajectories from the ISAM2 algorithm for the 3D case.

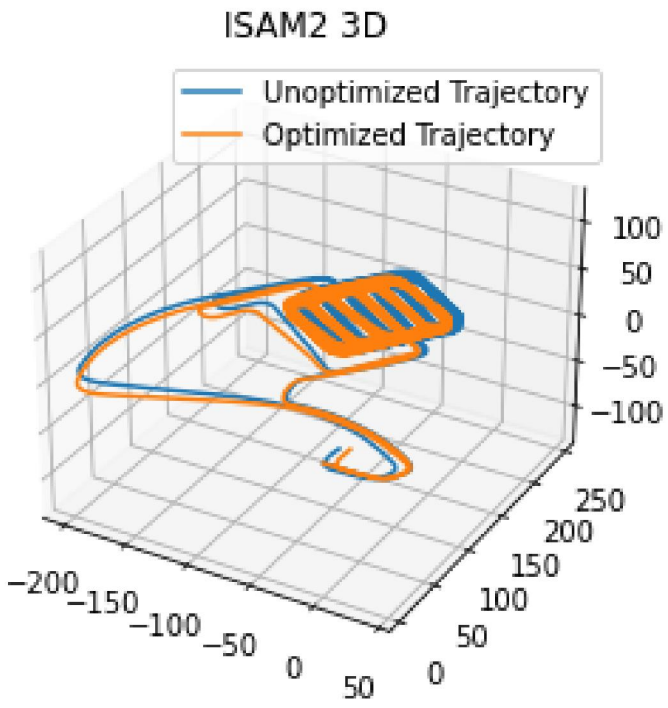
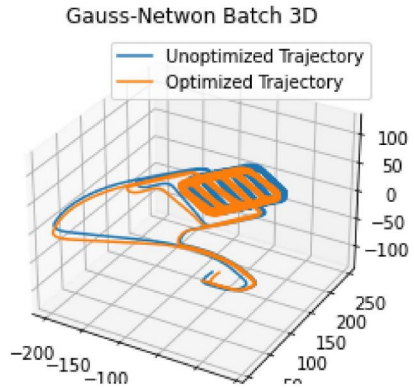
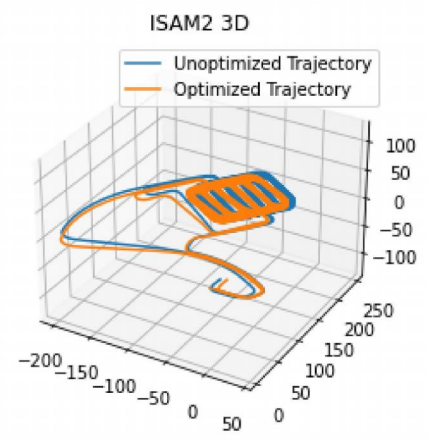


Figure 4: ISAM2 Optimized and Unoptimized Trajectory in 3D




```
graph = gtsam.NonlinearFactorGraph()
init = gtsam.Values()

params = gtsam.ISAM2Params()
isam = gtsam.ISAM2(params)

for ii, pose in enumerate(poses):
  if ii==0:
    r = gtsam.Rot3.Quaternion(pose[4][0], pose[4][1], pose[4][2], pose[4][3])
    t = gtsam.Point3(pose[1],pose[2],pose[3])
    init.insert(int(pose[0]),gtsam.Pose3(r,t))
    prior_model = gtsam.noiseModel.Diagonal.Variances(np.array([0.3, 0.3, 0.3,0.1,0.1, 0.1]))
    graph.add(gtsam.PriorFactorPose3(0, init.atPose3(0), prior_model))
  else:
    init.insert(int(pose[0]),result.atPose3(ii-1))

  for edge in edges:
    if edge[1]==ii:
      r = gtsam.Rot3.Quaternion(edge[5][0], edge[5][1], edge[5][2], edge[5][3])
      t = gtsam.Point3(edge[2],edge[3],edge[4])
      model = gtsam.noiseModel.Gaussian.Covariance(edge[6])
      graph.add(gtsam.BetweenFactorPose3(edge[0],edge[1],gtsam.Pose3(r,t),model))

  isam.update(graph,init)
  result = isam.calculateEstimate()
  graph.resize(0)
  init.clear()

opt_poses = gtsam.utilities.extractPose3(result)

ax = plt.axes(projection='3d')
plt.plot(poses[:,1],poses[:,2],poses[:,3])
plt.plot(opt_poses[:,9],opt_poses[:,10],opt_poses[:,11])
plt.title('ISAM2 3D')
plt.legend(['Unoptimized Trajectory', 'Optimized Trajectory'])

X = opt_poses[:,9]
Y = opt_poses[:,10]
Z = opt_poses[:,11]
max_range = np.array([X.max()-X.min(), Y.max()-Y.min(), Z.max()-Z.min()]).max() / 2.0

mid_x = (X.max()+X.min()) * 0.5
mid_y = (Y.max()+Y.min()) * 0.5
mid_z = (Z.max()+Z.min()) * 0.5
ax.set_xlim(mid_x - max_range, mid_x + max_range)
ax.set_ylim(mid_y - max_range, mid_y + max_range)
ax.set_zlim(mid_z - max_range, mid_z + max_range)
```

```
plt.show()
```



Colab paid products - Cancel contracts here

✓ 17s completed at 2:02 AM

2.3 2C **20 / 20**

✓ **- 0 pts** *Correct*

- 2 pts minor mistake in drawing

- 4 pts no description on graph construction process or its parameters

- 4 pts minor error in codes

- 10 pts No plots