Simultaneous Localization and Mapping Lab 1: SLAM for Karel in 1D

Beatriz Rosell Cortés
Erika Liced Rimacuna Castillo

February 10, 2025

Abstract

This report describes the work carried out in the context of Lab 1 of the Simultaneous Localization and Mapping (SLAM) course. The task was to implement SLAM for Karel, a robot that operates in 1D space. We used simulated odometry and sensor measurements to map the environment, handle beepers, and test different algorithms including Kalman filters and sequential map joining. We also analyzed the computational costs of the methods employed.

Contents

1 Introduction

In this section, provide a brief introduction to the problem, the significance of SLAM, and an overview of the task. Describe the problem context of Karel living in 1D space, how it interacts with its environment, and the importance of the sensors and algorithms used.

2 Initial Code Overview

In the initial code, the goal was to implement a basic framework for SLAM in a 1D world. The core functionality included:

- Defining robot and sensor characteristics such as motion error and sensor range.
- Simulating robot motion and measurements to a set of fixed features in the environment.
- Using a Kalman filter for robot localization and mapping, including updates and additions to the map.

This initial code set up the basic structure of the SLAM problem but lacked certain refinements such as handling multiple maps, improving the Kalman filter update, and optimizing computation.

3 Final Code Overview

The final version of the code includes several enhancements that improve the SLAM implementation. Key changes and additions include:

3.1 Kalman Filter Update

In the final version, the Kalman filter update is refined. The `update_map` function is now fully implemented to correctly calculate the Kalman gain K_k, the innovation y_k, and the covariance S_k. These updates allow the robot's belief about its state (position) and map (feature locations) to be corrected after each measurement.

3.2 Feature Addition and Management

The final code introduces a method for handling the dynamic addition of new features detected by the robot during the SLAM process. The function `add_new_features` ensures that newly detected features, which are not yet in the map, are properly incorporated into the map. This update is crucial for maintaining both the robot's position and the locations of the newly added features in the map.

The procedure for adding new features follows a series of steps:

1. **Check for New Beepers:** The first step in the process is to check whether the robot has detected new beepers (features). If no new beepers are detected, the process exits without making any changes to the map.

2. **Extract Distance and Uncertainty:** If new features are detected, the distance measurement (z_n) and the uncertainty of the sensor measurement (R_n) for the new feature(s) are extracted from the sensor data. These are crucial for updating the robot's belief about its position and the positions of the features.

To incorporate the new feature into the existing map, the following mathematical formulation is applied:

- Let $X_{k|k}$ represent the state vector (which contains both the robot's position and the map of features), and J_1 and J_2 be the Jacobian matrices that help propagate the uncertainty. The goal is to extend the map with the new feature, maintaining the robot's position as it was while adding new rows corresponding to the new feature.

- The Jacobian matrices J_1 and J_2 are used to compute the update:

$$X_{k|k} = \begin{bmatrix} x_{R_0R_k} \\ X_{R_0F} \\ x_{R_0R_k} + Z_n \end{bmatrix}_{1+m+n\times 1} = J_1 X_{k|k} + J_2 Z_n$$

  Where:

  – J_1 is designed to maintain the robot's position and the positions of existing features.

  – J_2 handles the incorporation of the new feature into the map.

  Form of the Matrices J_1 and J_2

  The matrices J_1 and J_2 are structured as follows:

  – J_1: This matrix ensures that the robot's position and the positions of existing features are preserved. The first $m + 1$ rows of J_1 form an identity matrix, corresponding to the robot's position and the features already in the map. The next n rows are set to 1 in the first element (to preserve the robot's position), and zeros in the rest.

  – J_2: This matrix is used to add the new feature to the map. It has zeros in the first $m+1$ rows (which correspond to the robot and the existing features), and an identity matrix in the last n rows, which correspond to the new feature.

  For example, let us assume that there are 2 features ($m = 2$) and we are adding 1 new feature ($n = 1$). The Jacobian matrices are constructed as follows:

$$J_1 X_{k|k} + J_2 Z_n = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} x_{R_0R_k} \\ x_{R_0F_1} \\ x_{R_0F_2} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} z_{n1} \\ z_{n2} \end{bmatrix}$$

resulting in:

$$X_{k|k} = \begin{bmatrix} x_{R_0R_k} \\ x_{R_0F_1} \\ x_{R_0F_2} \\ x_{R_0R_k} \\ x_{R_0R_k} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ z_{n1} \\ z_{n2} \end{bmatrix} = \begin{bmatrix} x_{R_0R_k} \\ x_{R_0F_1} \\ x_{R_0F_2} \\ x_{R_0R_k} + z_{n1} \\ x_{R_0R_k} + z_{n2} \end{bmatrix}$$

Here, J_1 ensures that the robot's position and the two existing features remain unchanged, while J_2 adds a new row for the newly detected feature.

Sparse Matrices

It is important to note that when constructing the matrices J_1 and J_2, they should be handled as sparse matrices. This is due to the majority of it's elements being zeros, therefore, using sparse matrix representations allows efficient computation. This construction becomes even more relevant as we scale the problem, handling larger maps or including more features in the SLAM process.

Update Rule

Finally, the robot's state vector $x_{k|k}$ is updated by the following rules:

$$\begin{aligned} \tilde{y}_k &= z_k - H_k \hat{x}_{k|k-1} \\ S_k &= H_k P_{k|k-1} H_k^T + R_k \\ K_k &= P_{k|k-1} H_k^T S_k^{-1} \\ x_{k|k} &= x_{k|k-1} + K_k \tilde{y}_k \\ P_{k|k} &= (I - K_k H_k) P_{k|k} \end{aligned}$$

This update rule incorporates the new feature into the map, ensuring that both the robot's position and the positions of the features are estimated accurately.

Implementation in MATLAB

The implementation in MATLAB requires careful handling of sparse matrices, and the update must be computed efficiently to avoid unnecessary computational overhead. The updated state vector and map are returned, along with the updated covariance matrix.

`add_new_features` uses these matrix operations to incrementally expand the state and covariance matrices, ensuring that new features are properly added without disturbing the existing map structure.

This process allows the SLAM algorithm to dynamically adapt to new features detected by the robot, providing an accurate estimate of the robot's position and the features in the environment.

3.3 Handling Multiple Maps

The final code also introduced the concept of handling multiple maps by segmenting the map into smaller regions. This is achieved using the `reset_map` function, which resets the map when a certain threshold is met, allowing the robot to build a map incrementally. The maps are then joined using the `join_maps` function, where the various segments are combined, and the map is updated accordingly. In general, this process can be represented mathematically as follows:

$$\mathbf{J}_1\mathbf{x}_1 + \mathbf{J}_2\mathbf{x}_2 + \dots$$

where each segment contributes to the overall map construction, ensuring a more manageable and efficient mapping process.

3.3.1 Exercise 11

In sequential mapping, different maps are built and later joined to form a global representation. However, a key challenge arises when common environmental features appear in multiple maps, as their independent estimations may lead to inconsistencies.

In this section we comment how we solved it.

At each step, after updating the current map and adding new features, a checkpoint mechanism verifies whether it is time to reset the map and store the accumulated information.

When a reset condition is met, the system evaluates the features present in the current map. Only the most recent and relevant features—those still visible in the environment—are preserved, while older, less relevant information is discarded.

Since the reset occurs immediately after adding visible features, in the next iteration, instead of reintroducing them as if they were newly observed (as was initially done when all features were removed and treated as unseen), the system now only updates their estimates.

Each segmented map is stored separately, maintaining an organized list of maps. Importantly, the uncertainty associated with each feature is also recorded, preserving the statistical properties of the observations. This prevents inconsistencies when maps are later combined.

Once all individual maps have been constructed, they are systematically merged into a single, coherent global map. This merging process aligns overlapping features by adjusting their reference frames and ensuring consistency in their estimations. A transformation mechanism is applied to integrate the feature estimates while preserving their associated uncertainties. This ensures that features appearing in multiple maps are represented consistently in the final merged map.

4 Analysis of Results

The results of the SLAM process were evaluated by varying the number of maps used during the process. The number of maps was chosen to be 1, 2, 8, 16, 32, and 64.

4.1 Correlation Matrix

Figure 1 shows the correlation matrix of size 1003×1003 between the covariance terms in the Kalman filter. As we can see, neighboring features exhibit high correlation (red

region along the diagonal), meaning that nearby points in the state estimation process strongly influence each other. As the distance between features increases, the correlation decreases smoothly (transition from red to green and finally blue), indicating that distant features have weaker dependencies.

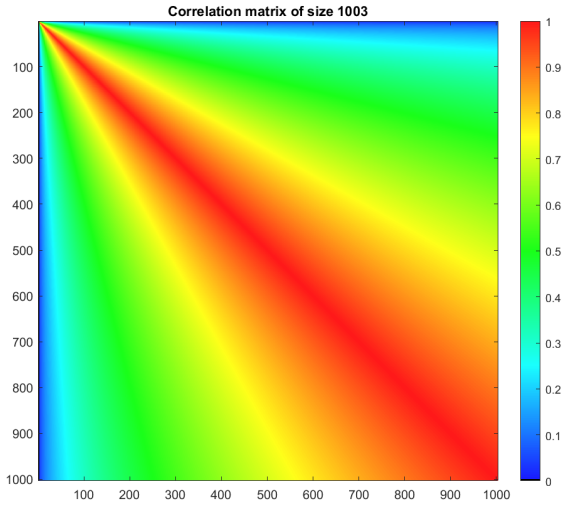


Figure 1: Correlation Matrix

4.2 Map Estimation Error

Figure 2 illustrates the estimation error of the map features along with the associated $\pm 2\sigma$ confidence bounds. As we can see, the increasing spread of the confidence bounds suggests that feature estimates become less precise as the mapping progresses. This behavior is expected in SLAM-based approaches where uncertainty accumulates over time. However, the fact that most errors remain within the $\pm 2\sigma$ range indicates that the estimation process is statistically consistent.

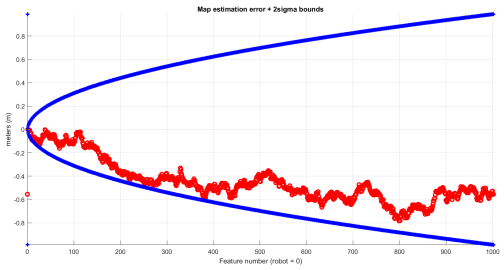


Figure 2: Map Estimation Error

4.3 Robot Estimation Error

Figure 3 presents the robot's estimation error over time along with the associated $\pm 2\sigma$ confidence bounds. The estimation error does not remain centered around zero, suggesting a drift in the estimated trajectory. This is a typical behavior in localization problems where small errors accumulate over time, leading to deviation from the true position.

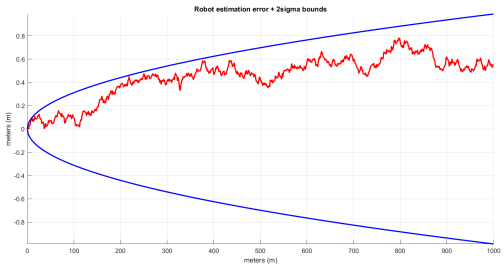


Figure 3: Robot Estimation Error

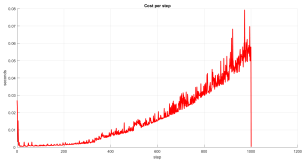
4.4 Performance Analysis

4.4.1 Cost Per Step

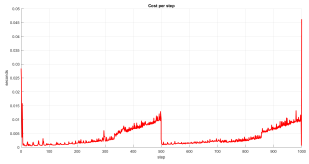
Figure 4 presents the computational cost per step when using different numbers of maps: 1, 2, and 8. The results show a clear trend in how computational efficiency improves as the number of maps increases.

- **1 Map (Figure 4a)**: When using a single map for the entire process, the computational cost increases significantly over time. This happens because the size of the state vector grows continuously, leading to higher computational complexity at each step.
- **2 Maps (Figure 4b)**: By splitting the mapping process into two segments, the computational cost still increases over time, but the reset between maps helps control the growth rate. The peaks in cost occur when a new map is introduced, resetting the accumulated state information and reducing the complexity temporarily.
- **8 Maps (Figure 4c)**: With more frequent resets (eight maps), the computational cost remains consistently low throughout the process. Each map contains only a limited number of features, preventing excessive state vector growth and maintaining efficient computations at each step.

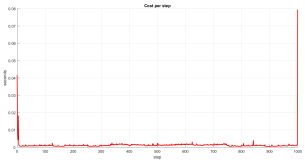
This trend confirms that segmentation improves computational efficiency by limiting the accumulation of state variables in each map. Instead of dealing with an ever-growing estimation problem, resetting and joining maps at appropriate intervals keep the cost manageable. This method ensures scalability while maintaining accuracy in state estimation.



(a) Cost Per Step 1 Map



(b) Cost Per Step 2 Maps



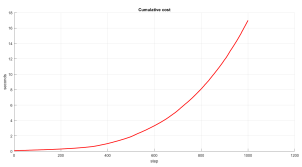
(c) Cost Per Step 8 Maps

Figure 4: Cost Per Step

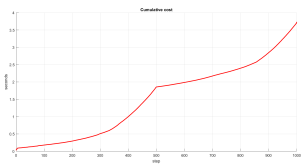
4.4.2 Cumulative Cost

Figure 5 illustrates the cumulative computational cost over time for different mapping strategies: using 1, 2, and 8 maps. The results highlight how segmenting the mapping process significantly impacts computational efficiency.

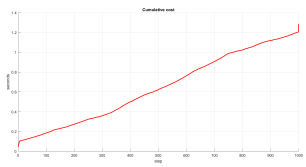
- **1 Map (Figure 5a)**: The cumulative cost grows exponentially as the number of steps increases. This is because a single map leads to an ever-growing state vector, making each computation more expensive over time.
- **2 Maps (Figure 5b)**: Splitting the mapping process into two reduces the cumulative cost considerably. The reset between maps limits the complexity, preventing excessive growth. The jumps in cost correspond to the transition between maps.
- **8 Maps (Figure 5c)**: When using eight maps, the cumulative cost grows almost linearly, remaining significantly lower than in the other cases. The frequent resets keep the computational burden minimal, preventing the rapid accumulation seen in the single-map case.



(a) Cumulative Cost 1 Map



(b) Cumulative Cost 2 Maps



(c) Cumulative Cost 8 Maps

Figure 5: Cumulative Cost

4.4.3 Elapse time

The elapsed time for different numbers of maps (1, 2, 8, 16, 32, 64 and 128) was averaged over 50 runs. The results were as follows:

Number of Maps	Average Elapsed Time (seconds)
1	20.88
2	4.49
4	1.88
8	1.93
16	1.60
32	2.71
64	2.22
128	2.65

Table 1: Average Elapsed Time for Different Numbers of Maps

As seen in the Figure 6, when the number of maps increases from low values (e.g., 1, 2, 4 maps), the elapsed time decreases sharply. This is because dividing the mapping process into smaller segments reduces the computational burden at each step, preventing excessive state growth.

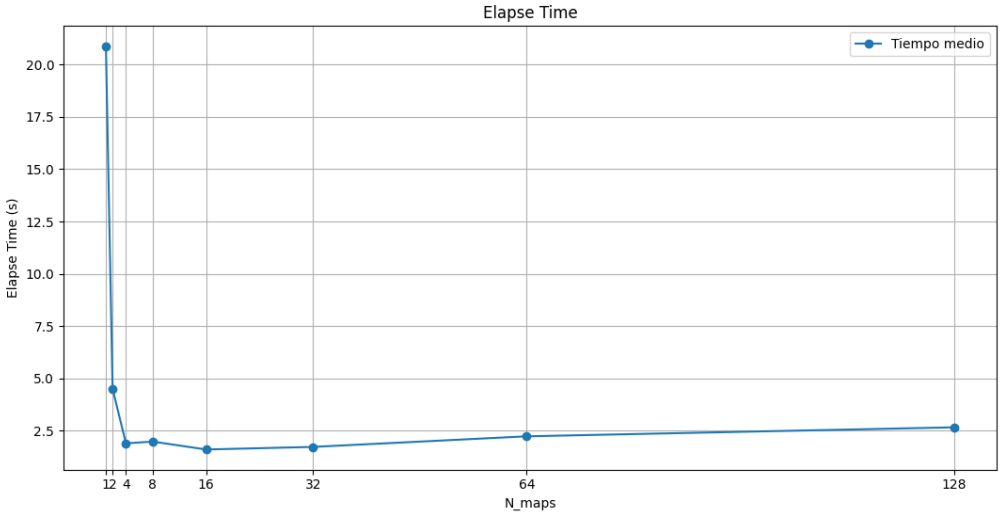


Figure 6: Elapse Time vs Num Maps

As N_{maps} continues to grow beyond a certain point (e.g., 32, 64, 128 maps), the elapsed time starts to increase again. This is likely due to the overhead introduced by frequent map resets and the merging process.

5 Conclusion

The final code version significantly improves the initial implementation by refining the Kalman filter update, adding new feature handling, and optimizing map segmentation and merging. Performance statistics such as cost per step and elapsed time were tracked and visualized in graphs, providing valuable insights into the computational complexity of SLAM with varying numbers of maps.