Graph Based SLAM with Visual Odometry Extensions

Awadhut Thube(athube), Mandeep Singh(msingh2), Poorva Agrawal(poorvaa), Tarang Shah(tarangs)

Project Report for CMU 16833 - Robot Localization and Mapping - Spring 2021

***Abstract*— Building a map of an environment while simultaneously localizing a mobile robot in the map is a well studied problem in the field of robotics. Traditionally, SLAM methods have relied on variants of the Bayes filter algorithm for estimating the state of the system. More recently, SLAM systems have evolved to implement graph based techniques to localize a robot in an environment while generating a consistent map of the environment. In the former case, the problem is modelled as a state estimation problem where only the latest state of the system is taken into consideration. In the latter case, the full trajectory of the mobile robot is refined using the entire measurement set. In this project, we have implemented an entire graph-based SLAM pipeline using 2D laser data and odometry information. We have also extended our implementation to accommodate visual data in our SLAM frontend.** [1]

I. INTRODUCTION

Simultaneous localization and mapping, or SLAM, is an important functionality for robotic systems that need to operate in environments where they rely merely on measurements from onboard sensors and cannot use any external references like GPS, and do not have a map of their surroundings. SLAM techniques are commonly classified into two categories: filtering and smoothing.

Filtering techniques, such as Kalman filters and particle filters, are online methods where the map and pose estimates are updated incrementally as new measurements are recorded. Smoothing techniques on the other hand build the entire robot trajectory first and optimize later using all the measurements recorded. Graph-based SLAM is an example of the latter and the main focus of our project. It was first proposed by Lu et al. [1] in 1997, and today forms the state-of-the-art techniques in SLAM by ensuring speed and accuracy of the system by utilizing tricks such as sparse linear algebra.

The SLAM problem is most commonly divided into two major parts, the frontend, and the backend. The frontend involves the creation of a factor graph which includes the poses of a mobile robot along its trajectory as nodes. The edges in the graph are derived from the observations made by the robot while moving along its trajectory. These edges serve as constraints between the robot poses and as the robot observations are noisy, these constraints may or may not be the exact constraints between the poses.

As soon as the robot reaches a previously visited position in the environment, it adds new constraints between the relevant poses. After such a graph is constructed, the problem is converted into an optimization problem which finds the configuration of the nodes, which best satisfies all the constraints in the graph. This causes the entire trajectory of the robot to be optimized while considering all the previously made measurements.

[1]Code available at https://github.com/HobbySingh/Graph-SLAM and https://github.com/awadhutthube/slam_project

II. RELATED WORK

To date, many algorithms have been proposed for both the frontend and the backend of the SLAM problem. For the backend, we closely follow the tutorial by Grisetti et al. in [2] to develop our own implementation of a pose graph optimizer. We also referred to [3] to get insight into the programming aspects of developing an efficient and fast optimizer, and [4] for the Python binding of the g2o framework. A summarized formulation of the graph SLAM backend is also available at [5]. For the frontend which utilized 2D laser and odometry data to build a pose graph, we referred to [6] for the overall structure of the pose graph construction algorithm and utilized the ICP implementation available at [7] to optimize the odometry-based poses used to form the vertices of the pose graph and to perform scan matching to detect loop closures. In order to incorporate visual data in our pose graph estimation, we referred to the technique laid out at [8]. The following sections detail the theory behind the optimization techniques and algorithms used, and the details involved in the implementation of the back-end and front-end submodules.

III. APPROACH

A. Dataset Description

1) 2D Laser and Odometry Logs: We use the 2D laser and odometry data provided in CARMEN format [9]. These include the raw odometry information. This dataset also includes manually added "relations", which serve as the ground truth. For our problem, we only use the raw data and evaluate the performance visually.

We first started by using the preprocessed g2o data from [10] to implement and test our backend graph optimization system. Once this was achieved, we moved on to using the raw data in [9] and building our own pose graph from the laser and odometry information. We used the Intel Research Lab data from the repository for the custom frontend implementation.

The raw data is in the CARMEN format where we have one entry per line. The first entry in the line could either be `ODOM` or `FLASER` (or `ROBOTLASER1`). The lines with `ODOM` include the odometry information and the lines with `FLASER` include the laser range finder information. The data is in 2D format, i.e. 2D position(x, y) and 1 axis of rotation(θ)

2) KITTI Dataset: In order to compute visual odometry, we have used sequences from the KITTI Visual Odometry dataset [11]. There are a total of 22 sequences, out of which we provide results on 5 of them. Each sequence contains pair of stereo images, LIDAR data, and ground truth poses. We will be using images from a single camera (left) and perform monocular SLAM with pose graph optimization.

B. Graph SLAM for 2D Datasets

The overall pipeline can be divided into the Frontend and Backend. The Frontend is responsible for creating the Pose Graph and adding additional contraints(edges) such as loop closures. The backend on the other hand is responsible for using the graph and optimizing it using a least squared approach. We can see an general schematic view of the pipeline on Fig. 1.

*1) **Backend**:* The backend in a Graph based SLAM system involves the Pose Graph representation and the optimization system. We run the backend optimization procedure each time we have a new vertex or edge added to the graph, including both regular vertex/edge additions and loop closure edges. We will discuss when these are added in the section III-B.2.

A graph consists of a set $\mathscr{V}$ with N vertices where each vertex v_i is associated with a pose $\mathbf{p}_i$. It also has a set ε of M edges where each edge e_j corresponds to a measurement $\mathbf{z}_j \in \mathscr{Z}$ which has an associated information matrix Ω_j. The residual corresponding to a measurement is given by the difference of the actual measurement and the measurement estimated from the poses $\hat{\mathbf{z}}_j(\mathbf{p_1}, \mathbf{p_2}, ...\mathbf{p_N})$. Graph SLAM aims to maximize the likelihood of poses given the measurements, i.e.

$$\underset{\mathbf{p_1},\mathbf{p_2},\cdots\mathbf{p_N}}{\arg\max}\ p(\mathbf{p_1}, ...\mathbf{p_N}|\mathscr{Z})$$

Using Bayes' rule we know that

$$p(\mathbf{p_1}, ...\mathbf{p_N}|\mathscr{Z}) \propto p(\mathscr{Z}|\mathbf{p_1}, ...\mathbf{p_N})$$

We model the measurements to have independent Gaussian noise and with zero mean and covariance Ω^{-1}

Thus,

$$p(\mathbf{z_j}|\mathbf{p_1}, ...\mathbf{p_N}) = \eta_j \exp(-\mathbf{e_j}^T \Omega_j \mathbf{e_j})$$

Thus, the expression to be maximised yields:

$$\underset{\mathbf{p_1},\mathbf{p_2},\cdots\mathbf{p_N}}{\arg\max}\ p(\mathbf{p_1}, ...\mathbf{p_N}|\mathscr{Z}) = \underset{\mathbf{p_1},\mathbf{p_2},\cdots\mathbf{p_N}}{\arg\min} \sum_{i=1}^{M} \mathbf{e_j}^T \Omega_j \mathbf{e_j}$$

Hence, we want to minimize

$$\chi^2 = \sum_{i=1}^{M} \mathbf{e_j}^T \Omega_j \mathbf{e_j}$$

Let $\mathbf{x_i}$ be a compact representation of pose $\mathbf{p_i}$, and let $\mathbf{x} = [\mathbf{x_1}, ..., \mathbf{x_N}]^T$. Then, this problem is solved iteratively with

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta\mathbf{x}$$

We first linearize the residuals about $\mathbf{x}^k$ and substitute this back into χ^2_{k+1} which yields:

$$\chi^2_{k+1} = \chi^2_k + 2\mathbf{b}^T\Delta\mathbf{x}^k + (\Delta\mathbf{x}^k)^T\mathbf{H}\Delta\mathbf{x}^k$$

$$\mathrm{H} = \sum_{e_j\in\varepsilon} \left(\frac{\partial(\mathbf{x}^k+\Delta\mathbf{x}^k)}{\partial\Delta\mathbf{x}^k}\right)^T \left(\frac{\partial\mathbf{e}_j(\mathbf{x}^k+\Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k+\Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=0}\right)^T \Omega_j \left(\frac{\partial\mathbf{e}_j(\mathbf{x}^k+\Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k+\Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=0}\right) \frac{\partial(\mathbf{x}^k+\Delta\mathbf{x}^k)}{\partial\Delta\mathbf{x}^k}$$

$$\mathbf{b}^T = \sum_{e_j\in\varepsilon} [\mathbf{e}_j(\mathbf{x}^k)]^T \Omega_j \left(\frac{\partial\mathbf{e}_j(\mathbf{x}^k+\Delta\mathbf{x}^k)}{\partial(\mathbf{x}^k+\Delta\mathbf{x}^k)}\bigg|_{\Delta\mathbf{x}^k=0}\right) \frac{\partial(\mathbf{x}^k+\Delta\mathbf{x}^k)}{\partial\Delta\mathbf{x}^k}$$

The update rule is thus given by:

$$\Delta\mathbf{x}^k = -\mathbf{H}^{-1}\mathbf{b}$$

The update rule is applied to the pose until convergence. We now discuss how we implement the frontend system using the raw inputs.

*2) **Frontend**:* The frontend system mainly involves reading the sensor and odometry data and updating the pose graph. We directly parse the raw data files that we discussed previously. These files include odometry and laser sensor measurements at multiple time stamps.

While parsing, we read all the values from the files. But, to ensure we perform the graph optimization in a reasonable time, we want to keep the size of our graph relatively smaller. One of the best ways to do this is to reduce the number of vertices in our graph (and as a result, reducing the number of edges).

To ensure we have a small number of vertices, we only add a vertex to the graph if the robot has performed "significant" motion. This "significant" motion is based on the odometry measurements. Though the odometry is a quick and easy way to calculate motion, it could be susceptible to errors if the odometry is known to be noisy. Currently the parameters are set to a displacement greater than $0.4m$ in the position or an angle change greater than 10°(0.2 rad).

We also have a 2D ICP routine which takes two sets of laser measurements and aligns them. This allows us to calculate a transformation between any two positions, as long as we have the laser data from that position. For the case where we add a new vertex, we use this ICP routine to find the relative transformations between the poses of the vertices. This transformation is stored in the edge connecting these 2 vertices.

*3) **Loop Closure**:* For any graph-based SLAM problem, loop closure detection is essential as it gives us valuable constraints that help greatly optimize the graph. A naive way to detect if the current location was "seen" previously, (i.e. we have closed a loop) would be to use the laser measurement at the current location and search through all the previous locations and check if the laser measurements match. As we increase the number of points, we can easily see that this will make the problem exponential and slow down the system. We use 2 methods to ensure that loop closure checks don't slow the system down. Firstly, instead of checking for loop closures at all new vertices, we only check for loop closures after a few vertices have been added to the graph. For example, in our current implementation,

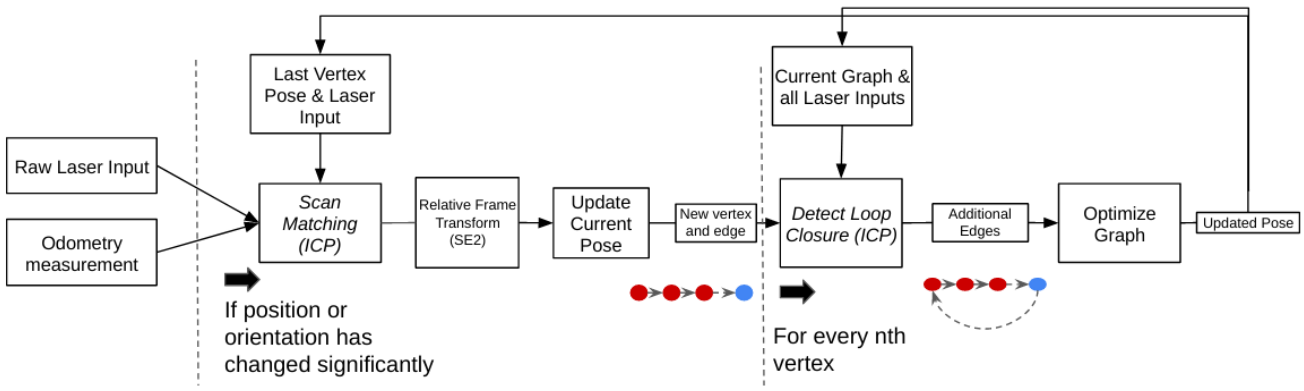


Fig. 1: 2D Graph SLAM Pipeline using Laser Sensors and Odometry inputs

we only check for loop closures after every 10 vertices. This ensures we don't waste computation in redundant locations. The second method use is to use a K-d Tree to reduce the search space for loop closures. At every loop closure check, we take the current vertex positions and create a K-d tree. Using this new K-d Tree, we search for past vertices that are near the current pose. Once we find the N closest vertices, we perform the previously discussed ICP routine and find the mean distance between the laser sensor detected points. If this mean distance is within a certain threshold, we can say that a loop closure was achieved following which we add an edge between the current and the past vertex. Now, since we are using poses (vertices) from the latest optimized graph, there is a high chance that our system is using less noisy poses. But, since we are still dependent on odometry between the current and previous pose, there is a chance that our search might not be accurate if the last odometry was very noisy (which would have resulted in a noisy edge).

*4) **Implementation Details**:* As part of the project, we first started with implementing a Graph SLAM backend. This included data structures and objects for storing the poses and odometry information in the vertex and edges of the graph. The core Graph SLAM and Pose Graph idea is based on [2]. This gave us a good theoretical understanding of the overall system. The custom optimization code and data structures are also inspired from [3], this repository was referred to for the backend code only.

For the frontend, we need to parse the log files and extract the relevant pose and odometry information at each time stamp. The poses represent the vertices of the graph. We also need to ensure the appropriate transforms are calculated and added as edges to the graph. Also, as an implementation detail, we need to ensure the computation occurs in reasonable time, especially since we need to optimize a large graph. We referred to [6] for the general structure and also used the ICP routine from that repository. We replaced the g2o based graph optimization and incorporated our own graph and graph optimizer into this frontend structure.

The overall logic and flow is as depicted in the pipeline and flow chart in Fig. 1.

C. Incorporating Visual Odometry

In order to extend the laser-based 2D pose graph optimization method, we further implemented visual odometry-based pose graph creation and optimization. This implementation uses the KITTI dataset described in Section III-A.2. There are 3 main parts of the implementation as described below.

Algorithm 1 Pseudocode for the Visual Odometry Pose Graph Optimization

First Frame - Initialize Pose Graph
$prev\_Rt = I$
$PoseGraph$.add_vertex($prev\_Rt, set\_origin = True$)
$ref\_points = SIFTDetector(frame1)$
Second Frame - Initialize Feature Tracker
$ref\_points, curr\_points = LKTracker(frame1, frame2, ref\_points)$
$curr\_Rt = get\_pose(ref\_points, curr\_points)$
$PoseGraph.add\_edge(curr\_Rt, prev\_Rt)$
Subsequent Frames - Loop Closure and Pose Graph Optimization
for $frame\ in\ FRAMES$ **do**
  $ref\_points, curr\_points$ = LKTracker($frame1, frame2$)
  $curr\_Rt$ = get_pose($ref\_points, curr\_points$)
  $PoseGraph$.add_edge($curr\_Rt, prev\_Rt$)
  if $ref\_points \leq 1500$ **then**
    ref_points = SIFT_Detector($frame$)
  end
  $found\_loop, rel\_pose$ = check_loop_closure($frame$) **if** $found\_loop$ **then**
    $PoseGraph$.add_loop_constraint($curr\_Rt, rel\_pose$)
    $PoseGraph$.optimize()
  end
end
return *PoseGraph*

*1) **Visual Odometry**:* The visual odometry frontend is built using 2D-2D feature correspondences between relative frames. We utilize only left camera images from the stereo pair provided in the KITTI dataset and perform monocular visual odometry. We use SIFT feature detector and Lukas Kanade tracker to track the features across frames. We

Fig. 2: SIFT Feature Matching

compute the relative pose between frames by using the Essential Matrix obtained after performing RANSAC. Using the computed poses we construct our Pose Graph. The overview of the algorithm is given in 1.

*2) **Loop Closure**:* In order to simplify the scope of the project, we use ground truth poses to detect loop closures. We use pre-computed adjacency list which maps every frame to those frames whose ground-truth poses are within 6m radius of each other. At every frame we extract the top match between neighbouring frames using SIFT features 2. If the length of the feature set is greater than a threshold for the best match we compute the relative pose between those frames. The *get_pose*() function also returns the number of inliers after reprojecting the features to the reference frame, which we use to decide whether to consider the relative pose constraint as successful loop closure or not.

D. Pose Graph Optimization

For visual odometry based SLAM we create pose graph structure based on g2o library's python wrapper [4]. In order to optimize the pose graph, we use Sparse Optimizer from g2o library.

Now we will discuss the results of above illustrated approaches for both inertial and LIDAR odometry and visual odometry based posed graph optimization.

IV. Results

A. 2D Graph SLAM

For the 2D Graph SLAM problem our main goal was to test the implementation and qualitatively evaluate the system. We aimed to create the graph and optimize it at multiple steps.

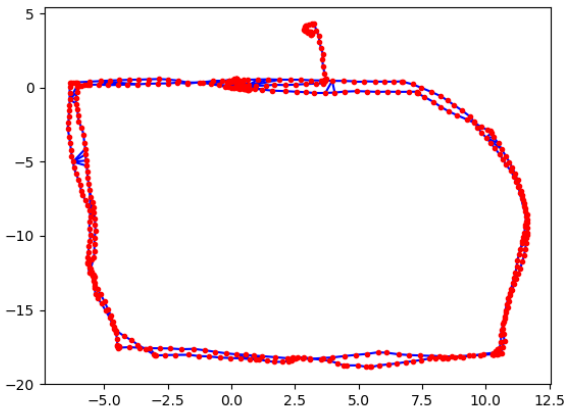


Fig. 5: 2D Pose Graph visualization for the Intel Dataset

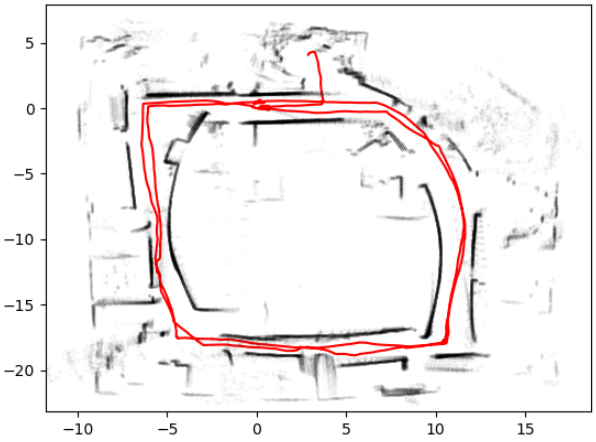


Fig. 6: 2D Map from the initial few frames of the Intel Dataset

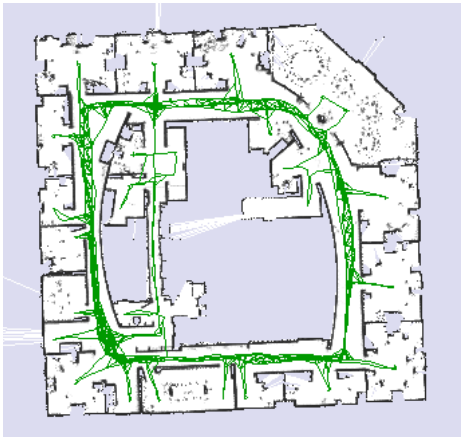


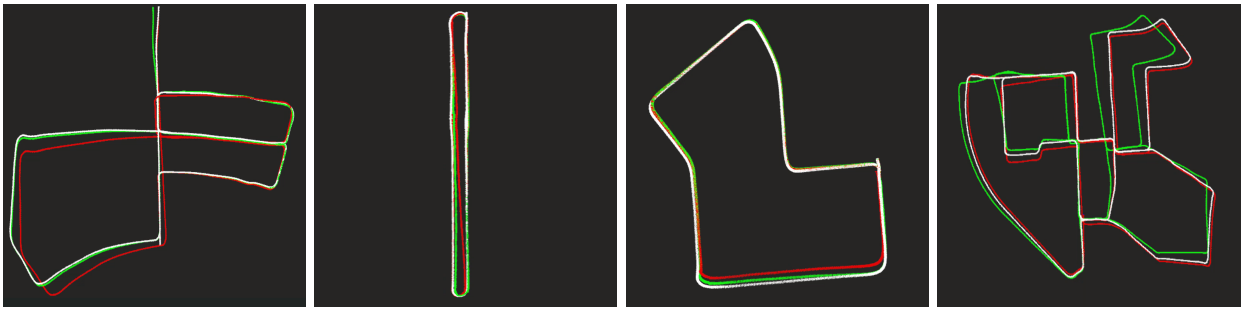
Fig. 7: Ground truth map for the Intel Dataset

In Fig. 5 we can see the vertices as red dots and edges in blue. Besides the simple edges between sequential poses we can also see some edges that are added during loop closures. Since we don't use the manually provided ground truth edges, the number of loop closures are quite few, but we can still see a good number of matches and the resulting map in Fig. 6 matches the ground truth in Fig. 7.

B. Visual Odometry Pose Graph Optimization

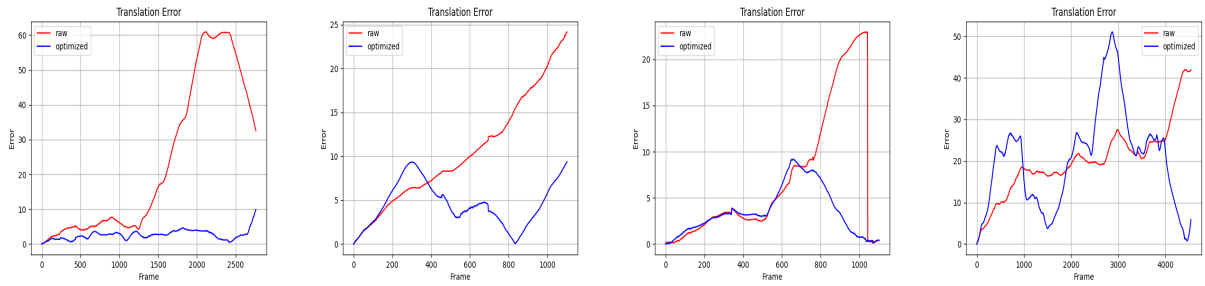
We tested our approach on 4 different sequences from the KITTI dataset. The optimized pose graph visualizations for each sequence can be seen in Figure 3. The white trajectory is the sequence of ground truth poses. The red trajectory denotes the pose graph with raw poses which only includes relative pose constraints and no loop closures. The green trajectory denotes the optimized pose graph satisfying the loop closure constraints. Qualitatively we can see that the green trajectory more closely matches with the white trajectory as compared to red trajectory.

Moreover we also present the quantitative translation errors for the raw pose and optimized poses with the ground truth poses in Figure 4. We can observe from the blue plot



(a) (b) (c) (d)

Fig. 3: Optimized Pose Graphs (a) Sequence 5 (b) Sequence 6 (c) Sequence 7 (d) Sequence 0



(a) (b) (c) (d)

Fig. 4: Translation Errors of optimized and raw poses with ground-truth poses (a) Sequence 5 (b) Sequence 6 (c) Sequence 7 (d) Sequence 0

that the optimized pose errors grow similar to raw poses until the loop closure. Once the loop closure constraint is added and graph is optimized we see significant decrease in the error compared to raw poses.

C. *Difficult Cases*

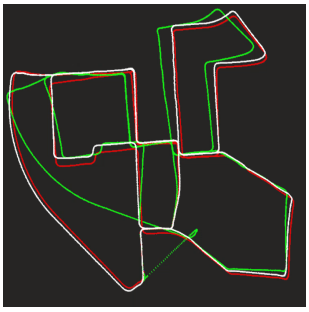


Fig. 8: Drifted Optimized Pose Graph

We also showcase a difficult test case which demonstrates the effect of inaccurate pose constraint on the graph optimization. As observed in Figure 4d, we can see that the optimized pose errors were decreasing till 2000^{th} step but due to a bad edge there is sudden increase in the error after that. We can also observe this visually in Figure 8 that the optimized graph is highly drifted from the ground truth trajectory. Interestingly our approach is able to recover successfully after a few final accurate loop closing constraints are added. The final optimized graph can be seen in Figure 3d.

V. Conclusion and Future Work

We successfully implemented a pose graph-based backend for the SLAM problem and utilized it to optimize the pose graph generated by the frontend using 2D laser scan and odometry data. We also implemented a custom frontend that could generate a pose graph using visual odometry data and used the g2o framework to optimize the same. In the future, we would like to incorporate meaningful information matrices in the frontend. In the 2D case, this corresponds to using sensor uncertainty while performing ICP for scan matching and updating the pose instead of the default identity matrices. For images, we can scale the identity matrix using the number of matched features across frames. In the 3D case, we can also obtain a raw map by performing rectified stereo reconstruction at each pose.

References

[1] F. Lu and E. Milios, "Globally consistent range scan alignment for environment mapping," *Autonomous Robots*, vol. 2, no. 4, p. 33–349, 1997.

[2] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based slam," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[3] "Python graph slam reference repository(backend)." https://github.com/JeffLIrion/python-graphslam.

[4] "g2o python wrapper." https://github.com/uoip/g2opy.

[5] "Graph slam formulation." https://github.com/AtsushiSakai/PythonRobotics/blob/master/SLAM/GraphBasedSLAM/graphSLAM_formulation.pdf.
[6] "Python graph slam reference repository(frontend)." https://github.com/goktug97/PyGraphSLAM.
[7] "Icp implementation for frontend." https://github.com/ClayFlannigan/icp.
[8] "Monocular visual slam." https://github.com/sakshamjindal/Monocular-MiniSLAM.
[9] "Repository of laser and odometry data in carmen format at uni freiburg." http://ais.informatik.uni-freiburg.de/slamevaluation/index.php.
[10] "g2o format datasets by luca carlone." https://lucacarlone.mit.edu/datasets/.
[11] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.