Pose-Graph SLAM Implementation

Kevin Robb

Contents

1 Introduction & Motivation

This project is an addition to my EKF-SLAM personal project, in which I developed a custom simulator for a 2D robot to respond to motion commands and publish landmark detections. This project is built on ROS Noetic in Ubuntu 20.04, and I have previously implemented nodes to perform EKF-SLAM, UKF-Localization, and UKF-SLAM.

For this project, I will re-frame the process in terms of manifold-valued quantities, generalize my architecture with an abstract `Filter` class, and implement pose-graph SLAM. This work serves as my final project for MATH-7223: Riemannian Optimization at Northeastern University.

2 EKF-SLAM Recap

The filters I have previously implemented are online Bayesian filters, so they were designed to run alongside the simulation, estimating the current state at each iteration, and throwing out all previous estimates. This "state" consisted of the current vehicle pose estimate as well as the current position estimate of all landmarks that have been observed at least once. These are manifold-valued quantities, but I represented them as a vector of floats,

$$x_t := \{\, x_{veh}, y_{veh}, \theta_{veh}, x_{l,1}, y_{l,1}, \ldots, x_{l,M}, y_{l,M} \,\}^T .$$

The true state representation in these types of filters is a multi-dimensional Gaussian distribution, so we track this state vector x_t as the mean, in addition to a covariance matrix P_t which carries a notion of uncertainty in our state estimate.

The usual Kalman filtering process then proceeds, with motion commands sent to the robot being used to propagate forward the state ("prediction step") and landmark measurements being used to verify our estimate ("update step"). This process is very memory-efficient, since we need only store the current state estimate, and some temporary values used for the prediction and update stages to compute the next state. However, since we don't keep any reference to past poses, we cannot directly utilize loop closures, which are an important method for reducing uncertainty and error in SLAM applications.

3 Pose-Graph SLAM (PGS) Overview

In pose-graph SLAM, we will store and estimate the entire vehicle pose history, rather than just the current pose. We do this by constructing a so-called "pose-graph", consisting of nodes and connections. The vehicle pose at each timestep will be a new node, and each landmark will be a node. We don't create new nodes for landmarks on re-detection, since they are static in the environment. An example of such a pose-graph is shown in Figure 1.

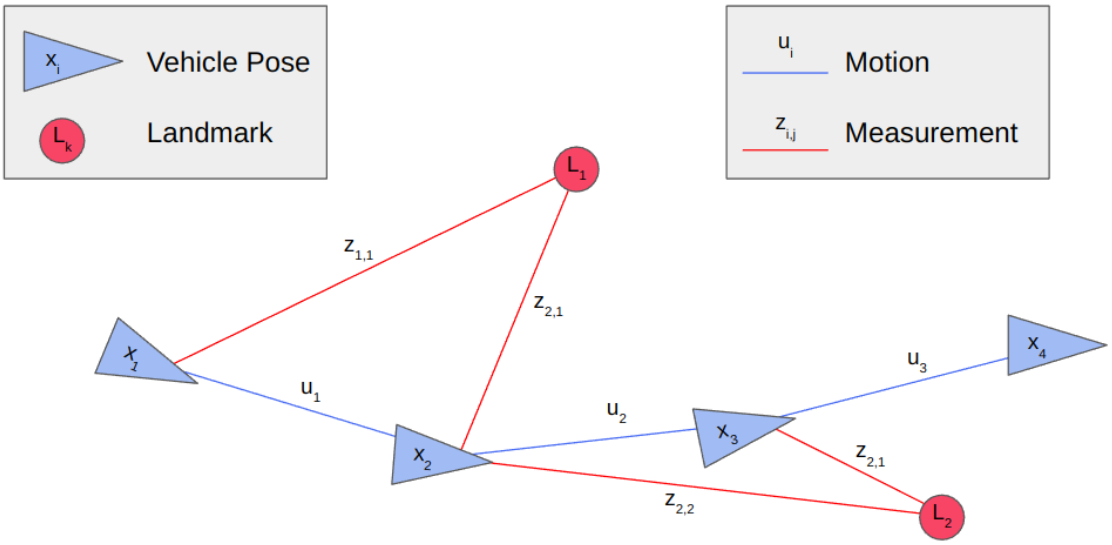


Figure 1: An example pose-graph that could have been constructed by our filter.

Our goal is to estimate:

- the full vehicle pose history, $x_i \quad \forall i \in [0, t]$
- all landmark positions, $L_k \quad \forall k \in [1, M]$

by minimizing error in all relative transforms:

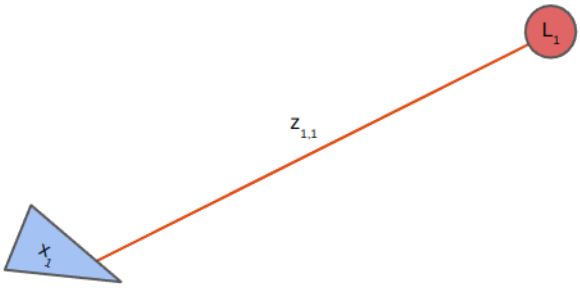
- motions, $u_i \quad \forall i \in [0, t-1]$
- measurements, $z_{i,j} \quad \forall i \in [0, t-1], \quad j \in [1, M] \quad$ (not all connections will exist)

Note that we could wait until the entire pose-graph has been built to run optimization, or we could solve every iteration. The latter will give better results but will of course increase runtime per iteration, especially as the number of nodes and connections in the graph rises as the run continues.

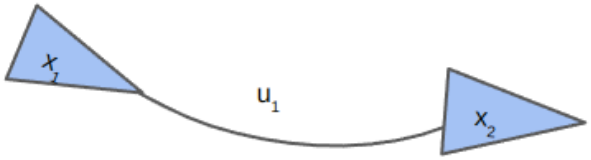
A crucial difference in this method compared to EKF-SLAM is the frequent loop-closures created by re-detections of landmarks. In EKF-SLAM, the measurements on each iteration are technically separate, even if the same landmark is detected that has been seen before; re-detections do help to reduce uncertainty and improve the result, but the Bayesian filtering process denies the possibility of directly relating two vehicle poses at non-adjacent timesteps. In the pose-graph, many vehicle poses will be related in the graph by connections to a shared landmark, and optimizing the graph will use this to estimate both poses more accurately.

4 High-Level: Building the Pose-Graph

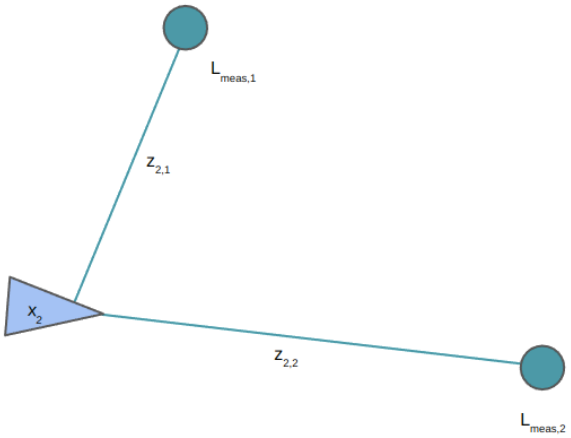
Figure 2 gives a high-level overview of a single iteration adding to the pose graph.



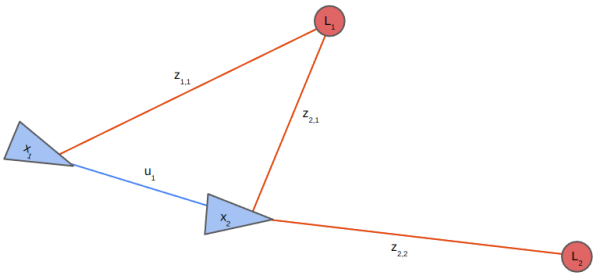
(a) Existing pose-graph at iteration $t = 1$.



(b) Motion commanded to move from $t = 1$ to $t = 2$.



(c) Landmark measurements at $t = 2$ from new vehicle pose.



(d) New pose graph structure at $t = 2$.

Figure 2: High-level overview of a single iteration's changes to the pose-graph.

Step (c) may require data association to relate a measured landmark to a particular unique landmark that may or may not have been detected previously. If we do not assume a measurement comes with that landmark's ID, we can use a simple distance heuristic to obtain an ID. This strategy works fairly well as long as our state estimate isn't too far off from the true vehicle pose. For some parameterized distance δ, we check this measurement against all previously detected landmarks, and find

$$\|L_{meas,1} - L_1\| < \delta,$$

so we decide $L_{meas,1} \equiv L_1$. Meanwhile, $L_{meas,2}$ does not pass this check for any existing landmark, so we insert a new landmark L_2 and declare $L_{meas,2} \equiv L_2$. We then proceed to step (d).

5 Manifold-Valued Representations

It's natural to approach the pose-graph SLAM problem with Riemannian optimization in mind, since all nodes and connections in the graph are manifold elements.

The vehicle pose is a 2D position and yaw-only orientation. We previously represented this with three float values, $(x_{veh}, y_{veh}, \theta_{veh})$, where the x and y components could take any float values, but θ as an angle in radians was restricted to the range $(-\pi, \pi]$ using the `remainder(`θ`, 2`π`)` function. This was fine for

EKF-SLAM, but for pose-graph SLAM we will use a more natural and technically-accurate representation.

5.a Vehicle Pose Nodes

Overall, a 2D vehicle pose is an element of $SE(2)$, which can be equivalently represented with a 2D position vector $t \in \mathbb{R}^2$ and a 2x2 orientation matrix $R \in SO(2)$.

$$x_i = \begin{pmatrix} R_i & t_i \\ 0 \quad 0 & 1 \end{pmatrix} \in SE(2),$$

where

$$t_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix} \in \mathbb{R}^2$$

and

$$R_i = \begin{pmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{pmatrix} \in SO(2).$$

Note the relationship between the rotation matrix R_i and the yaw scalar θ_i:

$$R_i = \exp(\theta_i) := \begin{pmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{pmatrix}$$
$$\theta_i = \log(R_i) := \arctan\left(\frac{R_\theta[1,0]}{R_\theta[0,0]}\right)$$

5.b Landmark Nodes

Landmarks have no inherent orientation, so each landmark will be simply a position vector.

$$L_k = \begin{pmatrix} x_k \\ y_k \end{pmatrix} \in \mathbb{R}^2 .$$

We can represent our overall "state" captured by the nodes of the pose-graph as an element of $SE(2)^t \times (\mathbb{R}^2)^M$, where t is the number of timesteps so far, and M is the number of detected landmarks.

5.c Motion Connections

From our EKF-SLAM architecture, we know there is a commanded motion at every timestep which allows us to predict the vehicle pose at the next timestep. In the pose-graph, this will take the form of a relative motion constraint between every pair of adjacent poses. So, if there are t vehicle poses in the graph, there will be $t-1$ motion connections.

Each commanded motion is a Twist velocity command, consisting of a linear (forward) speed in meters/sec and an angular speed in radians/sec. Using the known time difference between iterations, dt, we can convert these speeds to distances:

$$u_i = \{\, v_{fwd} \cdot dt, \quad v_{ang} \cdot dt \,\} = \{\, d_{fwd}, \quad d_{ang} \,\} \in \mathbb{R}^1 \times \mathbb{S}^1.$$

In EKF-SLAM, we used these values to form our state propagation matrix. We will do something very similar here to obtain a relative transformation in $SE(2)$ between adjacent vehicle poses.

$$u_i = \begin{pmatrix} \exp(d_{ang}) & \begin{pmatrix} d_{fwd} \\ 0 \end{pmatrix} \\ \begin{matrix} 0 & 0 \end{matrix} & 1 \end{pmatrix} \in SE(2).$$

We can see that this will give the relationship

$$x_{i+1} = u_i \cdot x_i,$$

just as we saw in the EKF-SLAM prediction step with the state transition matrix F_i (the motion model).

5.d Measurement Connections

Measurement connections are the trickiest to represent, because they must connect a vehicle pose in $SE(2)$ to a landmark in $\mathbb{R}^2$. The raw measurement values published by the simulator take the form

$$z_i = \{\, j_1, r_1, b_1, \ldots, j_n, r_n, b_n \,\}^T,$$

In each timestep, we may detect nothing, one landmark, or any number of landmarks, depending on what is within the FOV of the vehicle. Each landmark detection includes that landmark's unique ID, j, as well as the range r and bearing b of the landmark relative to the vehicle pose.

> Aside: We could compare each detection to all previous landmark detections and perform data association, rather than assuming the unique landmark ID is provided, but for now we skip this step in favor of focusing on accurate pose-graph construction and optimization.

We can split this into a detection for each landmark,

$$z_{i,j} = \{\, r_{i,j}, \quad b_{i,j} \,\} \in \mathbb{R}^1 \times \mathbb{S}^1,$$

and form this into a matrix to estimate the detected landmark's position.

$$z_{i,j} = \begin{pmatrix} \exp(b_{i,j}) & \begin{pmatrix} r_{i,j} \\ 0 \end{pmatrix} \\ \begin{matrix} 0 & 0 \end{matrix} & 1 \end{pmatrix} \in SE(2).$$

This gives the relation

$$L_j = z_{i,j} \cdot x_i,$$

but this has a flaw by assuming the orientation carries through, giving us a landmark estimate in $SE(2)$ instead of just $\mathbb{R}^2$. Depending on our specific implementation, we have two options here:

1. Use this $SE(2)$ representation, and set the rotation component's weight to 0. PG optimization will estimate the rotation for landmarks, but it will be meaningless and only there to ensure data-types match.

2. OR, use a different type of connection that strips out only the translation component.

This second choice essentially inserts static matrices to force dimensions & data-types to match:

$$L_j = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot (z_{i,j} \cdot x_i) \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

which does the job of extracting the resulting position vector from the $SE(2)$ result of $z_{i,j} \cdot x_i$.

In GTSAM, we can avoid doing this explicitly by using a specific type of connection that is provided for range-bearing measurements.

In SE-Sync, all nodes and measurements must be elements of $SE(2)$ (or $SE(3)$), so we must use option (1).

6 Noise

Noise is applied by our simulator both when applying requested motion commands and when determining measurements to send. This noise is also manifold-valued, and is sampled from a multi-dimensional Gaussian distribution. All noise distributions have mean zero, since we assume there is no systematic bias in our system, only random noise whose covariances we can change to experiment with performance as noise increases.

The terms q_{lin}, q_{ang}, w_{range}, and $w_{bearing}$ are parameterized through the config file. These are used both for the true noise distributions used by the simulator, as well as by all our filters to estimate the state and track uncertainty.

> Aside: In reality, these could not possibly match, since reality is not so simply parameterized and is never known, but this simplification allows us to focus more on the project objectives without losing the ability to compare different filters.

6.a Process Noise

Process noise reflects the inaccuracies in our motion model as it pertains to reality. This includes things like wheel slip, uneven surfaces, and other factors that we don't include in our simple motion model. We can also use the process noise to cover wrong assumptions that we make for simplicity, such as using a constant-velocity model. Since our simulation is not a fully-accurate physics sim, we apply a randomly-sampled noise to our commanded motion each timestep to emulate these effects.

This process noise is also needed as an uncertainty term that goes along with our motion connections in the pose-graph.

The process noise distribution has components both for translation and rotation. The translation term takes the form $t_q \sim \mathcal{N}(0, Q)$, where Q is a covariance matrix,

$$Q = \begin{pmatrix} q_{lin} & 0 \\ 0 & q_{lin} \end{pmatrix}.$$

Translation noise ($t_q \in \mathbb{R}^2$) is sampled and applied via addition to our position terms.

> Note the nonzero term $Q[1,1]$, which enables lateral motion in the vehicle, which is not possible to command for differential drive. This alone will cause truth to differ from our estimates if we do not

perform some kind of filtering beyond direct propagation of commanded motion.

The rotation term takes the form $R_q = \exp(\theta_q)$, where $\theta_q \sim \mathcal{L}(0, q_{ang})$. Here, $\mathcal{L}$ is the Langevin distribution; sampling from it gives an angle $\theta_q \in \mathbb{S}^1$. We can then obtain the rotation matrix $R_q = \exp(\theta_q)$, which is applied via left-multiplication to our orientation terms.

> Note: Since our sampled angle will be immediately passed through the exponential map to form R_q, it is not essential that it falls within the range of angles $(-\pi, \pi]$; i.e., it will suffice in our implementation to sample from a 1D Gaussian distribution centered at 0, and an angle outside the range will make no difference in the resulting rotation matrix's validity.

Noisy vehicle propagation in the simulator is then

$$x_{i+1} = \begin{pmatrix} R_q & t_q \\ 0 \quad 0 & 1 \end{pmatrix} \cdot u_i \cdot x_i.$$

6.b Sensing Noise

Sensing noise reflects inaccuracies in the sensors themselves, as well as any simplifying assumptions we may have made in our setup. Similarly to the process noise, our simulator will compute true values and apply a sampled sensing noise to these measurements before publishing them.

As our measurements are also composed of a linear term and an angular term, we will have a similar setup to the process noise for our sensing noise distribution. However, while the process noise affected the true vehicle pose, the sensing noise will not affect the true landmark poses. As such, we will not compute a transformation matrix, but will rather compute noise separately for each measurement component.

The translation term takes the form $t_q \sim \mathcal{N}(0, w_{range})$.

The rotation term takes the form $\theta_w \sim \mathcal{L}(0, w_{bearing})$, where $\mathcal{L}$ is again the Langevin distribution of angles.

Our published measurement for a given landmark is then

$$z_{i,j} = \{\, r_{i,j} + t_w[0], \quad b_{i,j} + \theta_w \,\} \in \mathbb{R}^1 \times \mathbb{S}^1.$$

7 Implementation in Code

As mentioned in the introduction, we implement a custom abstract `Filter` class, and each of our filters inherits from this class and implements any variables and functions unique to that class. All of this is done in C++. The EKF, UKF, and new pose-graph SLAM classes are among these, and we also implement an extremely basic "Naive" filter which directly propagates its pose estimate using motion commands. These other filters can be run simultaneously as a comparison with pose-graph SLAM.

We initially implement the pose-graph SLAM algorithm using the open-source GTSAM (Georgia Tech Smoothing and Mapping) library [1]. A second implementation using the SE-Sync library [2] is in progress but as of yet is not working correctly. The specifics of these implementations are detailed in the following sections.

My full code for this project is open-source on my GitHub. This particular project is centralized in the `ekf_ws/src/localization_pkg` directory, with all config values being editable in `ekf_ws/src/base_pkg/config/params.yaml`. Instructions for installation and running it can be found in the repository's README.

> Aside: I would also like to derive and implement a basic pose-graph SLAM algorithm entirely from scratch, as I've done with the other filters in this personal project, though that will be a greater undertaking.

7.a GTSAM

In GTSAM, we build a `NonlinearFactorGraph` with all connections, and insert all nodes into a vector of `Values` as an initial estimate. As mentioned, vehicle poses are $SE(2)$ elements, while landmarks are in $\mathbb{R}^2$.

GTSAM requires a prior to be specified, which fixes the pose-graph with a specific pose as the first vehicle node. This enforces the coordinate system of all results after pose-graph optimization to preserve a common origin with our other filters. We can also specify a noise on the prior to allow an aspect of uncertainty on the first pose, if desired.

Each motion connection between adjacent vehicle poses is implemented as `BetweenFactor<Pose2>`, using the $SE(2)$ relative transformation u_i and the process noise we have described.

Each measurement connection between a vehicle pose and a landmark is a `BearingRangeFactor<Pose2, Vector2>`, using as its arguments $\exp(b_{i,j})$, r, and the sensing noise model we have described.

We solve the pose graph using a `LevenbergMarquardtOptimizer`, which produces a vector of `Values` containing the resulting estimates of all graph nodes, including the full vehicle pose history and all landmark estimates.

As mentioned, we can run this optimization after all iterations have completed, or at the end of each iteration. This behavior is parametrized by a flag in our config file.

7.b SE-Sync

In SE-Sync, we build a vector of `measurements_t` with all connections. In contrast to GTSAM, we need not track any initial estimate of vehicle poses and landmark positions. SE-Sync solves the graph more generally, with no prior on the initial vehicle pose.

As mentioned in Section 5.d, all nodes and connections are members of $SE(2)$, and we place a weight of $\kappa = 0$ on the rotational component of landmark estimates to reflect their lack of orientation. Additionally, we make sure to run Se-Sync with the translation-explicit form, rather than the simplified form, as the rounding that occurs in the latter is incompatible with our hack to include landmarks.

Since there is no specific gauge or prior enforced in SE-Sync, unlike GTSAM, the first pose will not necessarily be at the origin of our overall coordinate system. We can transform the entire resulting pose-graph to shift the first pose to

$$x_0 = \begin{pmatrix} I_2 & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ 0 \quad 0 & 1 \end{pmatrix}$$

to ensure our final result can be meaningfully visualized alongside our other filter results for comparison.

> Note: This implementation is not yet functioning properly, and while I've run out of time to continue working on this for the end-of-semester deadline, I plan to pick this up again soon and work on finishing the integration effort. I believe that spending more time reading over the algorithm and the associated paper will help me to understand where in my implementation I've erred.

> Aside: A potential solution to my problems with SE-Sync would be to simply assign an orientation to my landmarks. This isn't unheard of, as real-world landmark-based algorithms often use visual fiducials such as AprilTags, which inherently have a position as well as an orientation. In fact, I've previously written an AprilTag detection node in ROS which returns the tag's pose as an $SE(2)$ transform relative to the vehicle. Changing my simulator to allow landmarks to have detectable orientations would enable me to more easily use SE-Sync, as then every quantity (vehicle poses, landmarks, commands, and measurements) will be an element of $SE(2)$.

8 Results

We can run pose-graph SLAM (PGS) alongside another of our filters to get a comparison between them. We will compare against the Naive filter and EKF-SLAM. We'll treat iterative PGS (solving every iteration) and one-time PGS (solving only at the end) as two different filters for this section.

We can also run with different levels of noise to see how that affects the relative performances. We define two noise regimes:

- "Low noise": $q_{lin} = 0.01$, $q_{ang} = 0.001$, $w_{range} = 0.01$, and $w_{bearing} = 0.001$.
- "High noise": $q_{lin} = 0.01$, $q_{ang} = 0.01$, $w_{range} = 0.01$, and $w_{bearing} = 0.01$.

The process noise values can be interpreted relative to our motion constraints; since in any iteration, we can move at most 0.1 units, q_{lin} sets a standard deviation of 10% our max value. Raising this value higher leads to rapid falloff in performance regardless of what filter we use. By a similar argument, we hold noise for range measurements at the same value for both regimes. The more impactful difference in these regimes lies in q_{ang}, which at its higher value causes very noticeable drift in dead-reckoning estimates, so the naive filter gets worse over time, and the EKF must have frequent re-detections of previous landmarks to stay on track.

The error for an iteration will be computed as the Euclidean distance between a pose estimate and the corresponding true vehicle pose for the same timestep. For simplicity, we consider only positional error. The average error for a filter's run is then computed as the mean of these errors across all 1000 iterations. We run every filter at least 10 times to compute a super-average of its average error across all runs. These values are shown in Table 1. Note that every run has a randomly generated environment of landmarks and a different trajectory computed to explore that particular environment.

	Naive	EKF-SLAM	One-Time PGS	Iterative PGS
Low Noise	0.2699	0.1883	0.1807	0.1802
High Noise	2.2772	1.3966	0.9265	0.7294

Table 1: Average error for the filter across 10 runs. All values are in meters.

We can see that the error for pose-graph SLAM is lower in both noise regimes than all other filters, and that

the iterative version beats out the one-time approach. Despite this, we can see that for low noise, the results are quite similar; although PGS has lower error, all filters perform reasonably well, since the noise is low enough that online estimators can suffice to generate a full pose history estimate. This is evident visually, such as in Figure 4.

Figure 3 shows direct comparisons between our filters running simultaneously on the same simulation.

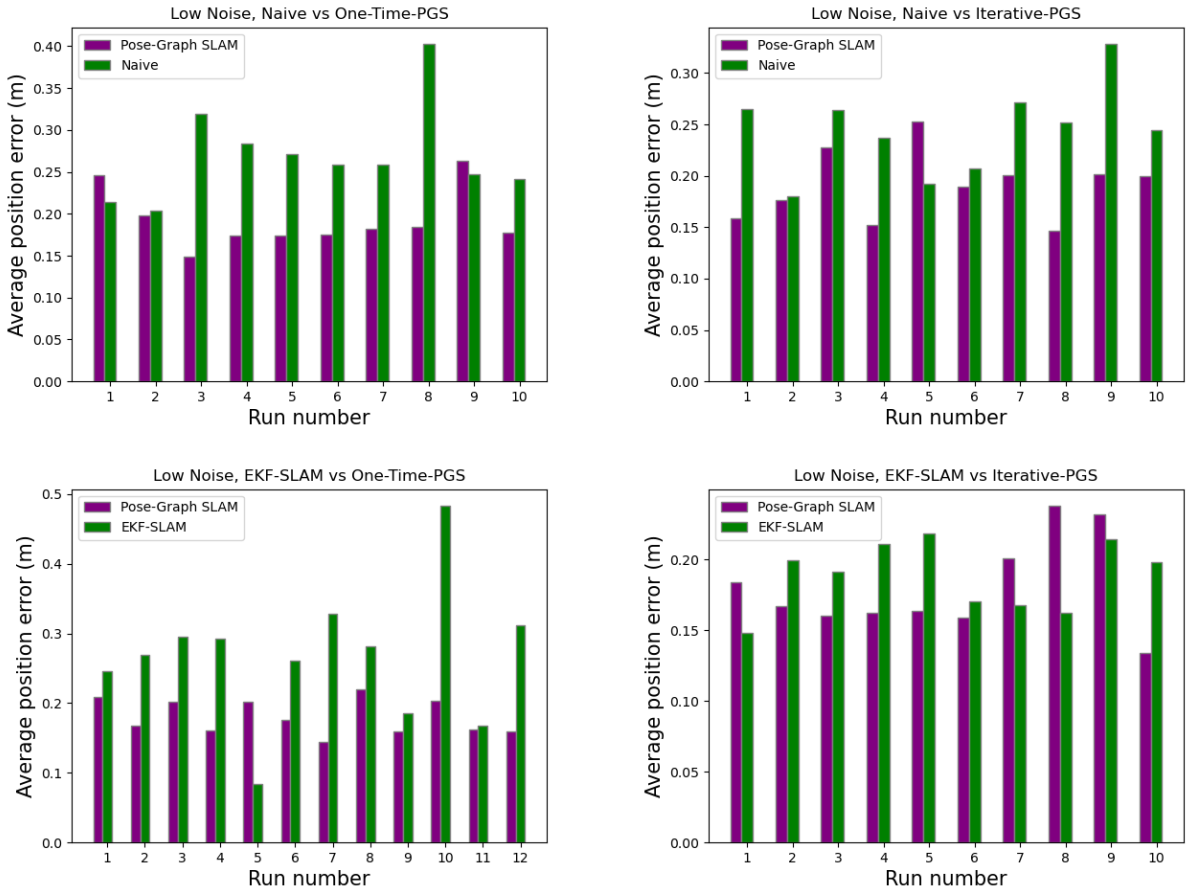


Figure 3: Error comparisons with low noise. We can see that the pose-graph is not always better in a single run, despite having lower super-average error in all match-ups.

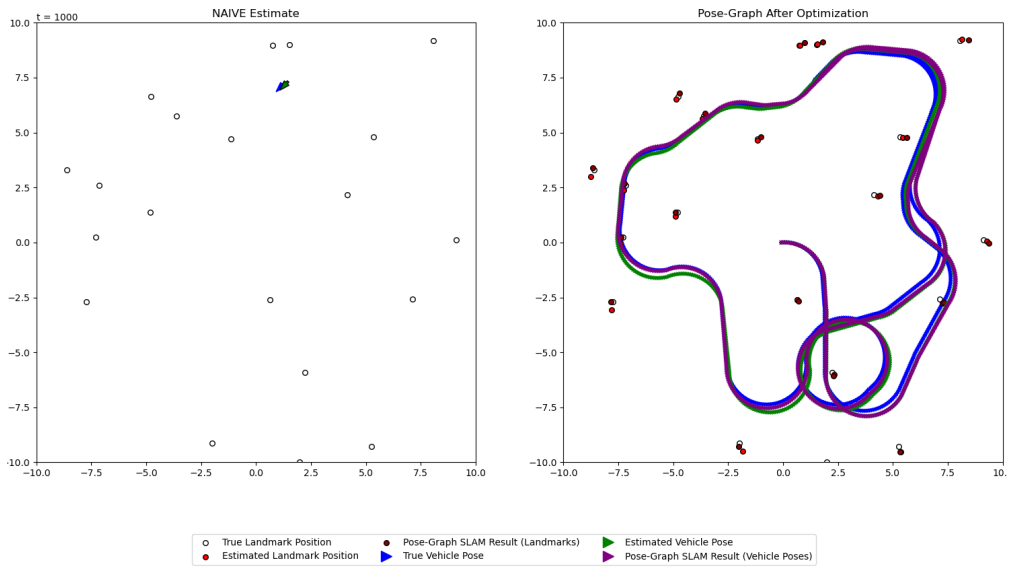


Figure 4: With low noise, one-time PGS is barely different from the online filter estimates.

In most high-noise runs, the one-time PGS solution either closely matches the initial estimate, or it finds a solution with lower error that still does not appear correct visually. An example of this is shown in Figure 5.

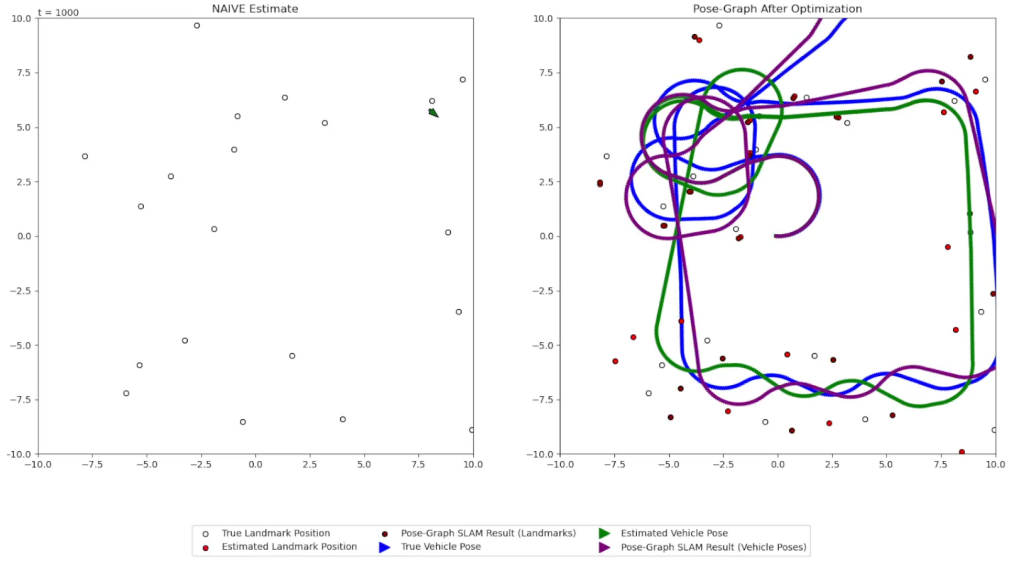
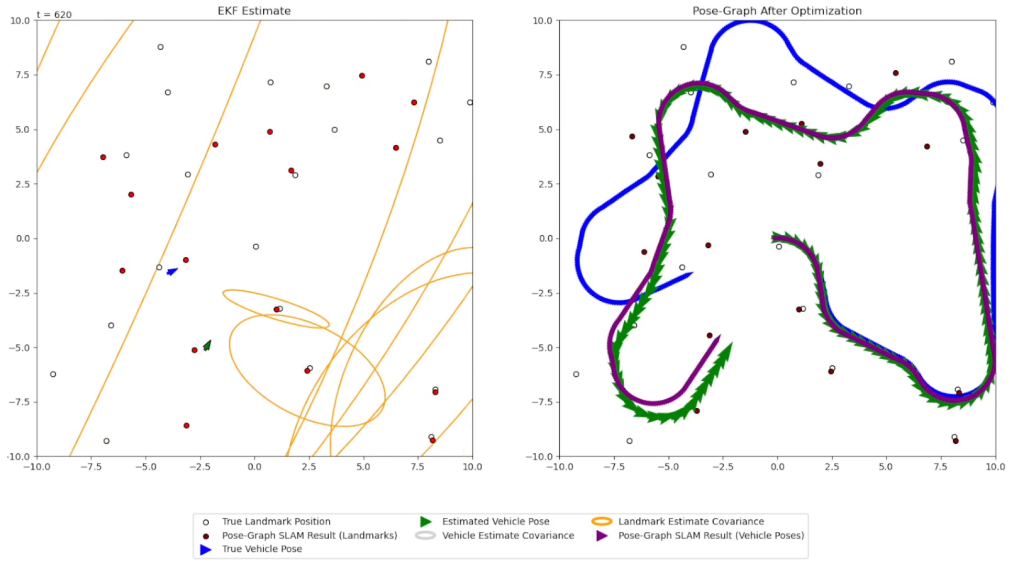
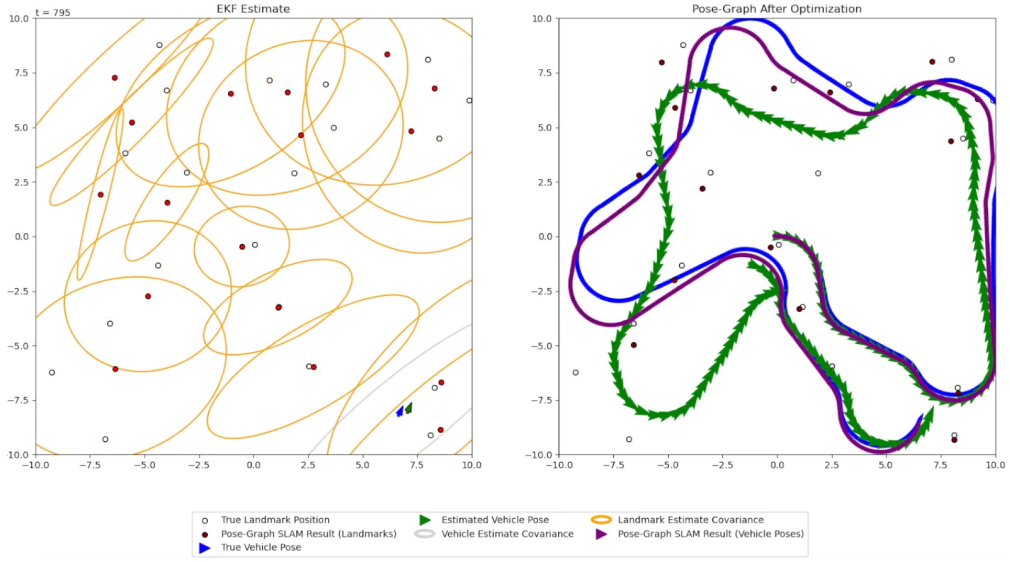


Figure 5: With high noise, one-time PGS outperforms the naive filter, but both have high error.

The incremental PGS solver, in contrast, is able to recognize an important loop closure and snap to a far better solution than the online filter was able to achieve. Note that the EKF also improves its pose estimate at these places, but this does not affect its pose history leading up to this time, which remains incorrect. This is shown in Figure 6.



(a) Long chain w/o loop closures.



(b) PGS solution after some loop closures.

Figure 6: Iterative PGS performance vs EKF-SLAM, with high noise.

For the high-noise regime, the benefits of pose-graph SLAM are much clearer, as it stands out with a much higher margin from the other filters. Batch runs for comparison are shown in Figure 7.

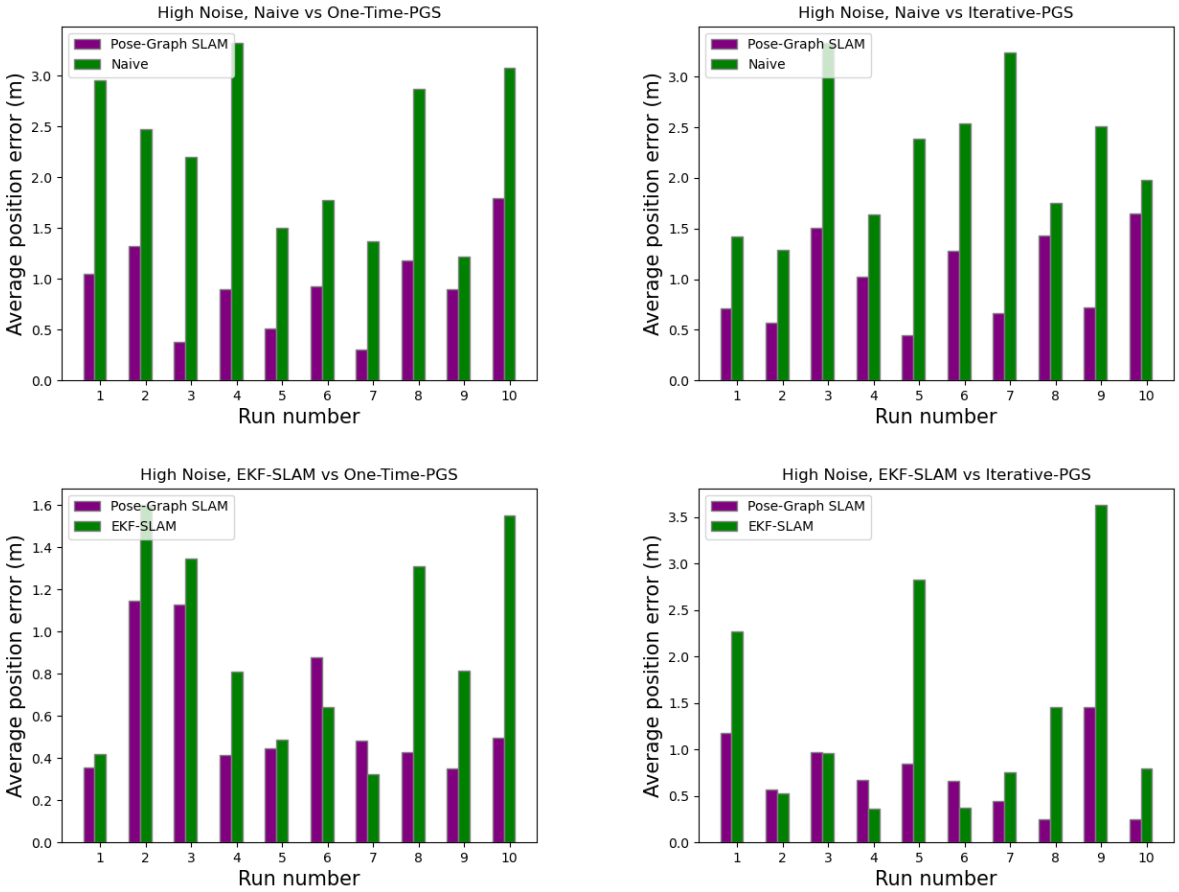


Figure 7: Error comparisons with high noise. We still see some individual runs in which the EKF outperforms PGS, though these are quite rare. This may be indicative that some particular characteristic of the environment or trajectory can lead to better EKF performance or worse pose-graph optimization.

A run for each direct comparison has been recorded and compiled into a video here. This video does a better job of conveying visually how this implementation performs under different circumstances than could be included in a static document.

As expected, there is a trade-off with iteratively solving the pose-graph. Performance is significantly improved and is almost always superior to an online EKF in estimating the full vehicle pose history, especially as noise increases, but the runtime noticeably slows as we approach 1000 poses in the graph. GTSAM is not optimized to run iteratively, and in our implementation the full graph is solved every iteration as if from scratch, despite using the previous result as an initial estimate. To combat this slowdown, a future project should investigate an algorithm designed specifically for repeated optimization, such as iSAM.

References

[1] Frank Dellaert and GTSAM Contributors. borglab/gtsam, May 2022.

[2] D.M. Rosen, L. Carlone, A.S. Bandeira, and J.J. Leonard. SE-Sync: A certifiably correct algorithm for synchronization over the special Euclidean group. *Intl. J. of Robotics Research*, 38(2–3):95–125, March 2019.