Lab5 - EKF Simultaneous Localization and Mapping

Mohammad Rami Koujan

1 Introduction

This lab assignment is based on the concept of Simultaneous localization and Mapping (SLAM) which is defined as the problem of building a map while at the same time localizing the robot within that map. In practice, these two problems cannot be solved independently of each other. Before a robot can answer the question of what the environment looks like given a set of observations, it needs to know from which locations these observations have been made. At the same time, it is hard to estimate the current position of a vehicle without a map. Therefore, in SLAM, a good map is needed for localization while an accurate pose estimate is needed to build a map.

In fact, the main objective during this lab is to program a SLAM algorithm in ROS using python language so as to help the turtlebot with the localization and mapping task, and to test this algorithm using rviz (Robot Visualization) which is a 3D visualizer for displaying sensor data and state information from ROS.

2 SLAM algorithm analysis

This section explains in details the different functions of the implemented SLAM algorithm. Basically, a number of functions were implemented to handle the different necessary steps for the appropriate work of this filter which are:

1. state prediction
2. Data association
3. State update
4. State augmentation

The work was done initially on a synthetic dataset (dataset 3) gathered from Kineckt sensor where a number of lines were extracted by split and merge algorithm from this dataset and used as features.

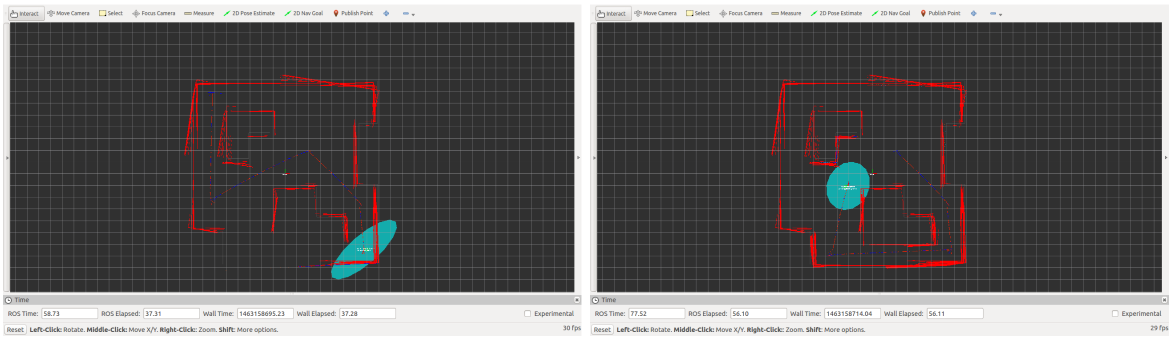
2.1 State Prediction

A function called "predict" was programmed to carry out the task of prediction where in this step the aim is to predict the new robot position, which is known initially, after a movement and reception of an odometry message. In order to do so, the following equations were used:

$$\hat{x}^B_{k|k-1} = f\left(\hat{x}^B_{k-1}, \hat{u}^{k-1}_k\right) \tag{1}$$

$$P^B_{k|k-1} = F_k P^B_{k-1} F^T_k + G_k Q_k G^T_k \tag{2}$$

The first equation is used in general to update the the mean state vector. This was done in this lab by using the odometry message after doing a composition step, the purpose of which is transform this message from the robot frame to the world frame in order to be added then to the state vector. The second equation takes care of the co-variance matrix which is built in the prediction step by the use of two jacobian matrices. The first one is with respect to the previous state (J1) and the second one is with respect to the odometry uncertainty (J2). Then, both of these matrices were augmented in the F_K and G_K respectively where F_K is of size (3+2n)x(3+2n) and G_K is of size (3+2n)x(3), n is the number of augmented features which is initially before doing state augmentation is equal to zero. The results of running only the prediction step is shown in figure 1.



ROS Time: 58.73 ROS Elapsed: 37.31 Wall Time: 1463158695.23 Wall Elapsed: 37.28

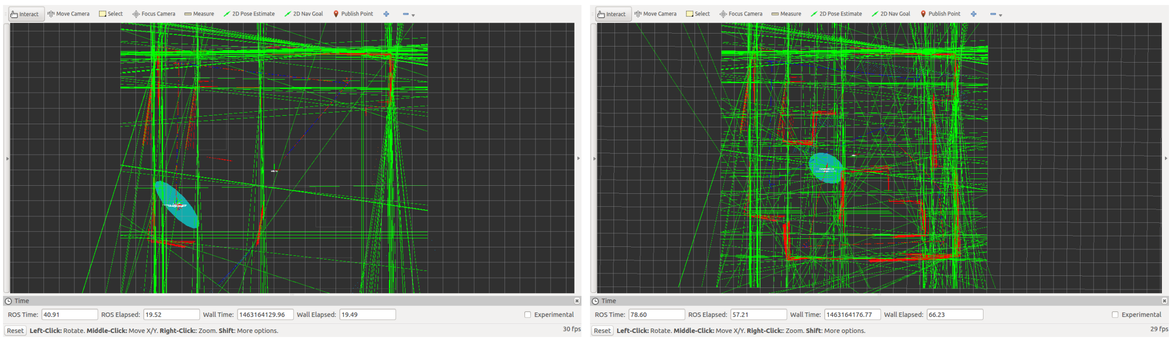
ROS Time: 77.52 ROS Elapsed: 56.10 Wall Time: 1463158714.04 Wall Elapsed: 56.11

Figure 1: Result of running the prediction step only

As it is shown in the previous figure, the uncertainty of the robot grows while it is moving since the only part that is taken into account is the prediction which causes the uncertainty to increase because of the odometry uncertainty.

2.2 State Augmentation

State augmentation is the process of adding not associated lines, which result from data association step, to the state vector. Not associated lines are those which are observed by the robot but are not used to update it is position due to their Mahalanobis distance, with the map features, that is greater than the defined threshold (*chi_threshold*). Consequently, those lines are used by the robot as map features, and they represent the mapping step of the SLAM algorithm. In fact, the state augmentation step has been implemented by first taking the lines observed by the robot and the indices which represent the not associated ones. Then, the polar coordinates of the not associated lines were computed with respect to the robot frame and after that they were transferred to the world frame in order to be appended to the state vector. However, to add those lines to the co-variance matrix of the state vector, equation 2 was used where *F_K* was built from an identity matrix whose size is (3+n*2)x(3+n*2) and for each not associated line the jacobian *H_tf*, returned by the tfPolarLine function, was vertically stacked with this matrix. On the contrary, *G_k* was constructed from a zeros matrix of size (3+n*2)x(2) with the jacobian *H_line*, returned by the tfPolarLine function, stacked vertically for each not associated line. *H_tf* and *H_line* are the jacobians with respect to the transformation and line respectively. Figure 2 shows the result of running the predict and state augmentation parts of the programmed SLAM algorithm.



ROS Time: 40.91 ROS Elapsed: 19.52 Wall Time: 1463164129.96 Wall Elapsed: 19.49

ROS Time: 78.60 ROS Elapsed: 57.21 Wall Time: 1463164176.77 Wall Elapsed: 66.23

Figure 2: Result of running the prediction and state augmentation steps

The previous figure demonstrates the effect of using the state augmentation part in SLAM algorithms. Green lines represent the lines that are added to the state vector as map features where all the observed lines are added in this case since the data association function has not been implemented up until this step. This explains the high number of green lines shown in the previous figure.

2.3 Data Association

The next step in this assignment is to implement the data association function which has the task of constructing the Jacobian and innovation matrices required to do the update step in this filter. Basically, this was done by following the next steps:

1. Iterating through each of the observed lines passed as an array of start and end points of each of those lines.
2. Inside the preceding loop, the polar coordinates of the current line was computed in the robot frame.
3. For each of the previous lines, another loop was used to iterate through each of the map features.
4. A function called lineDist was invoked to compute the jacobians, innovation, and Mahalanobis distance between the observed line and a map feature. lineDist function does this calculations by first converting the map line from the world frame to the robot frame and then using the jacobian of the transferred feature (with respect to to the transform and line) with the inverse jacobian obtained due to the conversion to construct the H matrix which will be used in the state update. Subsequently, the innovation, uncertainty of innovation, and the Mahalanobis distance were computed using the following equations:

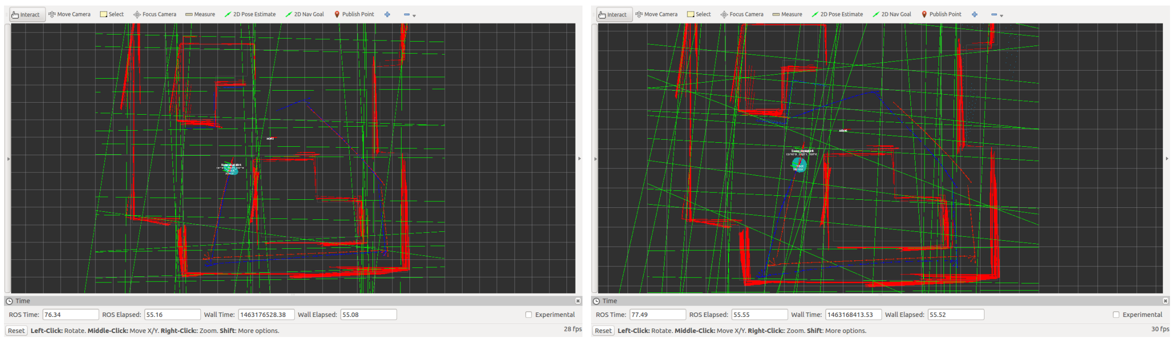
$$\begin{aligned} v_{ij} &= z_i - h(x_{k|k-1}) \\ S_{ij} &= H_j P_{k|k-1} H_j^T + R_i \\ D_{ij}^2 &= v_{ij}^T S_{ij}^{-1} v_{ij} \end{aligned}$$

5. Based on the previously obtained parameters, the map feature with smallest Mahalanobis distance with the current observed line was considered for association if this distance is smaller than *chi_thres* parameter. All the associated lines parameters were eventually appended into lists.

The value of *chi_thres* parameter has indeed the effect of increasing or decreasing the number of lines deemed as not associated. As a result, this gives a degree of freedom in the filter design that provides a balance between complexity and accuracy.

2.4 Update

The last step after doing the prediction, state augmentation, and data association is to update the state vector and the co-variance matrix. This is done by using the passed lists of innovation, jacobian, and measurements uncertainty of the associated lines. Eventually, the Kalman gain and innovation were used to update the position and the uncertainty of the robot. The result of running the final code with all the implemented steps is shown below.



ROS Time: 76.34 ROS Elapsed: 55.16 Wall Time: 1463176528.38 Wall Elapsed: 55.08

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Zoom. **Shift**: More options.

ROS Time: 77.49 ROS Elapsed: 55.55 Wall Time: 1463168413.53 Wall Elapsed: 55.52

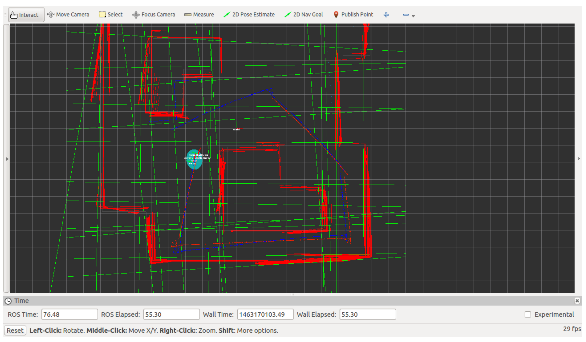
Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Zoom. **Shift**: More options.

Figure 3: Result of running the update step with *chi_thres*=0.2 (left) and *chi_thres*=0.5 (right)

The preceding figure reveals that using higher value of *chi_thres* causes more map features to be included in the state vector and consequently more memory and computations are required in this case.

3 Optional part

This section discusses and shows the results of implementing the optional part stated in the assignment. In order to improve the implemented SLAM algorithm, the weakness of the line extraction procedure, which results in adding some lines that are not in the environment to the scan, is taken into account. The idea is to count the number of times a feature has been observed and associated with one measurement, which was done in the data association function. Afterwards, the measurements that are associated with features that have been observed more than a minimum number (*min_observations*) are considered in the update step. Otherwise, those measurements were neglected. In fact, *min_observations* controls the rejection condition imposed and increasing this number will reject the less observed features. Therefore, the value of this number should be set in a way that reflects the accuracy of the extracted lines and consequently the used sensor. Figure 4 shows the result of executing the modified SLAM algorithm.



ROS Time: 76.48 ROS Elapsed: 55.30 Wall Time: 1463170103.49 Wall Elapsed: 55.30

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Zoom. **Shift**: More options.

29 fps

Figure 4: Result of running the modified SLAM

4 Conclusion

In this lab assignment, A simultaneous localization and mapping (SLAM) algorithm is implemented in python language and tested using ROS and rviz visualizer. The different steps are also analyzed and run to check their effects on the overall algorithm. Indeed, in those types of algorithms, a considerable attention should be paid to each step while implementing it in order to debug any possible error that could pop up while running the entire algorithm. Furthermore, parameter tuning is an an essential procedure that determines the overall complexity and accuracy of the final result.