Yet Another Implementation of Graph-based SLAM

Duy Tran
Khoury College of Computer Science
Roux Insititute at Northeastern University
Portland, Maine
tran.duy3@northeastern.edu

***Abstract*—**
***Index Terms*—Robotics, SLAM, Factor Graph, Backend**

I. Introduction

Graph-based SLAM is currently the state-of-the-art SLAM backend technique for optimizing pose graphs. This paper will discuss in further detail an implementation of the main algorithm introduced in [1] for 2D graphs and tested on Luca Carlone's 2D Pose Graph Optimization Dataset at lucacarlone.mit.edu/datasets [2]. The implementation can be found at github.com/rael346/graph-slam

II. Related Works

There have been many implementations of graph-based SLAM algorithm over the years in C++ like Ceres [3], which is a general non-linear optimization library, and g2o [4], which is a non-linear optimization library with a focus on SLAM-related optimization problem.

This project, similar to g2o, will implement the non-linear optimization algorithm with a focus on solving 2D pose graphs discussed in [1]. The project was also inspired by the work of Irion [5], implementing the algorithm entirely in Python.

III. Background

The crux of the algorithm is not in the optimization algorithm, which was discussed in detail in Algorithm 2 of [1], but the manifold representation of a robot pose. Therefore, this section will go into detail the manifold representation and the math behind it.

A. Special Euclidean Group SE(2)

A robot pose in space is described by its position and orientation with respect to an origin and axes. For 2D space, the position is a set of $(x, y) \in \mathbb{R}^2$ coordinates and a rotation angle $\theta \in [-\pi, \pi]$. This representation can also be understood as a transformation from the origin using the transformation matrix

$$T = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} \tag{1}$$

with R being the rotation matrix of the transformation

$$R = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{2}$$

and $t = (x, y)$ being the translation of the transformation.

Thus we have a compact representation (x, y, θ) and the full representation in matrix form from equation (1). The compact representation is mostly used to save memory since it requires only 3 floats compared to the full representation which requires 9 floats.

The full representation also belongs to a special group called Special Euclidean Group, which is mathematically defined as

$$\mathrm{SE}(2) = \left\{ T = \begin{pmatrix} R(\theta) & t \\ 0 & 1 \end{pmatrix} \mid R \in \mathrm{SO}(2), t \in \mathbb{R}^2 \right\} \tag{3}$$

$$\mathrm{SO}(2) = \left\{ R \in \mathbb{R}^{2\times 2} \mid RR^T = I, \det(R) = 1 \right\} \tag{4}$$

B. SE(2) Operations

This section will layout the main operations for poses in SE(2) form required to implement the optimization algorithm, which was discussed in length in [6]. The difference is that [6] developed the full mathematical framwork for Special Euclidean Group while this section only discuss the relevant operations for the algorithm. Furthermore, this section serves as a somewhat more comprehensive tutorial to the math dicussed in [1] for the 2D case, since [1] only show the resulting equation without much elaboration.

For the rest of the section, we will refer to $p = (x, y, \theta)$ as the compact representation of the robot pose and T as the full representation described in equation (1).

a) *Inverse:*

$$T^{-1} \overset{\text{full}}{=\!=} \begin{pmatrix} R(\theta)^\top & -R(\theta)^\top t \\ 0 & 1 \end{pmatrix} \overset{\text{compact}}{=\!=\!=} \begin{pmatrix} -R(\theta)^\top t \\ -\theta \end{pmatrix} \tag{5}$$

This can be verified by $TT^{-1} = I$. Note that $R(\theta)^\top = R(-\theta)$. While this operation wasn't used directly in the implementation, it is a useful one for later equation derivation.

b) *Composition:*

$$\begin{aligned} p_1 \oplus p_2 &= T_1 T_2 \\ &\overset{\text{full}}{=\!=} \begin{pmatrix} R(\theta_1 + \theta_2) & R(\theta_1)t_2 + t_1 \\ 0 & 1 \end{pmatrix} \\ &\overset{\text{compact}}{=\!=\!=} \begin{pmatrix} R(\theta_1)t_2 + t_1 \\ \theta_1 + \theta_2 \end{pmatrix} \end{aligned} \tag{6}$$

This operation also wasn't used directly in the implementation, but is the basis for the next operation.

c) *Inverse Composition:*

$$\begin{aligned} p_1 \ominus p_2 &= T_2^{-1} T_1 \\ &\overset{\text{full}}{=\!=} \begin{pmatrix} R(\theta_1 - \theta_2) & R(\theta_2)^\top (t_1 - t_2) \\ 0 & 1 \end{pmatrix} \\ &\overset{\text{compact}}{=\!=\!=} \begin{pmatrix} R(\theta_2)^\top (t_1 - t_2) \\ \theta_1 - \theta_2 \end{pmatrix} \end{aligned} \tag{7}$$

This operation is the crux of the optimization problem, since it is used in the error calculation. Note that the original tutorial [1] didn't use this operation, but instead use the inverse and composition of poses. The inverse composition is the same mathematically, but is represented more succinctly as a single operation. [6] used this operation extensively to describe the error calculation.

C. Error Function

In graph-based SLAM, the vertices are the poses of the robot at different timestamp, whereas the edges represent the diference between the two poses p_i and p_j. This difference can also be understood as the pose of the robot at time j relative to the pose at time i.

The edges also contain the measured difference z_{ij} gotten from the frontend of the SLAM system (usually processed from the raw sensor measurements). Thus the difference between the expected difference and the measured difference is defined as

$$e_{ij} = \left(p_j \ominus p_i\right) \ominus z_{ij} \tag{8}$$

Note that since the poses inverse composition is not commutative, the ordering of the poses here is actually important. This equation also corresponds with equation 30 in [1], which describes the error function in its full derivation.

D. Jacobians

This is another important section since the optimization algorithm relies heavily on the Jacobians calculated from the pose operations.

a) *Jacobians of Inverse Composition:*

With $\Delta t = R(\theta_2)^\top (t_1 - t_2)$ and $\Delta\theta = \theta_1 - \theta_2$ from equation (7) we have

$$\frac{\partial(p_1 \ominus p_2)}{\partial p_1} = \begin{pmatrix} \frac{\partial(\Delta t)}{\partial t_1} & \frac{\partial(\Delta t)}{\partial(\theta_1)} \\ \frac{\partial(\Delta\theta)}{\partial t_1} & \frac{\partial(\Delta\theta)}{\partial(\theta_1)} \end{pmatrix} = \begin{pmatrix} R(\theta_2)^\top & 0 \\ 0 & 1 \end{pmatrix} \tag{9}$$

$$\begin{aligned} \frac{\partial(p_1 \ominus p_2)}{\partial p_2} &= \begin{pmatrix} \frac{\partial(\Delta t)}{\partial t_2} & \frac{\partial(\Delta t)}{\partial(\theta_2)} \\ \frac{\partial(\Delta\theta)}{\partial t_2} & \frac{\partial(\Delta\theta)}{\partial(\theta_2)} \end{pmatrix} \\ &= \begin{pmatrix} -R(\theta_2)^\top & \frac{\partial(R(\theta)^T)}{\partial(\theta_2)}(t_1 - t_2) \\ 0 & -1 \end{pmatrix} \end{aligned} \tag{10}$$

b) *Jacobians of Error Function:*

Deriving the Jacobians of the error function (8) is simply using the chain rule

$$A_{ij} = \frac{\partial e_{ij}}{\partial p_i} = \frac{\partial e_{ij}}{\partial\left(p_j \ominus p_i\right)} \frac{\partial\left(p_j \ominus p_i\right)}{\partial p_i} \tag{11}$$

$$B_{ij} = \frac{\partial e_{ij}}{\partial p_j} = \frac{\partial e_{ij}}{\partial\left(p_j \ominus p_i\right)} \frac{\partial\left(p_j \ominus p_i\right)}{\partial p_j} \tag{12}$$

c) *Jacobians of Manifold:*

$$M_i = \frac{\partial p_i \boxplus \Delta p_i}{\partial \Delta p_i} = I \tag{13}$$

$$M_j = \frac{\partial p_j \boxplus \Delta p_j}{\partial \Delta p_j} = I \tag{14}$$

Author Note: this is the only part I couldn't understand since the original tutorial [1] didn't show any derivation. [6] didn't use this Jacobians at all so it is hard to verify the math here. It probably has something to do with the Lie Algebra and the Lie Group (SE(2) is a Lie Group) but I didn't have enough time to flesh this out.

IV. Implementation Details

A. Dataset Format

The dataset use the g2o format, the description of which can be found on Luca Carlone's dataset website. Essentially, each line of the file is either

- The poses of the graph in SE(2) and has the format `VERTEX_SE2 id x y theta`
- The edge of the graph and has the format `EDGE_SE2 IDout IDin dx dy dtheta i11 i12 i13 i22 i23 i33`
  - `IDout` and `IDin` are the IDs of the poses that the edges connected
  - `dx dy dtheta` is the measured difference between the two poses, aka z_{ij}
  - `i11 i12 i13 i22 i23 i33` is the 3x3 information matrix, which is triangular

B. Objects

After defining the core operations and Jacobians above, the implementation is fairly straight forward.

- A `SE2` object that implements the above operations and Jacobians
- An `Edge` object that represents the edge of the graphs and stores the infomation matrix and the meaasured difference
- A `Graph` object that parses the dataset in g2o format and stores the list of poses in `SE2` form as well as the edges of the graph, and implements Algorithm 2 in the original tutorial [1]

C. Libraries

This was implemented using Python 13 using Numpy for the matrix operations, Scipy for sparses matrix solver, and Matplotlib for visualizing the pose graph before and after optimization.

V. Results

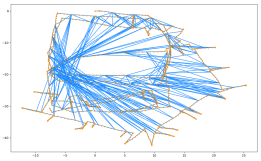

Fig. 1: Intel Research Lab Before Optimization

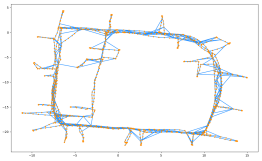

Fig. 2: Intel Research Lab After Optimization

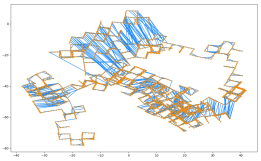

Fig. 3: M3500 Before Optimization

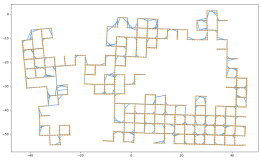

Fig. 4: M3500 After Optimization

For the first three datasets (Intel, MIT, M3500), which contains relatively less noise, the algorithm converges to the global minima.

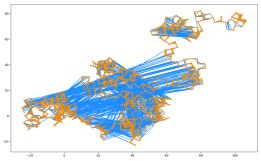

Fig. 5: M3500a Before Optimization

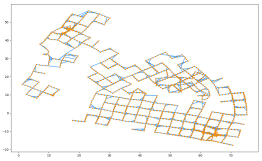

Fig. 6: M3500a After Optimization

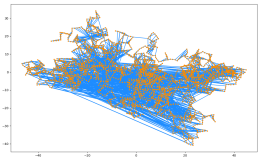

Fig. 7: M3500b Before Optimization

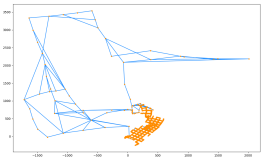

Fig. 8: M3500b After Optimization

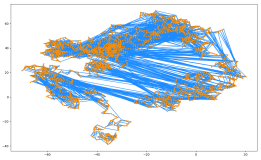

Fig. 9: M3500c Before Optimization

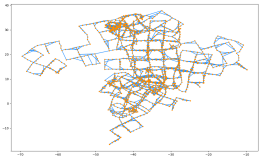

Fig. 10: M3500c After Optimization

For the variation of M3500 with extra noises to the relative orientation, the algorithm struggled to find the global minima and either converges to a local minima (Fig. 6) or not converges at all (Fig. 8).

All datasets were run with 20 max iteration, except for M3500b, which it still fails to converges. Interestingly, M3500c manages to converges (Fig. 10) to a local minima even though it has more noises than M3500b.

VI. Discussion

A. Future Works

Similar to gradient descent, the non-linear optimization approach presented here is still subjected to local minima convergence. Possible solutions to this problem includes apply learning rate during the pose update step so that the algorithm can slowly converges to the minima instead of bouncing around it; using an optimizer like Adam or AdamW to "nudge" the algorithm to a global minima. However, such solutions might not be necessary when taking into account the whole SLAM system. Within a SLAM pipeline, pose graph optimization is usually only triggered when the robot completes a loop (this process is called loop closure). Therefore, the pose graph is usually being optimized incrementally instead of all at once as implemented here, which can reduce the overall noise and graph complexity.

Another direction is implementing SE(3) poses, which represents the robot pose in 3-dimension space. However, the 3D case is significantly more complex than the 2D case because of the rotation being quarternions instead of a single angle θ.

Testing the algorithm by incorporating it with a frontend like Visual SLAM could be an interesting next step, since it show whether the algorithm is robust enough in real-time SLAM.

B. Implementation Challenges

One of the biggest challenges implementing this algorithm is verifying if the math is correct. The original tutorial [1] didn't show the full derivation of the pose operators, error function and jacobians, whereas [6] uses a completely different notation system and usually showed the full form of the result element-by-element matrices. This is fairly error prone since the full matrices are usually large, so a single plus/minus can derail the whole algorithm. This is the main reason why I dedicated a whole section about SE(2) above, since it significantly simplify the math by using the block matrices R and t instead of going element-by-element. In addition, having a reference implementation like Irion's [5] was very helpful in debuging the algorithm.

References

[1] 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, doi: 10.1109/MITS.2010.939925.

[2] L. Carlone and A. Censi, "From Angular Manifolds to the Integer Lattice: Guaranteed Orientation Estimation With Application to Pose Graph Optimization," *IEEE Transactions on Robotics*, vol. 30, no. 2, pp. 475–492, 2014, doi: 10.1109/TRO.2013.2291626.

[3] S. Agarwal, "ceres-solver: A large scale non-linear optimization library." Accessed: Apr. 19, 2025. [Online]. Available: https://github.com/ceres-solver/ceres-solver

[4] R. Kümmerle, "g2o: A General Framework for Graph Optimization." Accessed: Apr. 19, 2025. [Online]. Available: https://github.com/RainerKuemmerle/g2o

[5] J. Irion, "python-graphslam: Graph SLAM solver in Python." Accessed: Apr. 19, 2025. [Online]. Available: https://github.com/JeffLIrion/python-graphslam

[6] J. L. Blanco-Claraco, "A tutorial on SE(3) transformation parameterizations and on-manifold optimization," 2022, [Online]. Available: https://arxiv.org/abs/2103.15980