Project 2: LiDAR-Based SLAM

Behrad Rabiei
Dept. of Electrical and Computer Engineering
University of California San Diego
Email: brabiei@ucsd.edu

***Abstract*—This paper presents a comprehensive report on Project 2 for the ECE 276A (Sensing and Estimation in Robotics) course. The core aim of this project was to implement Simultaneous Localization And Mapping (SLAM) utilizing data from various sensors, including IMUs (Inertial Measurement Units), Encoders, Lidar, and RGBD cameras. Our approach involved executing Dead Reckoning with the assistance of IMU and Encoder data to estimate initial trajectories. We enhanced these estimations through the application of the Iterative Closest Point (ICP) algorithm on Lidar data, followed by the integration of factor graphs to refine our trajectory predictions further. Utilizing the refined trajectories, we generated Occupancy Grid maps and Texture maps. The texture maps effectively correlate pixel colors captured by the camera to locations within the global map, providing a robust framework for robotic navigation and environment interaction.**

I. Introduction

In the evolving field of robotics, the ability to accurately perceive and navigate within an environment is paramount. This capability is essential to numerous applications, from autonomous vehicles navigating city streets to robots performing complex tasks in unpredictable settings. The core challenge in achieving such autonomy lies in the robot's ability to simultaneously map its environment and ascertain its position within that map, a process known as Simultaneous Localization And Mapping (SLAM). We seek to address this challenge by leveraging a multiple sensors comprising Inertial Measurement Units (IMUs), Encoders, Lidar, and RGBD cameras.

Our approach begins with Dead Reckoning, an initial estimation process that combines IMU and Encoder data to track the robot's position and orientation over time. While straightforward, Dead Reckoning is prone to cumulative errors; thus, it serves primarily as a preliminary step. To refine these estimates, we apply the Iterative Closest Point (ICP) algorithm to Lidar data. ICP is a technique that iteratively aligns data points from successive Lidar scans, minimizing the difference between them to produce a more accurate trajectory estimate.

However, even with the improvements offered by ICP, further optimization is necessary to account for the inherent noise in sensor data and the complexity of real-world environments. Here, factor graphs come into play. A factor graph is a probabilistic graphical model that represents variables (such as robot positions) and factors (constraints or measurements relating these variables) to solve optimization problems efficiently. By integrating factor graphs, we leverage the full spectrum of sensor data in a coherent framework, optimizing our trajectory estimates to a higher degree of precision.

Finally, with these optimized trajectories, we generate two types of maps: Occupancy Grid maps and Texture maps. Occupancy Grid maps represent the environment as a grid of cells, each indicating the presence or absence of an obstacle, which is crucial for path planning and navigation. Texture maps, on the other hand, enrich this representation by associating each location with pixel colors from the RGBD camera, offering a visually detailed map that supports more complex interaction with the environment. Through these advanced techniques, our project not only tackles the technical challenges of SLAM but also contributes to the broader pursuit of creating autonomous systems capable of understanding and navigating their surroundings.

II. Problem Formulation

Our goal is to get a reasonably accurate estimate of the location of a robot as it moves around in an unknown environment and to make a map of that environment over time based on data that is attained from our sensors.

A. What we Have

*1) **Robot**:* The robot we are working with is a differential-drive robot that is equipped with encoders, IMU, 2-D Lidar, and a RGBD camera. We will assume the origin of the robot body is the geometric center of the robot body. This is because the differential-drive motion model assumes that the robot is rotating around its center.

*2) **Encoders**:* The encoders monitor the rotation of each of the four wheels at a frequency of 40 Hz, resetting the rotation count after every measurement. Consider, for instance, that a single wheel rotation equates to a distance of ℓ meters. Thus, a sequence of encoder readings such as 0, 1, 0, -2, 3 translates to a total displacement of $(0+1+0-2+3)\ell = 2\ell$ meters for that specific wheel. According to the specifications, the diameter of each wheel is 0.254 meters and with 360 ticks per complete revolution. The wheel advances 0.0022 meters per tick. When the encoder readings for the front-right, front-left, rear-right, and rear-left wheels are read, denoted as [$\boldsymbol{FR}$, $\boldsymbol{FL}$, $\boldsymbol{RR}$, $\boldsymbol{RL}$] respectively, the cumulative distance covered by the right-side wheels is computed as $\frac{(\boldsymbol{FR}+\boldsymbol{RR})}{2} \times 0.0022$ meters, and similarly, the left-side wheels cover a distance of $\frac{(\boldsymbol{FL}+\boldsymbol{RL})}{2} \times 0.0022$ meters.

*3) **IMU**:* The Inertial Measurement Unit (IMU) supplies data on linear acceleration and angular velocity. For our purposes, only the yaw rate, denoted by ω_t at time t, is utilized from the IMU data to estimate the robot's motion within

the differential-drive model framework. The utilization of additional measurements from the IMU is deemed unnecessary for this application.

4) ***Hokuyo****:* A horizontal LiDAR sensor, equipped with a 270° field of view and a maximum detection range of **30** m, acquires distances to surrounding obstacles. This sensor, the Hokuyo UTM-30LX, delivers **1081** range measurements per scan, details of which are accessible online. The sensor's position relative to the robot's chassis is detailed in the robot's specification document. Understanding the interpretation of LiDAR data is crucial, including the conversion of range measurements to Cartesian coordinates $(\boldsymbol{x}, \boldsymbol{y})$ in the sensor's coordinate frame, then transforming these to the robot's body coordinate frame, and ultimately to the global coordinate frame.

5) ***Kinect****:* An RGBD camera provides both RGB images and disparity images. The depth camera is positioned at $(0.18, 0.005, 0.36)$ meters relative to the robot's center, with its orientation described by roll 0 radians, pitch 0.36 radians, and yaw 0.021 radians. The intrinsic parameters of the depth camera are defined by the matrix $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} f_{su} & f_{s\theta} & c_u \\ 0 & f_{sv} & c_v \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 585.05 & 0 & 242.94 \\ 0 & 585.05 & 315.84 \\ 0 & 0 & 1 \end{bmatrix}$$

It is observed that the timing of Kinect data is irregular, with gaps up to 0.2 seconds, due to the logging software's inability to capture all data, resulting in dropped frames. The task is to align the closest SLAM pose with the timestamp of the current Kinect scan. Notably, the depth and RGB cameras are not co-located, necessitating a transformation to align color with depth data due to an x-axis offset between them. Given a disparity value d at pixel (i, j), the depth depth and the corresponding RGB pixel location $(\text{rgbi}, \text{rgbj})$ can be calculated as follows:

$$\text{dd} = (-0.00304d + 3.31)$$

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

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

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

These equations facilitate the conversion from disparity measurements to depth and align depth data with corresponding RGB color locations.

B. Motion Model

Since our robot operates on a differential-drive mechanism, its motion can be accurately represented using a differential-drive kinematic model. Specifically, the robot's movement over time is modeled through Euler discretization across time intervals of length τ_t, where τ_t denotes the time difference between two consecutive timesteps:

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

Where v_t is the linear velocity at time t. We would like to find $x_t \forall t \in \{0, \ldots, T\}$.

C. Occupancy Grid Map

The objective of occupancy grid mapping is to represent the environment as a discrete grid where each cell can either be occupied, free, or unknown. The grid is defined as a set of cells $\mathcal{G} = \{g_1, g_2, \ldots, g_N\}$, where each cell g_i has an associated occupancy state. The map is updated based on sensor measurements taken by a robot as it navigates through the environment.

The problem is formulated as follows: Given a sequence of sensor measurements $\mathbf{z}_{1:t} = \{z_1, z_2, \ldots, z_t\}$ and the corresponding robot poses $\mathbf{x}_{1:t} = \{x_1, x_2, \ldots, x_t\}$, estimate the occupancy state of each cell g_i in the grid. The occupancy state is represented using the log odds notation, which converts the probabilistic occupancy $P(g_i)$ into a log odds value $l(g_i)$:

$$l(g_i) = \log\left(\frac{P(g_i)}{1 - P(g_i)}\right) \quad (2)$$

where $P(g_i)$ is the probability that the cell g_i is occupied.

For each measurement z_t and pose x_t, the log odds update rule is applied, which adjusts the log odds value $l(g_i)$ based on the likelihood of the sensor measurement given the occupancy state:

$$l(g_i)_{t+1} = l(g_i)_t + \log\left(\frac{P(z_t|g_i, x_t)}{P(z_t|\neg g_i, x_t)}\right) - l_0 \quad (3)$$

where l_0 is the initial log odds value representing a prior belief of occupancy, and $\neg g_i$ denotes the event that the cell is not occupied. The term $P(z_t|g_i, x_t)$ is the likelihood of observing z_t given that the cell g_i is occupied, while $P(z_t|\neg g_i, x_t)$ is the likelihood of z_t given the cell is free.

The problem requires determining the log odds values for all cells over time, taking into account the measurements and poses, to achieve an accurate occupancy grid map.

D. Texture Map

The aim is to generate a 2-D color map representing the floor texture, which complements the occupancy grid map of an environment. This process involves utilizing RGBD images and the estimated robot trajectory.

Given a set of RGBD images acquired over time, along with an estimated robot trajectory, the problem involves several key steps:

1) **Depth Acquisition**: For each disparity pixel in the RGBD images, determine the depth using the intrinsic information.
2) **Projection**: Convert the 3-D points from the depth camera's coordinate frame to the world frame using the known robot poses and camera calibration parameters.

3) **Plane Identification**: Identify the floor plane within the world frame by thresholding the height data at (near) zero meters, which corresponds to the floor level.
4) **Texture Mapping**: Create a secondary grid map with the same resolution as the occupancy grid. For each cell in this grid, assign the RGB values based on the corresponding points projected onto the floor plane.

The resulting 2-D map should capture the texture of the floor by coloring each cell according to the RGB values derived from the RGBD images, aligned with the world frame, and adjusted for the robot's trajectory.

E. Factor Graph

This formulation concerns the enhancement of robot trajectory estimation via pose graph optimization with loop closure constraints, facilitated by the GTSAM framework.

The objective is to refine the trajectory estimation through the following steps:

1) **Factor Graph Construction**: Define a factor graph $\mathcal{G} = (\mathcal{X}, \mathcal{F})$, where $\mathcal{X}$ represents the set of robot poses $\{x_1, x_2, \ldots, x_n\}$, and $\mathcal{F}$ denotes the set of factors that encode constraints from odometry and loop closures.
2) **Odometry Factors**: For each consecutive pose x_{i-1} and x_i, introduce an odometry factor that reflects the motion model's prediction.
3) **Fixed-Interval Loop Closure**: After every predetermined number of poses, such as every 10 poses, enforce a loop closure constraint f_{loop} if a loop is detected.
4) **Optimization**:
Optimize the pose graph to find the set of poses $\mathcal{X}^*$ that minimizes the sum of squared residuals from all constraints:

The solution to this optimization problem yields an updated set of robot poses that provides an improved estimation of the robot's trajectory, which is crucial for accurate mapping.

Fig. 1. Differential-drive robot, equipped with encoders, IMU, 2-D LIDAR scanner, and an RGBD camera.

III. Technical Approach

A. Preliminary

Before we can fully use all the data that we have at our disposal, we must first synchronize the data that is coming from our sensors. All the data that we have from our sensors have corresponding time stamps. Using this fact, we will use the closest-in-the-past timestamp of the larger datasets. The exact method I used is as follows:

Given two sequences of time points, $time1 = \{t_1^{(1)}, t_2^{(1)}, \ldots, t_n^{(1)}\}$ and $time2 = \{t_1^{(2)}, t_2^{(2)}, \ldots, t_m^{(2)}\}$, the function *find_nearest_index* computes a sequence of indices $I = \{i_1, i_2, \ldots, i_m\}$ such that each index i_j corresponds to the closest time point in $time1$ to the time point $t_j^{(2)}$ in $time2$, with the additional consideration that if $t_j^{(2)} < t_{i_j}^{(1)}$, then the index is decremented by one to ensure $t_{i_j}^{(1)}$ is truly the nearest value that does not exceed $t_j^{(2)}$.

Formally, for each $t_j^{(2)} \in time2$, we find i_j satisfying:

$$i_j = \underset{i}{\operatorname{argmin}} |t_i^{(1)} - t_j^{(2)}|,$$

and adjust i_j as follows:

$$i_j = \begin{cases} i_j - 1 & \text{if } t_j^{(2)} < t_{i_j}^{(1)}, \\ i_j & \text{otherwise.} \end{cases}$$

The result is a vector of indices I, which maps each time point in $time2$ to its nearest time point in $time1$, taking into account the specified condition. This method was crucial for synchronizing IMU and encoders, encoders and lidar, and disparity and RGB. A crucial detail is that dataset 20 has more lidar data than encoders do so you must map lidar to ecoder data but this is reversed in dataset 21 as there are more encoder data than lidar data.

B. Dead Reckoning

Our goal is to estimate the trajectory of a robot by synthesizing data from Inertial Measurement Units (IMUs) and Encoders. The process commences with the derivation of time intervals between consecutive encoder readings, symbolized as τ_t, to encapsulate the temporal dynamics of the robot's motion. The encoder data, encompassing the rotation counts of the robot's wheels, is necessary for calculating the robot's average linear velocity, denoted as v. The velocity v_t at time t is computed by utilizing the encoder data to determine the average distance traveled by the robot's wheels within the time interval τ_t. This interval represents the time difference between consecutive encoder readings. The value of ℓ, which is the distance covered per encoder tick, is defined by the wheel's physical characteristics, specifically the distance the wheel travels per tick. In the provided context, ℓ is explicitly given as 0.0022 meters per tick. The formula for computing the velocity v_t is thus revised as follows:

$$v_t = \frac{\left(\sum_{i=FR,FL,RR,RL} \frac{encoder\_counts_i}{4}\right) \times \ell}{\tau_t}$$

Here, τ_t is precisely the time interval between consecutive encoder measurements, and $encoder\_counts_i$ represents the tick counts for each wheel (Front Right FR, Front Left FL, Rear Right RR, and Rear Left RL). This equation calculates the robot's average linear velocity by dividing the average distance traveled (factoring in the distance per tick ℓ) by the time interval τ_t.

For motion estimation, a kinematic motion model is employed, represented by the function f_d, which projects the robot's next state based on its current velocity v and orientation, as well as the yaw rate ω derived from the IMU data. The robot's state at any time t is defined by its position (x, y) and orientation (θ), with the initial state presumed to be at the origin. The subsequent positions and orientations are iteratively predicted using this motion model, adhering to the principles of Euler discretization.

C. Scan Matching (ICP)

To accurately estimate the relative transformation between consecutive robot poses using point cloud data from Lidar scans, we employ an approach that integrates the Kabsch algorithm within the Iterative Closest Point (ICP) framework.

1) Initial Setup for Relative Transformation: The process starts by constructing homogeneous transformation matrices T_t and T_{t+1} for consecutive poses at times t and $t+1$, encapsulating the robot's pose in a global reference frame. These matrices are defined as:

$$T_t = \begin{bmatrix} \cos(\theta_t) & -\sin(\theta_t) & 0 & x_t \\ \sin(\theta_t) & \cos(\theta_t) & 0 & y_t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{t+1} = \begin{bmatrix} \cos(\theta_{t+1}) & -\sin(\theta_{t+1}) & 0 & x_{t+1} \\ \sin(\theta_{t+1}) & \cos(\theta_{t+1}) & 0 & y_{t+1} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where θ_t, x_t, and y_t denote the orientation and Cartesian coordinates of the robot at time t, respectively. The relative transformation $T_{relative}$ between these poses is then computed by $T_{relative} = T_t^{-1} \cdot T_{t+1}$.

2) Iterative Closest Point (ICP) Algorithm: ICP iteratively aligns two point clouds—source (m) and target (z)—extracted from Lidar scans to minimize their discrepancy. Initial estimates for the rotation matrix (R_k) and translation vector (p_k) facilitate the alignment of the source point cloud towards the target.

3) Kabsch Algorithm Integration:

1) **Centroid Calculation**: Determine the centroids $\bar{m}$ and $\bar{z}$ of point clouds m and z, respectively.
2) **Translation to Centroids**: Shift both point clouds to position their centroids at the origin.
3) **Cross-Covariance Matrix**: Construct the cross-covariance matrix H between the translated point clouds:
$$H = \sum (m_i - \bar{m}) \cdot (z_i - \bar{z})^T$$
4) **Singular Value Decomposition (SVD)**: Decompose H via SVD, where $H = U\Sigma V^T$.
5) **Optimal Rotation Matrix**: Derive the optimal rotation matrix R_{opt} as:
$$R_{opt} = U \cdot I \cdot V^T$$
Adjust I as necessary to ensure a proper rotation if the determinant of R_{opt} is not 1.
6) **Translation Vector**: The optimal translation vector $\vec{p}_{opt}$ aligns the centroids of the original point clouds under R_{opt}:
$$\vec{p}_{opt} = \bar{m} - R_{opt} \cdot \bar{z}$$

4) Pose Update: Following ICP convergence, the obtained rotation (R_{opt}) and translation $(\vec{p}_{opt})$ update the initial pose estimates, refining the relative transformation between consecutive robot poses.

D. Occupancy Grid Mapping

In our technical approach to occupancy grid mapping, we mathematically formulate the environment as a discrete grid, where each cell represents a potential occupancy state. This discretization is achieved by dividing the environment into a matrix of cells with a specific resolution, denoted as Δ, which defines the physical size of each cell. The process begins by establishing the boundaries of the map, defined by x_{min}, x_{max}, y_{min}, and y_{max}, and the resolution Δ. The dimensions of the grid, in terms of cells, are computed as $size_x = \lceil (x_{max} - x_{min})/\Delta \rceil + 1$ and $size_y = \lceil (y_{max} - y_{min})/\Delta \rceil + 1$, ensuring that each cell can be indexed and accessed to represent a portion of the physical space.

To relate the robot's continuous spatial coordinates to discrete grid indices, a transformation function is utilized. This function converts a point (x, y) in the robot's coordinate frame to grid coordinates (r, c) based on the map's resolution and minimum bounds. Specifically, the row index r is calculated as $r = \lfloor (x - x_{min})/\Delta \rfloor + 1$ and the column index c as $c = \lfloor (y - y_{min})/\Delta \rfloor + 1$, effectively mapping the continuous space to the discrete grid.

The Bresenham2D algorithm plays a critical role in identifying which cells in the grid are affected by a given sensor measurement. Originating from computer graphics for drawing lines on discrete displays, this algorithm is adapted to trace the path of a sensor's ray from its start point (sx, sy) to its endpoint (ex, ey). The algorithm iteratively determines the grid cells that the line intersects, facilitating efficient marking of cells as free or occupied based on the sensor data. The algorithm accounts for the steepness of the line to ensure accuracy in both horizontal and vertical orientations. It calculates the difference in x and y coordinates (dx and dy), determining the direction of iteration. Through cumulative addition of decision variables, Bresenham's algorithm efficiently identifies the set of grid cells (x, y) that form the straight-line path between the sensor and the observed point, enabling precise updates to the log odds values of the occupancy grid.

In our approach to updating the occupancy grid map with Lidar data, we systematically transform Lidar range measurements from polar to Cartesian coordinates, accounting for the robot's orientation and sensor offset, and then update the map based on these transformed measurements. The Lidar data, represented as a series of ranges at specific angles, is first transposed to align with the processing sequence. Each range r at an angle ϕ from the Lidar scan is combined with the robot's pose (x, y, θ) to compute the position of detected points in the environment.

The transformation of Lidar measurements into the robot's frame involves adjusting the angle of each measurement by the robot's current orientation θ and converting polar coordinates (range and angle) to Cartesian coordinates (x', y') using the following equations:

$$x' = x + r\cos(\theta + \phi) + d_x$$
$$y' = y + r\sin(\theta + \phi) + d_y$$

where d_x and d_y are the offsets of the Lidar sensor from the robot's center, accounting for the physical location of the sensor. This offset ensures measurements accurately reflect their origin from the sensor's perspective relative to the robot's center.

The angles ϕ are defined within a range, typically covering the Lidar's field of view (in this case, from $-135°$ to $135.25°$, converted to radians), and these angles are adjusted by the robot's orientation θ to ensure alignment with the global frame.

Once the Cartesian coordinates of the Lidar points are determined, we employ the Bresenham2D algorithm to trace a path from the robot's location to each detected point. This algorithm, by iterating over the grid, identifies which cells the Lidar ray intersects until it reaches the detected obstacle. For each Lidar ray, this process involves incrementing or decrementing the log odds values of the cells based on whether they are along the ray path or at the endpoint of the ray, respectively. The log odds update for a cell g_i along the ray path is computed as:

$$l(g_i)_{\text{updated}} = l(g_i) - \Delta\lambda$$

And for the endpoint cell indicating an obstacle:

$$l(g_i)_{\text{updated}} = l(g_i) + \Delta\lambda$$

where $\Delta\lambda$ is a constant used to adjust the log odds value, signifying the cell's increased or decreased likelihood of being occupied. The values of $\Delta\lambda = 4$ and $\lambda_{\text{max}} = 20$ are constants that control the rate of change in the log odds values and the maximum allowable value, respectively, ensuring that the log odds do not become excessively large or small, which could skew the map's accuracy over time.

This procedure is repeated for each selected Lidar ray (e.g., every tenth ray to balance detail and computational load), updating the map's representation of the environment incrementally. Through this method, the occupancy grid map dynamically evolves, enhancing the robot's environmental understanding and navigational capabilities. Note that we need to filter out some of the lidar rays as they are outside the regions where the lidar is valid (i.e. less than 0.1m or greater than 30m)

E. Texture Mapping

In our texture mapping process, we integrate depth information from disparity images with RGB textures, based on camera geometry and the robot's motion, to construct an enriched 3D representation of the environment. This process involves several computational steps and transformations, utilizing specific values and equations to ensure accurate mapping.

The transformation begins by calculating the depth (z) for each pixel in the disparity images. This calculation is derived from the disparity values using the equation:

$$z = \frac{1.03}{-0.00304 \times disparity + 3.31}$$

Here, the constants 1.03 and 3.31 are calibration parameters specific to the disparity-to-depth conversion process, while -0.00304 is the disparity gradient that relates pixel disparity to physical depth.

With the depth information, we compute the 3D coordinates (X, Y, Z) in the camera's coordinate system for each pixel. The intrinsic parameters of the camera, $f_x = 585.05108211$ and $f_y = 585.05108211$, represent the focal lengths in pixels along the x and y axes, respectively. The optical center of the camera is given by $(c_x = 315.83800193, c_y = 242.94140713)$. These parameters facilitate the transformation from pixel coordinates (u, v) to 3D coordinates:

$$X = (u - c_x)/f_x \cdot z$$
$$Y = (v - c_y)/f_y \cdot z$$

This transformation effectively positions each pixel within a 3D space relative to the camera's viewpoint.

The next step involves adjusting these 3D coordinates to account for the camera's orientation and position on the robot, using a predefined rotation matrix o_R_r and a translation vector p. This adjusts the 3D coordinates from the camera's optical frame to the robot's body frame, considering the camera's mounting angles and its displacement from the robot's geometric center.

To map the 3D points from the robot's body frame to the global frame, we use the robot's pose at the time of each image capture. The pose includes the robot's position (x, y) and orientation (θ), which are used to construct a rotation matrix R. This matrix rotates the points to align with the robot's orientation in the global frame, and then translates them based on the robot's position, effectively placing the 3D points within a global context.

For the texture mapping, we calculate the corresponding RGB pixel coordinates for each depth pixel using the equations:

$$rgbu = \left\lfloor \frac{u \times 526.37 + dds \times (-4.5 \times 1750.46) + 19276.0}{f_x} \right\rfloor$$

$$rgbv = \left\lfloor \frac{v \times 526.37 + 16662.0}{f_y} \right\rfloor$$

These equations adjust the depth image coordinates to align with the RGB image, accounting for any disparities between the two images due to the camera setup. The constants involved, such as 526.37, 1750.46, 19276.0, and 16662.0, are calibration parameters specific to the camera and its configuration, ensuring accurate alignment of depth and RGB data.

Valid pixel indices are then used to filter out points that do not have a corresponding texture in the RGB image, ensuring that only meaningful textures are mapped to the 3D points. This textured 3D map provides a detailed visual representation of the environment, combining spatial and color information for enhanced robotic navigation and interaction.

F. Factor Graphs and Loop Closure

In the development of our graph-based SLAM (Simultaneous Localization and Mapping) solution, we leverage the GTSAM (Georgia Tech Smoothing and Mapping library) framework to model the robot's trajectory and map the environment. This approach utilizes factor graphs to represent the relationships between robot poses and landmark observations, where nodes represent robot poses or landmarks, and factors represent constraints or measurements between these entities.

1) Graph and Prior Setup: A 'NonlinearFactorGraph' object is instantiated to hold our SLAM problem's factors. We introduce a prior factor with minimal uncertainty (0.00001 in all dimensions) on the initial robot pose to anchor our solution, effectively setting a strong belief that the robot starts at the origin $(0, 0, 0)$ in the 2D plane.

2) Odometry Factors: For each consecutive pair of poses, we add odometry (Between) factors that encode the relative motion, modeled with a noise level defined by $\sigma_x = \sigma_y = 0.1$ and $\sigma_\theta = 0.01$. These factors represent our belief in the motion between successive poses based on the robot's internal sensors, such as encoders or IMUs.

3) Loop Closure and ICP: Every few steps, determined by 'dis' (distance), we perform a loop closure check to correct for drift in the robot's trajectory. I experimented with different values for 'dis' and found 5 to be the best. We use the Lidar measurements at these intervals to identify loop closures. Valid Lidar ranges are filtered based on a predefined distance criterion ($0.1m < range < 30m$) for both the source and target point clouds. The point clouds are transformed from polar to Cartesian coordinates, incorporating a constant offset to adjust for the Lidar sensor's position on the robot.

Using these point clouds, we perform Iterative Closest Point (ICP) to estimate the relative transformation between the two poses. This involves calculating the rotation matrix R_k and translation vector p_k that best align the source point cloud from the previous pose to the target point cloud at the current pose. The outcome of the ICP provides a refined estimate of the relative pose, which is used to add a loop closure factor to the graph if the estimated rotation is within acceptable bounds ($\pm\frac{\pi}{4}$). I assumed that any rotation outside of this bound would not be physically possible for the robot in 5 timesteps.

4) Initial Estimates and Optimization: Initial estimates for the robot poses are populated based on motion estimates from the ICP. These estimates serve as the starting point for the optimization process. The 'GaussNewtonOptimizer' is then employed to iteratively refine the robot poses and the map by minimizing the overall error in the factor graph, adhering to both odometry and loop closure constraints.

IV. Results

We can now visualize the findings from our approaches and compare them.

A. Dead Reckoning

For the dead reckoning, the trajectories that were found by using the motion model for both datasets can be found in Figure 2. The trajectories seem rather reasonable with no sign of sudden jumps or turns. Using these positions, you can create a map that represents the floor plan and project lidar scans at those positions and orientations to get a rough estimate of what the actual map looks like. This is demonstrated in Figure 3.

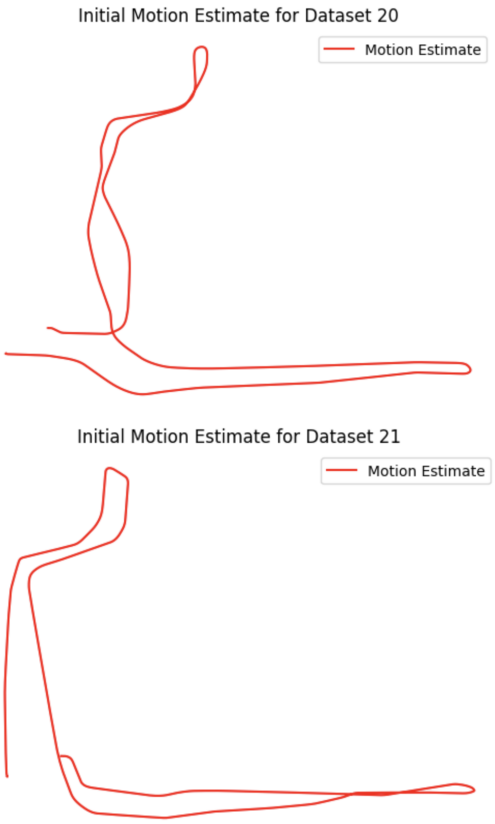

Fig. 2. Initial Motion Estimates for Dataset 20 and 21

B. Scan Matching (ICP)

1) Warm-up: The resulting plots of the warm-up for the drill and the liquid container are shown in Figures 4 and 5 respectively. We can see that our ICP implementation matches

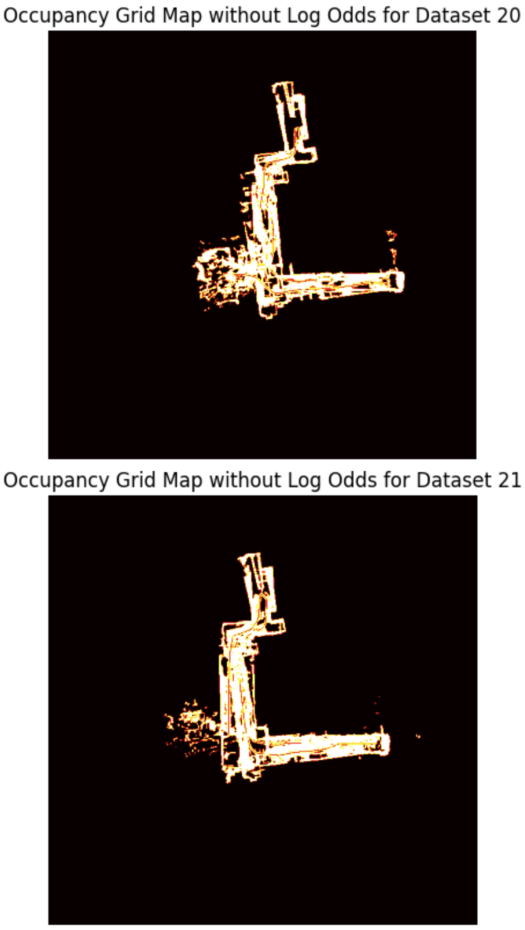

Fig. 3. Projected Lidar Scans for Initial Motion Estimates for Dataset 20 and 21

the cloud points with relatively good accuracy. I would like to point out that ICP is very sensitive to initialization and depending on what you choose the final result can be different.

2) Scan Matching: The resulting trajectories for both datasets are shown in Figure 6. Figure 6 demonstrates a comparison between the motion model estimates(Red) and the updated ICP estimates(Blue). By visual inspection, it is clear that the ICP does not perform as well as the motion model. This is most likely due to errors propagating in the ICP.

C. Occupancy Grid Mapping

Figures 7 and 8 show all 3 occupancy grid maps (motion model, ICP, GTSAM) for both datasets. By visual inspection, it becomes even more clear that the motion model still had the best estimate out of the 3 methods and GTSAM only makes the map slightly better.

D. Texture Mapping

Figures 9 and 10 show all 3 texture grid maps (motion model, ICP, GTSAM) for both datasets. By visual inspection, it becomes even more clear that the motion model still had the best estimate out of the 3 methods and GTSAM only makes the map slightly better. This result aligns with what we saw with the occupancy grid maps.

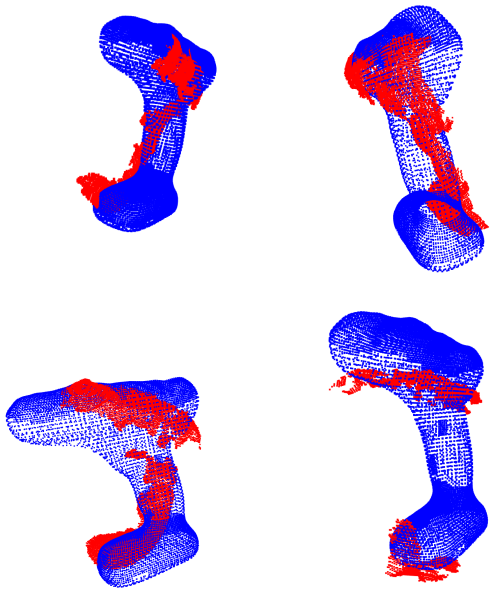

Fig. 4. Drill Scan Matching for Warm-up

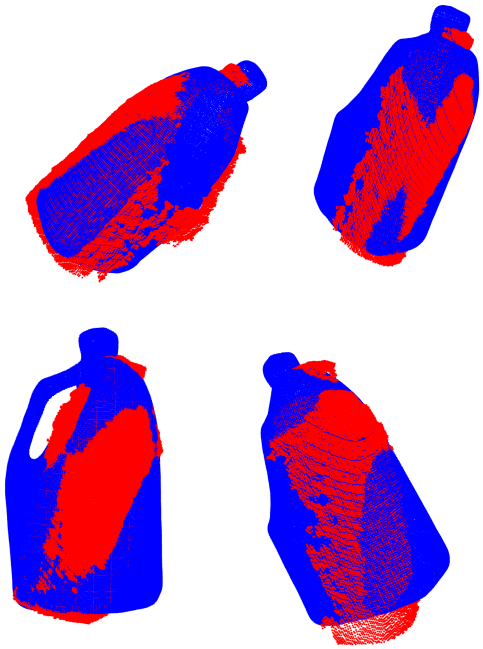

Fig. 5. Liquid Container Scan Matching for Warm-up

E. Factor Graphs and Loop Closure

The result of factor graph optimization with GTSAM is demonstrated and compared with the other approaches for both datasets in Figure 11. We can see that we have a slight improvement over ICP but still not close to the motion

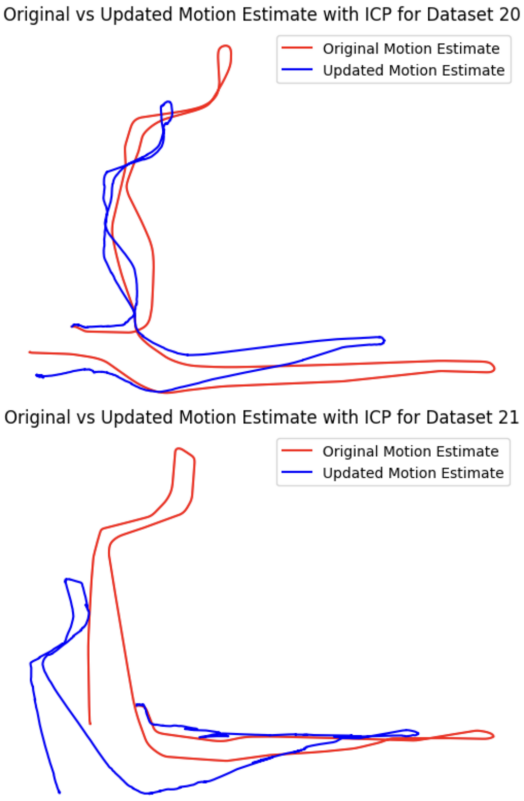

Fig. 6. Initial Motion Estimates(Red) VS ICP(Blue) for Dataset 20 and 21

model estimates. Further improvements might be possible if proximity-based loop closure is implemented.

V. Conclusion

In this project, our objective was to execute Simultaneous Localization and Mapping (SLAM) leveraging a comprehensive suite of sensors, including Encoders, an Inertial Measurement Unit (IMU), Lidar, and an RGBD camera. Initially, we employed Dead Reckoning to acquire a preliminary approximation of the robot's trajectory, subsequently refining this estimate by integrating Lidar data through Iterative Closest Point (ICP) scan matching techniques. To further enhance the accuracy of our trajectory predictions, we adopted factor graph optimization, a method that improved the precision of our estimates. Armed with these refined trajectories, we succeeded in constructing detailed occupancy grid maps, which delineated the occupied from the unoccupied regions. Moreover, we developed texture maps by projecting RGB data onto a global frame, thereby enriching the spatial representation of the environment. In prospective developments, I anticipate the incorporation of the Extended Kalman Filter as a means to achieve similar objectives, potentially offering a sophisticated alternative to our current methodologies.

References

[1] https://en.wikipedia.org/wiki/Spherical coordinate system
[2] https://natanaso.github.io/ece276a/
[3] https://gtsam.org

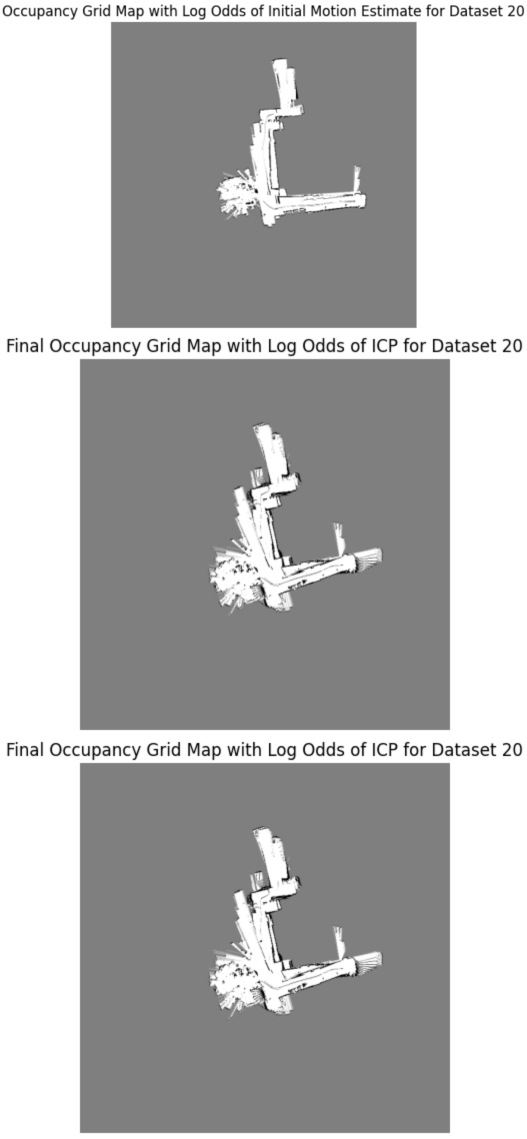

Fig. 7. Occupancy grid map with Initial Motion Estimate, ICP, and GTSAM for Dataset 20

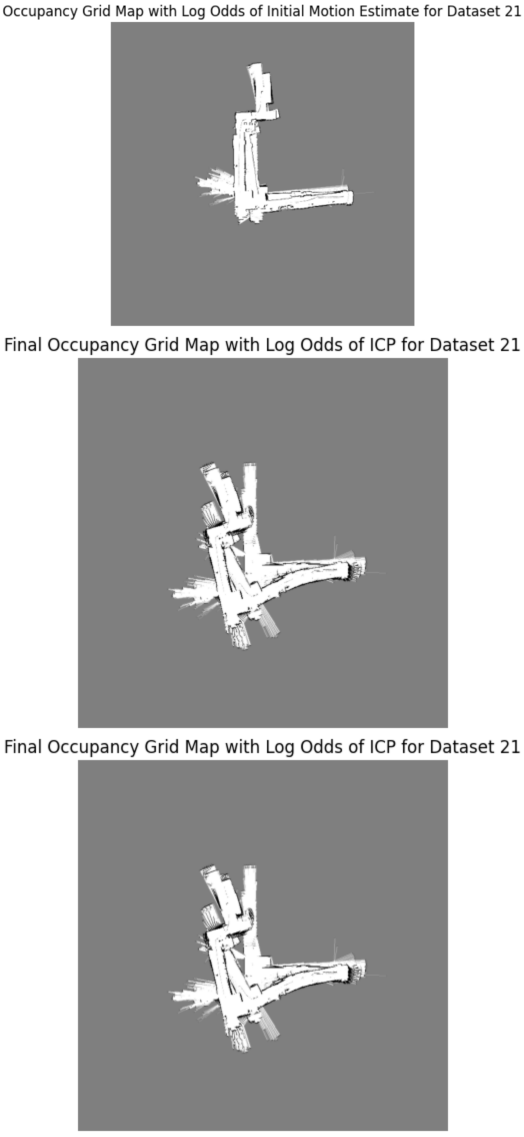

Fig. 8. Occupancy grid map with Initial Motion Estimate, ICP, and GTSAM for Dataset 21

Fig. 9. Texture grid map with Initial Motion Estimate, ICP, and GTSAM for Dataset 20

Fig. 10. Texture grid map with Initial Motion Estimate, ICP, and GTSAM for Dataset 21

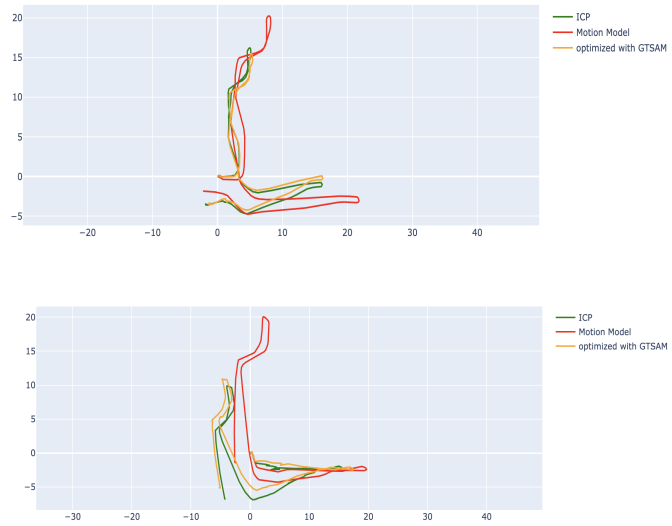

Fig. 11. Initial Motion, ICP, and GTSAM Trajectories for Dataset 20 and 21

CONTENTS