LiDAR-Based SLAM

ECE 276a Project 2

Pedram Aghazadeh
Department of Electrical and Computer Engineering
University of California, San Diego
La Jolla, CA, U.S.
paghazadeh@ucsd.edu

***Abstract*—This paper addresses the problem of LiDAR-based Simultaneous Localization and Mapping (SLAM) on a differential-drive robot equipped with an Inertial Measurement Unit (IMU), an encoder for each wheel, a Light Detection and Ranging (LiDAR) sensor, and an RGB-D Kinect camera. The objective is to estimate the robot's trajectory and construct a 2D occupancy map and a texture map of the environment. The SLAM pipeline consists of four key stages: (1) initial odometry estimation using encoder and IMU measurements, (2) refinement of the trajectory via LiDAR scan matching using the Iterative Closest Point (ICP) algorithm, (3) occupancy grid and texture mapping based on the estimated trajectory, and (4) trajectory optimization using factor graph optimization with loop-closure constraints in GTSAM. The final trajectory correction improves localization accuracy and enhances the quality of the generated maps. Experimental results demonstrate the effectiveness of the proposed approach in achieving accurate localization and mapping.**

***Index Terms*—SLAM, Factor Graph Optimization, Iterative Closest Point, Odometry, LiDAR, Robotics**

INTRODUCTION

Simultaneous Localization and Mapping (SLAM) is a fundamental problem in robotics, enabling autonomous systems to build maps of an unknown environment while simultaneously tracking their position. LiDAR-based SLAM is widely used in applications such as autonomous navigation and mobile robotics due to its high-precision environmental sensing. This project focuses on implementing a complete LiDAR-based SLAM pipeline, integrating multiple sensor modalities and optimization techniques.

The project is structured into four key components. First, odometry estimation is performed using encoder and IMU data based on a simple differential-drive motion model. Second, an improved trajectory estimate is obtained by applying Iterative Closest Point (ICP) for LiDAR scan matching using the odometry estimation as an intial guess for the ICP. Third, the estimated trajectory is utilized to construct a 2D occupancy grid map and a texture map using Kinect RGBD images. Finally, the trajectory is enhanced using pose graph optimization with loop-closure constraints implemented via GTSAM, improving localization accuracy and map consistency.

This report details the implementation of each stage, highlighting problem formulation, technical approach, and results obtained at each stage. Furthermore, if possible, at each step example of initial results and challenges faced are provided.

PROBLEM FORMULATION

Mapping

Given robot state trajectory $\mathbf{x}_{0:T}$ and sensor measurements $\mathbf{z}_{0:T}$ with observation model h, build a map $\mathbf{m}$ of the environment:

$$\min_{\mathbf{m}} \sum_{t=0}^{T} \|\mathbf{z}_t - h(\mathbf{x}_t, \mathbf{m})\|_2^2$$

Localization

Given a map $\mathbf{m}$ of the environment, sensor measurements $\mathbf{z}_{0:T}$ with observation model h, and control inputs $\mathbf{u}_{0:T-1}$ with motion model f, estimate the robot state trajectory $\mathbf{x}_{0:T}$:

$$\min_{\mathbf{x}_{0:T}} \sum_{t=0}^{T} \|\mathbf{z}_t - h(\mathbf{x}_t, \mathbf{m})\|_2^2 + \sum_{t=0}^{T-1} \|\mathbf{x}_{t+1} - f(\mathbf{x}_t, \mathbf{u}_t)\|_2^2$$

SLAM

Given initial robot state $\mathbf{x}_0$, sensor measurements $\mathbf{z}_{1:T}$ with observation model h, and control inputs $\mathbf{u}_{0:T-1}$ with motion model f, estimate the robot state trajectory $\mathbf{x}_{1:T}$ and build a map $\mathbf{m}$:

$$\min_{\mathbf{x}_{1:T}, \mathbf{m}} \sum_{t=1}^{T} \|\mathbf{z}_t - h(\mathbf{x}_t, \mathbf{m})\|_2^2 + \sum_{t=0}^{T-1} \|\mathbf{x}_{t+1} - f(\mathbf{x}_t, \mathbf{u}_t)\|_2^2$$

Given the dataset (x_0, z, u) where x_0 is the initial pose at the origin, z represents encoders and IMU angular velocities, z represents Lidar scans, and Kincet RGB-D images we aim to find the map m and robot state trajectories x_i by finding the pose transformation from one frame to another. This pose transformation is referred to as

$$_{t+1}T_t = \begin{bmatrix} R & \mathbf{p} \\ \mathbf{0}^T & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4}$$

where $R \in \mathbf{SO}(3)$ is a 3-d rotation matrix and $\mathbf{p} \in \mathbb{R}^3$ in the translation. We also show the pose at time step t as T_t, with $T_0 = I_4$ being the initial pose of the robot in the world frame. Note that the datasets provided all came with time-stamps but

were not synced. To fix this issue, all time-stamps were synced by the most important dataset at each stage using closest in the past time-stamp or smallest absolute value difference.

TECHNICAL APPROACH

A. Encoder

The encoders at each wheel record the number of rotations at a 40HZ and is reset after each reading, meaning that if after each rotation the wheel travels l meters and encoder records $0, 1, 0, 2, 3$, it corresponds to $(0 + 1 + 02 + 3)l = 2l$ meters of travel. All four wheels are identical and have a diameter of 0.254m and there are 360 ticks per revolution. Therefore, the wheels travel 0.0022 meter per tic recorded by encoder. Given encoder counts FR, FL, RR, RL corresponding to the front-right, front-left, rear-right, and rear-left wheels, the distances for each side are

$$D_R = \frac{(FR + RR) * 0.0022}{2}$$

$$D_L = \frac{(FL + RL) * 0.0022}{2}$$

hence the speeds for each side of the robot can be calculated using the time difference of consecutive recordings

$$V_R = \frac{D_R}{\tau}$$

$$V_L = \frac{D_L}{\tau}$$

and the speed of the center of the robot is

$$V = \frac{V_R + V_L}{2}$$

based on the differential drive robot shown in Fig -A.

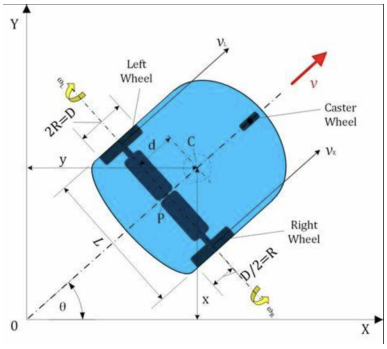


Figure -A: Kinematics of a Differential Drive Robot

B. Motion Model

Since the robot is *almost always* on a flat surface, from IMU we only used ω_{yaw} angular velocity to measure the rotation of the robot, which is recorded in rad/sec units. By syncing the time-stamps of the $Encoder$ with IMU we define the pose (x, y, θ) of robot and update using the following differential-drive motion model based on Fig -A:

$$f(x, y, \theta, v, \omega_{yaw}, \tau) = (x', y', \theta')$$

where

$$x' = x + v * \cos(\theta) * \tau$$

$$y' = y + v * \sin(\theta) * \tau$$

$$\theta' = \theta + \omega_{yaw} * \tau$$

C. Odometry from LIDAR points

The LiDAR sensor that was used in the robot has a field of view of 270 in the range of $[-135, 135]$. All the point clouds are considered to be at the same height (same height as the sensor from the ground) are at a distance between 0.1 and 30m, and all points closer or farther than this distance are discarded. All the 1081 points are also divided equally from -135 to 135 degree counter-clockwise (about 0.00436332). Given the distance d and angle θ of a point we extract the x, y coordinates in the LiDAR frame

$$LiDAR(d, \theta) = (d\cos\theta, d\sin\theta, 0)$$

and after that the points can be translated to the robot frame by adjusting the height of points. Then we use these points in our ICP algorithm to estimate the pose from time t to time $t + 1$, or rather find ${}_{t+1}T_t$:

$$X_{t+1} =_{t+1} T_t X_t$$

1) Kabsch Algorithm: The Kabsch algorithm finds the optimal transformation (rotation and translation) that aligns two sets of corresponding points by minimizing the mean-square error. Given points z, m our goal is to find rotation $R \in \mathbf{SO}(3)$ and translation $p \in \mathbb{R}^3$ such that

$$R, p = argmin_{R,p} f(R, p)$$

where

$$f(R, p) := \sum_{i=1}^{n} w_i ||m_i - (Rz_i + p)||_2^2$$

In order to solve this optimization problem, we first solve it with respect to p, that is given a valid rotation R, what is the optimal translation. First, by computing the centroids we get

$$\bar{m} = \frac{\sum_{i=1}^{n} w_i m_i}{w_i}, \bar{z} = \frac{\sum_{i=1}^{n} w_i z_i}{w_i}$$

and solving for

$$\nabla_p f(R, p) = 0$$

we get to

$$p = \bar{m} - R\bar{z}$$

Now our problem can be solved by optimizing with respect to R only, and after that by setting $p = \bar{m} - R\bar{z}$ we will successfully find the desired R, p. First, by plugging in $p = \bar{m} - R\bar{z}$:

$$f(R, p)$$

$$= f(R, \bar{m} - R\bar{z})$$

$$= \sum_{i=1}^{n} w_i ||m_i - (Rz_i + \bar{m} - R\bar{z})||_2^2$$

$$= \sum_{i=1}^{n} w_i ||(m_i - \bar{m}) - R(z_i - \bar{z})||_2^2$$

By centering the points

$$\delta_m = m - \bar{m}, \delta_z = z - \bar{z}$$

we can rewrite the objective function as

$$f(R) = \sum_{i=1}^{n} w_i ||R\delta_{z_i} - \delta_{m_i}||_2^2$$

and our goal would be to find

$$min_{R \in \mathbb{SO}(3)} \sum_{i=1}^{n} w_i ||R\delta_{z_i} - \delta_{m_i}||_2^2$$

by expanding the term inside the summation

$$\sum_{i=1}^{n} w_i ||R\delta_{z_i} - \delta_{m_i}||_2^2$$

$$= \sum_{i=1}^{n} w_i ||\delta_{z_i}^T R^T R \delta_{z_i} - 2\delta_{m_i}^T R \delta_{z_i} + \delta_{m_i}^T \delta_{m_i}||_2^2$$

$$= \sum_{i=1}^{n} w_i (\delta_{z_i}^T \delta_{z_i} - 2\delta_{m_i}^T R \delta_{z_i} + \delta_{m_i}^T \delta_{m_i})$$

note that $\delta_{z_i}^T \delta_{z_i}$ and $\delta_{m_i}^T \delta_{m_i}$ are constant with respect to R. Therefore, our goal would be to find R corresponding to

$$min \sum_{i=1}^{n} -w_i 2\delta_{m_i}^T R \delta_{z_i}$$

$$= max \sum_{i=1}^{n} w_i \delta_{m_i}^T R \delta_{z_i}$$

$$= max \sum_{i=1}^{n} w_i \mathbf{tr}(\delta_{m_i}^T R \delta_{z_i})$$

$$= \max \mathbf{tr}((\sum_{i=1}^{n} w_i \delta_{m_i}^T \delta_{z_i}) R)$$

This problem is also known as **Wahba's problem**. The goal would be to determine $R \in \mathbf{SO}(3)$ such that $tr(Q^T R)$ is maximized. In our case

$$Q = \sum_{i=1}^{n} w_i \delta_{m_i} \delta_{z_i}^T$$

.

The solution to this problem is described in the Kabsch algorithm detailed below using the singular value decomposition of Q matrix:

Require: Two point sets z, m with known correspondences
Ensure: Rotation matrix R and translation vector t

Compute centroids: $\bar{z} = \text{mean}(z)$, $\bar{m} = \text{mean}(m)$
Center the points: $\delta_z = z - \bar{z}$, $\delta_m = m - \bar{m}$
Compute covariance matrix: $Q = \delta_z^T \delta_m$
Perform SVD: $[U, S, V^T] = \text{SVD}(Q)$
Compute rotation matrix: $R = VU^T$
if $\det(R) < 0$ **then**
    Adjust for reflection: $V[:, -1] * = -1$
    Recompute $R = VU^T$
end if
Compute translation vector: $p = \bar{m} - R\bar{z}$
return R, p

2) Iterative Closest Point (ICP): The ICP algorithm iteratively refines the transformation between two point sets when correspondences are unknown.

3) Algorithm Steps:

1) Initialize transformation: $R = I, p = 0$.
2) Repeat until convergence:
   - Find nearest neighbors from z for each point in m.
   - Compute the best transformation using the Kabsch algorithm. This can be done using nearest neighbor or **KDTree** from *cv2* package.
   - Apply transformation: $\hat{m} \leftarrow Rz + p$.
   - Check convergence (e.g., error change $<$ threshold).

Iterative Closest Point (ICP)

Require: Source point set z, target point set m
Ensure: Optimal transformation R, p

Initialize: $R = I, p = \mathbf{0}$
repeat
    Find nearest neighbors for each point in z from m
    Compute optimal transformation using Kabsch algorithm
    Update transformation: $\hat{m} \leftarrow Rz + p$
    Check for convergence
until convergence
return R, p

D. Occupancy and Map generation

After applying ICP over point clouds the poses found are reasonably good for our purpose to construct an occupancy map and a texture map, where we use the LiDAR scans and RGB-d images respectively.

For the texture map we use the same LiDAR points translated in the world frame and a ray tracing algorithm. In our implementation the Bresenham algorithm for ray tracing was used to detect the empty cells in the map. Our map is set as a 700×700 grid, initialized by zero. For each point (x, y) detected on the map (ignoring the height of points or the $z - axis$) viewed from (x_{robot}, y_{robot}) we find the empty cells on the way of the beginning and end of the rays with Bresenham2D which returns a list of coordinates. Needless to say, to have higher quality in the map, the points have been centered and also scaled up by a factor. To account for the noise and uncertainty of the coordinates, we use the log odds to determine whether a cell is empty or full. The idea of log-odds can be described mathematically as followed:

$$\textbf{Occupancy map: } m \in X^{700 \times 700}$$

where $m_i = 1$ stands for occupancy and $m_i = -1$ stands for free cell and X is a Bernouli random variable. Probabilistically, we can use $\gamma_{i,t} = p(m_i = 1 | z_{0:t}, x_{0:t})$ to show the probability of occupancy and $1 - \gamma_{i,t}$ shows the probability of

a free cell at time t given the observations and trajectories up until that point. Using Bayes rule we have

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

$$= \frac{1}{\eta_t} p_h(z_t|mi = 1, xt) p(m_i = 1|z_{0:t-1}, x_{0:t-1})$$

$$= \frac{1}{\eta_t} p_h(z_t|m_i = 1, x_t)\gamma_{i,t-1}$$

similarly

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

$$= \frac{1}{\eta_t} p_h(z_t|m_i = -1, x_t)(1 - \gamma_{i,t})$$

By considering the odds ration, we no longer need to calculate the normalizing factor η_t:

$$o(m_i|z_{0:t}, x_{0:t}) = \frac{p(m_i = 1|z_{0:t}, x_{0:t})}{p(m_i = -1|z_{0:t}, x_{0:t})}$$

$$= \frac{\gamma_{i,t}}{1 - \gamma_{i,t}} = \frac{p_h(z_t|m_i = 1, x_t)}{p_h(z_t|m_i = -1, x_t)} \cdot \frac{\gamma_{i,t-1}}{1 - \gamma_{i,t-1}}$$

By defining observation model odds ratio $g_h(z_t|m_i, x_t)$:

$$g_h(z_t|m_i, x_t) = \frac{p_h(z_t|m_i = 1, x_t)}{p_h(z_t|m_i = -1, x_t)}$$

using Bayes rule again this is equal to

$$\frac{p_h(m_i = 1|z_t, x_t)}{p_h(m_i = -1|z_t, x_t)} \cdot \frac{p(m_i = -1)}{p(m_i = 1)}$$

where the second fraction is the map prior odds ratio that can be set to 1 (occupied and free cells are equally likely) or < 1 (optimistic about free space).

Taking the log-odds of the Bernouli random variable m_i:

$$\gamma_{i,t} = \gamma(m_i|z_{0:t}, x_{0:t}) = \log o(m_i|z_{0:t}, x_{0:t})$$

$$\gamma_{i,t} = \log p_(m_i = 1|z_t, x_t) p(m_i = -1|z_t, x_t) - \gamma_{i,0} + \gamma_{i,t-1}$$

which can be written as

$$\gamma_{i,t} = \gamma_{i,t-1} + (\Delta\gamma_{i,t} - \gamma_{i,0})$$

and in our implementation $\Delta\gamma_{i,t}$ is increased by $\log 4$ is the cell is occupied and decreased by the same amount when cell is marked as free, and the values are clipped in the range $[-20, 20]$ to avoid overflow. The final probability for each cell can be computed by applying sigmoid function $\sigma(x) = \frac{e^{-x}}{1+e^{-x}}$ and we use the probability in the gray-scale to plot occupancy map.

For the texture map, the process is mostly similar to occupancy map. First, we need to find the coordinates of points in the camera frame using the intrinsic parameters of the depth camera K.

$$K = \begin{bmatrix} f_{s_i} & f_{s_\theta} & c_u \\ 0 & f_{s_v} & c_v \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 585.05 & 0 & 242.94 \\ 0 & 585.05 & 315.85 \\ 0 & 0 & 1 \end{bmatrix}$$

. The RGB values are given in a normal image and the depth values are provided via a disparity image of similar resolution. Given the values d at pixel (i, j) of the disparity image we use the following ocnversion to obtain the depth and the pixel location $(rgbi, rgbj)$ of the associated RGB color:

$$dd = (-0.00304d + 3.31)$$

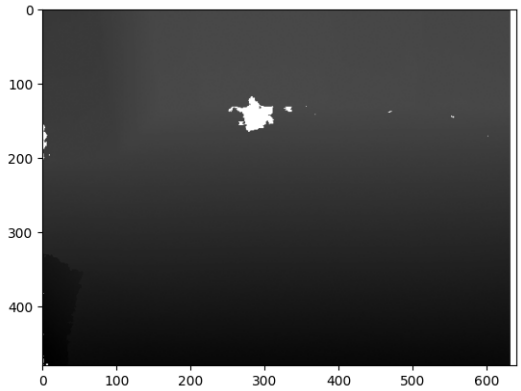
$$depth = \frac{1.03}{dd}$$

$$rgbi = \frac{526.37i + 19276 - 7877.07dd}{585.051}$$

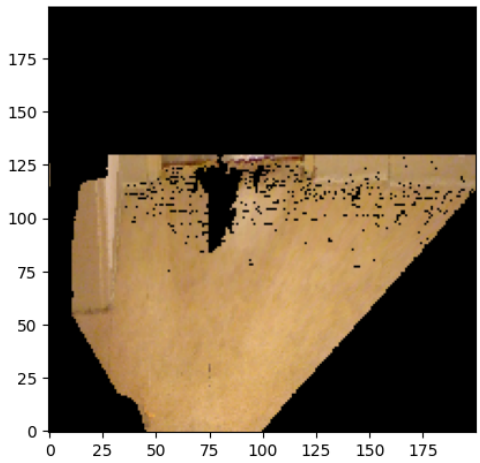
$$rgbj = \frac{526.37j + 16662}{585.051}$$

and using the values $(rgbj, rgbi, 1)$ we get the coordinates in the camera frame by $\begin{bmatrix} -y \\ -z \\ x \end{bmatrix} = K^{-1} \begin{bmatrix} rgbj \\ rgbi \\ 1 \end{bmatrix}$ Using the pose of the camera with respect to the robot frame, we translate the coordinates to robot frame and from there we use the poses estimated via ICP to find the points in the world frame. Once we have $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ coordinates in the world, we simply threshold over the distance of the points as we are interested in the points on the ground for the texture map and not the vertical feature like doors and walls. An example of the RGB and disparity images as well as their correct projection as a map is provided below.

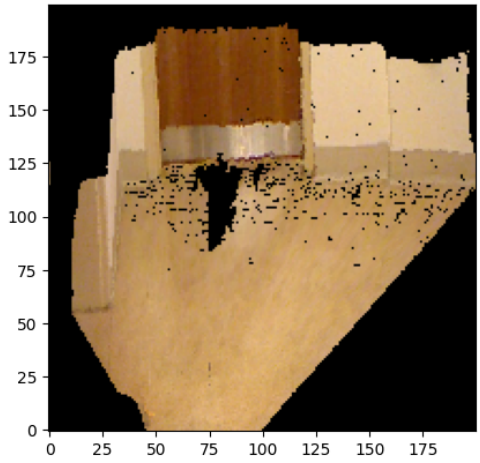
A sample of RGB image



A sample of disparity image



A sample of correct texture map with thresholding over heights



A sample of wrong texture map without thresholding over heights

E. Factor Graph SLAM

In order to improve the poses found by ICP, we employ factor graph optimization in SLAM.

A **factor graph** is a bipartite graph consisting of:

- **Nodes**: Representing the variables to be estimated, such as robot states $\mathbf{x}_t$ and landmark states $\mathbf{m}_j$.
- **Factors**: Representing constraints or measurements that relate the variables based on inputs $\mathbf{u}_t$ (controls) and observations $\mathbf{z}_t$.

A factor graph can be expressed as a collection of factors that encode relationships between variables:

- **Motion factor**: Encodes the error between the predicted and actual robot states:

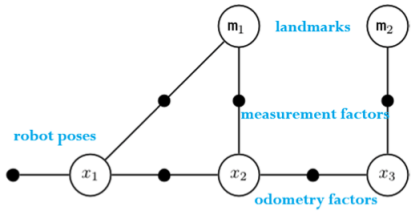
$$e_f(\mathbf{x}_{t+1}, \mathbf{x}_t) = \mathbf{x}_{t+1} \ominus f(\mathbf{x}_t, \mathbf{u}_t) \tag{1}$$

where $f(\mathbf{x}_t, \mathbf{u}_t)$ represents the motion model, and $\ominus$ represents a difference operation that respects the geometry of the space.

- **Observation factor**: Represents the difference between the observed and predicted measurement:

$$e_h(\mathbf{x}_t, \mathbf{m}_j) = \mathbf{z}_{t,j} \ominus h(\mathbf{x}_t, \mathbf{m}_j) \tag{2}$$

where $h(\mathbf{x}_t, \mathbf{m}_j)$ models the expected observation given the robot state and the landmark position.



Factor Graph

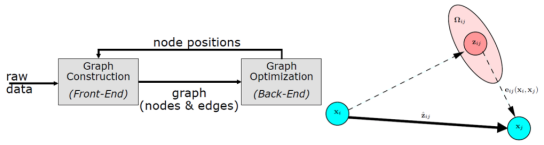
The operator $\ominus$ accounts for the geometry of the space:

$$y \ominus x = y - x, \quad \text{for } x, y \in \mathbb{R}^d \tag{3}$$

but follows a different formulation in Lie groups, such as $\mathbb{H}_*$.

Factor graph SLAM consists of two main components:

- **Front-end**: Constructs the factor graph using raw data from odometry, laser scans, and feature matching.
- **Back-end**: Optimizes the graph to estimate the variables $(\mathbf{x}_{0:T}, \{\mathbf{m}_j\})$.



Factor Graph Optimization

The process can be visualized as:

1) Constructing the graph using sensor data (front-end).
2) Formulating an optimization problem over the graph (back-end).
3) Solving for the optimal trajectory and landmarks.

F. Graph Optimization

Given a set of variables $\mathbf{x}_i$ associated with graph nodes and error factors $e(\mathbf{x}_i, \mathbf{x}_j)$ associated with graph edges, we define an optimization problem:

$$\min_{\{\mathbf{x}_i\}} \sum_{(i,j)\in\mathcal{E}} \phi_{ij}(e(\mathbf{x}_i, \mathbf{x}_j)) \tag{4}$$

where ϕ_{ij} is a distance function. A common choice is:

$$\phi_{ij}(e) = e^T \Omega_{ij} e \tag{5}$$

where Ω_{ij} is a positive-definite matrix representing the information (inverse covariance) of the measurement.

This optimization problem is typically solved using non-linear least squares methods such as Gauss-Newton or Levenberg-Marquardt.

In our factor graph implementation, we use the motion model 2d poses as initial guesses and use the poses from ICP as edges from one node of the graph to another as 3 dimensional poses. Note that in the initial implementation if ICP, the poses were saved cumulatively ${}_{world}T_i$ instead of ${}_{i+1}T_i$. Even though theoretically this way of saving the poses should not matter, due to numerical instability of numpy and

the inaccuracy of matrix inversion in Python, the outcomes were not useful. In theory, we have

$$_{i+1}T_i = (_{world}T_{i+1})^{-1} \cdot _{world}T_i$$

and for loop closure we'd need to compute $_{i+gap}T_i$:

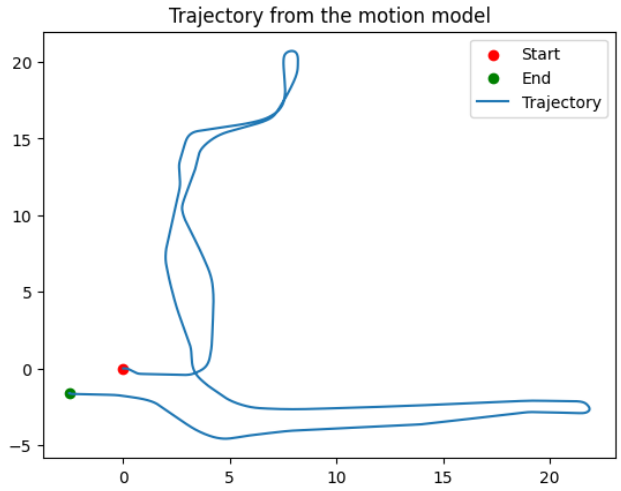
$$_{i+gap}T_i = (_{world}T_{i+gap})^{-1}_{world}T_i$$

but in practice this didn't work. Instead, the poses from ICP were computed again and stored frame-wise as $_{i+1}T_i$. There are two approaches for the loop closure, fixed-interval and proximity-based. In the former one we simply add an edge every 10 or 12 vertices which improves the results of ICP. For the latter one, we measure the similarity between LiDAR scans and if they are similar to a certain threshold, we add an edge between two corresponding vertices. The proximity-based with fixed-interval didn't change the results as much and the results of it by itself were slightly different from the ones done by fixed-interval.

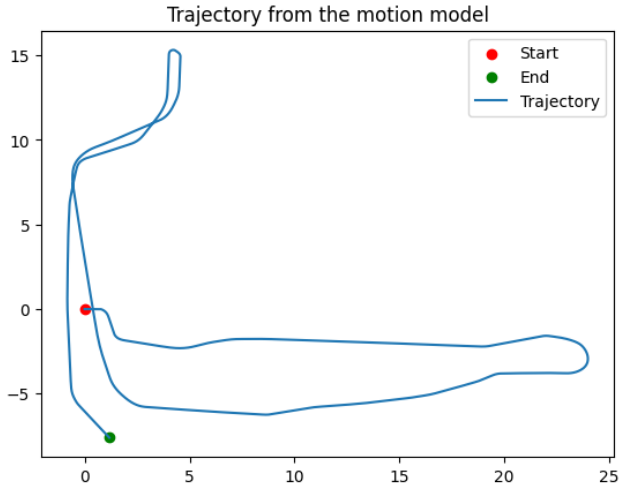
I. Results

A. Motion Model pose estimation

The motion model estimates of poses for both datasets provided are shown below:



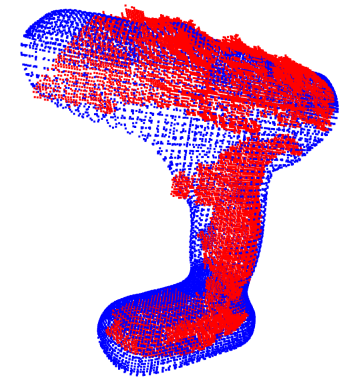
Poses from motion model on dataset 20



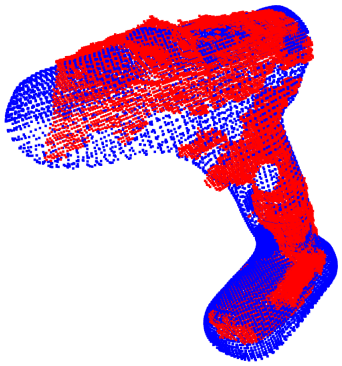
Poses from motion model on dataset 21

B. ICP-Warm up

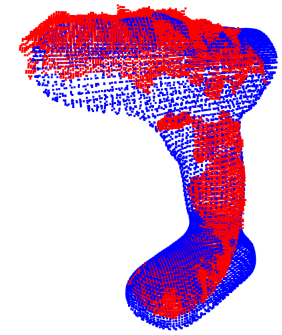
The results below are from applying the ICP on source LiDAR point clouds (blue) and 4 different target point clouds (red) of a drill and a liquid container. For the drill the results are found with initilizations on yaw only.



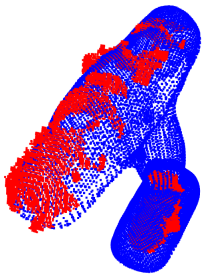
Drill 1



Drill 2

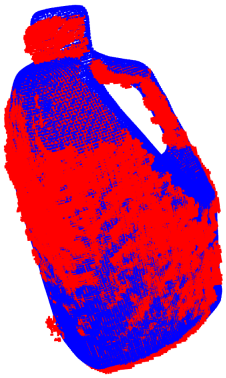


Drill 3

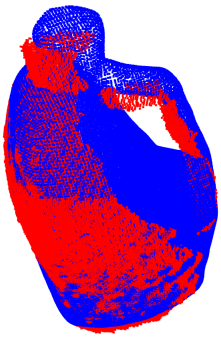


Drill 4

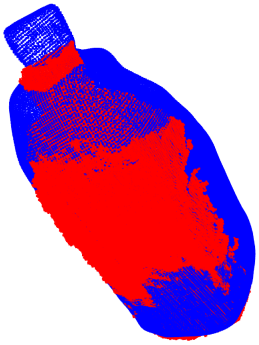
The liquid container scans include more point clouds and the results with yaw only initializations are decent but not as good as drill ones. Some of the yaw only results are



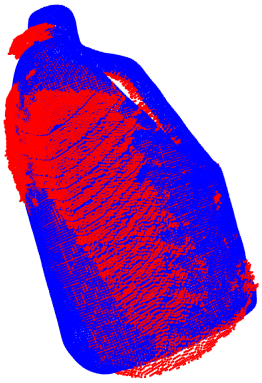
Liquid container 1 with yaw initializations



Liquid container 2 with yaw initializations

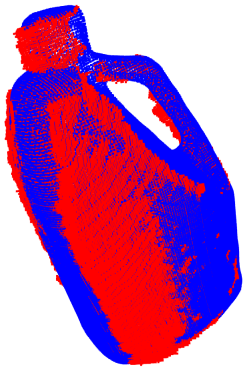


Liquid container 3 with yaw initializations

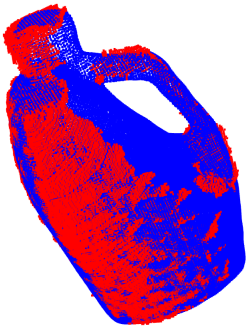


Liquid container 4 with yaw initializations

By adding roll to the initialization the ICP will converge to a better solution



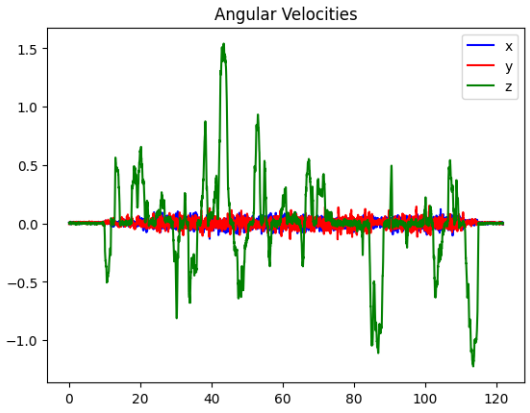
Liquid container 2 with roll and yaw initializations



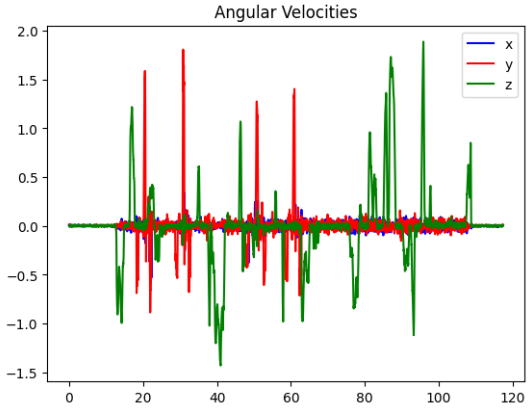
Liquid container 4 with roll and yaw initializations

C. ICP initialized with motion model poses

Checking the angles measure by IMU from both datasets we can see minimal changes in roll and pitch that make the initial pose estimation from simple motion model valuable and also help us to restrict the search space for the ICP initializations.

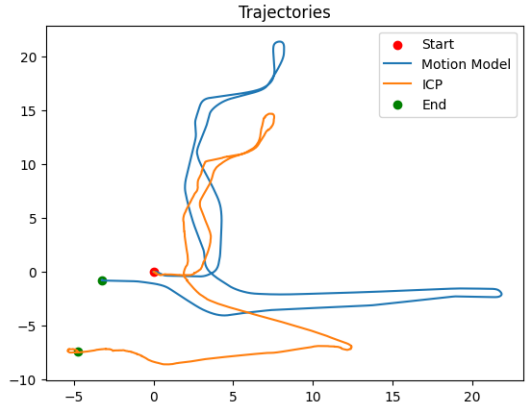


Angular velocities of dataset 20

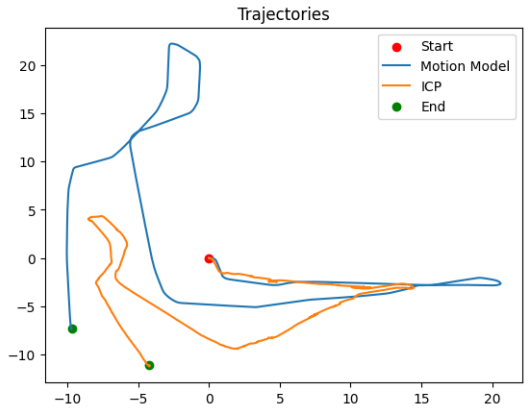


Angular velocities of dataset 21

Below are the results of ICP initialized with poses found with motion model. Initially, the yaw initializations for dataset 20 were covering a wide range $[-\pi, \pi]$ with small number of samples (5) and few iterations (50). This resulted in an error especially after the second U-turn of the robot which can bee seen below which took about 5 minutes for the entire dataset:



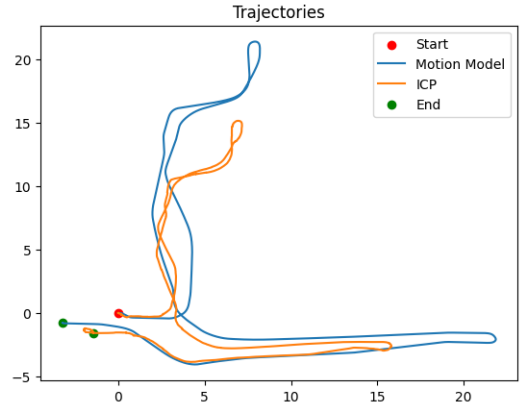
Undesirable ICP with initialization on dataset 20



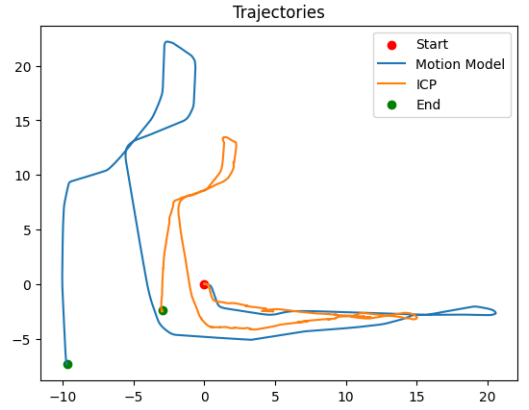
Undesirable ICP with initialization on dataset 21

In dataset 21 spikes over y axis can be seen which is caused by the robot climbing up and down an slide. This will not be an issue as ICP will be able to find a (possible local) minima for the pose tranformation which includes rotation along y-axis.

Since the rotations from one frame to another is not that great, the range was set to $[-\frac{\pi}{6}, \frac{\pi}{6}]$ with more points (15) and more iterations (70). This resulted in a much longer ICP process (3hrs for each dataset) but the results were run overnight and saved as $.npy$ files to be used for the next phase.

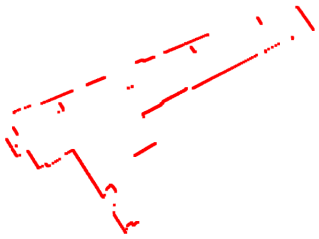


Final ICP with initialization on dataset 20

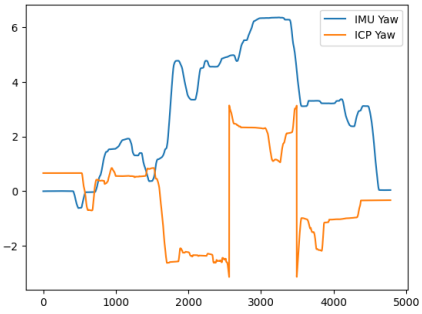


Final ICP with initialization on dataset 21

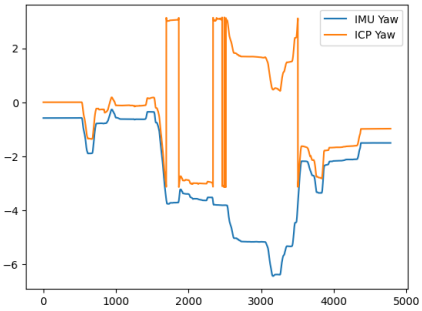
Finally, we compare the rotations along z-axis or yaw angles for both ICP and our initial motion model estimation.



A sample of LiDAR points from dataset 21



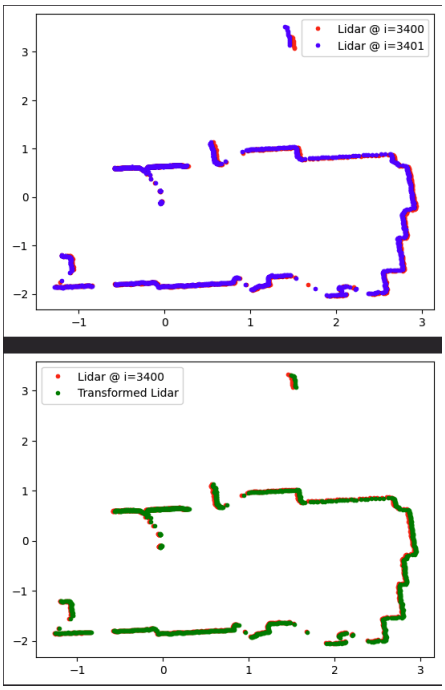
Yaw angles of dataset 20



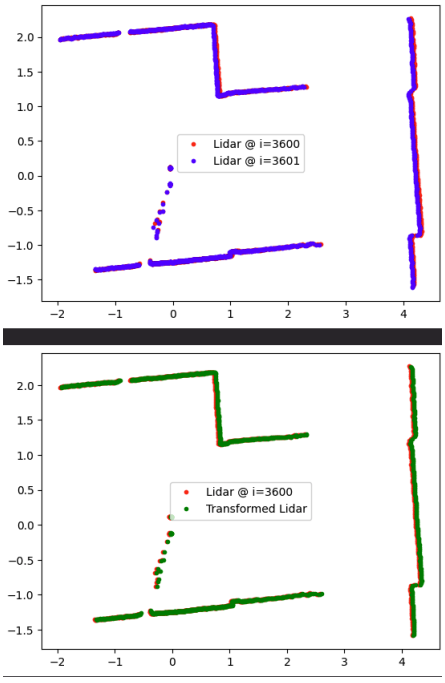
Yaw angles of dataset 21

As seen, the angles match each other closely when considering the equivalency of $-\pi \sim \pi$ or $-2\pi \sim 0 \sim 2\pi$. Then, occupancy maps are generated using the Bresenham2D and log-odds discussed earlier:

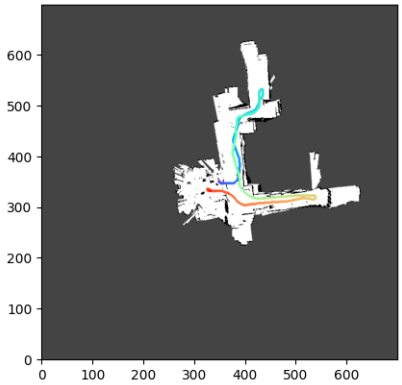
Furthermore, we check how well the LiDAR's match after the poses are applied



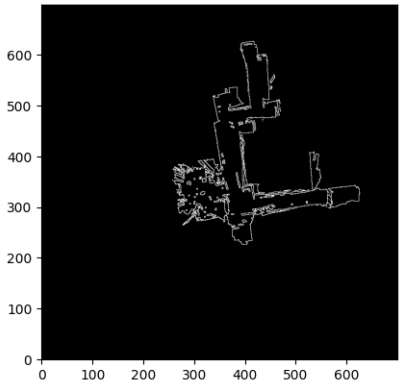
LiDAR verification dataset 20



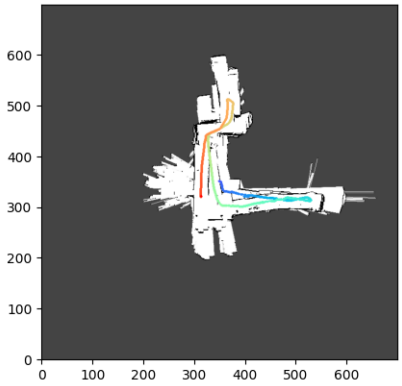
LiDAR verification dataset 21



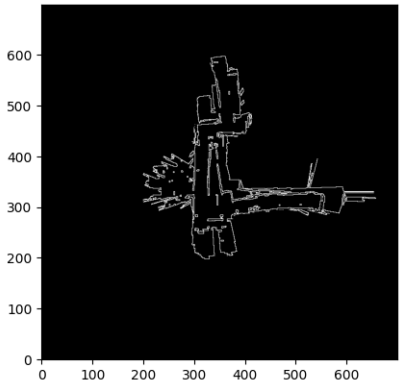
Occupancy map of dataset 20 with the traversed path shown in rainbow colors



Occupancy map's edges of dataset 21 with the traversed path shown in rainbow colors

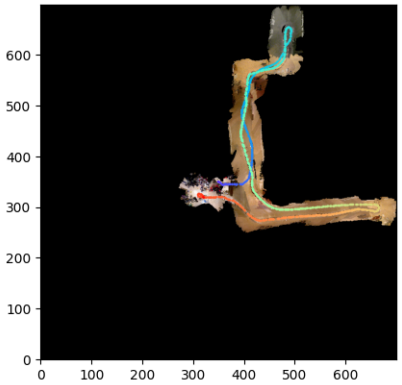


Occupancy map of dataset 21 with the traversed path shown in rainbow colors

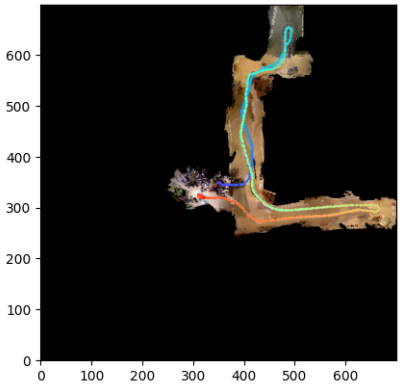


Occupancy map's edges of dataset 21 with the traversed path shown in rainbow colors

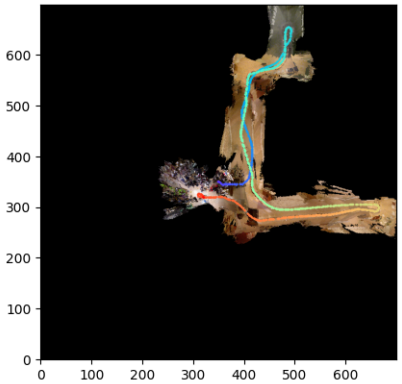
Below we can see the results for texture maps with different threshold levels:



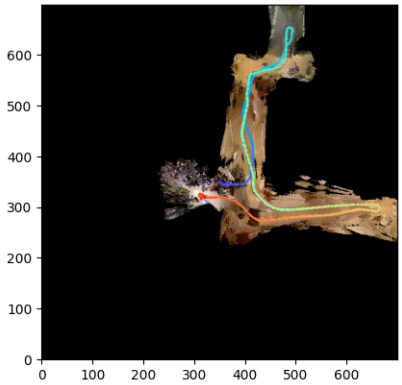
Texture map with a threshold of 1 for dataset 20



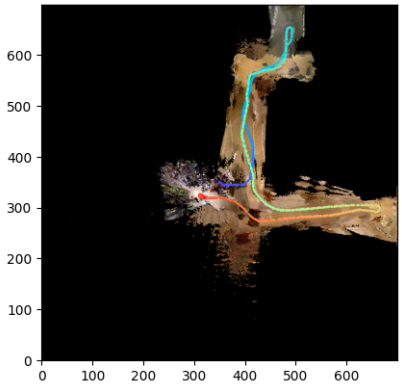
Texture map with a threshold of 2 for dataset 20



Texture map with a threshold of 5 for dataset 20

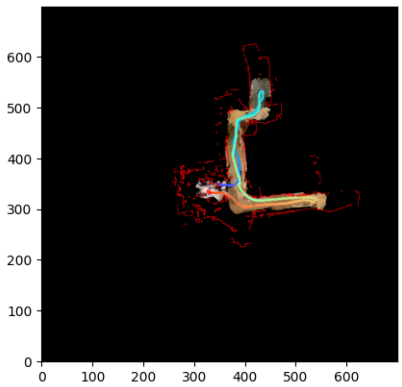


Texture map with a threshold of 10 for dataset 20



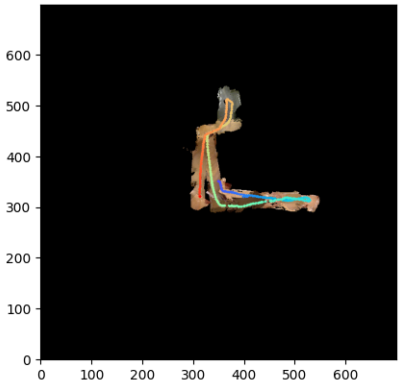
Texture map with a threshold of 100 for dataset 20

And by overlaying the occupancy map we get the map below

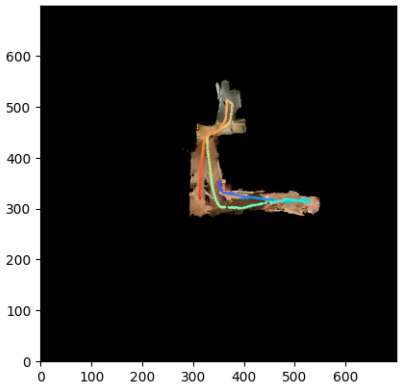


Occupancy map overlayed on texture map for dataset 20

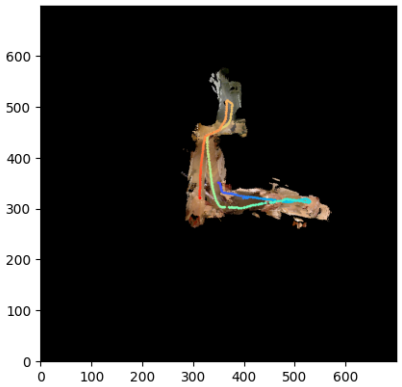
For dataset 21:



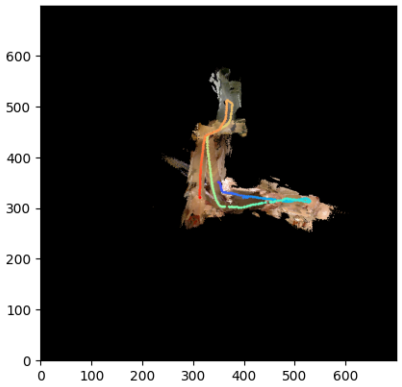
Texture map of dataset 21 with a threshold of 1



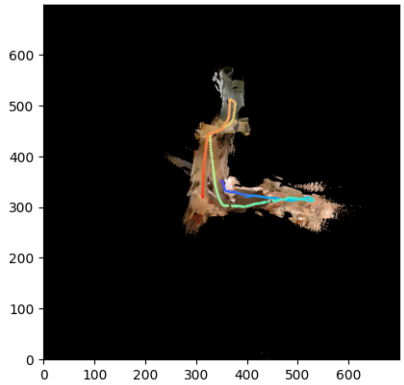
Texture map of dataset 21 with a threshold of 2



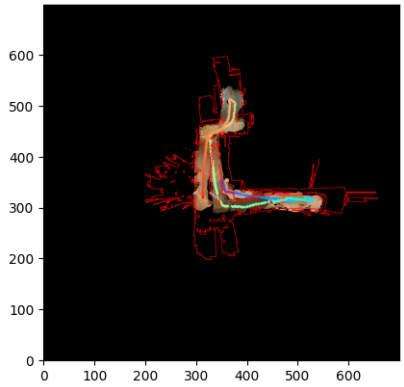
Texture map of dataset 21 with a threshold of 5



Texture map of dataset 21 with a threshold of 10



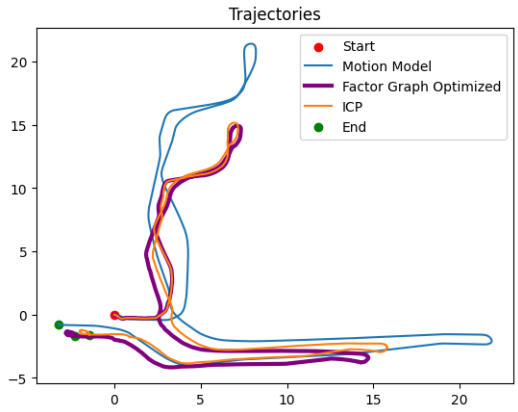
Texture map of dataset 21 with a threshold of 100



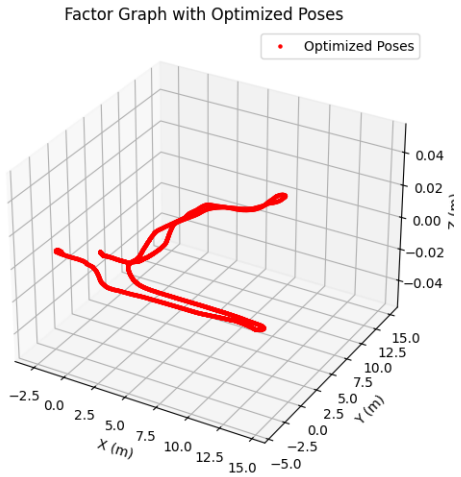
Occupancy map overlayed on texture map for dataset 21

Dataset 21 is more complicated as there are obstacles (visible in the videos provided in the repository) and pitch angles in the poses.

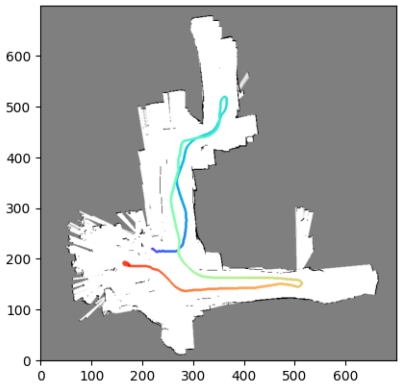
Factor graph optimized ICP poses look better than the ICP poses when using the not so great ICP poses for dataset 20. However, using the better poses for dataset 20 the difference is minimal.



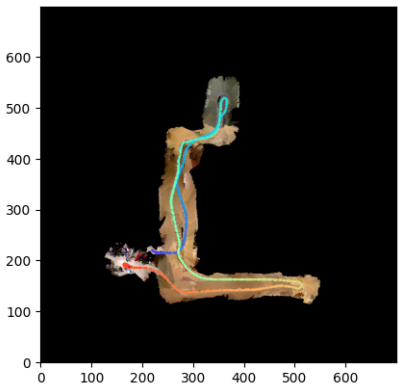
Dataset 20 with fixed-interval=12 loop closure



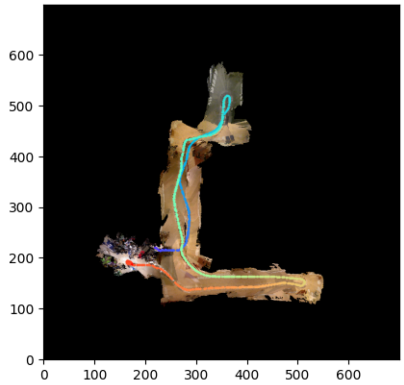
Dataset 20's 3d poses; The yaw angles are visible.



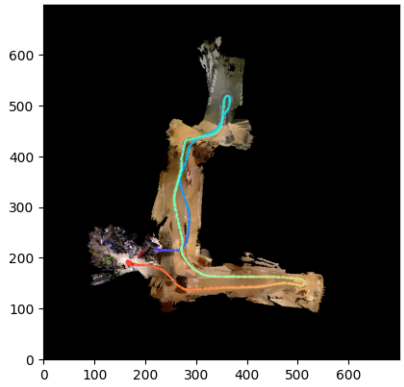
Occupancy map from dataset 20 with fixed-interval loop closure



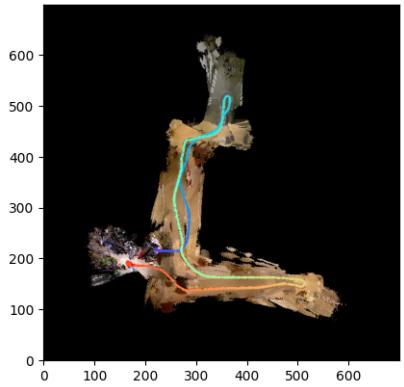
Texture map from dataset 20 on a threshold 1



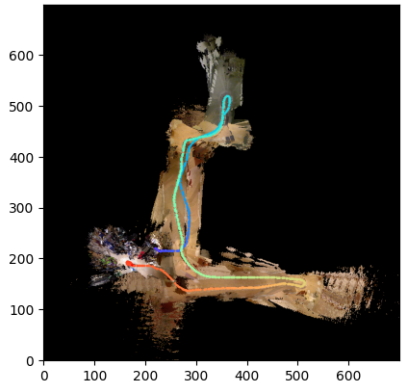
Texture map from dataset 20 on a threshold 2



Texture map from dataset 20 on a threshold 5

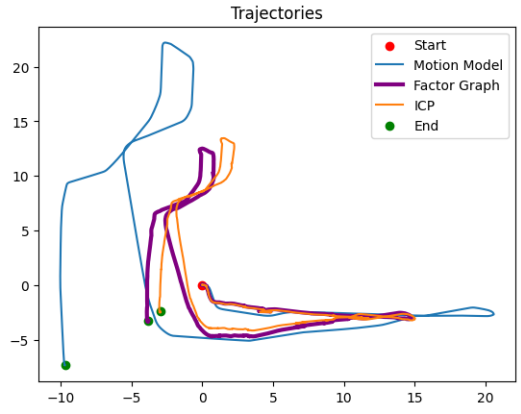


Texture map from dataset 20 on a threshold 10

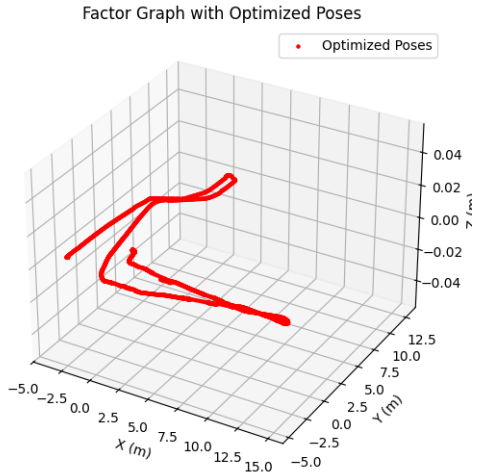


Texture map from dataset 20 on a threshold 100

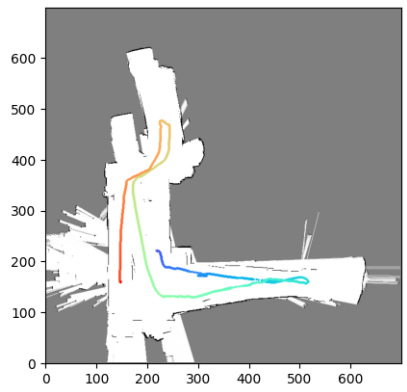
In dataset 21, the results of factor graph optimization are considerably better than ICP and Motion Model



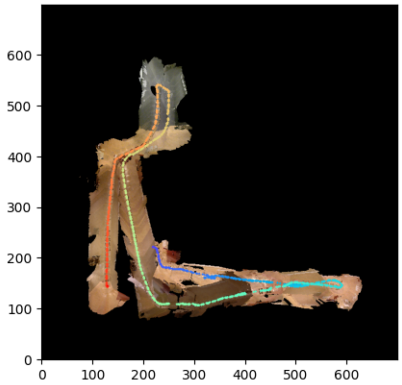
Dataset 21 with fixed-interval=10 loop closure



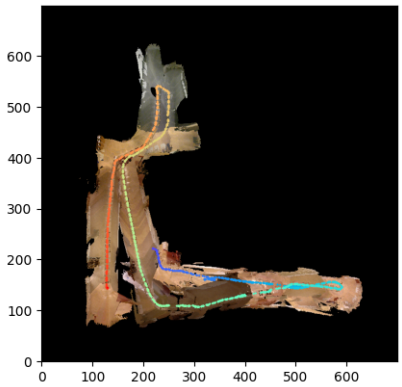
Dataset 21's 3d poses; The pitch angles are visible.



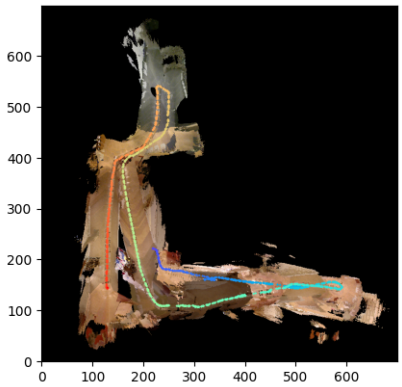
Occupancy map from dataset 21 with fixed-interval loop closure



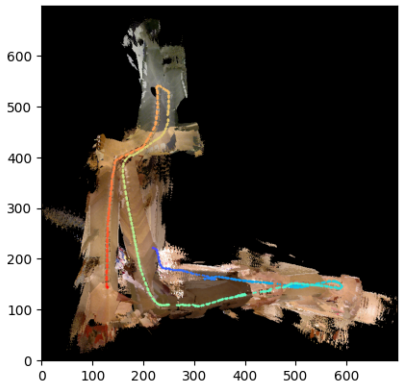
Texture map from dataset 21 on a threshold 1



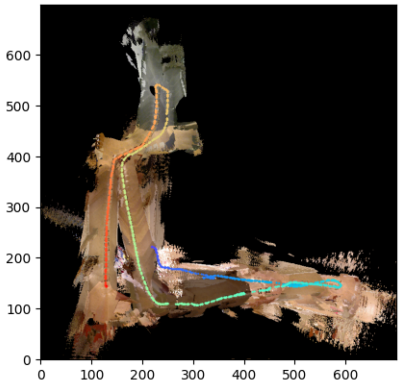
Texture map from dataset 21 on a threshold 2



Texture map from dataset 21 on a threshold 5

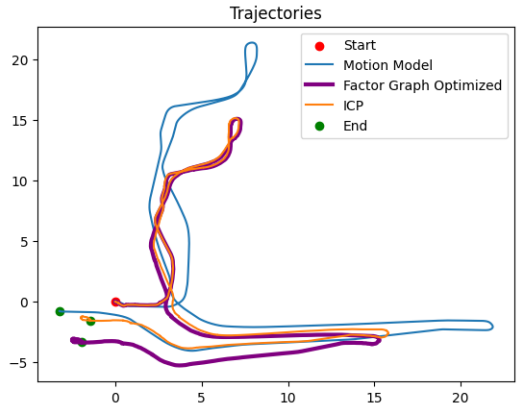


Texture map from dataset 21 on a threshold 10

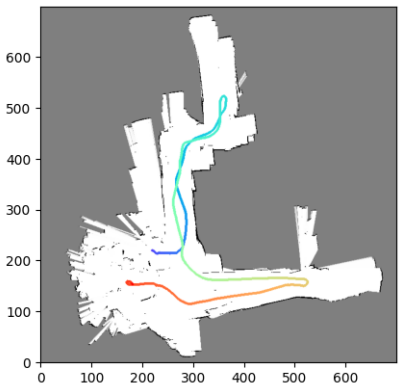


Texture map from dataset 21 on a threshold 100

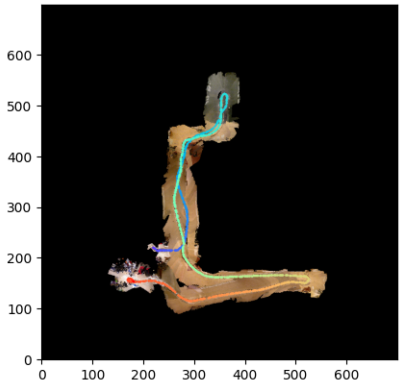
The results of the proximity-based loop closure are not better from the fixed-interval loop closure:



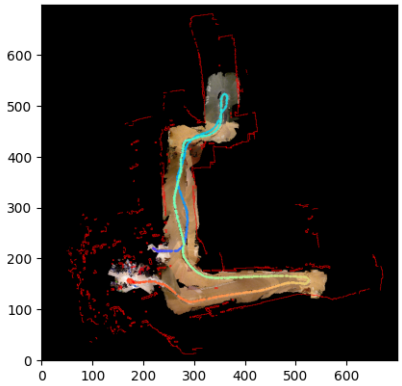
GTSAM with proximity-based loop closure on dataset 20



Occupancy map of GTSAM with proximity-based loop closure on dataset 20



Texture map of GTSAM with proximity-based loop closure on dataset 20



Overlayed map of GTSAM with proximity-based loop closure on dataset 20