Simultaneous localization and mapping (SLAM) and texture mapping

Per Kvinnesland Omvik
PID: U07974631
UC San Diego

***Abstract*— This report showcases a procedure of using input from an IMU encoders and a lidar to implement a particle filter that can predict and correct the movement of a real world robot moving in the x,y-plane. Using information about the robot's position and orientation, texture mapping of the 2D-space is also discussed.**

I. INTRODUCTION

Being able to create a map of the immediate surroundings of a robot, vehicle or a vessel, and locating the object within that map is an ability that is beneficial in a plethora of situations. For any self driving car for instance, knowing the distances to the other cars and their velocity, is not only something that increases safety and facilitates collision detection, but it is instrumental to the autonomous operation of the car. As SLAM can be used different cases, there are many different approaches to it. These include the use of different light sensors (e.g. lidars, cameras, etc.), observation models, motion models, and filtering techniques.

Based on the resources [1], [2] and [3], this project will highlight the implementation of a particle filter to determine the position and orientation of a robot driving around, based on information from physical sensors, and a lidar. An Inertial Measurement Unit (IMU) and rotary encoders are used to predict the movement of the robot. A lidar situated on the robot itself is used to generate a 2D occupancy grid which is applied to correct the positional noise that can be expected from the sensors. This correction is performed through correlating new lidar readings to the existing ones, thus ensuring that the predicted position makes sense.

After the robot's trajectory is determined, depth and color information from a RGBD camera(Microsoft Kinect), along with homogeneous transformations in 3D space, is utilized to determine and color points in the occupancy grid that belong to the ground plane. The mathematical formulations of the problem, and the proposed solution is written in sections II and III. The results and discussions about them are found in section IV

II. PROBLEM FORMULATION

A. Occupancy grid

From the data that is captured by the lidar, we would like to detect walls and create the occupancy grid we use in the localization step. If we have an estimate of the current position of the robot (starting at $(0, 0)$, and angle $0°$) and the lidar scan at time t, we can use transformations to get the positions of the lidar readings in the world frame, which are points the lidar detects as being walls or objects. As with any sensor, the lidar cannot be fully trusted, and might change readings because of various noise in the system. Instead of a binary representation of the map, we can create a map of the probabilities that a certain cell is occupied(1) or free(0). The probability of a cell m_i's state given observations/lidar scans $z_{0:t}$ is expressed as:

$$p(m_i|z_{0:t}) = \gamma_{i,t}^{m_i}(1 - \gamma_{i,t})^{1-m_i} \tag{1}$$

How we find $\gamma_{i,t}$ is discussed in the next section.

B. Particle filter

We use the previous pose of a particle in the particle filter together with information from the encoders and IMU to predict the next pose of the robot/particle:

$$s_{t+1|t} = f\left(s_{t|t}, u_t + \epsilon_t\right) \tag{2}$$

u_t can be considered as the input to the movement of the robot. In this project the inputs themselves are unknown, but we can treat the angular velocity around the z-axis(yaw), measured by the IMU, and the rotational speed of the wheels, measured by the encoders, as inputs to the pose. At each update of the robot pose, we can generate a transformation between the robot's body frame and the world frame. Because we also know the transformation from the lidar to body frame, we can convert all points from the lidar scan "through" the body frame, and get their coordinates in the world frame, illustrated by a map that we can plot.

After reading from the lidar, and estimating a pose for the robot at time t, we can use the laser correlation model:

$$p_h(z_{t+1}|x_{s+1|t}, m) \propto exp\left(\textbf{corr}\left(y_{t+1}, m\right)\right) \tag{3}$$

To update/correct the pose of the particle to best correlate with the scan obtained by the lidar. y is a vector of the occupied cells in the new lidar scan and m is the existing occupancy grid.

C. Texture mapping

To be able to apply the texture map, we find the points that are a part of the floor using a transformation between a pixel p of the disparity image and a point in the 3D world frame:

$$[X_w, Y_w, Z_w]^T = TZ_oK^{-1}p \tag{4}$$

K is a matrix containing the intrinsic parameters of the disparity camera, Z_o is the depth in the optical frame captured by the disparity camera in the kinect. T is a

homogeneous transformation matrix from the the optical frame, via the robot body frame, and finally to the world frame.

III. TECHNICAL APPROACH

A. Occupancy grid

Keeping track of the probabilities of occupied cells in the occupancy grid described in (1) is equivalent to accumulating a log-odds value for each cell that is increased for every time that particular cell is observed, and decreased otherwise:

$$\lambda_{i,t+1} = \lambda_{i,t} + log\ g_h(z_{t+1}|m_i, s_t) \tag{5}$$

$g_h(z_{t+1}|m_i, x_t)$ is a measure on how much we trust the new measurement. To create the occupancy grid from the log-odds map this generates, we can set all cells with positive log-odds to 1(occupied) and negative to 0.

B. Particle filter

Before we can use the occupancy grid to correlate to the next lidar scan, we need to predict the next pose for each of the particles that will be used in (3). We determine the average linear and angular velocities v_t, ω_t since last encoder measurement by averaging the yaw rate and converting the encoder counts to meters, then dividing by the time τ since last measurement. Using these, we can implement the discrete-time differential drive model with current state s_t, to predict next state s_{t+1}:

$$s_{t+1} = s_t + \tau \begin{bmatrix} v_t \mathbf{sinc}(\frac{\omega\tau}{2}) cos(\theta_t + \frac{\omega\tau}{2}) \\ v_t \mathbf{sinc}(\frac{\omega\tau}{2}) sin(\theta_t + \frac{\omega\tau}{2}) \\ \omega_t \end{bmatrix} \tag{6}$$

In our case we also apply a gaussian noise with an arbitrary standard deviation of 0.01 to the state to account for noisy measurements from the IMU and encoders. This model is then applied to each particle in the particle filter to aqcuire $\mu^{(k)}_{t+1|t}$ (particle k's state after prediction step).

When a particles position is predicted, we can generate a transform that transforms points in the particle's body frame into our world frame. This comes in hand when doing the "update" step of the particle filter. When the lidar scan at $t+1$ arrives, for each particle, we transform the scan to the world frame. The world x, y coordinates is then fed in to the correlation function in (3). For all particles we check which nearby cell(location) is the most coherent with the current occupancy grid, and move the particle to this position. Furthermore, we choose which particle has achieved the highest correlation (also referred to as weight) for this scan, and updates the occupancy and log-odds grids according to the current pose of this particle. All K weights (α) are fed into the softmax function (7) to get normalized to a pdf.

$$\sigma(\alpha)_j = \frac{e^{\alpha_j}}{\sum_{k=1}^{K} e^{\alpha_j}}, \quad \text{for } j = 1, ..., K \tag{7}$$

If one particle has a substantially higher weight than all the others, resampling of the particles is performed to align them with the particle with the current highest weight

C. Texture mapping

The texture mapping procedure can be performed simultaneously with SLAM, but because of limited computational power, I chose to create it afterwards using saved trajectory and occupancy grids.

To find which pixels in the Kinect images that belongs to the ground plane, a threshold can be applied to the Z_w values resulting from the transform in (4). The matrix T is the total transformation from the cameras' optical frame, to the world frame

$$T = \begin{bmatrix} R_{oc}R_{wc}^T & -R_{oc}R_{wc}^T p_{wc} \\ \mathbf{0} & 1 \end{bmatrix}^{-1} \tag{8}$$

Where R_{oc} is the transformation from the regular to the camera frame, R_{wc} is the transform from the camera to the world frame defined using the current pose of the particle.

IV. RESULTS

A. Occupancy grid

The progress in this project was a gradual one, and performed on a step by step basis. The first step was converting the ranges and angles from the lidar scan to make sense in the x,y-plane. The log-odds map after the first scan is displayed in fig 1. Each cell of the log-odds map was initially set to -4 to avoid potentially stray lidar scans adding noise to the occupancy grid. This worked marginally better than initializing the map to zero.

B. Particle filter

The next step was constructing a motion model for the robot to use in predicting the following states. The trajectories for datasets 20,21 and 23 are displayed in figures 2-4. Examining the figures, we can conclude that the differential drive model described in (6) proved to be a good model, as the trajectories seem reasonable. The noise added to the state parameters (x, y, θ) was fairly small, and did not affect the trajectories significantly. For a robust particle filter, one could probably increase the noise parameters, without too much compromise. Because of the relatively strong performance using only the prediction step. The occupancy grids using this approach were good, but not perfect. The case with only one particle is displayed in figures 5-7.

The final step of the particle filter, the update step proved to be a difficult one. After extensive tweaking and testing, I never got results that outperformed SLAM using dead-reckoning. A recurring trend was that the procedure originally performed well, but once the robot entered a challenging area of the map, the model spun out of control. This is illustrated in figure 8. In these instances, I suspect that the issue was that the `mapCorrelation` function used to find the map cell in which to correct the particles' positions to, picked a cell that was coincidentally better than the optimal, but caused the robot to veer off course and becoming unable to correct itself. Also, there could have been issues in the way the particles were resampled, because I experienced frequent resampling and was unsure if this was

expected behaviour. Due to time limitations, and computing constraints I was unable to find my underlying issue.

The gifs of the occupancy grid being formed over time along with the robots position are linked in [4], [5] and [6].

C. Texture mapping

Because of the unsatisfactory performance of the particle filter, the texture mapping had to be performed using an imperfect robot trajectory. That being said, the implemented method worked well, and gave reasonable results in both cases, shown in figures 9 and 10. In both cases, the threshold of the points in the z-axis was set to $0.2m$. The reason for not setting the value closer to zero, as would have been intuitive, was the excessive amount of false negative "floor points". This caused the texture map to become sparser than wanted.

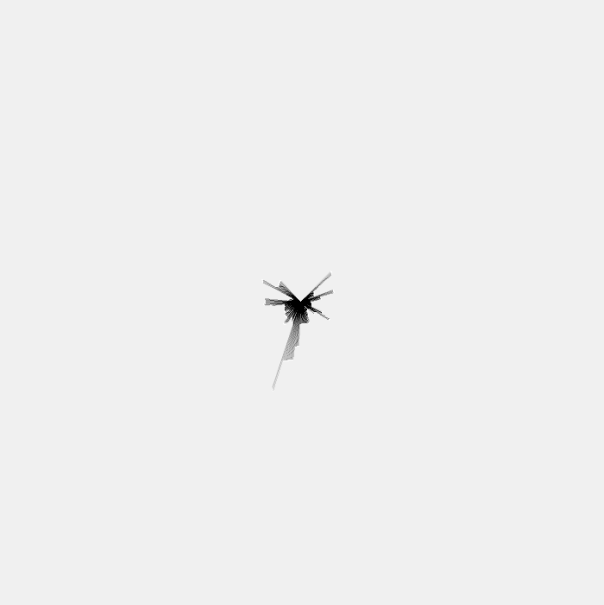

Fig. 1: Log-odds map after first lidar scan (trainset 20)

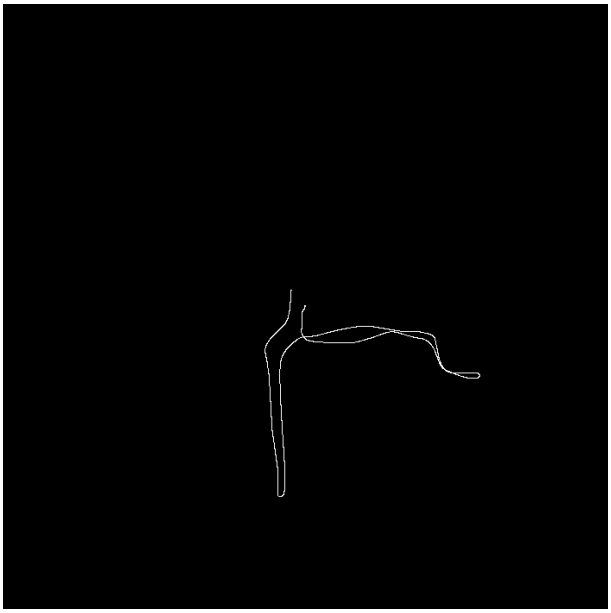

Fig. 2: Dead-reckoning trajectory in trainset 20

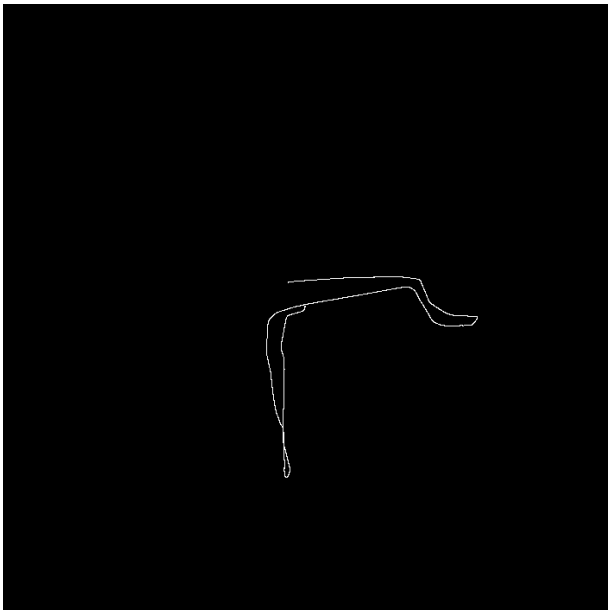

Fig. 3: Dead-reckoning trajectory in trainset 21

REFERENCES

[1] Antanasov, Nikolay, Lecture 6: Bayesian and Particle Filtering, UC San Diego, 2019

[2] Antanasov, Nikolay, Lecture 8: Motion and Observation Models, UC San Diego, 2019

[3] Antanasov, Nikolay, Lecture 9: Particle Filter SLAM, UC San Diego, 2019

[4] Gif of trainset 20

[5] Gif of trainset 21

[6] Gif of trainset 23

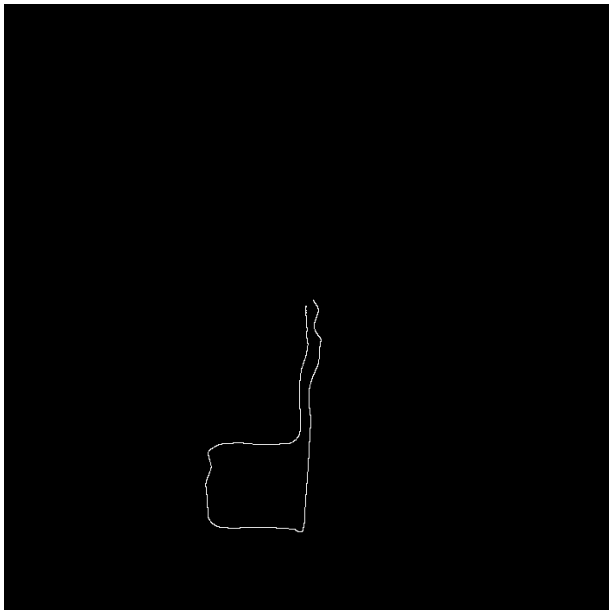

Fig. 4: Dead-reckoning trajectory in trainset 23

Fig. 5: Occupancy grid for trainset 20

Fig. 6: Occupancy grid for trainset 21

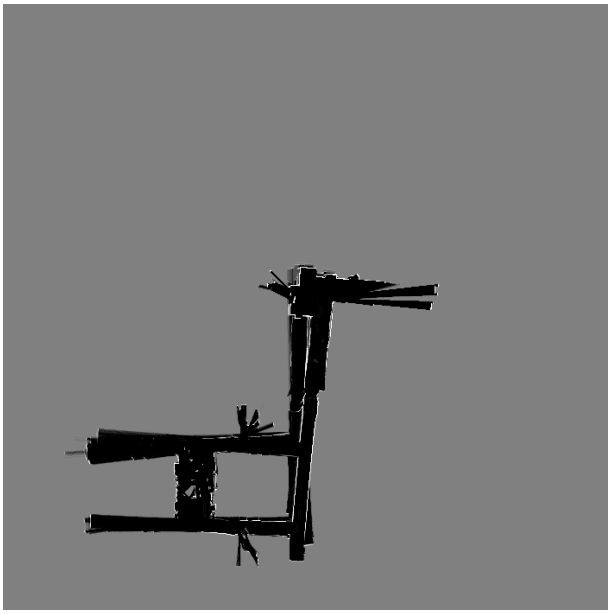

Fig. 7: Occupancy grid for trainset 23

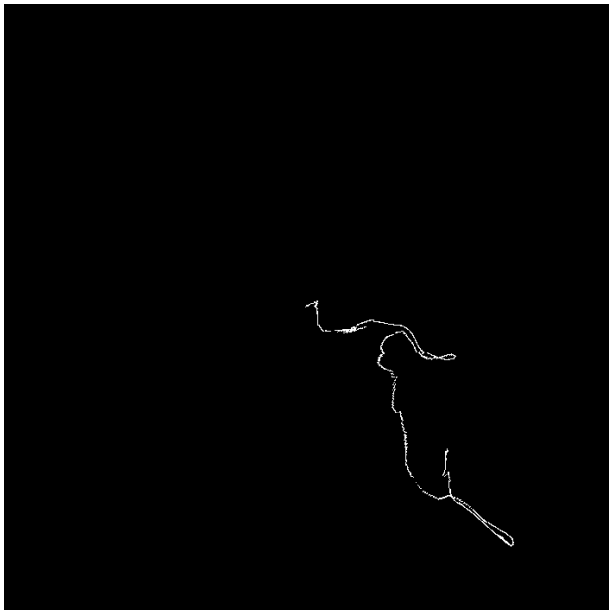

Fig. 8: The trajectory is similar to dead-reckoning in the top left, but spirals later

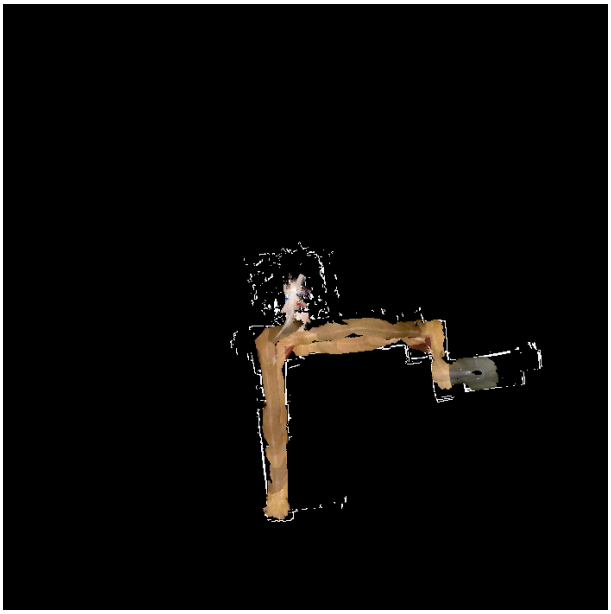

Fig. 9: Texture map for trainset 20

Fig. 10: Texture map for trainset 21