ECE 276a PR 2 - LiDAR-Based SLAM

Kenneth Vuong
PID: A15552808
February 28th, 2025

***Abstract*—This project is an exercise in learning SLAM encoder and IMU odometry, point cloud registration, occupancy and texture mapping, and loop closures.**

I. Introduction

Simultaneous localization and mapping (SLAM) is a fundamental problem seen in mobile robot autonomy everywhere. From self-driving cars to vacuum robots, this technique gives a robot vision and a sense of the environment around it. As autonomy grows increasingly prevalent today, SLAM remains at the forefront of developments to push further to design even better robots.

We will be implementing LiDAR-Based Slam, utilizing encoder and IMU measurements, a lidar, and a Kinect RGBD camera to reconstruct a point cloud which describes the environment around a robot as it follows a trajectory.

A. Experiment Setup

The experiment setup is as follows. We utilize a differential-drive robot, which is equipped with the following items.

- Encoders - there are encoders placed at each wheel to measure rotation.
- IMU - an IMU is onboard the differential drive robot
- Hokuyo UTM-30LX Lidar - a horizontal lidar with 270○ degree field of view and maximum range of 30 m
- Kinect - RGBD camera which provides RGB images and disparity images.

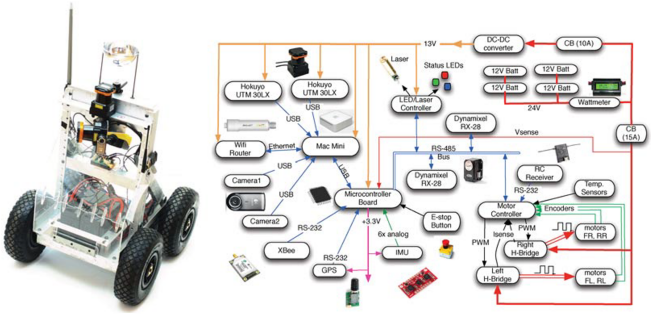


Fig. 1. Robot Electronics

We will assume the robot's origin is at the geometric center of the robot, i.e. the centered midpoint between the axles. This has some practical benefits, as the differential drive model we will be using assumes that the robot is rotating around its center. To solve the SLAM problem, I will be taking several steps. First, I will use the encoder and IMU data to generate a predicted trajectory the robot undergoes via the differential-drive motion model. Then, I will use the LiDAR scans to further refine this trajectory, using the calculated odometry to initialize the localization model under the iterative closest point (ICP) algorithm. Then, I will use the LiDAR data to generate a occupancy grid map that describes the environment in the world frame. This is done through utilizing the point clouds obtained from the LiDAR and combining it with the ICP trajectory, then using the Bresenham algorithm to determine the free space between the robot and any obstacles. A texture map is also created here using the same generated trajectories and through projection of the RGBD points onto the floor of the world frame. Finally, the GTSAM library is utilized to provide an optimized trajectory and the occupancy grid and texture maps are regenerated.

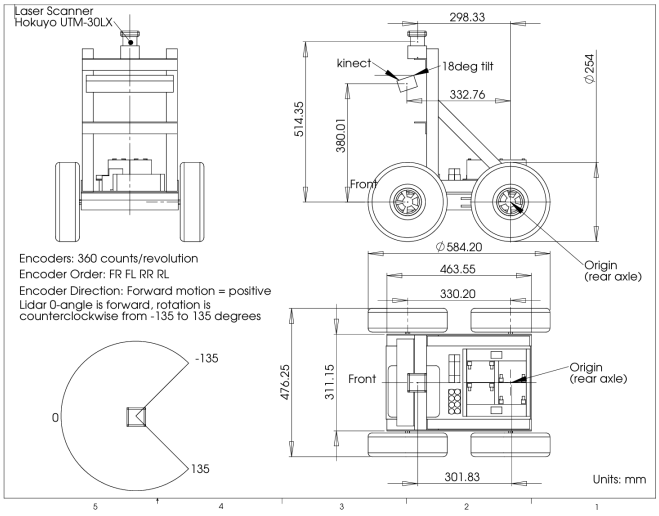


Fig. 2. Robot Drawing

II. Problem Formulation

We will go through data preprocessing to retrieve relevant data then go into the differential drive model, iterative closest points algorithm, occupancy grid probabilities, Bresanham algorithm, texture mapping transformation, and loop closure logic. For the robot, the initial pose is assumed to be the identity pose.

A. Data Preprocessing

1) Encoder and IMU Data: The data given for both the encoder and IMU are assumed to be calibrated. We assume

that the only relevant angular velocity is yaw, which is given. We can relate the rotation of each wheel (front right, front left, rear right, rear left) to determine the overall velocity of the robot. The encoders count at 40 Hz, the counter is reset after each reading. The datasheet shows that there are 360 ticks per revolution and that the diameter of our wheel is 0.254 meters. The distance travelled over one revolution of the wheel is $0.254 * \pi$, so the around per tick is $\frac{0.254*\pi}{360} = 0.0022m$. Given encoder counts $[FR, FL, RR, RL]$, We can average the distance for the left and the right wheels as follows:

$$D_{right} = 0.0022 * \frac{FR + RR}{2}$$

$$D_{left} = 0.0022 * \frac{FL + RL}{2}$$

From this, we can retrieve, the velocities V_{left} and V_{right} by differentiating over time. In this case, we discretely divide change in distance over change in time to retrieve the velocities. All other data can be used readily and are in meter units.

2) LiDAR Data: Each LiDAR scan contains 1081 measured range values, spaced evenly across a 270 degree field of view. Counter clockwise is positive and clockwise is negative. Because the robot is pointed towards the center of this 270 degree field of view, we need to express coordinates accordingly. Knowing θ and r, where r represents the range to a point, we can convert from 2D spherical coordinates to 2D cartesian coordinates by:

$$x = r\cos(\theta)$$

$$y = r\sin(\theta)$$

3) RGBD Data: The Kinect Dataset gave values within it's own frame of reference. First the dataset has to be rotated with the rotation matrix:

$$oR_r = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix}$$

where in practice we take the transpose of the rotation matrix to go from Kinect frame to regular frame. Then the data needs to be translated to the robot frame for use, covered in the technical approach section.

B. Differential Drive Model and Odometry

The differential drive model is a model describing the motion of a robot driven on the left and right sides. The model can be seen below. We approximated the left and right velocities using the encoder counts at the four wheels. We are also given the angular velocity ω_t. To predict the pose at time $t+1$, we solve for the Euler discretization over time interval of length τ_t:

$$\mathbf{x}_{t+1} = \begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = f_d(\mathbf{x}_t, \mathbf{u}_t) := \mathbf{x}_t + \tau_t \begin{bmatrix} v_t\cos(\theta_t) \\ v_t\sin(\theta_t) \\ \omega_t \end{bmatrix}$$

We will assume our pose at time $t = 0$ is the identity matrix, i.e. robot frame is aligned with the world frame.

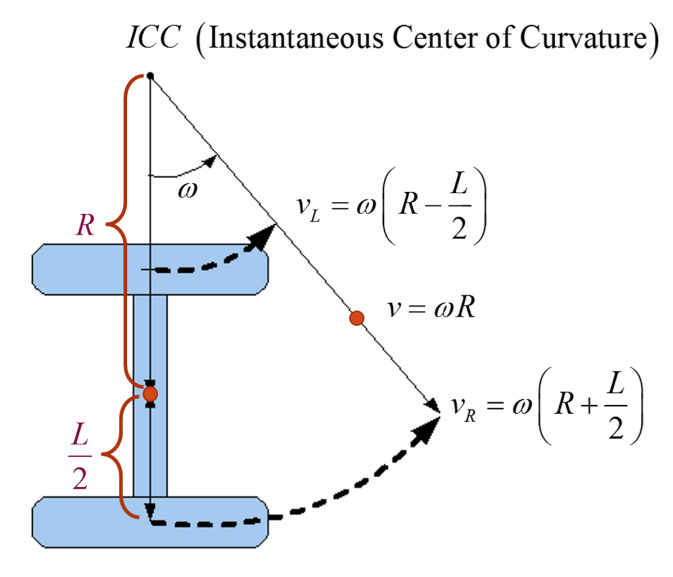


Fig. 3. Differential Drive Model

C. Iterative Closest Points Algorithm

The iterative closest points algorithm is an algorithm that is used to describe the transformation between two point clouds. It alternates between finding point associations and applying the Kabsch algorithm to determine intermediate poses that it then iterates upon again. The descriptions are found below. Typically the ICP algorithm can be run until an error threshold is met, in practice I implement an arbitrary number of iterations for the algorithm to run.

1) Closest Points Approach: First, ICP finds correspondences between two point clouds, matching close points.

$$i \leftrightarrow \arg\min_j \|m_i - (R_k z_j + p_k)\|_2^2$$

2) Kabsch Algorithm: The Kabsch algorithm finds the best pose between two sets of points. It aims to solve this problem by finding the rotation and pose which minimizes the squared norm between points.

$$\min_{R \in SO(d),\, p \in \mathbb{R}^d} f(R,p) := \sum_i w_i \|(Rz_i + p) - m_i\|_2^2$$

To do so, we first solve for the centroids between two points.

$$\bar{m} := \frac{\sum_i w_i m_i}{\sum_i w_i}$$

$$\bar{z} := \frac{\sum_i w_i z_i}{\sum_i w_i}$$

Once the centroids are found, we can solve for the gradient of the cost function with respect to the position to get:

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

We then find the centered point clouds which we will use to normalize our points:

$$\delta\mathbf{m}_i := \mathbf{m}_i - \bar{\mathbf{m}} \qquad \delta\mathbf{z}_i := \mathbf{z}_i - \bar{\mathbf{z}}$$

But to determine the rotation R that aligns two associated centered point clouds δm_i and δz_i, we need to solve a linear optimization problem in SO(d):

$$Q := \sum_i w_i \delta \mathbf{m}_i \delta \mathbf{z}_i^\top$$

$$\max_{R \in SO(d)} \operatorname{tr}(Q^\top R)$$

After computing Q, we can find the rotation through the singular value decomposition. Let the following be the SVD of Q:

$$Q = U \Sigma V^\top$$

We then solve for the rotation matrix through:

$$R = U \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & \det(UV^\top) \end{bmatrix} V^\top$$

We then can recalculate the position based off the equation above and retrieve the transformation between two sets of points.

D. Occupancy Mapping

The occupancy grid is a vector that describes whether a cell in an occupancy map is occupied or free. In an occupancy

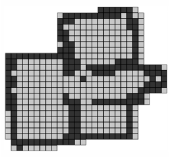


Fig. 4. Occupancy Map

grid mapping, the occupancy grid is unknown and must be estimated using observations and robot trajectory. We employ an independence assumption that cell values are independent conditioned on the robot trajectory.

$$p(m \mid z_{0:t}, x_{0:t}) = \prod_{i=1}^{n} p(m_i \mid z_{0:t}, x_{0:t})$$

We represent our probability of a cell being occupied using log-odds occupancy grid mapping, where the probability mass function is equal to accumulating several log odds ratio from each individual reading, where $\lambda_{i,t}$ is the log odds of cell i at time t.

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

Here, we the occupancy grid will store the log probability of a cell being occupied or not, we update this by:

$$\Delta\lambda_{i,t} = \log \frac{p(m_i = 1 \mid z_t, x_t)}{p(m_i = -1 \mid z_t, x_t)}$$

$$= \begin{cases} +\log 4, & \text{if } z_t \text{ indicates } m_i \text{ is occupied} \\ -\log 4, & \text{if } z_t \text{ indicates } m_i \text{ is free} \end{cases}$$

In practice, I adjusted the log occupancy update to add and subtract different values.
By iterating through every LiDAR scan, it is possible to construct this grid given the robot pose in the world frame at time t. The LiDAR points are described in the LiDAR frame, which then will be converted to the robot body frame, which is then expressed in the world frame:

$$\text{points}_{\text{world}} = {}_wT_b \cdot {}_bT_l \cdot \text{points}_{\text{lidar}}$$

E. Bresenham Algorithm

The Bresenham algorithm is an algorith for tracing a straight line between two points on a grid. It incrementally determines the closest pixel to the ideal line. In the context of occupancy grid mapping, the algorithm is used to determine which cells lie along the line-of-sight between the robot and a detected obstacle. Cells along this path are marked as free space, while the endpoint of the line is marked as occupied. Given the starting (x_0, y_0) and end point (x_0, y_0), we compute differences $dx = x_1 - x_0$ and $dy = y_1 - y_0$. We then determine the slope, if $|dx| > |dy|$ then we iterate over x, otherwise we iterate over y. In practice, we also have an initial error term $p = 2dy - dx$ to decide when to increment y. This gives us the discretized version of a line on a grid.

F. Texture Mapping

Texture mapping is done by projecting points from the Kinect into world coordinate then thresholding to obtain points that are on the floor. After which, RGB colors associated with said points are converted to pixel coordinates in an image which is masked by the occupancy grid map. To convert points from the Kinect to the world frame we have:

$$points_{world} =_w T_b \cdot_b T_r \cdot_r T_k \cdot points_{kinect}$$

Note that there is an extra transformation between the Kinect frame and the body frame as the Kinect's data coordinate frame is misaligned with the robot body's.

G. Factor Graph Optimization

To optimize a set of poses $x = \begin{bmatrix} x_1^\top & \cdots & x_n^\top \end{bmatrix}^\top$, we follow an optimization of the cost function

$$\min_x \sum_{(i,j) \in E} \phi_{ij}(e(x_i, x_j))$$

where $\phi_{ij} : \mathbb{R}^d \to \mathbb{R}$ is some distance function.
Our approach follows pose graph optimization,

$$\min_{\{T_i\}} \sum_{(i,j) \in \mathcal{E}} \left\| W_{ij} \log\left(\bar{T}_{ij}^{-1} T_i^{-1} T_j\right) \vee \right\|_2^2$$

with the distance function $\phi_{ij}(\mathbf{e}) = \mathbf{e}^\top W_{ij}^\top W_{ij} \mathbf{e} = \|W_{ij}\mathbf{e}\|_2^2$. This is known as the Levenberg-Marquardt algorithm, which we use GTSAM to solve for.

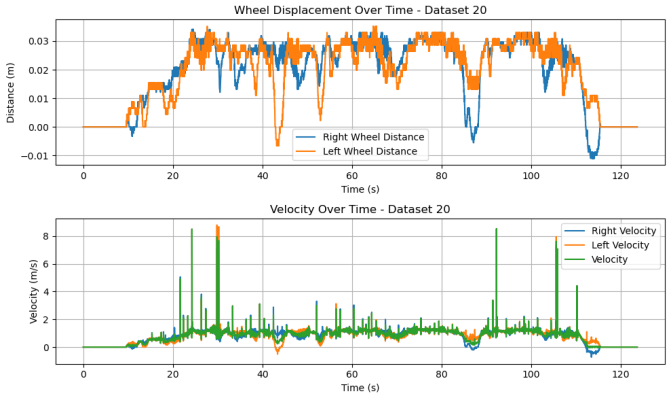


Fig. 5. Wheel Speed - Dataset 20

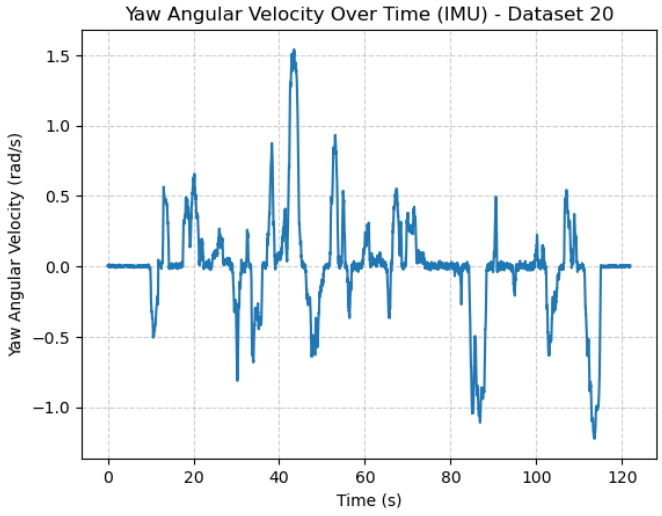


Fig. 6. Yaw Angular Velocity - Dataset 20

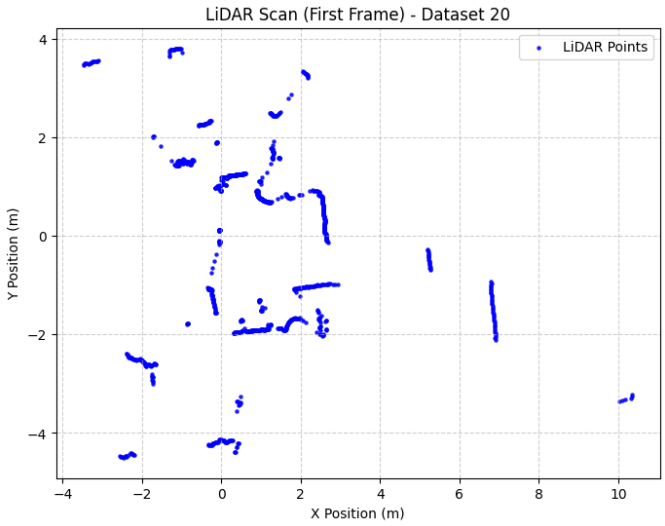


Fig. 7. Lidar Point Scan First Frame - Dataset 20

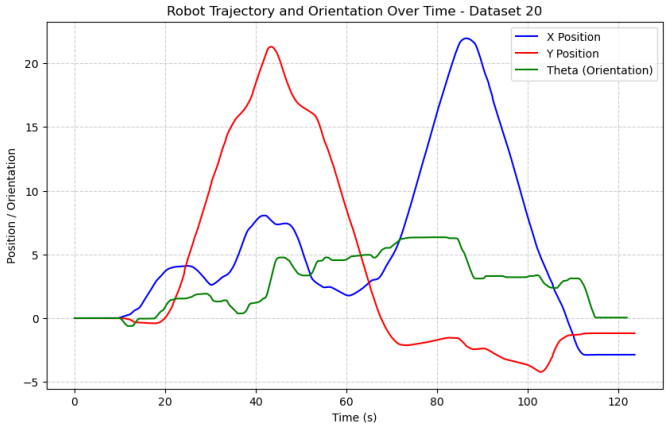


Fig. 8. X, Y, and Theta Values

III. Technical Approach

A. Data Preprocessing

The differential motion model described above was implemented. First, a time vector was established. Given UNIX time stamps for both encoder and IMU readings, I created time vectors by subtracting the first entry from the entire array for both arrays. This normalizes the time to start at $t = 0$. Following the method described above, I was able to determine the left side, right side, and total velocity of the body. This is seen below: The IMU data is displayed below. The lidar point scan data is shown below. Note that because the size of the datasets were different, I synced them by matching the closest time steps then filtering datasets based off the indices retrieved. This is done later on with the Kinect dataset as well.

B. Differential Drive Odometry

With the synced time I calculated the pose at every time step. I first calculate the interval between time steps, then I use the preprocessed velocity and orientation to find position and orientation updates at time $t+1$. I then combine the data into a pose matrix. The individual values and trajectory can be seen below.

C. Point-cloud registration via ICP

1) Warm-up: The ICP algorithm was implemented and tested against some test sets. The ICP algorithm is sensitive to the initial point cloud positions between two dataset. Meaning, different starting poses can lead the algorithm to iterate to a local minima that is incorrect. To combat this, I implemented a discrete yaw function that rotates the source point cloud about yaw and matches points between the two point clouds to get an error score. The yaw angle with the lowest angle would then be selected to initialize ICP. To assist in getting the distances between points and error between point clouds, I utilized the scipy.spatial.KDTree function from the scipy library. Values for both the test drill and test container are shown below.

2) Scan Matching: Given that I calculated the robot odometry estimated using the motion model, I utilized my ICP algorithm to implement LiDAR scan matching. The LiDAR

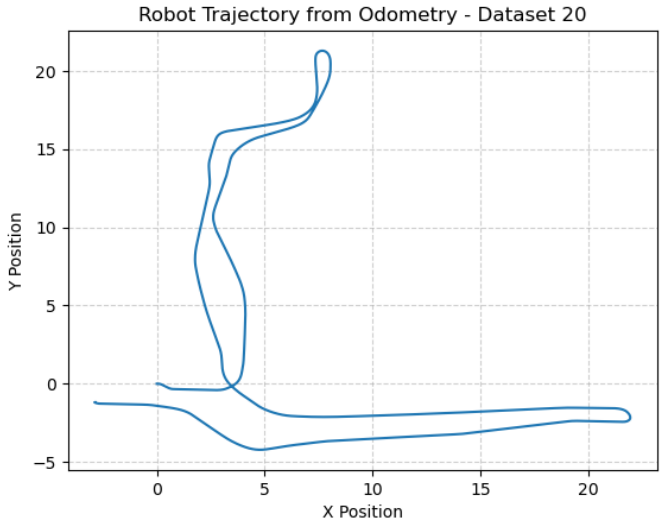


Fig. 9. Odometry Trajectory

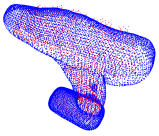


Fig. 10. Drill 1

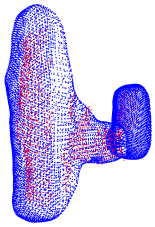


Fig. 11. Drill 2

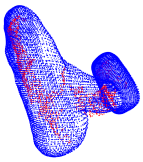


Fig. 12. Drill 3

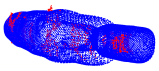


Fig. 13. Drill 4

dataset had to first be synced with the odometry dataset, as the odometry dataset had fewer data entries. I then clipped the LiDAR dataset such that values too close or too far are removed. Using the odometry data, I retrieved the transformation matrix $\boldsymbol{T}_{delta}$ between poses $\boldsymbol{x}_t$ and $\boldsymbol{x}_{t+1}$. Since the LiDAR

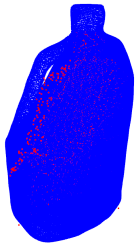


Fig. 14. Container 1

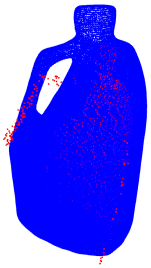


Fig. 15. Container 2

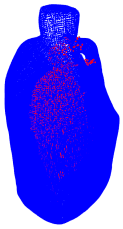


Fig. 16. Enter Caption

points are expressed in the LiDAR frame, I transformed $\boldsymbol{T}_{delta}$ to match the LiDAR frame to the world frame $\boldsymbol{T}_{transformed}$, and used that to transform my source LiDAR datapoints to initialize the ICP algorithm.

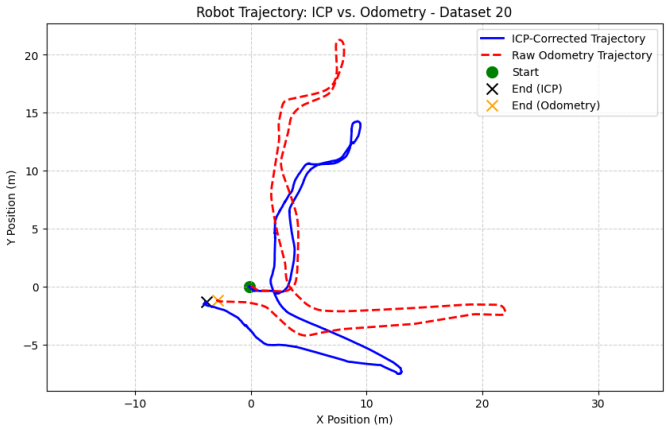


Fig. 17. ICP vs Odometry Trajectory - Dataset 20

D. Occupancy and Texture Mapping

I used the first texture map to display an occupancy grid map to verify that my transforms were correct. To do this, I took the LiDAR data points and implemented the Bresenham algorithm, assuming the center was the start point. The LiDAR points are treated as occupied and the line traced is treated as free. I determined a grid size and resolution, and scaled my LiDAR measurements from world units to pixel units. I initialized an occupancy grid map that stored the log values and incremented the log odd values of cells which were determined to be occupied and decremented the others. This was done with +4 for increments and -0.5 for decrements.

Fig. 18. Occupancy Grid Map First Frame - Dataset 20

Once the first frame was verified. I then constructed an occupancy grid map of the entire trajectory. This was done by syncing each LiDAR scan to it's respective trajectory pose in the trajectory derived from ICP in the previous section. Through applying a coordinate transform, I was then able to express the origin of the robot and point cloud in the world frame, convert that to grid coordinates, run the Bresenham algorithm, and update log odds for every cell per each time step. Here are the occupancy grid maps with the trajectories

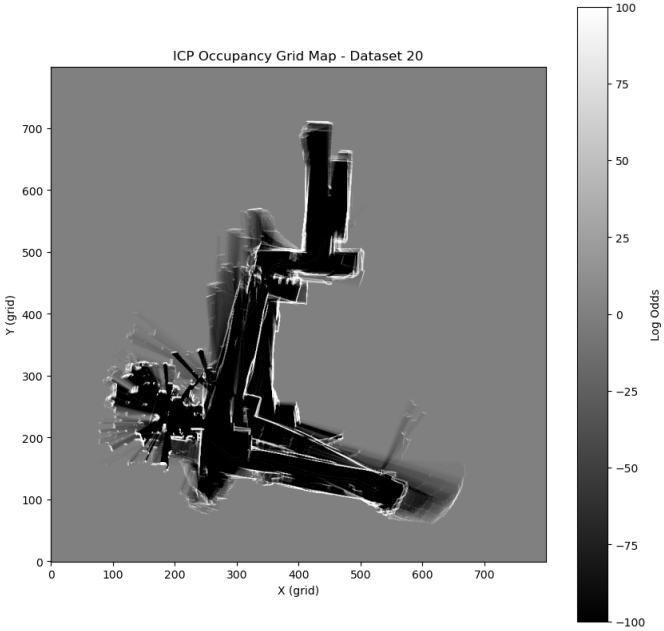


Fig. 19. ICP Occupancy Grid Map - Dataset 20

drawn on for both the odometry and ICP trajectories.

Once the odometry was defined, I generated a texture map from the Kinect disparity and RGB images. I first test the RGBD data for a single frame. I transform the data from the Kinect coordinate convention to the robot coordinate convention, then from the Kinect coordinate frame to the robot coordinate frame, then from robot coordinate frame to world.

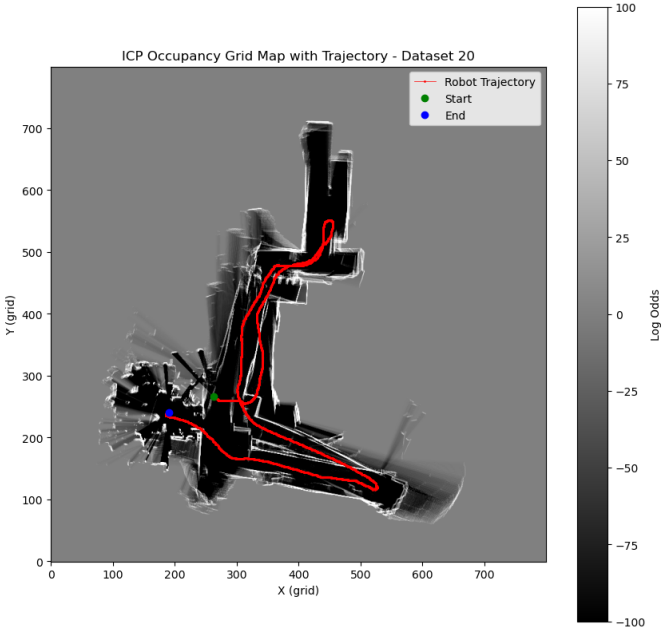


Fig. 20. ICP Occupancy Grid Map with Trajectory - Dataset 20

Fig. 21. Odometry Occupancy Grid Map with Trajectory - Dataset 20

I then threshold for points that have a z value near the ground (near 5 cm) to filter points. I then apply this method for all points along the trajectory of the robot. I first had to sync the data to the lowest count dataset, which was the RGB image dataset. Then I constructed a second occupancy grid with the same size as the first, and scaled filtered world RGBD points to the grid pixel coordinates. I did this by simply selecting for the x and y coordinates of each filtered point then applying the scaling. I then color the second occupancy grid pixel coordinate with the RGB value of the filtered RGBD point. After the entire dataset has been iterated through, a texture map is generated. At the end, I mask the texture map with the occupancy map to retrieve the following results.

E. Pose Graph Estimation and Loop Closure

A factor graph was implemented using the GTSAM library with the nodes representing robot poses at different time steps. The poses obtained from the ICP trajectory was used to intialize the graph. Noise was added inbetween each pose to introduce some deviation from the ICP trajectory. Two loop closure detection methods are used to refine the trajectory.

Fig. 22. Filtered Ground Points

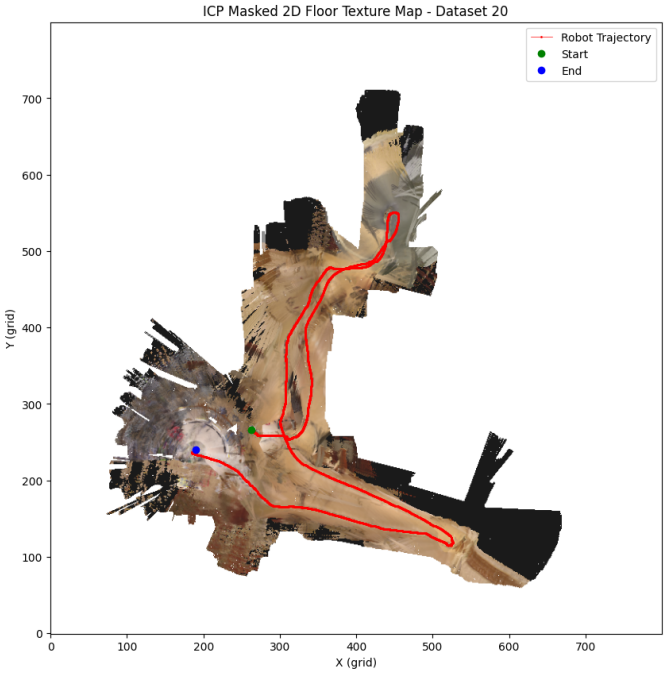


Fig. 23. ICP Texture Map - Dataset 20

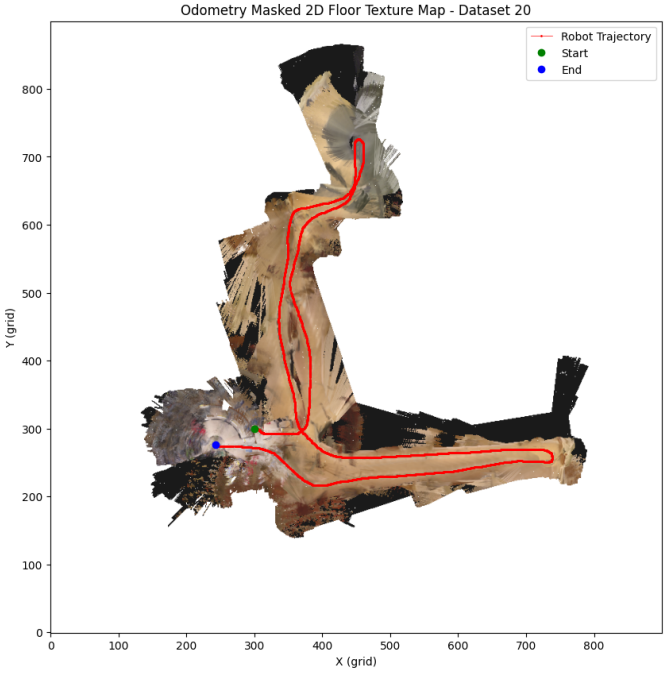


Fig. 24. Odometry Texture Map - Dataset 20

Fixed interval-loop closure was implemented with a pose interval of ten and a error threshold of 0.3. Every ten poses the current pose would be compared to the pose 10 iterations ago and ICP was run with an initialization of the pose difference between them. If the error was low enough, then a node would be added to the graph. Proximity-based loop closures were also implemented that ran ICP on nearby poses. Neighbors were determined by simply looking at the previous several iterations of poses. If a neighbor point cloud aligned with the current point cloud with low enough error, then an edge in the factor graph was added to represent a loop closure. Finally, the optimized trajectory was retrieved and a occupancy grid map and texture map were constructed.

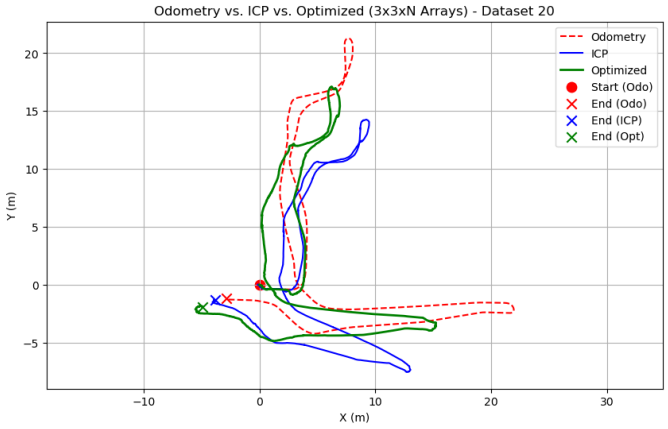


Fig. 25. GTSAM Trajectory - Dataset 20

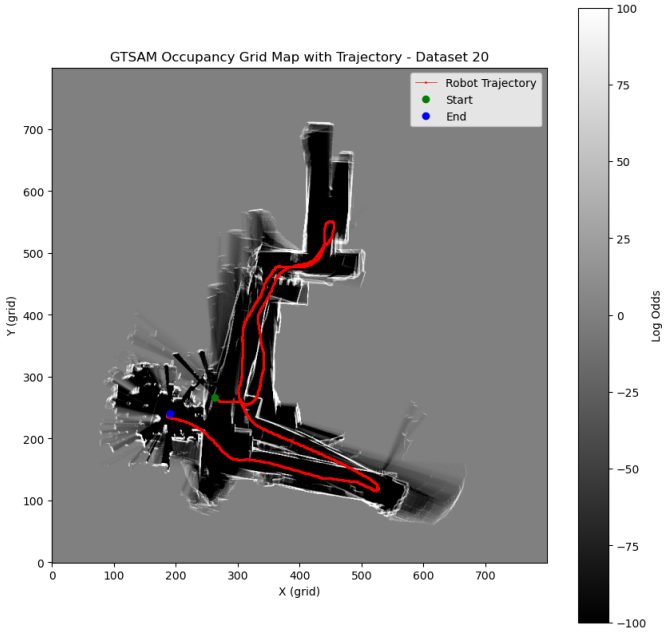


Fig. 26. GTSAM Occupancy Grid Map with Trajectory - Dataset 20

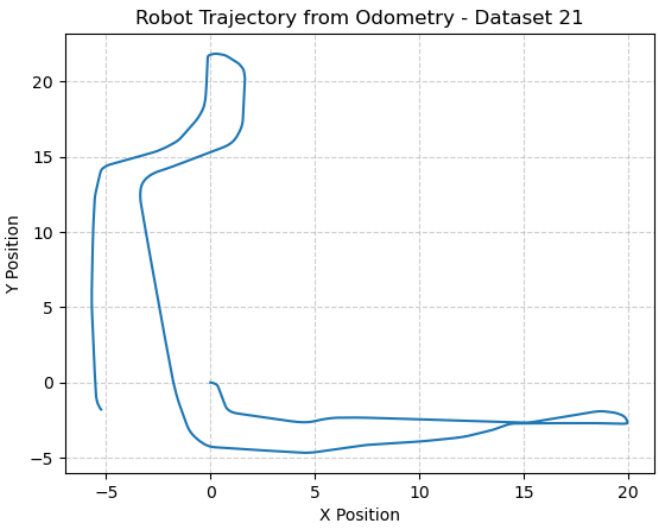
Fig. 27. GTSAM Texture Map - Dataset 20

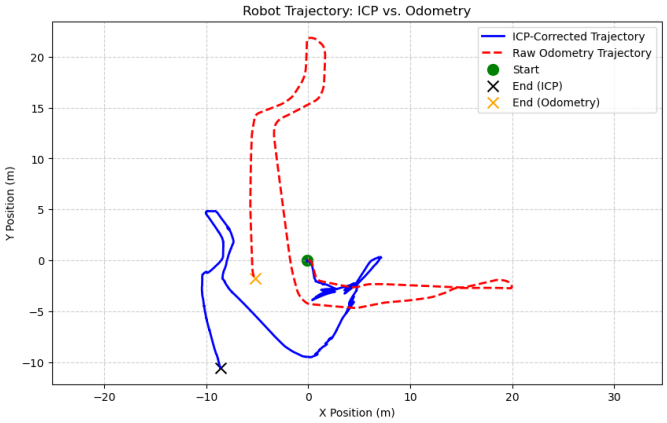
IV. Conclusion

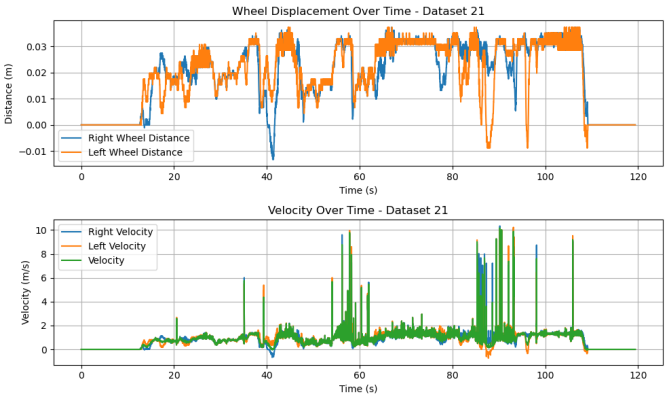
LiDAR-Based SLAM was implemented from the encoder, IMU, LiDAR, and RGBD data provided. ICP was implemented and tested on test object, these results match showing ICP to work. A trajectory, occupancy grid map, and texture

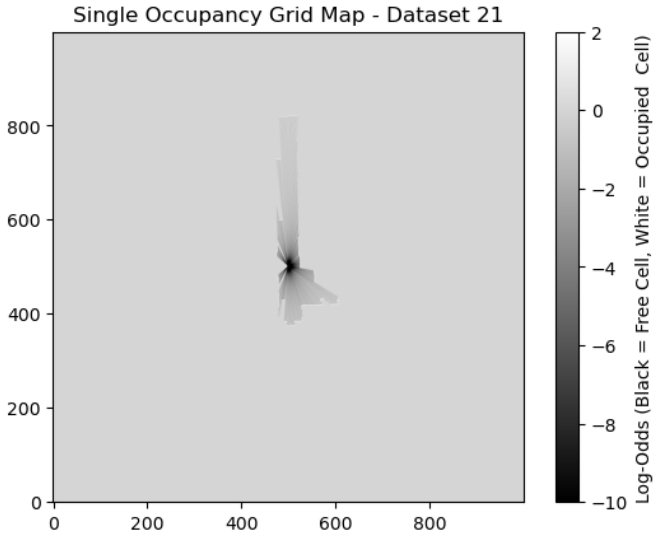
map was defined for the odometry, ICP, and GTSAM models. It is seen that the odometry model is the most smooth, this can be attributed to the fact that the model doesn't take into account the same amount of noise that the lidar data sees. It can be observed that the GTSAM model is in between the odometry model and the ICP model. This makes sense, as the ICP model was initialized based off the odometry model and the GTSAM model was initialized based off the ICP model. The occupancy grid maps show that the LiDAR scan is indeed working as the trajectory fits entirely within the free space generated in the occupancy grid map for all three methods. The texture maps do look discontinuous, this can be due to the overlap of pixels as the texture map is being generated. The texture map of the optimized trajectory is smoother than the ICP occupancy grid map and texture map. This shows that loop closure detection and pose graph optimization increased the accuracy and that the GTSAM optimization did indeed optimize robot trajectory. Several obstacles had to be solved in the implementation. The main of which is performance. Due to the large datasets being used, I first implemented calculations and updates of arrays through loops, but that proved to be too demanding for my computer. I had to leverage vectorization to perform matrix operations and employ tricks like real time occupancy map and texture map updates as loading all RGB images and disparity images would take too much memory. Improvements that can be made to this program would be a more robust point cloud matching algorithm (smoothing between points so that the trajectory generated isn't so sensitive to noise), better memory management, and better visualization such that pixels know when to overlap and when not too. Please reference dataset 21 below.

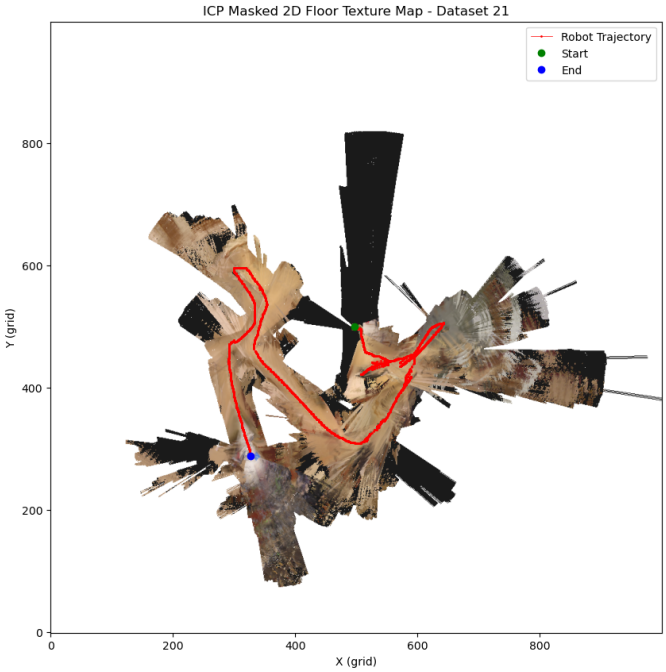
V. Appendix - Dataset 21 Results

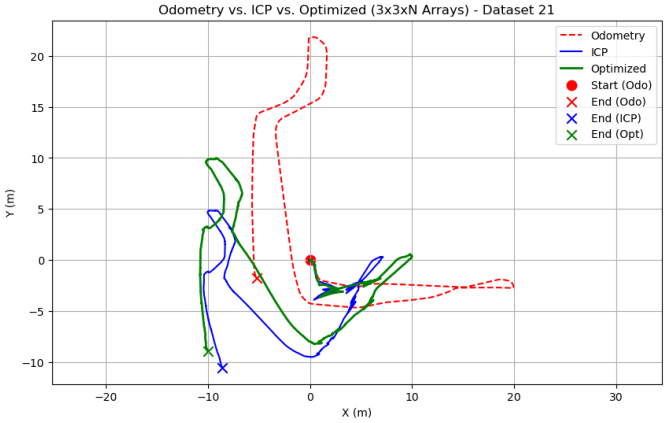


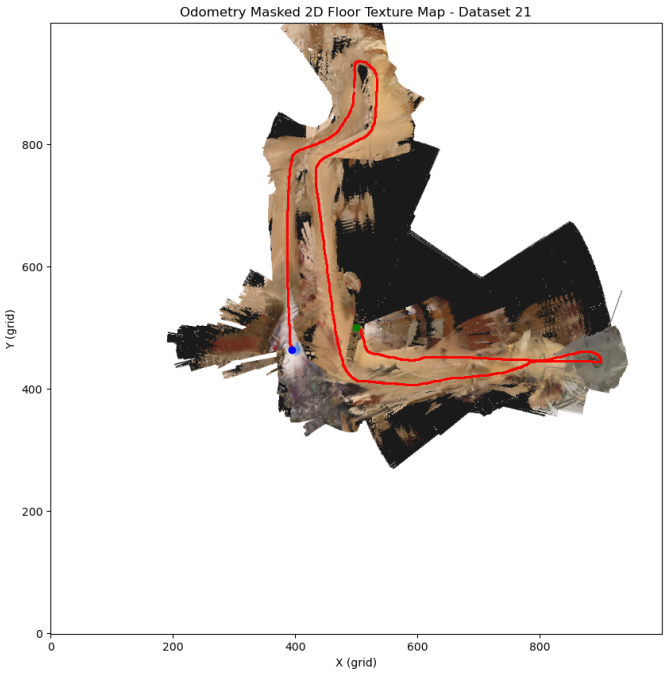












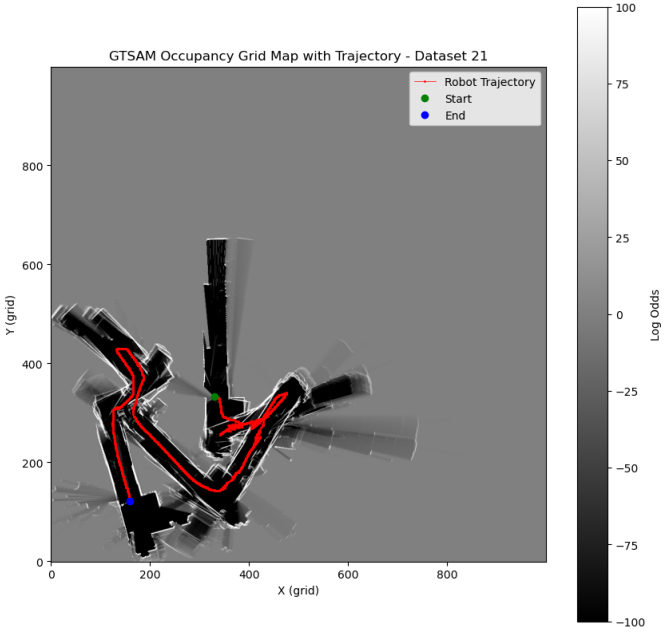
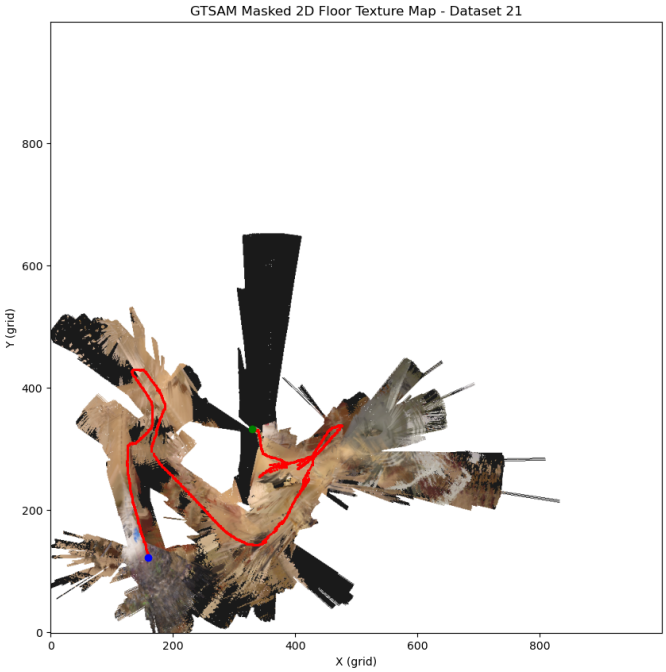


Fig. 28. Enter Caption


GTSAM Masked 2D Floor Texture Map - Dataset 21
Y (grid)
X (grid)
0
200
400
600
800