LiDAR Based SLAM

Charvi Shukla
Computer Science and Engineering Department
University of California, San Diego
cshukla@ucsd.edu

***Abstract*—Simultaneous Localization and Mapping (SLAM) is an important capability for autonomous systems to be able to navigate unknown environments. In this project, we develop a LiDAR-based SLAM implementation that utilizes multi-modal sensor data from a differential-drive robot to produce a map of the environment. We establish an initial trajectory using control data from an encoder and IMU, and a discrete time differential drive kinematic model, which is subsequently refined via sequential Iterative Closest Point (ICP) scan matching. We recognize that frame-to-frame matching accumulates angular drift when the robot is turning. Therefore, we use GTSAM to enforce loop closure constraints using Pose Graph Optimization. The corrected trajectory is then used to generate an occupancy grid and a floor texture map that is consistent with the environment.**

***Index Terms*—Robotics, Simultaneous Localization and Mapping, Sensors, Autonomous**

I. Introduction

Oftentimes in the real world, an autonomous robot or an autonomous vehicle, needs to function in a place that it has not seen before. Therefore, it must be able to localize itself in its surroundings, while also constructing an accurate map of where it is. Simultaneous Localization and Mapping (SLAM) is a fundamental problem in robotics, as it provides the framework to achieve this dual objective. While sensors such as IMUs and encoders provide valuable and high frequency motion estimates, we cannot solely rely in their dead-reckoning estimates. Over time, their uncorrected drift distorts the robot's estimated trajectory, leading to fragmented, slanted, and physically impossible maps.

LiDARs are very useful in this field as they direct, high-precision distance measurements, and are robust to various lighting conditions. LiDARs are also active sensors and capture dense and accurate geometric representations of the surroundings. Therefore, in this project, we utilize this useful sensor and we implement LiDAR based SLAM for a differential-drive robot equipped with wheel encoders, an IMU, a 2D LiDAR scanner, and an RGBD camera. Our objective is to use these multi-modal sensor readings to accurately track the robot's trajectory and then to construct two of the following representations of the environment: a 2D occupancy grid map mapping, and a 2D color texture map of the floor surface.

To solve the SLAM problem and mitigate sensor drift, we follow a multi-step process:

1) Data pre-processing for synchronization and for reading/deriving the control inputs.
2) IMU and Encoder odometry (using a motion model) for dead reckoning estimation of the robot trajectory.
3) ICP scan matching for improving the dead reckoning estimation .
4) Pose Graph optimization utilizing GTSAM to enforce fixed-interval and proximity-based loop closure constraints.
5) Occupancy grid and texture mapping for visual verification and simultaneous mapping.

II. Problem Formulation

A. Robot Description

We have data from a differential drive robot equipped with the following sensors:

- Wheel Encoders (counting at 40Hz)
- Inertial Measurement Unit (IMU)
- 2-D LiDAR scanner (Hokuyo UTM-30LX) with 270° FOV
- RBG-D camera (Kinect)



Fig. 1. Differential Drive Robot

B. Data Pre-Processing

A fundamental challenge in this trajectory estimation problem is the lack of hardware-level temporal synchronization; each sensor operates asynchronously at its own independent frequency. To perform accurate state estimation and mapping, these disparate data streams must be unified into a cohesive timeline. We utilize a combination of nearest-neighbor assignment and linear interpolation to temporally align the measurements.

We want to estimate the sequence of robot poses over time $x_{0:T} = \{x_t, y_t, \theta_t\}_t^T$, using the provided sensors. To use a

differential drive kinematic model to obtain these poses, we need to first find the control input $\mathbf{u}$ containing the linear velocity v_t and angular velocity ω_t:

$$\mathbf{u} = \begin{bmatrix} v_t \\ \omega_t \end{bmatrix}$$

The encoders measure the rotation of the four wheels, reporting data as a vector of cumulative tick counts C_t at a given time step:

$$C_t = \begin{bmatrix} FR & FL & RR & RL \end{bmatrix}^\top$$

representing the front-right, front-left, rear-right, and rear-left wheels, respectively. We want to use this data to derive the robot's linear velocity v_t.

From the IMU, we are directly given the angular velocity for $t \in [T]$:

$$\omega_t = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^\top \in \mathbb{R}^3$$

Since the motion model for a differential drive robot assumes planar motion, we are primarily interested in using the yaw-rate ω_t.

With these control inputs, we want to estimate the sequence of "dead-reckoning" poses using the discrete time differential drive kinematic motion model:

$$x_{t+1} = \begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = x_t + \Delta t = \begin{bmatrix} v_t sinc\left(\frac{\omega_t \Delta t}{2}\right) \cos\left(\theta_t + \frac{\omega_t \Delta t}{2}\right) \\ v_t sinc\left(\frac{\omega_t \Delta t}{2}\right) \sin\left(\theta_t + \frac{\omega_t \Delta t}{2}\right) \\ \omega_t \end{bmatrix}$$

C. Point Cloud Registration

Given a target point cloud of landmark positions $\mathcal{M} = \{\mathbf{m}_i\}_{i=1}^{N_m} \subset \mathbb{R}^d$ and a source point cloud of relative position measurements $\mathcal{Z} = \{\mathbf{z}_j\}_{j=1}^{N_z} \subset \mathbb{R}^d$ with an unknown data association mapping Δ, the objective of point cloud registration is to determine the optimal rigid body transformation that aligns the source set to the target set.

Specifically, we seek to find the translation vector $\mathbf{p} \in \mathbb{R}^d$, rotation matrix $R \in SO(d)$, and correspondence set Δ that minimize the sum of squared Euclidean distances between the associated points. Mathematically, this is formulated as the following joint optimization problem:

$$\min_{\mathbf{p} \in \mathbb{R}^d, R \in SO(d), \Delta} \sum_{(i,j) \in \Delta} \|(R\mathbf{z}_j + \mathbf{p}) - \mathbf{m}_i\|_2^2$$

We apply this ICP algorithm in the following two cases:

1) **3D Point Cloud Matching**: Determining the 3-D poses of objects from depth images (here, $d = 3$)
2) **Scan matching:** estimating the 2-D relative pose changes, ${}_tT_{t+1}$, between consecutive LiDAR scans at time t and $t+1$ (here, $d = 2$).

Once this relative transformation ${}_tT_{t+1}$ between two time steps t and $t+1$ is determined, we wish to estimate the the robot pose T_{t+1} at time $t+1$.

D. LiDAR Based Occupancy Grid Mapping

The objective of occupancy grid mapping is to estimate the map m of the environment given a known robot trajectory $x_{0:t}$ and observations $z_{0:T}$. The environment is divided into a regular grid of n cells, where each entry m_i indicates whether the i-th cell is free ($m_i = 0$) or occupied ($m_i = 1$). We assume that cell values are independent conditioned on the robot trajectory:

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

Thus, we want to track the probability of being occupied,

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

for each cell $i = 1, \ldots, n$. To have "additive" updates, we formulate this problem as the estimation of the log-odds $\lambda_{i,t}$ for each cell $i = 1, \ldots, n$:

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

Then, across time, this estimation becomes:

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

where $\Delta\lambda_{i,t} = g_h(\mathbf{z_t}|m_i, \mathbf{x_t})$ is the observation model log-odds for the i-th cell at time t. For LiDAR based mapping, we need to determine the set of cells intersected by each measurement ray to update the log-odds of both the *perceived* free space, the occupied dead ends, and unexplored areas.

E. Texture Mapping

We want to create a 2D color map of the floor surface by fusing the RGB-D camera measurements with the optimized robot trajectory. That is, at time step t, we are given an RGB image $I^{(t)}$, a corresponding depth map $D^{(t)}$, and the optimized robot pose $T_t \in SE(2)$. Then, we want to find:

- For a given image I_t, the set, S of all coordinates (u, v) in the image that geometrically intersect with the ground plane.
- A function f that maps a valid floor pixel $(u, v) \in S$, its measured depth $z = D^{(t)}(u, v)$, and the current robot pose T_t to a 2D discrete cell (m_x, m_y) in the global world frame:

$$(m_x, m_y) = f(u, v, z, T_t)$$

- The texture map M populated such that the color of the cell (m_x, m_y) matches the corresponding RGB pixel value at time t:,

$$M(m_x, m_y) = I_t(u, v) \quad \forall (u, v) \in S, \forall t$$

F. Pose Graph Optimization

To enhance the accuracy of our robot trajectory estimation and correct for accumulated drift, we do Pose Graph Optimization with loop closure constraints. The pose graph is defined as a bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where the components represent our variables and measurements:

- The vertex set $\mathcal{V}$ represents the robot's sequential states $\{T_i\}_{i=0}^{N}$. Each node $T_i \in SE(d)$ corresponds to the robot's pose in the world frame, $\{W\}$ at time step i.
- The edge set $\mathcal{E}$ represents the measured relative poses $\bar{T}_{ij} \in SE(d)$ between the state at node i and the state at node j.

Additionally, the edges (i.e. the relative pose factors) are obtained from two sources:

- Odometry factors: relative poses ${}_i\bar{T}_{i+1}$ between consecutive time steps $(i, i+1)$, are obtained via frame-to-frame ICP scan matching from Section II.B.
- Loop Closure factors: non-sequential relative poses $\bar{T}_{ij}$ between non-adjacent time steps where $|i-j| > 1$. These edges are conditionally introduced when the robot revisits a previously mapped area.

For any measurement factor $(i, j) \in \mathcal{E}$, the expected relative pose computed from the current state variables is,

$$ {}_i\bar{T}_j = T_i^{-1} T_j $$

Furthermore, the relative pose vectors $\mathbf{e}(T_i, T_j)$ is defined as:

$$ \mathbf{e}(T_i, T_j) = \log(\bar{T}_{ij}^{-1} T_i^{-1} T_j)^{\vee} $$

.

The general back-end optimization problem seeks to find the optimal configuration of node variables $\{T_i\}$ by minimizing a distance function $\phi_{ij}(\mathbf{e}) = \mathbf{e}^{\top} W_{ij}^{\top} W_{ij} \mathbf{e} = \|W_{ij}\mathbf{e}\|_2^2$, where $W_{ij}^{\top} W_{ij}$ represents the confidence of the specific measurement. It is *how* much trust we have in the sensor. Substituting the pose graph factors into this general formulation, our final PGO objective is defined as the following nonlinear least-squares problem:

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

By minimizing this objective function, we seek to find a consensus between the measured relative poses and the estimated ones, and improve our trajectory.

III. Technical Approach

A. Data Synchronization

IMU to Encoder Synchronization. Since the IMU samples at a higher frequency than the 40 Hz encoders, we align the rotational data to the encoder timeline using a nearest-neighbor search. For each discrete encoder time-step t_k, we locate the closest IMU timestamp t_j at index j to extract the corresponding applied angular velocity:

$$ j^*(k) = \arg\min_j |t_k - t_j| \quad \omega_k = \Omega_{z,j^*(k)} $$

Odometry to LiDAR Interpolation. To correct the accumulating drift of the dead-reckoning model, we utilize 2-D LiDAR scan matching. Because ICP is highly sensitive to initialization, it requires a precise relative odometry guess at the exact time of each LiDAR scan, t_L. Since LiDAR timestamps rarely align perfectly with the discrete kinematic updates, we linearly interpolate the absolute dead-reckoning pose x_L between the bounding encoder timestamps $t_k \le t_L < t_{k+1}$:

$$ \gamma = \frac{t_L - t_k}{t_{k+1} - t_k}, \quad x_L = (1-\gamma)x_k + \gamma x_{k+1} $$

The relative geometric transformation between the previous scan's interpolated pose x_{L-1} and the current pose x_L is subsequently extracted to serve as the initialization prior $(\bar{\mathbf{p}}^{(0)}, \bar{\theta}^{(0)})$ for the ICP solver.

LiDAR Spatial Pre-processing. Finally, before the scan matching objective is evaluated, the raw LiDAR data must be filtered and mapped to Cartesian space. For a raw scan consisting of ranges r_i and angles α_i, we discard invalid measurements falling outside the physical bounds of the Hokuyo sensor ($r_i < r_{\min}$ or $r_i > r_{\max}$). The remaining valid ranges are projected into Cartesian coordinates in the local sensor frame:

$$ \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} r_i \cos\alpha_i \\ r_i \sin\alpha_i \end{bmatrix} $$

These filtered Cartesian coordinates construct the point clouds $\mathbf{z}_j$ utilized in the optimization step.

B. Encoder and IMU Odometry Estimation

Linear Velocity. First, we begin by estimating the linear velocity v_t from the wheel encoders. We have that the diameter of the robot's wheel is 0.254, and the encoder has a resolution of 360 ticks per revolution.

$$ C = \pi \cdot d = \pi \cdot 0.254 \approx 0.79796 \text{ meters/revolution} $$

Then the distance per tick can be computed:

$$ \text{Distance per tick} = \frac{0.79796 \text{ meters}}{360 \text{ ticks}} \approx 0.002216 \text{ meters/tick} $$

We compute the physical distance traveled by the right and left sides of the robot by averaging the counts of the respective front and rear wheels as follows:

$$ \Delta d_{R,k} = \frac{FR_k + RR_k}{2} \cdot 0.0022 $$

$$ \Delta d_{L,k} = \frac{FL_k + RL_k}{2} \cdot 0.0022 $$

The incremental linear displacement of the robot's center Δs_k over the time interval $\Delta t_k = t_k - t_{k-1}$ is the average of the left and right wheel displacements:

$$ \Delta s_k = \frac{\Delta d_{R,k} + \Delta d_{L,k}}{2} $$

Therefore, the instantaneous velocity at timestep k is given by:

$$ v_k = \frac{\Delta s_k}{\Delta t_k} $$

Angular Velocity. The angular velocity vector at a given time t is directly obtained from the Inertial Measurement Unit (IMU). The raw IMU data provides a $3 \times N$ array of angular velocities, representing the rotational rates about the x, y, and z axes $(\omega_x, \omega_y, \omega_z)$, coupled with an N-length array of UNIX time stamps. We isolate the yaw rate, which corresponds to the third row of the $3 \times N$ array of the IMU angular velocity data.

Application of the motion model: With the linear and angular velocities extracted, the robot's state $x_k = [x_k, y_k, \theta_k]^\top$ is propagated forward in time. Assuming that the velocities are held constant over the small time interval Δt_k, we utilize the discrete-time kinematic motion model as follows:

$$x_k = x_{k-1} + v_k \cos(\theta_{k-1} + \frac{\omega_k \Delta t_k}{2})$$

$$y_k = y_{k-1} + v_k \sin(\theta_{k-1} + \frac{\omega_k \Delta t_k}{2})$$

$$\theta_k = \theta_{k-1} + \omega_k \Delta t_k$$

We apply this iteratively, starting at an initial starting pose $x_0 = [0, 0, 0]^\top$. This generates a baseline dead reckoning trajectory shown in Figure 2 and 3.

C. Iterative Closest Point Algorithm (ICP) for Point Cloud Registration with Unknown Data Association

In this section, we discuss how we solve the following joint optimization problem to find R and $\mathbf{p}$ that align the point clouds $\{\mathbf{m}_i\}_{i=1}^{N_m} \subset \mathbb{R}^d$ and $\{\mathbf{z}_j\}_{j=1}^{N_z} \subset \mathbb{R}^d$ with an unknown data association Δ:

$$\min_{\mathbf{p} \in \mathbb{R}^d, R \in SO(d), \Delta} \sum_{(i,j) \in \Delta} \|(R\mathbf{z}_j + \mathbf{p}) - \mathbf{m}_i\|_2^2$$

where all Δ, R and $\mathbf{p}$ are all initially unknown. The ICP algorithm iterates between doing the following (and we initialize a guess p_0 and R_0):

1) Given $\mathbf{p_k}$ and R_k, **find correspondences** $(i, j) \in \Delta$ based on closest points:

$$i \leftrightarrow \arg\min_j \|\mathbf{m}_i - (R_k \mathbf{z_j}) + \mathbf{p_k}\|$$

2) Given a data association Δ, use the Kabsch Algorithm to **find the transformation** R_{k+1} and $\mathbf{p_{k+1}}$ that align $\{\mathbf{m}_i\}_{i=1}^{N_m}$ and $\{\mathbf{z}_j\}_{j=1}^{N_z}$

Next, we discuss the (i) Kabsch Algorithm and (ii) how we find the data correspondences.

(i) Kabsch Algorithm. With a given data association Δ, we do the following 6 steps:

1) Compute centroids:

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

2) Center point clouds around the mean:

$$\delta m_i = m_i - \bar{m}, \quad \delta z_i = z_i - \bar{z}$$

3) Formulate Wahba's Problem:

$$W = \sum_{i=1}^{n} \delta m_i \delta z_i^\top$$

4) Singular Value Decomposition of W: Computed directly using `NumPy`'s implementation

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

5) Compute the optimal (R): To ensure the resulting matrix constructed from U and V is a valid physical rotation (i.e. it is $R \in SO(3)$ with determinant $+1$) and to strictly avoid reflection, we apply a correction factor $\det(UV^\top)$ to the last column. Thus, R is given by:

$$R = U \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \det(UV^\top) \end{bmatrix} V^\top$$

6) Compute the optimal ($\mathbf{p}$): With the optimal rotation R, we back-substitute it to find the optimal translation p:

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

(ii) Determining data correspondences Δ. Given some R and $\mathbf{p}$, we want to find a Δ. To achieve fast computation, we utilize the `SciPy` implementation of a k-dimensional tree (KD-Tree) data structure. This allows us to spatially partition the target point cloud $\mathcal{M}$. Then, the KD-Tree recursively subdivides the $\mathbb{R}^d$ search space into distinct half-spaces using axis-aligned hyperplanes.

By querying this tree, the search space for any transformed source point $\mathbf{z}'_j = R_k \mathbf{z}_j + \mathbf{p}_k$ is drastically bounded. The tree traversal eliminates large, distant segments of the target cloud from our search consideration. This allows the nearest-neighbor objective:

$$i^*(j) = \arg\min_i \|\mathbf{m}_i - \mathbf{z}'_j\|_2^2$$

to be evaluated in $O(\log N_m)$ time on average. Therefore, the overall time becomes $O(N_z \log N_m)$, where N_z is the total number of points in the source point cloud $\mathcal{Z}$ (the current LiDAR scan) and N_m is the total number of points in the target point cloud $\mathcal{M}$ (the reference map or prior scan).

This general Point Cloud Registration technique is then used in the two contexts mentioned before:

- **3D-ICP (Warm-up):** The algorithm aligns point clouds in three-dimensional space. The points are $\mathbf{m}_i, \mathbf{z}_j \in \mathbb{R}^3$, and the transformation parameters are translation $\mathbf{p} \in \mathbb{R}^3$ and rotation $R \in SO(3)$. The registration objective is:

$$\min_{R \in SO(3), \mathbf{p} \in \mathbb{R}^3, \Delta} \sum_{(i,j) \in \Delta} \|(R\mathbf{z}_j + \mathbf{p}) - \mathbf{m}_i\|_2^2$$

  Provided point clouds for a drill ($\sim 14,000$ 3D points) and a liquid container ($\sim 42,000$ 3D points) are used to verify the implementation of ICP in 3 dimensions, and the implementation is generalizable to 2 dimensions for scan matching.

- **2D-ICP (Scan Matching):** The algorithm aligns sequential LiDAR scans to correct local dead-reckoning drift between time steps $t-1$ and t. Assuming planar motion, the source point cloud at time t, $\mathbf{z}_{t,j} \in \mathbb{R}^2$, is aligned to the target point cloud from the previous time step, $\mathbf{z}_{t-1,i} \in \mathbb{R}^2$. The relative translation is denoted as ${}_{t-1}\mathbf{p}_t \in \mathbb{R}^2$, and the relative rotation $R({}_{t-1}\theta_t) \in SO(2)$ is parameterized by the change in yaw ${}_{t-1}\theta_t$:

$$R({}_{t-1}\theta_t) = \begin{bmatrix} \cos({}_{t-1}\theta_t) & -\sin({}_{t-1}\theta_t) \\ \sin({}_{t-1}\theta_t) & \cos({}_{t-1}\theta_t) \end{bmatrix}$$

Since ICP is highly sensitive to the initial guess, the relative odometry derived from the kinematic model is supplied as the initialization $({}_{t-1}\mathbf{p}_t^{(0)}, R({}_{t-1}\theta_t^{(0)}))$. The reduced 2-D optimization objective becomes:

$$\min_{{}_{t-1}\theta_t \in \mathbb{R}, {}_{t-1}\mathbf{p}_t \in \mathbb{R}^2, \Delta} \sum_{(i,j)\in\Delta} \|({}_{t-1}R_t \mathbf{z}_{t,j} + {}_{t-1}\mathbf{p}_t) - \mathbf{z}_{t-1,i}\|_2^2$$

The number of iterations for each call of the ICP algorithm was set to `MAX_ITER=50`. The following practical considerations were made in our implementation:

1) `MAX_DIST` Threshold: A threshold was applied in the step where we find correspondences. Between two frames, there should not be *too* large of a distance between corresponding points. If there is, it is likely sensor noise, and we should discard those correspondences. This prevents Kabsch Algorithm from minimizing error over "bad" matches. This value was hand tuned.
2) Multi Start optimization. ICP is a local optimizer and is highly sensitive to the initial guess. If the robot turns too quickly between scans, the kinematic model's guess falls into a local minimum. We see this specifically in LiDAR scans 2100 - 2200. We test multiple initial orientation offsets:

$$\theta_{initial} = \theta_{odometry} + \delta\theta, \quad \delta\theta \in \{0, \pm 30^\circ, \pm 45^\circ, \pm 60^\circ\}$$

The algorithm runs ICP for each offset and selects the $(R, \mathbf{p})$ pair that yields the lowest final Mean Squared Error.
3) `STRIDE =5`. We use a stride to skip over every 5 scans. This makes the process significantly faster, and we get a enough difference in consecutive scans to not get stuck in a local minima.

D. Constructing the Occupancy Grid Map

We set up the map as a $50m$ by $50m$ grid with a resolution of $0.05m$ per cell. Continuous world coordinates (x, y) of the robot and LiDAR points are mapped to discrete 2D array indices (u, v) by shifting the coordinate origin to the center of the grid and scaling by the resolution.

To identify the sequence of cells intersected by a given LiDAR measurement, we draw a ray from the robot's current estimated pose to the global coordinate of the LiDAR hit. We implement this using OpenCV's line-drawing function `cv2.line`. The cell in which the ray ends represents the physical point where the laser beam reflected off an object or a wall.

Using a mask generated from the ray tracing, we apply the additive log-odds update to the global map $\lambda_{i,t}$:

$$\Delta\lambda_{free} = -\ln(4.0) \Delta\lambda_{occ} = \ln(4.0)$$

Furthermore, unexplored cells receive no update. Next, to visualize the map, our log-odds values are converted back into a probability space $P \in [0, 1]$ using the Sigmoid function :

$$P(m_i = 1) = 1.0 - \frac{1}{1 + \exp(\lambda_{i,t})}$$

Then, we simply use `NumPy` to visualize this probability map in gray-scale, resulting in our map.

We realized that, sometimes, very high probabilities were being assigned to cells that came from noisy LiDAR rays. Therefore, to make the resulting occupancy map look better, we clamp he log-odds of every cell to the interval $[-5.0, 5.0]$. This allows the map to be able to recover if we see LiDAR scans that do not agree with a previous one.

E. Constructing the Texture Map

To construct the texture map, we first convert the image from camera frame to the robot frame, and then to the world frame. The steps are outlined below.

1) **Getting depth values from disparity.** We compute the true depth values,Z (in meters) for each pixel, from the given disparity using the following conversion: :

$$d' = -0.00304 \cdot d + 3.31$$

$$Z = \frac{1.03}{d'}$$

2) **Projection into Camera Frame:** Using the standard pinhole camera model and the known intrinsic parameters ($f_x = f_y = 585.05$, $c_x = 315.84$, $c_y = 242.94$), we project the 2D pixels (u, v) and their corresponding depths Z into 3D points in the camera's optical frame, $\{C_{opt}\}$:

$$\mathbf{p}_c = [x_c, y_c, z_c]^\top$$

.
3) **Extrinsic Transformation and Floor Thresholding:** We convert the 3D points from the camera's optical frame $\{C_{opt}\}$ (where the z-axis represents depth) into the robot's base link frame $\{B\}$. We apply the extrinsic rotation matrix, and add a spatial translation of $[0.18, 0.005, 0.36]^\top$ meters to account for the camera's mounting offset. To isolate the ground plane, we filter out any point whose height relative to the wheels exceeds 0.30 m ($|z_r| \geq 0.30$). This leaves us with the set S.
4) **Pixel Alignment:** Because the depth sensor and the RGB camera are spatially offset, a pixel (u, v) in the depth image does not map exactly to the same coordinates in the RGB image. So, for the valid floor points,

we calculate their corresponding RGB pixel indices (i_{rgb}, j_{rgb}):

$$i_{rgb} = \text{round}\left(\frac{526.37u + 19276 - 7877.07d'}{585.051}\right)$$

$$j_{rgb} = \text{round}\left(\frac{526.37v + 16662}{585.051}\right)$$

We sample the color $\mathbf{c}$ from the RGB image at these indices.

5) **Converting to the world frame** Then, we convert from the camera optical frame $\{C_{opt}\}$ to the world frame $\{W\}$ by applying the relative pose ${}_tT_{t+1}$ between two timesteps. This could be the dead reckoning, ICP or GTSAM optimized trajectory pose. We discretize the world coordinates to our resolution: `0.05 m/pixel`. We shift these $\frac{1}{2}$ the map dimension to have everything centered in the middle. To make the map look visually better, we combine it with the occupancy grid map and remove any outlier RGB colors from the map.

F. Pose Graph Optimization

To implement Pose Graph Optimization, we utilize the Georgia Tech Smoothing and Mapping (GTSAM) library. We will talk about the two main phases of our approach: (i) Front-end graph construction, and (ii) the back-end optimization.

Variable Representation The graph is constructed incrementally as the robot navigates the environment. Since, in our case, we have 2D poses, we define the state variable as: $T_i \in SE(2)$, containing the x, y coordinates and yaw angle θ:

$$T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & x_i \\ \sin\theta_i & \cos\theta_i & y_i \\ 0 & 0 & 1 \end{bmatrix}$$

and it is stored as the following to keep our code fast:

$$\mathbf{x}_i = \begin{bmatrix} x_i \\ y_i \\ \theta_i \end{bmatrix}$$

Additionally, the edges are the measured relative poses $\bar{T}_{ij} \in SE(2)$ between nodes. Below, we discuss how we add these edges. **Noise Modeling.** The confidence of each measurement is defined by a diagonal noise model Σ. In our implementation, we have have the following noise:

1) Prior Noise (10^{-6}): High-certainty constraint that anchors T_0 to the origin. This is done to prevent global coordinate drift.
2) Odometry Noise (0.1m, 0.05 rad): Lower-certainty weight for sequential ICP factors.
3) Loop Closure Noise (0.01m, 0.01 rad): High-certainty weight that forces the trajectory to align when revisiting known areas.

Front-End: Graph Construction We incrementally build the bipartite graph as we go through each scan

1) Sequential Edges: We add factors between T_{i-1} and T_i using relative poses that we compute from the ICP optimized odometry.
2) Loop Closure factors: These are added between T_i and T_j ($|i - j| > 1$) when the robot revisits a place that it has seen before. We have two types of loop closures:
   - Proximity based loop closure: We compare LiDAR scans at nearby poses. If the scans match well, this indicates a loop closure.
   - Fixed interval loop closure: After every fixed number of robot poses (we use 10 in our program). This is done by matching the current LiDAR scan to a historical one 10 scans before

   We only add the edge to our graph if the ICP "fitness" score (MSE) between two point clouds P_S and P_t is below a specific threshold. This fitness metric for this consistency check is defined as:

$$F(P_s, P_t) = \frac{1}{N}\sum_{i=1}^{N} \|(R\mathbf{q}_i + p) - \mathbf{k}_i\|^2$$

   where $\mathbf{q}_i$ is a point in the source point cloud, $\mathbf{k}_i$ is the corresponding nearest neighbor in the target point cloud and N is the number of points in the source cloud. This threshold is set to 0.05 in our implementation, and was tuned by hand.

Back-end optimization. We solve the back end optimization problem described in Section II. F. This is done by utilizing GTSAM's implementation of the Levenberg-Marquardt optimizer. The overall objective is to minimize the total non-linear least-squares error across all factors:

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

where $\phi_{ij}(\mathbf{e}) = \mathbf{e}^\top W_{ij}^\top W_{ij}\mathbf{e}$, and W_{ij} is the information matrix derived from our assigned noise models. GTSAM iteratively updates poses by doing the descent: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\delta\mathbf{x}^{(k)}$. To find the optimal step $\delta\mathbf{x}^{(k)}$ at each iteration, the GTSAM linearizes the error and solves the following system:

$$\left(\sum_{ij} J_{ij}^\top W_{ij}^\top W_{ij} J_{ij} + \lambda D\right)\delta\mathbf{x}^{(k)} = -\sum_{ij} J_{ij}^\top W_{ij}^\top \mathbf{e}(\mathbf{x}_i^{(k)}, \mathbf{x}_j^{(k)})$$

Where:

- $J_{ij} = \frac{\partial \mathbf{e}(\mathbf{x}_i, \mathbf{x}_j)}{\partial \mathbf{x}}$ is the Jacobian matrix of the error function evaluated at the current estimate $\mathbf{x}^{(k)}$.
- λ is some dynamic damping factor
- D is a diagonal matrix.

Finally, when the minimization algorithm converges, GTSAM optimizer outputs the optimal configuration of the state variables, T_i. We iterate through the optimized factor graph to extract the updated global pose $\mathbf{x}_i = [x_i, y_i, \theta_i]^\top$ for each time step. This newly refined trajectory replaces our initial dead-reckoning and sequential ICP estimates.

IV. Results

A. Encoder and IMU odometry

Using the synchronized data from the wheel encoders and IMU, and our discrete time differential drive motion model, we get the following dead-reckoning trajectory for Dataset 20, shown in Figure 2. Furthermore, for Dataset 21, we also get a similar trajectory (shown in Figure 3) Here, we can imagine that the robot in moving through "L" shaped hallways, making this estimation seem reasonable.

Furthermore, we also construct a Occupancy Grid map using the dead reckoning trajectory of the robot, along with the LiDAR data, We see that this map is very noisy, and contains a lot of "ghost" cells – i.e. areas that should not be explored, appear as free cells. This is because there is a lot of drift in the dead reckoning estimation. While the general shape of the hallways is maintained, the hallway looks "thicker" because of the drift. In the later sections, we use ICP and Factor Graph optimization to correct for this.

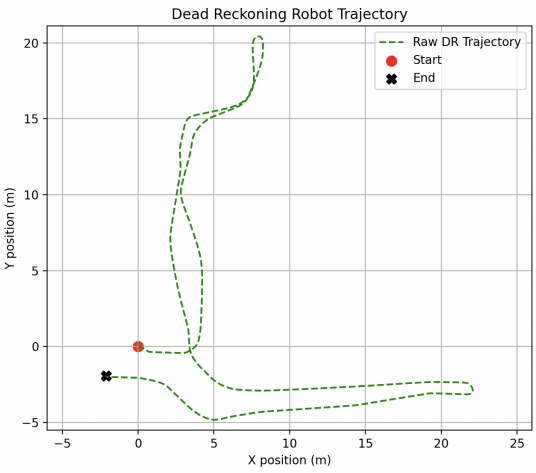


Fig. 2. The robot's trajectory using dead reckoning (Dataset 20).

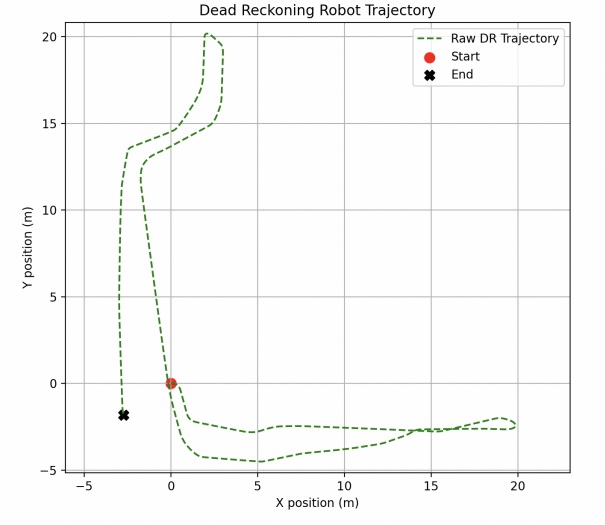


Fig. 3. The robot's trajectory using dead reckoning (Dataset 21).

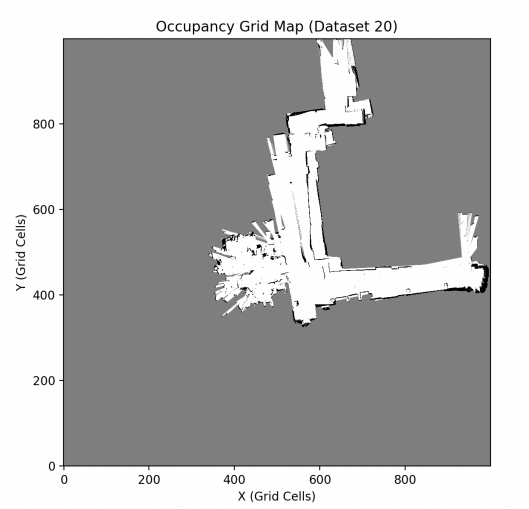


Fig. 4. Occupancy Grid Map using the robot's dead reckoning trajectory (Dataset 20).

B. Point-cloud registration via iterative closest point (ICP): Warm Up

We tested our ICP implementation on point clouds of a drill and a liquid container. As we were implementing the ICP algorithm, we found that doing a plain nearest neighbor search for finding data correspondence (even with `NumPy`'s vectorized operations), was slow and intractable for dense point clouds. During our initial runs of this algorithm without a KD-Tree, we found that the algorithm took 20 mins to complete for matching one sample of the drill. and over 1 hour to match one sample of the liquid container as it had more points. This indicates how computationally intensive the ICP algorithm really is.

From Figure 6 and 7, we demonstrate the matching between the source (blue) and the target (red) point clouds. Overall, the algorithm achieves a reasonably good alignment for both the drill and the liquid container models. However, as visualized in the point cloud overlays, the registration is not always perfect. For example, in Figure 6, panel 2, we can see we observe a clear mismatch at the base of the handle. Instead of correctly wrapping to the outer boundary points, the surface points of the source point cloud (blue) are have matched to the internal, *volumetric* points of the target point cloud's handle (red). This could indicate that the ICP algorithm had reached a local minima, and decided that this was the best matching.

When processing denser point clouds, such as the liquid container shown in Figure 7, we achieve a noticeably better alignment. This is likely because there is more geometric information in dense point clouds. Since this basic implementation of the ICP Algorithm seemed to work, we moved back to the 2 dimensional case of scan matching for estimating the trajectory of our robot.

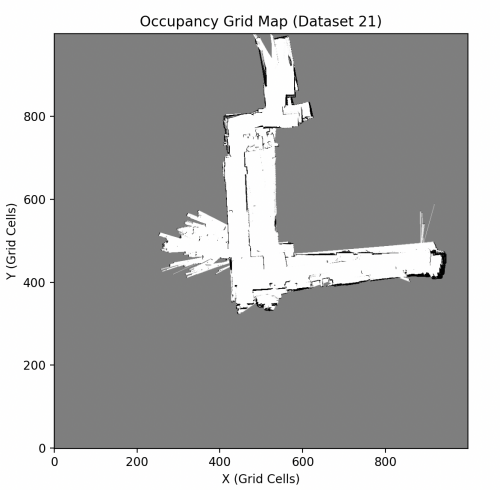


Fig. 5. Occupancy Grid Map using the robot's dead reckoning trajectory (Dataset 21).

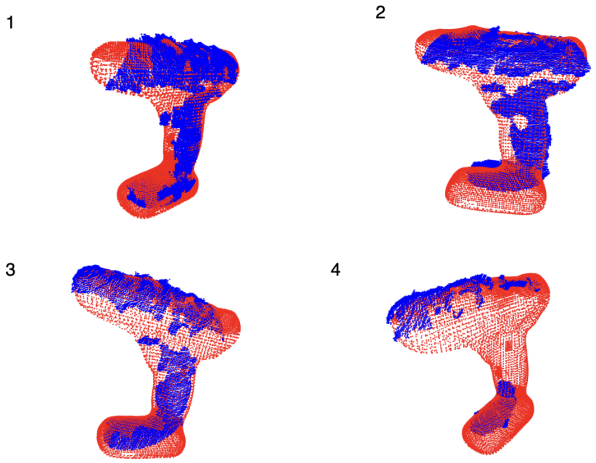


Fig. 6. 3D point cloud matching with drill. Red is the target point cloud and blue is the source.

C. *Point-cloud registration via iterative closest point (ICP): Scan Matching*

As we mentioned before, we use the estimates from the encoder and IMU odometry as initializations for the ICP algorithm. The differential drive dead reckoning motion model already provides a robust initial pose estimate. Thus, there is very little left to correct for the algorithm. During standard translations, the initial alignment is strong, and the ICP cost generally converges as expected. However, rotational movements naturally introduce larger odometry errors. In these turning scenarios, rather than strictly decreasing to a minimum, we observe that the ICP cost often increases and subsequently flattens out. This plateauing behavior indicates that the algorithm is struggling to align point clouds between rotational motion. It is likely getting caught in local minima due to poor observation, sudden or fast movements or just due to confusion about the geometry of the environment. In

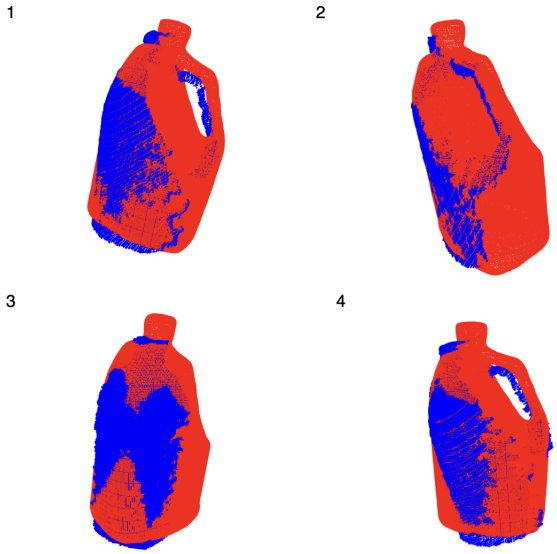


Fig. 7. 3D point cloud matching with liquid container. Red is the target point cloud and blue is the source.

the occupancy and texture map sections, we show how this suboptimal convergence leads to a poor map construction and accumulation of drift.

In Figure 8 10, a clear visualization of this phenomenon across three different somewhat consecutive (100 apart) scans. In the first row, for $k = 2100$, the robot is moving straight, and is leaving a room. Here, the mean squared error cost curve decreases smoothly and converges to a stable minimum, indicating a successful point cloud registration. In the second row, for $k = 2200$, we can see that the robot has undergone rotational motion. This is evident from the fact that the entire point cloud looks rotated, and we can see that a hallway starts to form. Instead of a monotonic descent, the ICP cost initially drops but then goes up again and finally plateaus at a suboptimal cost. This shows that the algorithm is trapped in a local minimum. Once the robot completes the turn and resumes moving straight down the hallway (step $k = 2300$ in row 3), the ICP cost curve converges smoothly again. The angular error accumulated around $k = 2200$ adds to the drift in the trajectory. This drift is visualized in the occupancy grid map and texture map discussed in later sections.

Furthermore, after collecting the optimized R and $\mathbf{p}$ at each iteration, we end up with the trajectory curves for datasets 20 and 21, as shown in Figures 9

D. *Occupancy Grid Map and Texture Map Results (ICP)*

We can compare our post-ICP occupancy grid maps shown in Figures 12 and 14 with the the dead reckoning occipancy grid maps shown in Figures 4 and 5, we can see that there has been a huge improvement in the estimation of our map. Qualitatively, we can see that there are significantly fewer "ghost" hallways and the characteristic "L" shape of the hallway is more evident. We can see that our resuts for Dataset

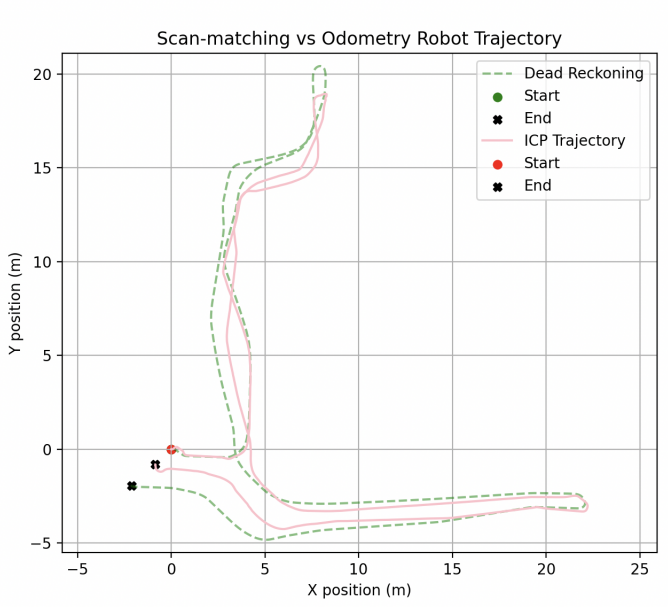


Fig. 8. Plot of dead reckoning trajectory and ICP optimized trajectory For Dataset 20

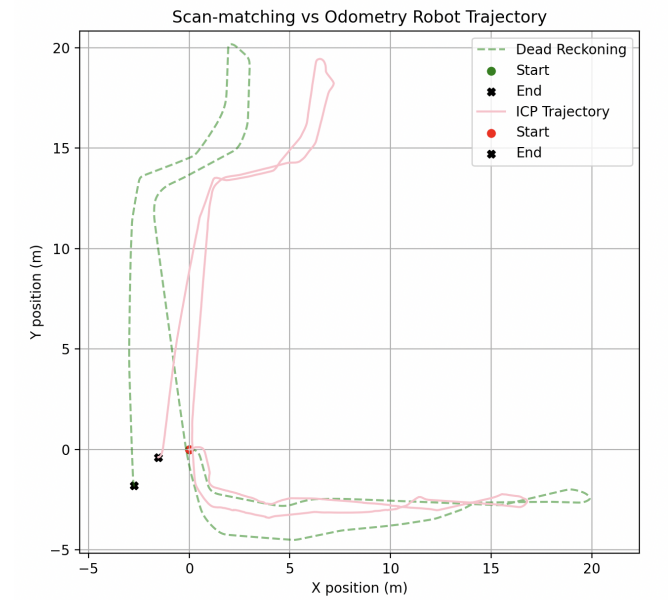


Fig. 9. Plot of dead reckoning trajectory and ICP optimized trajectory For Dataset 21

20 are much better than those for Dataset 21. For the former, the hallways look almost perfectly perpendicular. However, for the latter, the hallway looks slightly slanted. This could be due to ICP getting stuck in a suboptimal minima when the robot rotates to take a turn (as described in 10). This leads to the extra ghost cells we see at the intersection of the hallways.

Figures 13 and 15 show the texture maps, which are achieved by projecting the RGB floor data onto the global coordinate frame using our ICP estimated trajectory. The floors look smooth in general, showing acceptable trajectory estimation. In areas where we do have drift, we see that the floor is not very smooth and continuous.

Interestingly, we note the significance of using the `MAX_DIST` rule outlined in Section III B. This threshold tells ICP to ignore correspondences that have a very large distance. Since we are running this algorithm frame by frame, there should not be a very large difference between corresponding points. In Figure 11, we show how setting a very generous `MAX_DIST` leads to the trajectory shooting off in the wrong direction, leading to incorrect maps. Thus, that check is very important for the performance of this SLAM implementation.

E. ICP Texture Map Videos Over time

You can see the construction of the graph in real time on these screen recordings.

- Dataset 20: https://youtu.be/VlbKOFCMRlE
- Dataset 21: https://youtu.be/uroUpcuDuRM

We see, in both cases, that the construction of the graph is smooth over time, but drift starts accumulating when the robot takes a 90° turn within the hallways. Interestingly, as the map updates, the texture map effectively traces the robot's estimated trajectory in real time. For Dataset 20, we can see that as it returns down the first hallway, it starts accumulating some angular drift. This phenomenon is more enhanced in dataset 21. Overall, we can also visualize how the robot explores the area, while simultaneously building a reasonably accurate map of the environment.

F. Pose Graph Optimization Results

From Figure 16, we note that the dashed-green line demonstrating the dead reckoning trajectory has a lot of drift. ICP trajectory (pink line) significantly improves local consistency and scales the map correctly, it still leads to some drift, which is evident in the construction of maps using the ICP trajectory. The GTSAM-optimized trajectory (blue line) shows an even greater improvement on the previous two trajectories. We see that the path is more "straight" and follows the desired "L" shape.

Applying this refined trajectory to our mapping modules yields better maps of the environment. While there is still some error, it is less compared to our previous results. Most importantly, we have minimized the angular drift. We no longer see "ghost" cells around the areas where the robot did a 90 degree turn. The corridors now exhibit perpendicular intersections, reflecting the true physical geometry of the building. Because the underlying global poses are now highly accurate, the texture map created with the RGB-D floor data is also much more smooth. While the trajectory for Dataset 21 is not perfect, it is also a massive improvement from our previous estimations, where the width of the corridor looked twice its true size

G. PGO Texture Map Videos

You can see the construction of the graph in real time on these screen recordings:

- Dataset 20: https://youtu.be/7fcoYnrF8NI

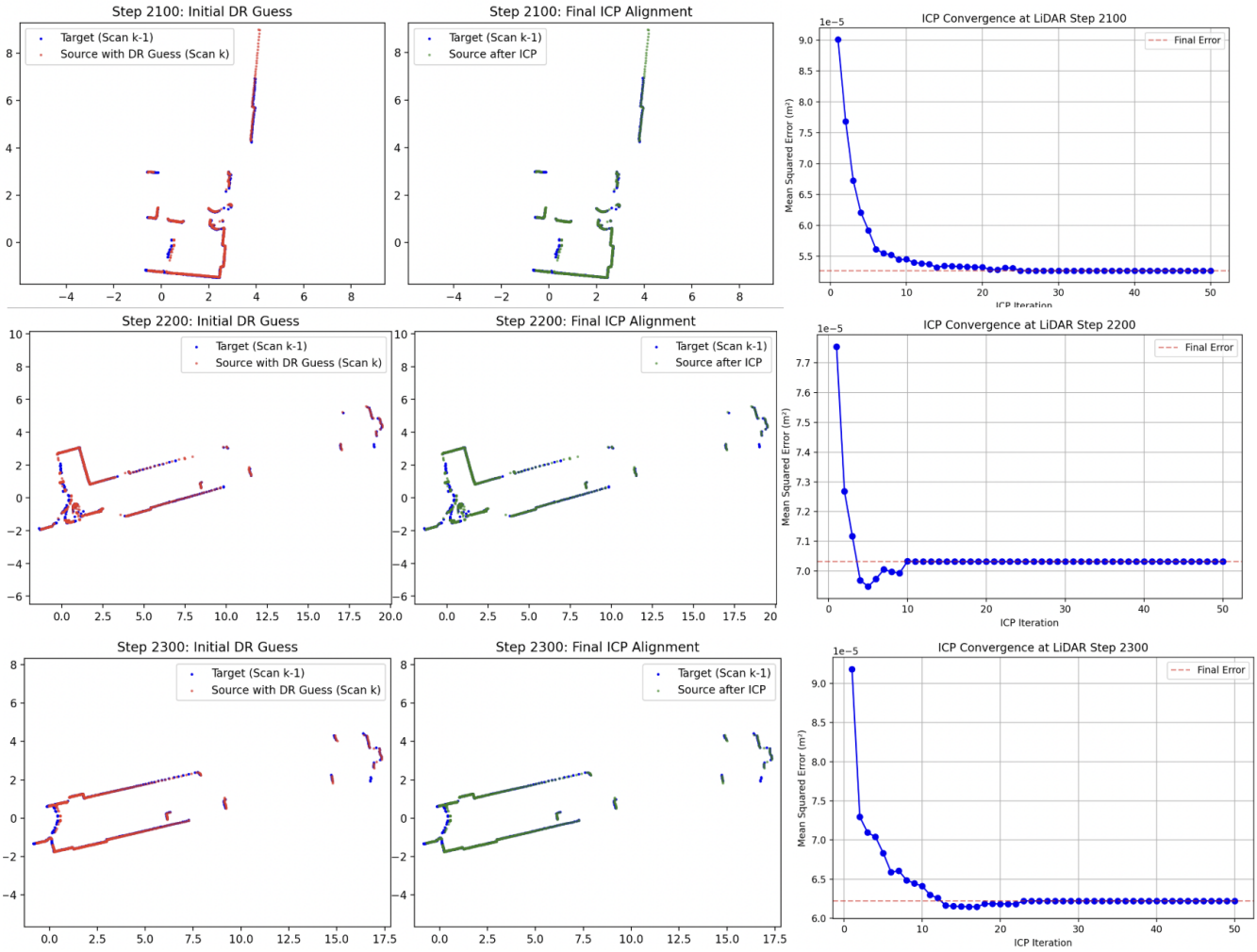


Fig. 10. LiDAR Point clouds before and after running the ICP algorithm for scans k=2100, 2200, 2300 (Left). ICP objective cost across iterations for each frame (right)

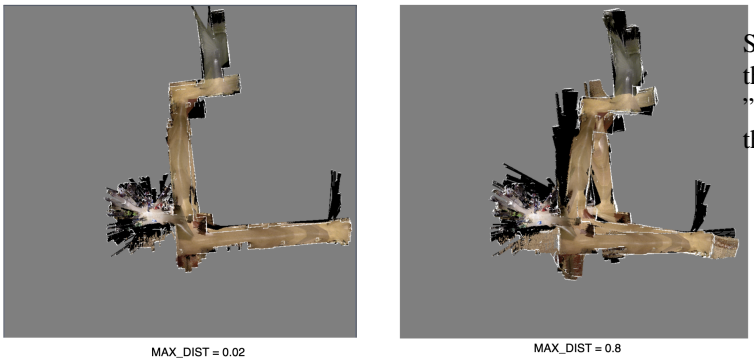


Fig. 11. Texture maps built with ICP trajectories built with different values for MAX_DIST. Allowing ICP to have correspondences between points that have a high euclidean distance leads to the Kabsch Algorithm estimating R and $\mathbf{p}$ based on bad associations, leading to very poor estimation.

- Dataset 21: https://youtu.be/BcJX1bN4RRA

Similarly, in these videos, we can see that the Texture map and the robot's trajectory updates and corrects itself over time. The "legs" of the PGO optimized trajectory are much straighter than before, indicating an improvement.

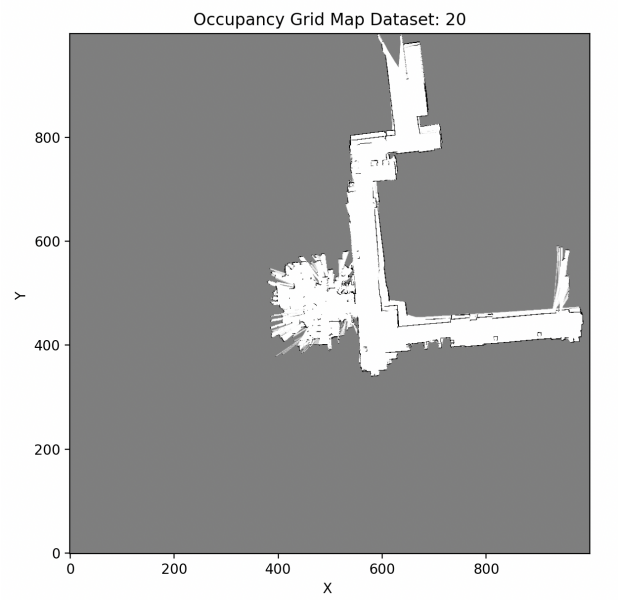


Fig. 12. Occupancy Grid Map of robot trajectory with ICP optimization: Dataset 20

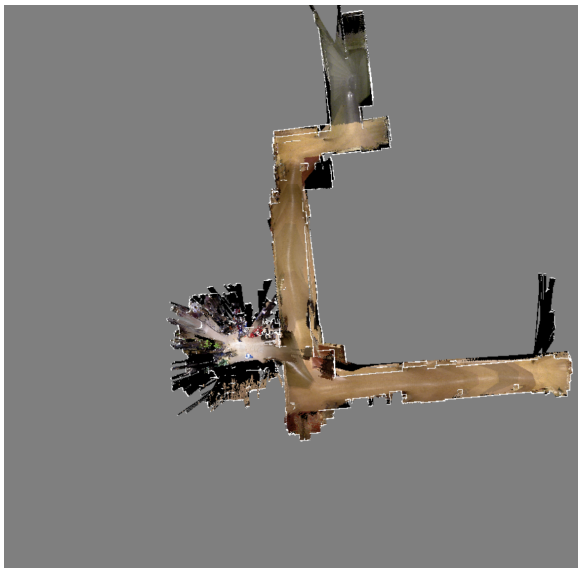


Fig. 13. Texture Map of robot trajectory with ICP optimization: Dataset 20

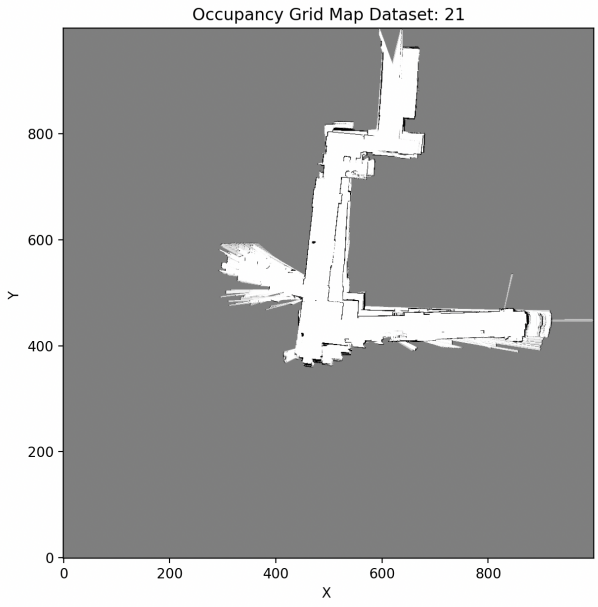


Fig. 14. Occupancy Grid Map of robot trajectory with ICP optimization: Dataset 21

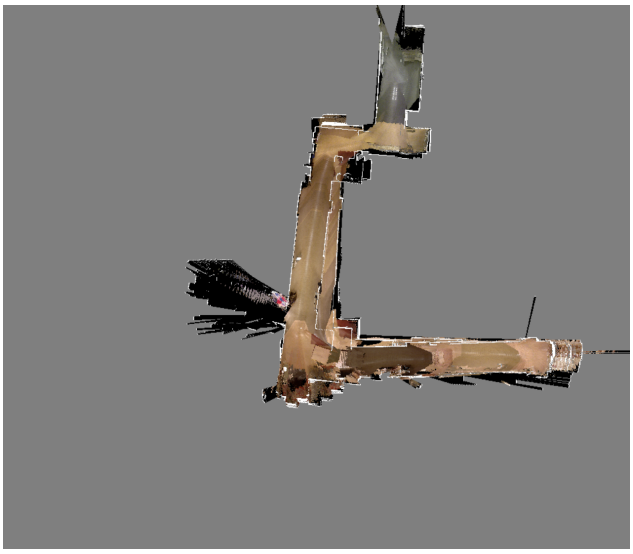


Fig. 15. Texture Map of robot trajectory with ICP optimization: Dataset 21

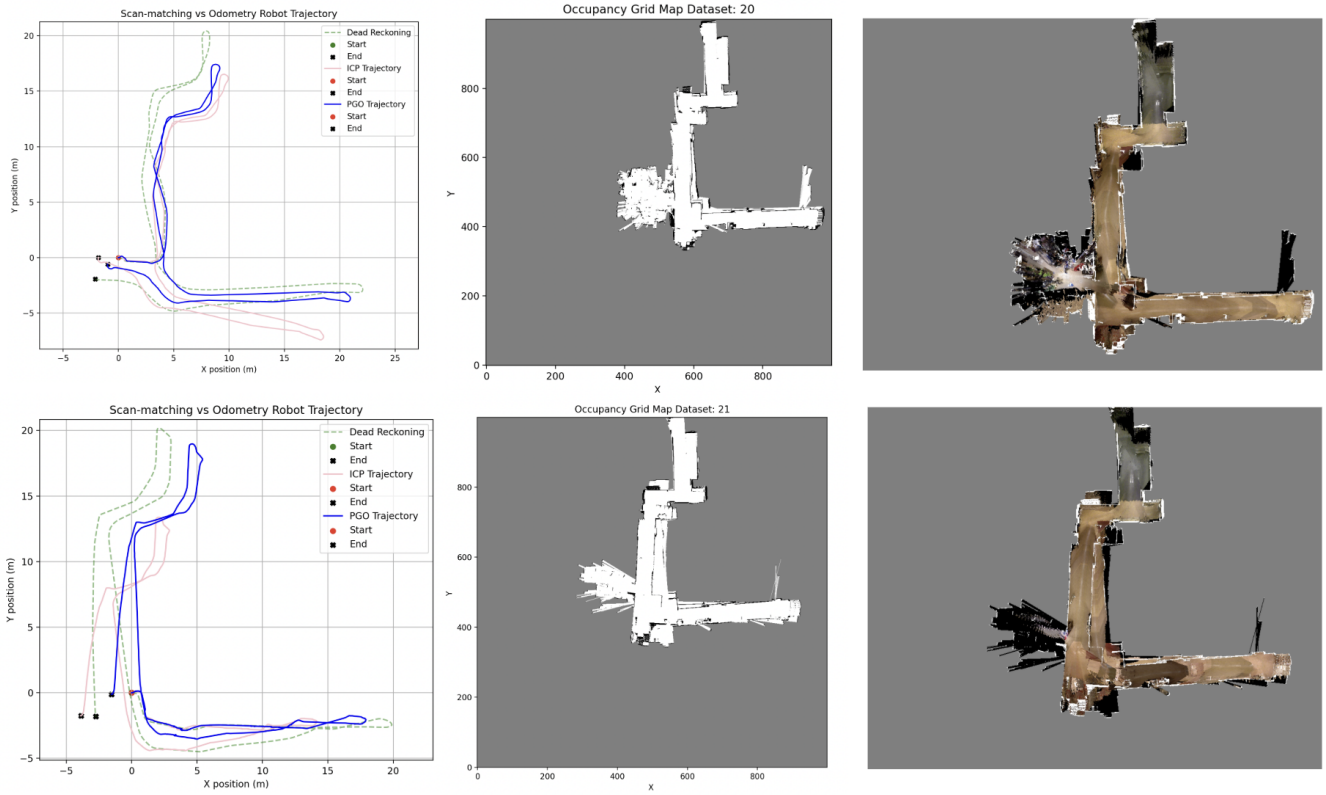


Fig. 16. Pose Graph optimization results, and optimized trajectory plots for Dataset 20 (top) and Dataset 21 (bottom)

REFERENCES

[1] Localization and Odometry from Point Features Lecture https://natanaso.github.io/ece276a/ref/ECE276A_6_LocalizationOdometry.pdf
[2] Factor Graph SLAMhttps://natanaso.github.io/ece276a/ref/ECE276A_5_FactorGraphSLAM.pdf
[3] Motion and Observation Models Lecture: https://natanaso.github.io/ece276a/ref/ECE276A_4_MotionAndObservationModels.pdf
[4] Piazza Questions: https://piazza.com/class/mhr7oh6rr7c3k8/post/174, https://piazza.com/class/mhr7oh6rr7c3k8/post/162, https://piazza.com/class/mhr7oh6rr7c3k8/post/148
[5] Factor Graph GTSAM https://gtsam.org/tutorials/intro.html
[6] CV2 Documentation: https://docs.opencv.org/4.x/db/deb/tutorial_display_image.html, https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html