Simultaneous Localization and Mapping of a 3D Space into a 2D Map

Rahul Agrawal Bejarano
University of Michigan
rahulab@umich.edu

Luke Cohen
University of Michigan
lukecohe@umich.edu

Fanzhong Kong
University of Michigan
kongfz@umich.edu

Yin Wang
University of Michigan
dianawww@umich.edu

Abstract

Simultaneous Localization and Mapping (SLAM) is a problem that requires creating a map of the environment while estimating the robot's or system's current location within this environment as it moves throughout it. We have created a SLAM algorithm that uses a dataset that consists of RGB-D images matched with IMU data that we use to estimate our trajectory and position, corrected using an Extended Kalman Filtering algorithm. This algorithm ultimately outputs a 2D map of the robot's movement throughout it's environment.

1. Introduction

A problem in the world of autonomous robotics is the ability for a fully autonomous robot to be able to perceive and understand its current state and through that information perform some kind of task. It plays an important role in robotics, especially when the robot is designed to work in some new environments, such as Mars, deep oil wells, and place in ruins. SLAM, simultaneous localization and mapping, is an algorithm used to map and localize an environment of a robot using on board sensors in order to allow the device to understand its current, and previous, state. A basic SLAM algorithm is able to read and process data from sensors then estimate the map and trajectories based on back-end optimization methods. It involves Computer Vision knowledge because as the development of CV, more algorithms rely on camera input data as it provides more information about objects and environments, compared to Lidar and Radar.

For this project, our group has researched the EKF-SLAM algorithm and instantiated our own implementation of it that uses RGB-D images matched with IMU data to output a 2-dimensional map of the robots movement throughout it's environment. Our project is broken down into three main sections: Landmark Extraction, Prediction, and Correction. Throughout this paper, we will discuss these sections and how they allow us to input an RGB-D video along with sensor readings and accurately output a 2-dimensional map of movement. [1]

2. Contribution

During this project, we re-implement the EKF-based SLAM algorithms over the datasets. The whole project includes data alignment, feature extraction, matching and projection, as well as EKF filter implementation. The datasets we use are from existing online creditable resources; but we do the data clean and data alignment on our own, which will be covered in the next section. We use the SIFT API in **OpenCV** to extract features; but the feature alignment is implemented by ourselves. The original EKF-SLAM algorithm is invented by [2] for stereo images as input, however, since we change the data set that takes monocular RGB-D images as our input, the model are slightly different from the one used in paper and we re-derive a few modified mathematical equations including feature projections by ourselves. We implement all the steps for each part without any help from 3rd party libraries like **scipy, filterpy**, except **numpy**. Finally, we combine all parts together and test over the datasets.

3. Data

3.1. Overview

Our data was taken from the Computer Vision Group, Department of Informatics at the Technical University of Munich. We downloaded their RGB-D dataset freiburg2 pioneer slam and RGB-D dataset freiburg2 pioneer slam2 databanks. This data was collected by from an Xbox Kinect camera mounted on a pioneer robot traversing through a maze that consisted of tables, containers and walls. It recorded a frames in RGB formats as well as depth mea-

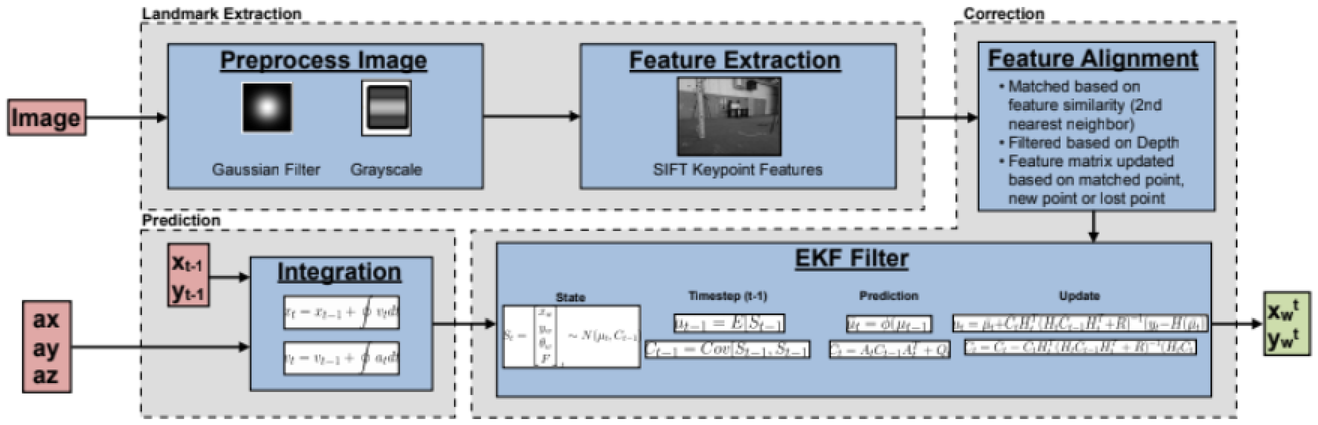
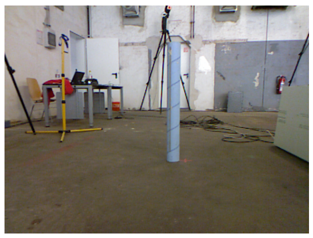
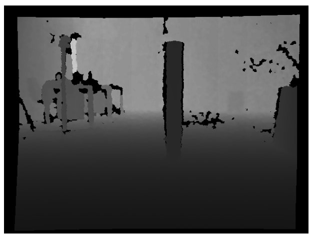

Landmark Extraction
Correction
Image
Preprocess Image
Gaussian Filter
Grayscale
Feature Extraction
SIFT Keypoint Features
Feature Alignment
• Matched based on feature similarity (2nd nearest neighbor)
• Filtered based on Depth
• Feature matrix updated based on matched point, new point or lost point
Prediction
x_{t-1}
y_{t-1}
ax
ay
az
Integration
$x_t = x_{t-1} + \int v_t dt$
$v_t = v_{t-1} + \int a_t dt$
EKF Filter
State
Timestep (t-1)
Prediction
Update
$S_t = [x_w, y_w, \theta_w, F] \sim N(\mu_t, C_{t-1})$
$\mu_{t-1} = E[S_{t-1}]$
$C_{t-1} = Cov[S_{t-1}, S_{t-1}]$
$\mu_t = \phi(\mu_{t-1})$
$C_t = A_t C_{t-1} A_t^T + Q$
$\mu_t = \mu_t + C_t H_t^T (H_t C_{t-1} H_t^T + R)^{-1} (y_t - H(\mu_t))$
$C_t = C_t - C_t H_t^T (H_t C_{t-1} H_t^T + R)^{-1} (H_t C_t)$
x_w^t
y_w^t

Figure 1: Framework

surements for each pixel in a frame respectively, examples of which can be seen in figures 1a and 1b. This was complemented with inertia measurement unit (IMU) data recording the movement of the robot. The movement of the robot was restricted moving in the x-y plane and rotations of the camera about the z axis. The dataset also included the groundtruth measurements of the position and trajectory of the robot for comparison and experimentation later in the project.


(a) 330rd RGB Frame


(b) 330rd Depth Frame

3.2. Data Alignment

The RGB and depth image frame collections were already lined up together on a timestamp basis however the IMU data was recorded on a much higher clock rate. Specifically, the RGB camera reads approximately 30 frames per second, and the IMU sensor reads approximately 500 frames per second. Considering that the camera and IMU sensor take data independently and there are more IMU being clocked, we had to manually align the correct IMU data to the frames we used. To do this, we iterated through the IMU data at each timestamp using a dynamic model, run the predict part of EKF and then move onto the next timestamp. If at some timestamp we are given an RGB frame, we run the update part of EKF and use the features from image to eliminate the covariance of predicted states. The alignment between timestamps of different sensors is simply the nearest neighbor search without any interpolation. This virtually "online" method is slightly different from a real model working on robots that requires identifying if there is data input at current timestamp of processor, but it should be enough for our demo.

3.3. Preprocessing and Corner Cases

We preprocessed the images by using a Gaussian filter to each image in order to reduce the noise in them. While data was close to ideal there were some corner cases that did affect how we found key features and matched them across frames, namely this was when the robot would cross over some cables in the environment causing the camera to shake. However since only happens once throughout the video we considered it negligible. For the IMU data, we passed it by a low-pass filter to roughly eliminate the noise of high frequency.

4. Methodology

There are many algorithms targeted to solve the optimization problem of SLAM. Among them, we have the three most popular methods: FastSLAM, EKF-SLAM, Graph-SLAM. Both FastSLAM and EKF-SLAM algorithms predict then only update the current state (also called online-SLAM), and the main difference is that EKF-SLAM uses first-order Taylor Expansion to approximate a nonlinera motion while FastSLAM adopts sampling method. In contrast, Graph-SLAM is more powerful but expensive since it updates the full map periodically using all valid observations (also called full-SLAM). Our final decision is to implement an EKF-SLAM by ourselves because it is efficient and easy to be implemented.

Our EKF-based SLAM algorithm can be divided into three parts: prediction, feature extraction and correction, as described in Figure 1.

1. **Prediction**. For IMU data at each timestamp, we apply

the low-pass filter on acceleration data and then use purely dynamic integration to predict the next robot / camera pos. This is not a final state since the IMU data contains noise, which should be eliminated in the correction part.

2. **Feature Processing**. For each frame of the video, we preprocess the video frame and extract the features. We then match the features from current frame with the one from previous frame and project the pairs onto a 2D plane (the ground surface).

3. **Correction**. If the IMU data timestamp is aligned to frame timestamp, we perform the correction (update) step to modify the states' distribution by eliminating the difference between 2D location of features extracted from current frame and ones integrated from previous frame by a dynamic model. If no image data is achievable at the current IMU timestamp, we just take the predicted states as the input for next timestamp.

This is an overview of our EKF-based SLAM, and the detailed algorithm will be discussed below.

4.1. Predictions

For the EKF model, first we define the states at time t as

$$S_t = \begin{bmatrix} X_t \\ Y_t \\ \theta_t \\ \{\mathbf{F}\}_t \end{bmatrix} \sim \mathcal{N}(\mu_t, \Sigma_t) \tag{1}$$

where X_t, Y_t are the the x and y coordinates of a robot (camera) in the 2-Dimensional world coordinate system, which is defined to be the ground surface with center at the initial state of camera. θ_t is the camera direction and $\{\mathbf{F}\}_t$ represents the location of features. $\{\mathbf{F}\} = [\ldots, X_t^{(i)}, Y_t^{(i)}, \ldots]^T$, where $X_t^{(i)}, Y_t^{(i)}$ are the the x and y coordinates of i-th feature in the 2-D world coordinate system. Hence there will be totally $(3 + 2N)$ variables stored in the state, and N represent the number of features that is tracked in this system.

In Extended-Kalman filter, we assumes the state satisfies Gaussian distribution, with μ_t, Σ_t as the current mean and covariance. For simplicity, the correlation between features are set to zeros and the prior distribution of states is $\mathcal{N}(\mathbf{0}, I_N)$.

Note that for a static feature, its global location should never change whatever the pos of camea is. Therefore during prediction, we only modify the first three states that

$$\begin{bmatrix} \bar{X}_t \\ \bar{Y}_t \\ \bar{\theta}_t \end{bmatrix} = \begin{bmatrix} X_{t-1} \\ Y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} v_t T \cos(\omega_t T + \theta_{t-1}) \\ v_t T \sin(\omega_t T + \theta_{t-1}) \\ \omega_t T \end{bmatrix} + \mathcal{N}(0, R_t) \tag{2}$$

where $\bar{(\cdot)}$ represents the predicted state using a dynamic model, and v_t, w_t are velocity and angular velocity approximated by the IMU accelerator data,

$$v_t = v_{t-1} + \int_{t-1}^{t} a_v dt \approx v_{t-1} + a_v \delta t$$
$$w_t = w_{t-1} + \int_{t-1}^{t} a_w dt \approx w_{t-1} + a_w \delta t$$

$\mathcal{N}(0, R_t)$ is the Gaussian noise and the variance R_t should satisfy $R_t = V_t M_t V_t^T$ with V_t being the Jacobian matrix of (X_t, Y_t, θ_t) wrt. (v_t, w_t), which represents the error propagating from velocities to the positions and can be directly derived from Equation 2.

$$\begin{aligned} V_t &= \frac{[X_t, Y_t, \theta_t]^T}{[v_t, w_t]^T} \\ &= \begin{bmatrix} \delta t \cos(\theta_{t-1} + \delta t w_t) & -\delta t^2 v_t \sin(\theta_{t-1} + \delta t w_t) \\ \delta t \sin(\theta_{t-1} + \delta t w_t) & \delta t^2 v_t \cos(\theta_{t-1} + \delta t w_t) \\ 0 & \delta t \end{bmatrix} \end{aligned}$$

M_t is then the variance matrix of accelerations that satisfying $u_t = [v_t, w_t]^T \sim \mathcal{N}(0, M_t)$, which should be obtained through testing. In experiment, we just tuned it and pick up the best one.

These error variance matrices are then used to predict the new covarince and increase the uncertainty of the states

$$\mathrm{Cov}([X_t, Y_t, \theta_t]^T) = G_t \,\mathrm{Cov}([X_{t-1}, Y_{t-1}, \theta_{t-1}]^T) G^T + R_t \tag{3}$$

where

$$G_t = \begin{bmatrix} 1 & 0 & -v_t \sin(\theta_{t-1} + w_t \delta t) \\ 0 & 1 & v_t \cos(\theta_{t-1} + w_t \delta t) \\ 0 & 0 & 0 \end{bmatrix} \tag{4}$$

Note that the mean of first three states $\mathbb{E}[[X_t, Y_t, \theta_t]^T]$ is simply Equation 2 without Gaussian noise. With the calculated mean and covariance of the camera pos, we can modify the first three variables of predicted mean $\bar{\mu}$ and covariance $\bar{\Sigma}$.

4.2. Feature Processing

Extraction and Matching

We used the SIFT algorithm from a third party library to generate and find different key feature points across the frames the RGB frames. After this we calculated relative distances, measuring relative similarity, of descriptors between consecutive frames. If two feature points were similar enough, described by a threshold hyperparameter, and the second nearest neighbor criteria we considered these two feature points matched.

Additionally we found that the depth images seemed to have a black border around them, seen in Figure 1b. This is where depth discontinuity happens and the depth camera fails to estimate its depth in the neighborhood, which would then misrepresent our depth perception of pixels. Therefore we filtered the key points we found by checking whether they were higher than a depth threshold to make sure not to get any key points on this border.

Furthermore we made sure to hold a 10 pixel upper limit between matches in order to assure accuracy. For efficiency, we only keep the top $N = 50$ aligned features as input for EKF, to avoid any unnecessary time for matrix computation. Note that the current features that are aligned with previous features have priority to be picked up as the input, since a multi-frame tracking leads to higher accuracy than bi-frame one. In this way we could start describing how the the pioneer robot was moving throughout the map by seeing how the key points moved on a frame to frame basis.

Due to the movement of robot, the aligned features sometimes move out of the image and are not trackable anymore. In this case, we just remove those features from the state vector and use new ones to substitute them. In order to do this we maintained and updated a matrix of all the key point features found/matched and another "flag" matrix that described whether these key points were new points found, or matched with a previous frames.

Projections

The projection of our method is slightly different from the reference paper in [2]. Since we are using a depth camera, the information we used from every RGB-D frame can be summarized as

$$x^{(i)} = \begin{bmatrix} u^{(i)} \\ v^{(i)} \\ z^{(i)} \end{bmatrix} \sim \mathcal{N}(0, N_t) \tag{5}$$

where u, v represents the horizontal and verticel pixel indices of ith feature, and $z_{(i)}$ is the depth value at (u, v) in depth image. N_t is the covariance of signal by Gaussian assumption. In this project, we use

$$N_t = \begin{bmatrix} (\sigma_u^{(i)})^2 & 0 & 0 \\ 0 & (\sigma_v^{(i)})^2 & 0 \\ 0 & 0 & (\sigma_z^{(i)})^2 \end{bmatrix}$$

The standard deviation are hyperparameters that needs to tune. Next, we can derive the 3D coordinates of a pixel by

$$\begin{bmatrix} X_r^{(i)} \\ Z_r^{(i)} \\ Z_r^{(i)} \end{bmatrix} = \begin{bmatrix} (u^{(i)} - c_x)z^{(i)})/(f_x f_{\text{scale}}) \\ z^{(i)}/f_{\text{scale}} \\ ((v^{(i)} - c_y)z^{(i)})/(f_x f_{\text{scale}}) \end{bmatrix} + \mathcal{N}(0, Q_t) \tag{6}$$

where c_x, c_y are the optical center and f_x, f_y are the focal lengths of the RGB camera, and $f_{\text{scale}} = 5000$ is a depth scale factor for this dataset. These intrinsic parameters are all available from the website of dataset we used. Note that the sub "r" notation means the derived location is in robot coordinates system and should be transferred to the wrold coordinate system by

$$\begin{bmatrix} X_w^{(i)} \\ Y_w^{(i)} \end{bmatrix} = \begin{bmatrix} X_r^{(i)} \\ Y_r^{(i)} \end{bmatrix} \tag{7}$$

$$= \begin{bmatrix} \sqrt{X_r^{(i)2} + Y_r^{(i)2}} \cos\left(\theta_t - \text{atan2}\left(X_r^{(i)}, Y_r^{(i)}\right)\right) \\ \sqrt{X_r^{(i)2} + Y_r^{(i)2}} \sin\left(\theta_t - \text{atan2}\left(X_r^{(i)}, Y_r^{(i)}\right)\right) \end{bmatrix} \tag{8}$$

Figure 3 represents a relationship of camera coordinate and world coordiante system.

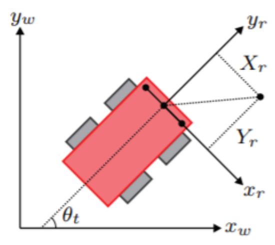


Figure 3: Landmark in camera/world coordinate system

For variance estimation, we estimate the covariance Q_t by

$$Q_t = W_t N_t W_t^T$$

where W_t is the Jacobian matrix of $(X_r^{(i)}, Y_r^{(i)})$ wrt. $(u^{(i)}, v^{(i)}, z^{(i)})$. Again, the expectation of feature positin $\mathbb{E}[[X_r^{(i)}, y_r^{(i)}]^T]$ is exactly the expression in Equation 6, the expected position of every feature at robot coordinate system is then transformed to the world frame by Equation 7.

4.3. Correction

The correction part can be summarized in the following algorithm.

Correction Algorithm

```
if $i \geq maxval$ then
end
$i \leftarrow 0$ else
end
$i + k \leq maxval$ $i \leftarrow i + k$
```

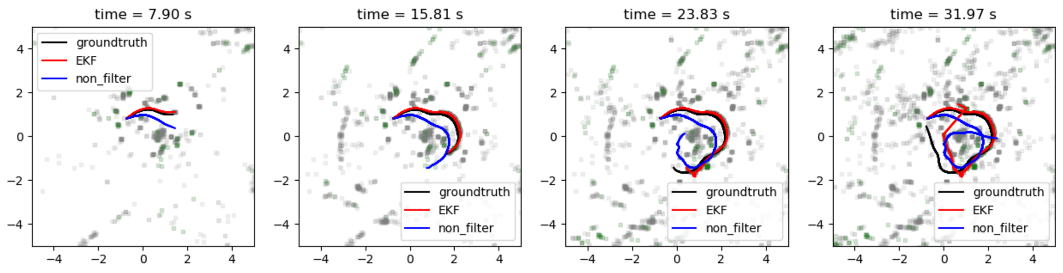


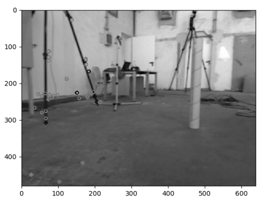
Figure 4: The mapping we create during different phases

5. Experiment

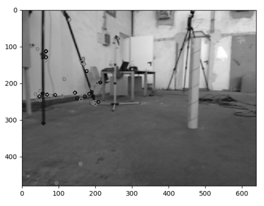
5.1. Feature Extraction and Alignment

We qualitatively analyzed judged our key point extraction and alignment across frames by looking at how close the key point matches were across the frames. We made sure to identify that these matches were close to their parent key points in the previous frame and that no other matches were found where no key points were found without parent key points.

An example of key point matching across frames can be seen in Figures 3a and 3b, gray circles show new key points found and the black circles are the key points matched in the subsequent frame.



(a) Key Points in frame 8



(b) Key Points in frame 9

5.2. Localization and Mapping

5.2.1 Localization

We run the EKF-SLAM over the dataset and get the prediction results. To better illustrate the strength of the EKF-SLAM, we run the EKF-SLAM against the baseline which only takes the prediction steps (non filter version). The resulting trajectories are listed below:

As we can see from the graph, EKF approximates the ground truth better than the non filter version. As time goes through, the EKF behaves more and more better than the non filter one. This is because the t timestamp prediction is based on t-1 timestamp prediction and the prediction error is cumulative through the iteration. Therefore, non-filter version produces much worse result because of lacking of correction. However, at the middle of the trajectory, the EKF generates a circle and produce not perfect matching after it. We reason it is because at that frame, the camera holder experiences a shake movement and most features cannot be aligned or even worse, generating misaligned pairs.

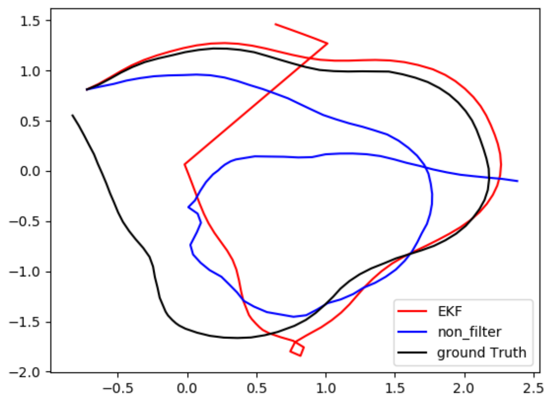


Figure 6: Trajectories generated from different algorithm

To better see which position parameter prediction (x, y, θ) results the issue, we produce the comparison on each position parameter.

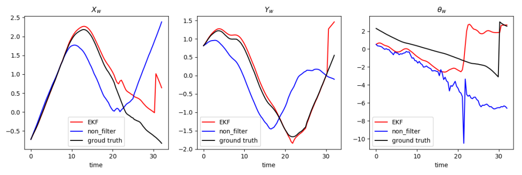


Figure 7: Trajectories generated from different algorithm on parameters

We see as x,and y approximates the ground truth greatly at beginning. However, at the time t = 20, the θ generates a bad prediction and consequently, ruining the sequential results. It shows the degree prediction is not robust when experiencing the shake. We reason it is because the shake produces a extreme large angular acceleration.

To quantitatively analyze the error, we calculate the square distance between the prediction trajectory and the ground truth. Then, take the average of all points.

	EKF SLAM	Non-Filter (Only prediction)
MSE(m)	0.304 (17.5%)	1.738 (100%)

Table 1: The MSE error of different algorithms

We can see the EKF SLAM version reduce the error to 17%, which marks the success of the EKF SLAM correction.

5.3. Mapping

We visualize the mapping we create on different timestamps. The pictures are shown below: The gray points in the figure 4 represent the features we detect but unable to align. The green points represent the features we detect and able to align, which will be used in EKF correction part. We notice that we can only use a very small portion of the features to improve the model.

For the left most graph, we notice that even though the overall direction is towards right, but it also discovers the features in the opposite direction, which seems unreasonable (because camera doesn't have chance to face in that direction). However, it is correct because the camera holder actually makes an rotation at the beginning which cannot be shown in the trajectory graph. And all abnormal features are detected at that moment.

Note the mapping is not very informative. It is because we use SIFT to extract features, which are some discrete points rather than large objects themselves. If we can extract features on object level, the mapping will be more straight forward and informative.

6. Conclusion

In conclusion, we were able to accomplish our goals and through feature extraction, and position prediction being combined in an Extended Kalman Filter, we were able to take a series of RGB and Depth images and map the path that the robot was traveling onto a 2D plane. Given that there was a large amount of matching points between frame t and frame t-1, our algorithm was able to output a highly accurate map of the trajectory of the robot onto a 2D plane.

While we may be able to achieve a strong mapping in perfect conditions, we have found that our algorithm is not very robust to a large amount of variation in frames. We think this could be due to both there being very few matching points between two consecutive frames and also because the EKF algorithm is only of first order accuracy and, although it is efficient, it will never be as robust as a graph-based optimization method.

Future work regarding this project would include making our 2D trajectory map a more accurate 3D trajectory map that would take into account changes in height as well as movement in the x, y plane. We would also be interested in making our feature detection model better so we can generate a map of the environment as well as a map of the trajectory. This is often done with lidar sensors that give an accurate representation of the surroundings. With these changes, we would be able to generate a 3D trajectory map of our robot that also maps the robots surroundings as well.

References

[1] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J.J. Leonard. Past, present, and future of simultaneous localization and mapping: Towards the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.

[2] W brink W Brink, C Van Daalen. Stereo vision as a sensor for ekf slam, 2011.