Project 2: LiDAR-Based SLAM

Justin Volheim
Electrical and Computer Engineering Department UCSD
jvolheim@ucsd.edu

Abstract

This project goes over the steps and methods utilized to accurately preform SLAM utilizing sensor data from encoders, LIDAR, IMU's, and a RGBD camera. To do this we will implement several methods for localization such as encoder/IMU estimation and ICP scan matching. We also create occupancy grids using both lidar and camera approximations. Finally, we optimize the path estimations using factor graph optimization. The methods discussed in this project can be widely applied to localization and mapping projects as they are built on basic principles and therefore easily generalized.

Introduction

In this project, our objective revolves around the ubiquitous challenge of robotic perception and mapping, where we aim to fuse sensor data to accomplish Simultaneous Localization and Mapping (SLAM) on a differential-drive robot. Our primary challenge is to precisely localize the robot within its environment while simultaneously constructing a detailed map using data from encoders, an IMU, a 2-D LiDAR scanner, and an RGBD camera. The significance of accurate localization and mapping extends far beyond robotics, permeating industries ranging from autonomous vehicles to industrial automation. To tackle this multifaceted challenge, our approach integrates various methodologies, including encoder and IMU odometry, LiDAR based point-cloud registration, RGBD image processing and factor graph optimization. These techniques form the backbone of our SLAM framework, facilitating the creation of an accurate and informative map. This methodology is applied to our differential drive robot to accurately illustrates its environment and movements over time, returning various path estimations and occupancy grids shown in this document.

Problem Formulation

The tasked problem in this project is to utilize implement SLAM for the below seen robot as it maneuvers throughout an environment.

To accomplish this task, we utilize various sensors on the robot which are as follows.

Encoders: This robot offers 4 encoders with one on each wheel returning a set of data in the form of [[FR, FL, RR, RL]] with an associated timestamp for each datapoint.

IMU: This IMU offers the standard 6-axis angular velocities and linear accelerations however in this project we will only utilize the yaw angular velocity.

Hokuyo Lidar: This lidar has a range of 30m and a viewing angle of 270 degrees.

This data is in the form of vectors with size 1081, where each index is the distance at an angle. Each of these vectors has an associated timestamp.

Kinect: The Kinect camera data comes in the form of two images one RGB and the other disparity ie.(depth estimates) both with associated timestamps.

The specific aspects of SLAM implemented in this project can be broken down into four categories.

Encoder and IMU odometry: The problem of extracting odometry from the wheel encoder and IMU values will require a simple iterative algorithm. This algorithm utilizes the yaw angle from the IMU and the average motion of the wheels to predict change in coordinates and change in rotation at the next timestep. This result can be utilized to estimate the odometry over all timesteps.

LIDAR Scan Matching: In the problem of Lidar scan matching, we are tasked with estimating the change in pose of our robot between scans and to combine these poses to recreate an overall localization of the robot in the environment. To do this we implement the iterative closest point (ICP) algorithm. This algorithm will allow us to accuracy estimate the pose transformation between two-point clouds taken from the lidar scans. Running this algorithm for all pairwise sequential lidar scans will allow us to recreate the overall odometry of the robot.

Occupancy grid estimation: The problem of creating an occupancy grid in this project will be solved in two alternative ways. The first will be to utilize the lidar scans and odometry to get estimates of points in our world frame that contain objects and points that don't. Using this we can update the likelihood of that location being occupied resulting in a log-odds occupancy grid. The second method is to utilize the Kinect cameras RGBD values with the robots odometry to approximate where the floor the robot is traversing exists. Method will require us to get 3d point clouds from the Kinect and threshold those clouds leaving only the values along the plane representing the floor. This should result in a full color image of the occupancy grid.

Pose graph optimization: The problem of pose graph optimization is simply a method we are using to improve our overall results by removing compounding errors from our estimation. The compounding errors arise when for example calculating the odometry using scan matching we could get one scan that is 180 degrees off from the actual rotation. This would result in all transformation matrices from that point forward being rotated by 180 degrees just due to a single error. We will implement interval loop closer and an optimizer to correct for such errors.

Technical Approach

Data Preprocessing: Throughout the technical approach each of the problems relies on the sensor data. This sensor data requires some preprocessing to be usable in the following tasks.

The first problem with the data is that the interval at which it records and saves data is not constant across all the sensors. Therefore, the first task is to utilize the timestamps of each sensor to associate the data into one sample set. The method we will use is to simply find the closest sample is each of the sensor data sets using the encoder samples as the base. Meaning we will assign IMU, lidar, and RGBD samples to each encoder sample via the timesteps.

The second issue with the data is that the sensors are placed at various locations on the body of the robot and therefore need to all be transformed to the body frame. Specifically, the following transformation matrix is used to transform the lidar datapoints to the body frame whereas the

IMU is used as the origin and the RGBD transformation will be covering in the texture mapping section respectively. In order to put the lidar coordinates into the body frame we first have to convert them from the semi-cylindrical coordinates they are to x, y coordinates. This can be done using the following equation

$$\begin{bmatrix} x \\ y \end{bmatrix} = distance_i * \begin{bmatrix} \cos(\theta_i) \\ \sin(\theta_i) \end{bmatrix}$$

Where the distance is the ith value of the vector and the ith theta is the ith value from the vector of angles [-135, ….,135] equally spaced and of the same size as the distance vector. We also remove data points whose distances were outside the range for the lidar i.e. 0.1<distance<30. Finally, the transformation matrix for lidar to body for each point is as follows.

$$body\ T\ lidar = \begin{bmatrix} 1 & 0 & 0 & 0.29833 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.51435 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying this to all the datapoints results in the correct body frame coordinates of the lidar datapoints.

Encoder and IMU odometry:

To implement encoder and IMU odometry we first need to extract the following values from the matching time step encoder and IMU sensor readings.

$$\boldsymbol{Tau_t = time\ between\ steps}$$

$$\boldsymbol{w_t = imu\ yaw\ angle}$$

$$\boldsymbol{v_t = \frac{0.0022(FR + FL + RR + RL)_t}{4 * Tau_t}}$$

Now that we have the correct datapoints we can implement an iterative odometry algorithm to calculate the changing pose of the robot over time shown here.

$$\boldsymbol{given\ p_t = [x_t, y_t, \theta_t]}$$

$$\boldsymbol{initialized\ with\ p_o = [0, 0, 0]}$$

We will loop over all vales of t using the following update equation.

$$\boldsymbol{p_{t+1} = p_t + tau_t * \begin{bmatrix} v_t * cos(\theta_t) \\ v_t * sin(\theta_t) \\ w_t \end{bmatrix}}$$

$$\boldsymbol{Tx_t = \begin{bmatrix} R_t & p_t \\ 0^T & 1 \end{bmatrix}}$$

This algorithm results in an estimated pose of the robot for all timesteps and will be used to illustrate its movement along a path in the results section.

Scan Matching:

To implement scan matching we first need to create a method to do Point-cloud registration. The method used here is the iterative closest point (ICP) algorithm. The algorithm starts by inputting an initial guess of the pose between the two point-clouds.

$$\boldsymbol{T_o = \begin{bmatrix} R_o & p_o \\ 0^T & 1 \end{bmatrix}}$$

Without this initialization the algorithm will default to an identity rotation matrix and the position being the difference between the means of each point-cloud.

$$\boldsymbol{p_o = \frac{1}{d_z}\sum z_i \ - \frac{1}{d_m}\sum m_i}$$

Where z represents the source point-cloud and m represents the target point-cloud with d being their respective number of points. Using either initialization the algorithm starts by transforming the source point cloud and finding the closest associated points in the target point-cloud as follows.

$$R = R_o \quad P = p_o$$

Iterative algorithms start-

$$z_{transf} = R \cdot z + P$$

To associate the points between the two clouds we utilize the following equation.

$$indexes = argmin\left(\left|\left|z_{transf} - m\right|\right|_2^2\right)$$

As these point clouds could be of different sizes, we will only take the indexes of the smaller of the two point-clouds. Using these indexes, we can create a new set of point-clouds.

$If\ d\_z > d\_m:$
$$z_{new} = \{z \mid z_i \in z, i \in indexes\}$$
$else$:
$$m_{new} = \{m \mid m_i \in m, i \in indexes\}$$

$$with\ order\ preserved\ for\ both$$

This if statement basically says that the smaller of the two point-clouds will have all its points while the larger will be shrunk only to the points that associate to the other. With these new associated points, we preform kabsch algorithm to estimate the transformation matrix between them. This algorithm works first by finding the means of each point cloud and centering them about the origin.

$$\boldsymbol{z_{mean} = \frac{1}{d_z}\sum z_{new_i}}$$

$$\boldsymbol{m_{mean} = \frac{1}{d_m}\sum m_{new_i}}$$

$$\boldsymbol{z_{centered} = z_{new} - z_{mean}}$$

$$\boldsymbol{m_{centered} = m_{new} - m_{mean}}$$

Using these new point centered point-clouds we can construct the following Q matrix and calculate it's SVD.

$$\boldsymbol{Q = m_{centered} \cdot z_{centered}^T}$$

$$\boldsymbol{SVD(Q) = U \cdot \Lambda \cdot V^T}$$

These SVD matrices allow us to estimate the rotation and translation between the two point-clouds using the following equations.

$$\boldsymbol{A = \begin{bmatrix} 1 & & \\ & \cdots & \\ & & det(U \cdot V^T) \end{bmatrix}}$$

$$\boldsymbol{R = U \cdot A \cdot V^T}$$

$$\boldsymbol{P = m_{mean} - R \cdot z\_mean}$$

With these new estimates of the rotation and translation we can jump back up to the start of the ICP iterative algorithm and run it until the algorithm converges.
The convergence or stopping condition for the algorithm is a maximum number of iterations of around 1000 or when the change in loss between iterations is less than 1e^-6. The loss for each iteration is as follows.

$$\boldsymbol{loss = RMSE(z\_new - m\_new)}$$

We will test this algorithm in the ICP warmup section by matching point-clouds in the shape of drills and water jugs.
With this the ICP algorithm is complete, and we simply need to show how it is applied to our lidar scans to preform accurate pose estimation.
We initialize the ICP algorithm as follows where Tx_t represents the odometry transformation matrices from the encoder/IMU section.

Loop start-

$$\boldsymbol{R_o = r_z\left(yaw(Tx_t(R)) - yaw(Tx_{t-1}(R))\right)}$$

The $yaw(Tx_t(R))$ quantity represents the yaw angle from the rotation matrix in Tx_t and the $r_z(\theta)$ function represents the z axis rotation matrix given an angle theta.

$$\boldsymbol{P_o = Tx_t(P) - Tx_{t-1}(P)}$$

The value $Tx_t(P)$ represents the position portion of the matrix Tx_t

$$\boldsymbol{T_o} = \begin{bmatrix} \boldsymbol{R_o} & \boldsymbol{p_o} \\ \boldsymbol{0^T} & \boldsymbol{1} \end{bmatrix}$$

With this initialization we can set the source $z = scan_t$ and target $m = scan_{t-1}$ resulting in the following function.

$$\boldsymbol{T_{temp}} = \boldsymbol{icp}(\boldsymbol{z}, \boldsymbol{m}, \boldsymbol{T_o})$$

With this new translation between each of the lidar scans we can calculate the translation from the initial position using the following equation

$$\boldsymbol{T_t} = \boldsymbol{T_{t-1}} \cdot \boldsymbol{T_{temp}}$$

Loop end-

Now utilizing the above algorithm, we can loop over all lidar scans and create an accurate estimate of the robots odometry and pose at any timestep.

Occupancy grid:

The occupancy grid essentially uses the lidar to determine empty and non-empty locations along the robot's trajectory. It utilizes these values to update the log odds of there being an object in that location.

For each lidar scan utilize the robots matching transformation matrix from body to world to get the position of the robot and to transform the body frame lidar scans to the world frame.

$$WorldlidarCords = T_t * (BodylidarCords)$$

$$P_x,\ P_y = T_t[0,3]\ ,\ T_t[1,3]$$

This set of lidar points now consists of all the locations at this timestep that the lidar sees an object at. The information these points offer can also be expanded because by recognizing a point at some distance away it is also saying that the points between itself and that point are empty. Therefore, we can find all the points where the lidar says no object exists using bresenham2D algorithm which will return all the grid points in a straight line between two points.
Now that we have 2 sets of points one where an object was detected another where no object should be we can update the log odds of our occupancy grid.
For each element in the object found set add log(4) to its corresponding coordinate in the Occupancy grid image. This value is chosen to represent an 80% certainty in the lidar values.

$$image[x, y] += \log(4)$$

For each element in the no object found set we will subtract log(4).

$$image[x, y] -= \log(4)$$

Now that we have an updated log odds for one lidar scan we will restrict the image to a max and min of +-20log (4) to prevent over fitting of the data.
Following this method and looping over all lidar scans we can create a complete occupancy grid.

Texture mapping:

Creating a transformation of image pixels to the world frame in order to make an alternative to the previous log odds occupancy grid requires us to utilize the following rotations and translation matrices. The first three shown here are R and p which represent the rotation and translation from the body frame to the camera frame.

$$R = R_x(0)R_y(0.46)R_z(0.021)$$

$$p = \begin{bmatrix} 0.18 \\ 0.005 \\ 0.36 \end{bmatrix}$$

The following two matrices are the rotation from the camera frame to the optical frame and the cameras characteristic matrix.

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

$$K = \begin{bmatrix} 585.05 & 0 & 242.94 \\ 0 & 585.05 & 315.84 \\ 0 & 0 & 1 \end{bmatrix}$$

Computing the optical Z_o coordinates require you to use the following equations to convert the disparity image into an image containing the Z_o i.e. depth at the u, v coordinate.

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

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

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

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

Using these equations for all i, j in the disparity image will result in a new array with the depth z_o at each pixel. With this depth image i.e.(z_o) and the previously defined matrixes we know all the values in the following two equations except the body frame coordinates.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \frac{1}{Z_o} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_o \\ Y_o \\ Z_o \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_o \\ Y_o \\ Z_o \\ 1 \end{bmatrix} = \begin{bmatrix} oRr \cdot R^T & -oRr \cdot R^T p \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X_b \\ Y_b \\ Z_b \\ 1 \end{bmatrix}$$

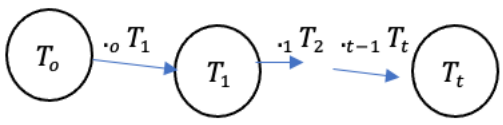
These equations however do not lend themselves to being converted into an expression for the body frame coordinates easily due non-invertibility of the 3x4 matrix and the requirements to separate z_o out however this can be easily accomplished in code by reversing the equations piece by piece. It is also important to filter out any points whose depth coordinate is outside the range of the RGBD camera i.e. 0.05<depth<5

After computing all the coordinates from the image, we threshold the z coordinates removing any point above -0.1 and flatten the remaining data by setting their z value to zero. This results in an image of the floor in front of the robot in the body frame. Using this method and the transformation matrixes in the previous sections we transform these points to the world frame for each image creating a visual representation of the floor along the path of the robot. This resultant image should resemble a decent occupancy grid.

Pose graph optimization and loop closure:

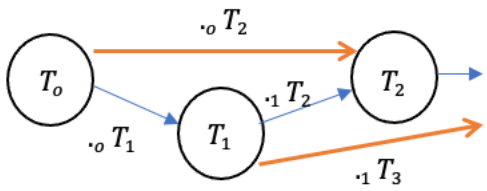
To implement the pose graph optimization, we will be utilizing the gtsam library in python. This library gives the tools required to implement the graph with nodes, edges, and optimization functions. In general, however the construction of the graph is as follows.

When constructing our graph there are two main components we will be using. The first is the nodes. For our graph we will be using the pose estimates at each time step for our nodes. The second are the edges. Four our graph we will be using the results of the ICP algorithm for the transformation between each time step as the edges. An example can be seen here



This however when optimized wouldn't change anything as all the nodes are

linearly connected with no loops or shortcuts.
To fix this we interval loop closures. An interval loop closure is simply an added edge between nodes at a set interval if the transformation is possible. If the set interval was 1 then the following would be an example of the first loop closure in the iteration.

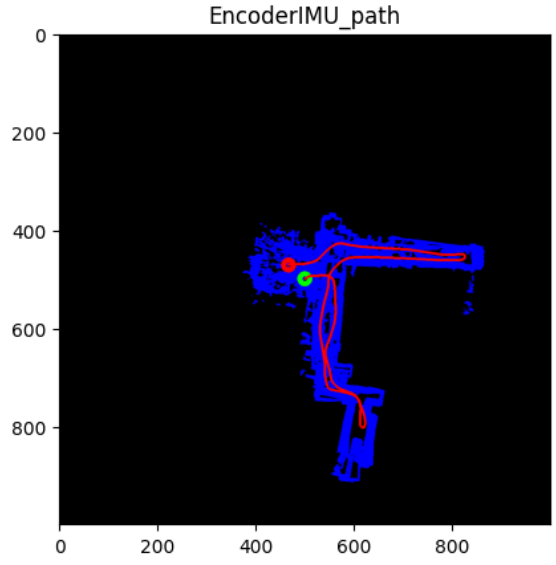


Where each of the new transformation edges is calculated using the ICP algorithm between each of the lidar scans. This allows for a node or nodes to be corrected if the overall graph improves. For our graph we used an interval of 20 and constrained the Transformations matrices by 0.04 max rotation and 0.2 max translation.

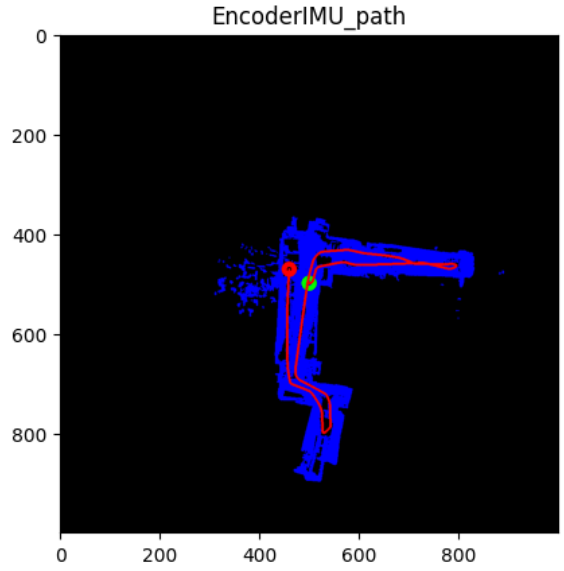
Results

Encoder and IMU odometry:
These two plots show the path and lidar scans for datasets 20 and 21 respectively

(Dataset 20)

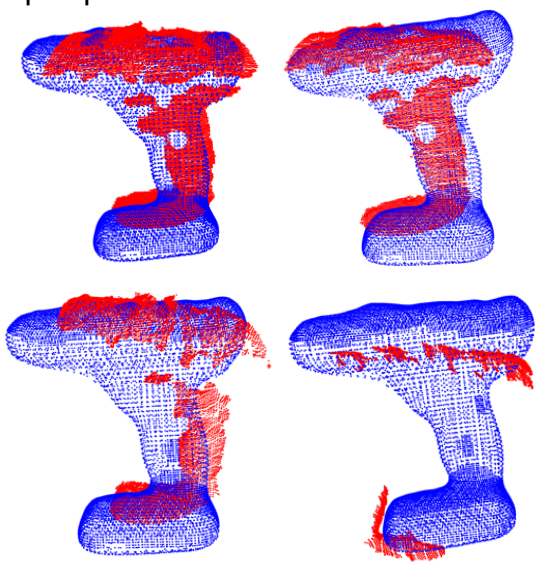


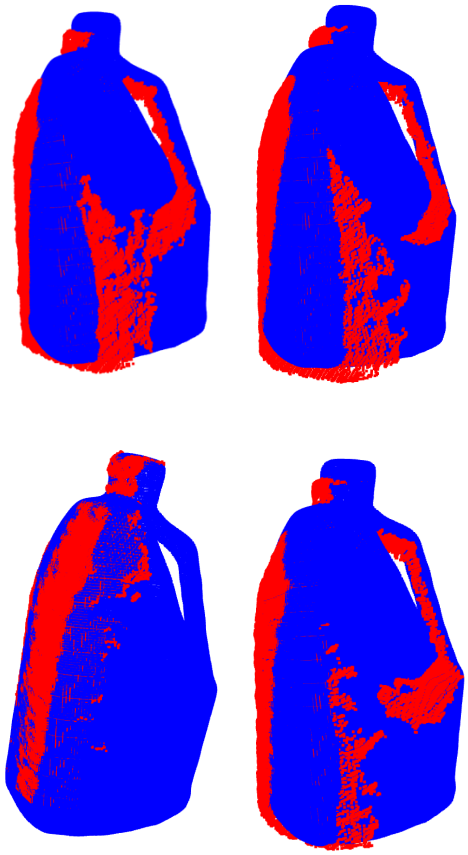
(Dataset 21)



Analysis: Based on these images we can see that the robot odometry calculated based on IMU and encoder values is very accurate making it a good method for localization.

ICP warmup:
These 8 images show the results from testing the ICP algorithm on drill and liquid point-clouds.

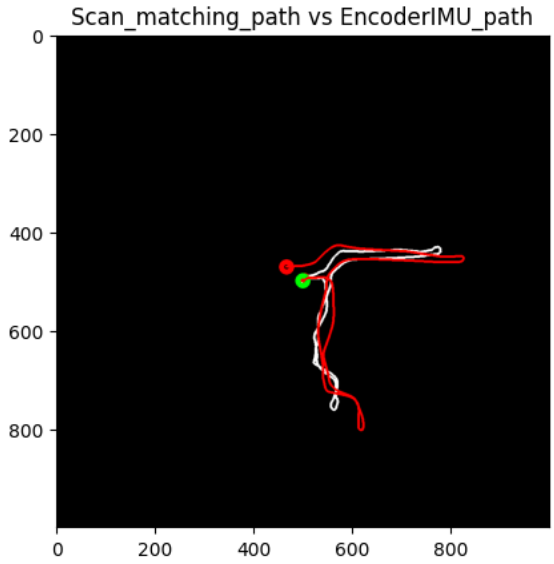




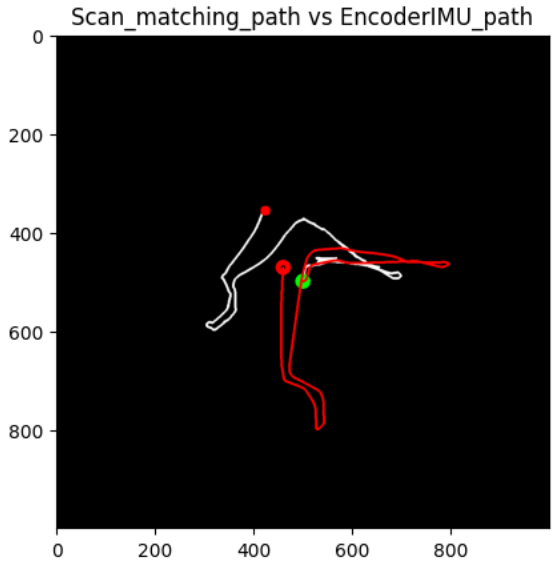
Analysis: Based on these images we can see that the ICP algorithm does a good job at predicting the transformation matrices for the Liquid containers however it struggles with the drill. These results however are sufficient for our purposes. We could improve these results by increasing the complexity of our ICP algorithm.

Scan Matching:
The following two graphs show the results of the lidar scan matching in white vs the IMU odometry in red of the previous section.

(Dataset 20)

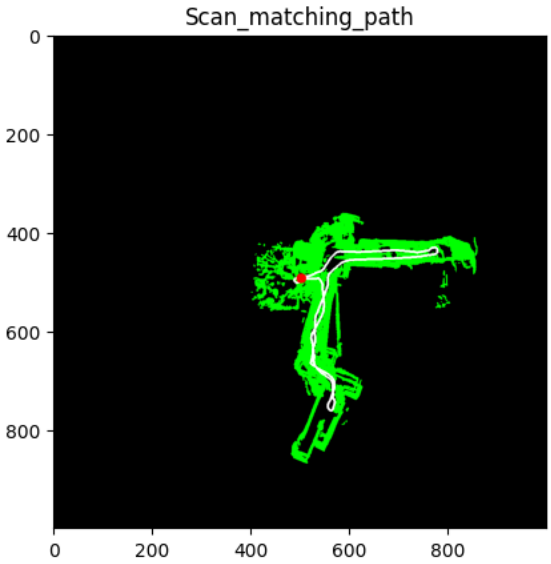


(Dataset 21)

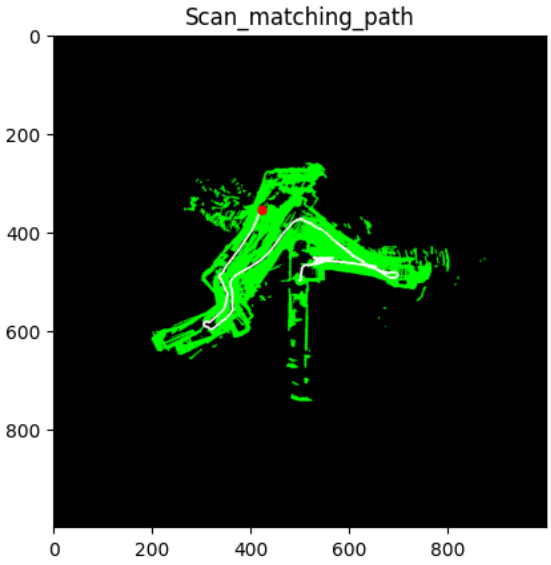


The following two graphs show the updated lidar scans along the with the scan matching trajectories.

(Dataset 20)



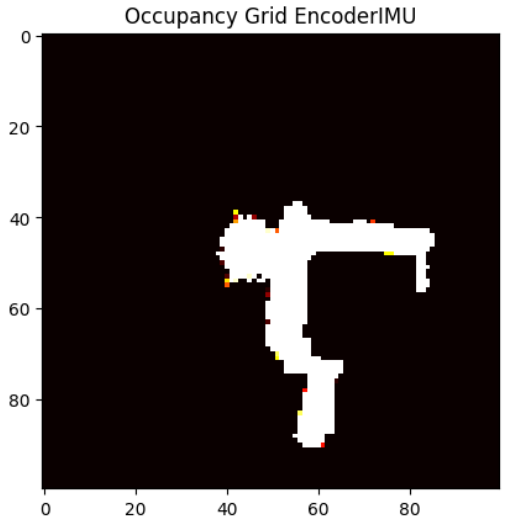
(Dataset 21)



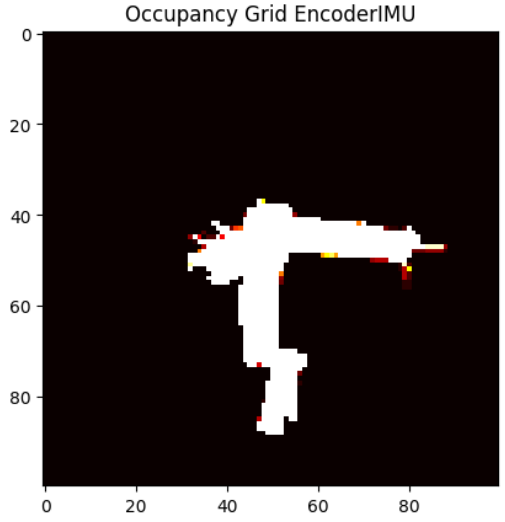
Analysis: Based on the previous 4 images we can see that our Scan matching algorithm preforms well on Dataset 20 with accurate estimations of the odometry. It does however seem to underestimate the distance traveled when comparing it to the Previous odometry model. This is likely due to the long hallways. For dataset 21 our method does not perform as well. This is likely due to the mismatch in lidar scans to encoder readings. Ie (when matching the lidar scans the timestamps were considerably off so we ended up reusing a previous lidar scan around 400 times for dataset 21.) All together the shape of the environment can be gleaned from both datasets using this method.

Occupancy and texture mapping:

These two plots show the occupancy grids for dataset 20 and 21 respectively
(Dataset 20)

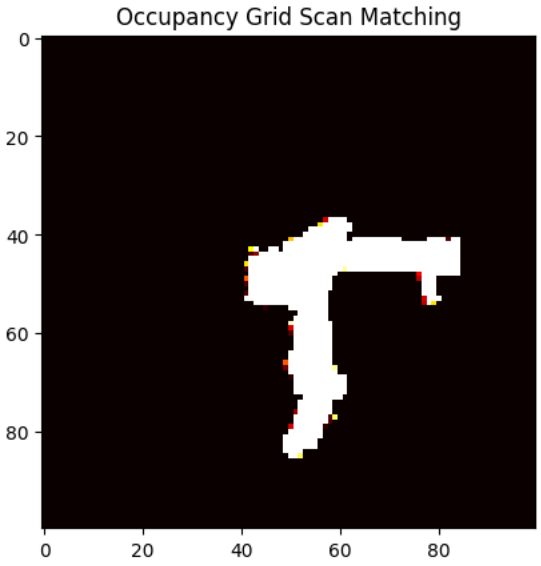


(Dataset 21)

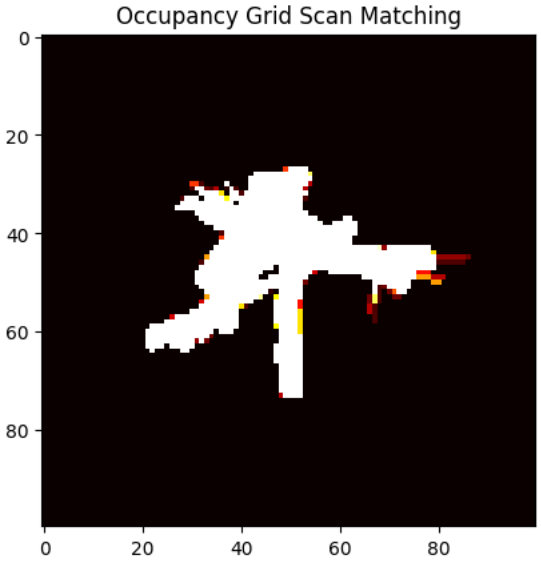


The following two graphs show the occupancy grid for the scan matched lidar grids.

(Dataset 20)

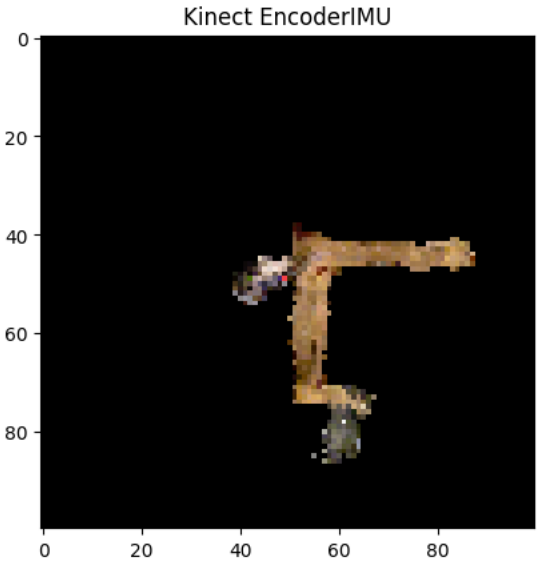


(Dataset 21)

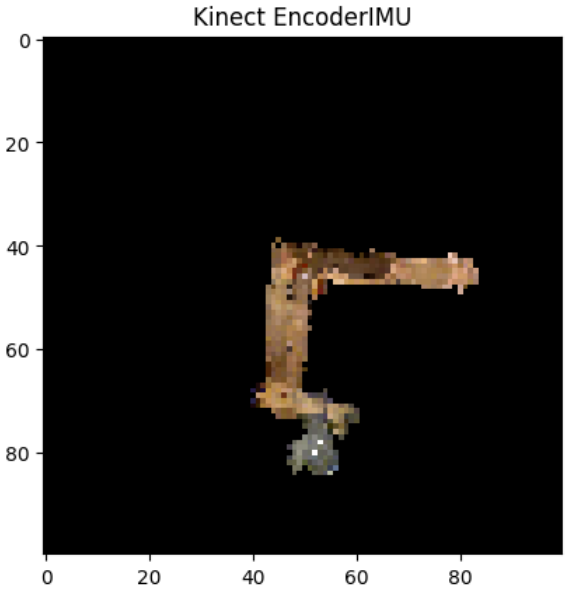


The following two graphs show the texture mapping for each of the two datasets with encoder odometry.

(Dataset 20)

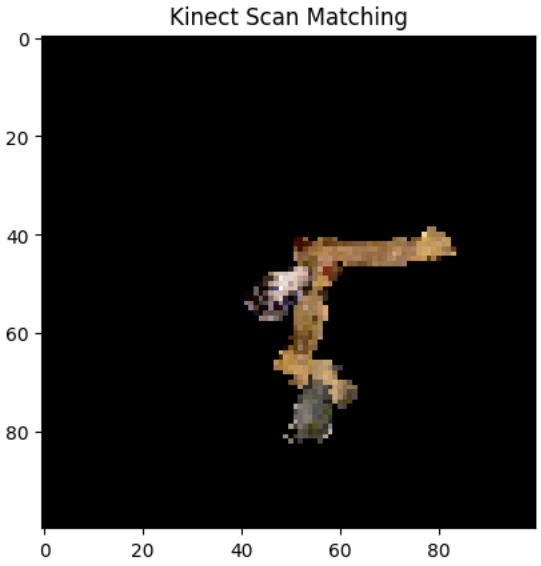


(Dataset 21)

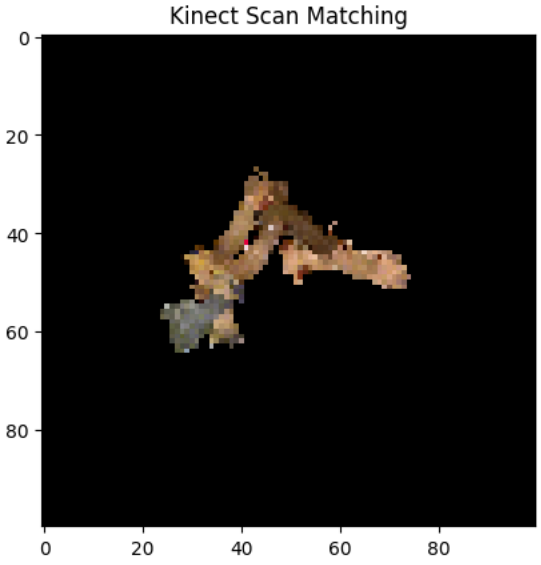


following two graphs show the texture mapping of the two datasets with scan matching odometry.

(Dataset 20)



(Dataset 21)

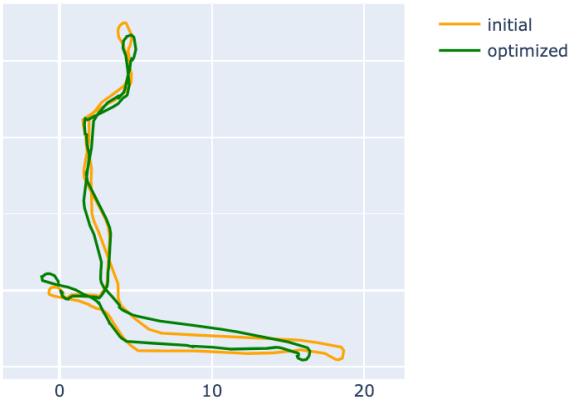


Analysis: Based on these 4 images we can see that the log odds occupancy grid tends to contain more noise however the texture mapping method is much more limited in range. They both however do a sufficient job at mapping the environment.

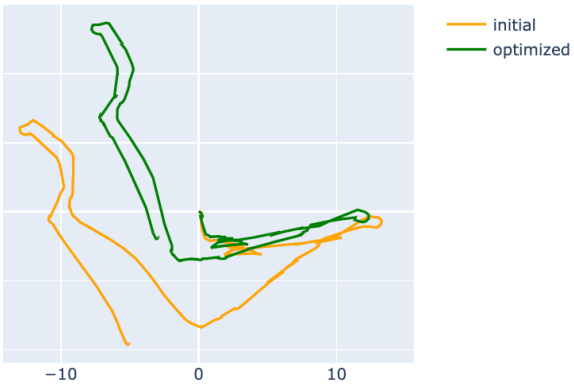
Pose graph optimization and loop closure:

The following two graphs show the initial path vs the optimized graph for the scan matched odometry data.

(Dataset 20)

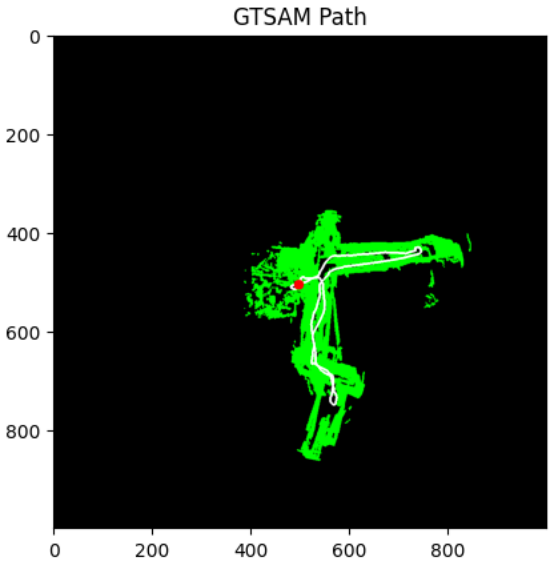


(Dataset 21)

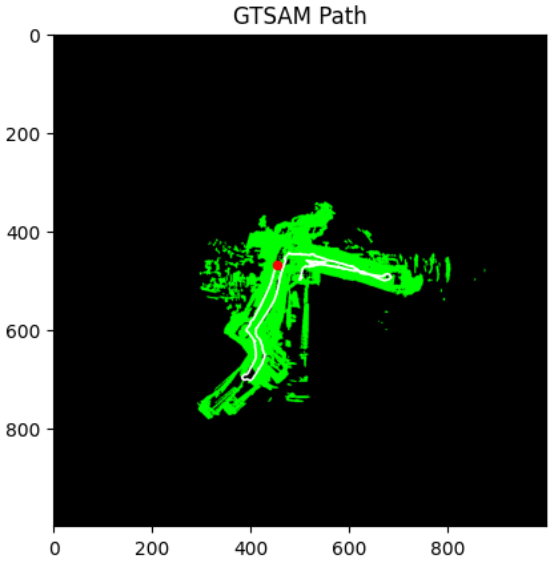


The flowing two images show the optimized odometry and corresponding lidar scans.

(Dataset 20)

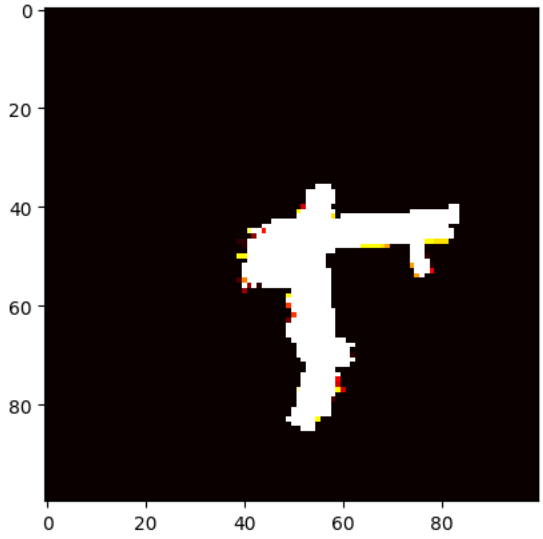


(Dataset 21)

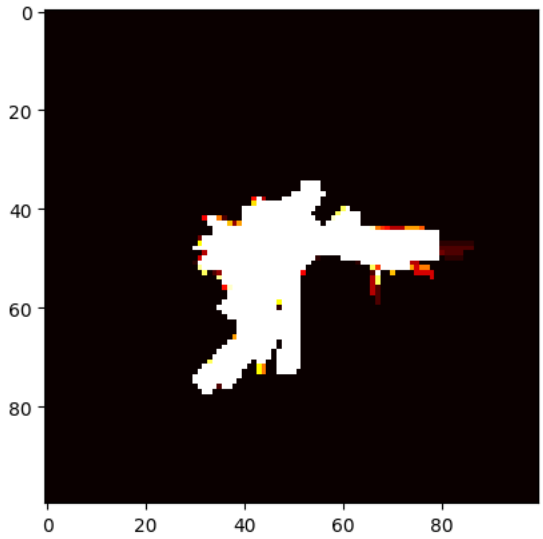


The following 4 images show the occupancy grids using log odds and texture mapping of the optimized odometry.

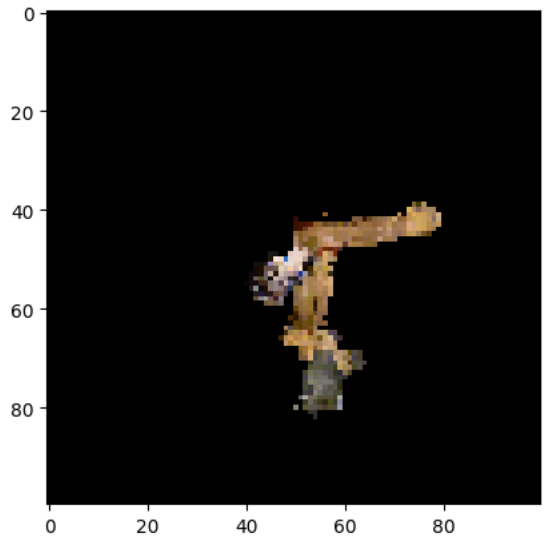
(Dataset 20)



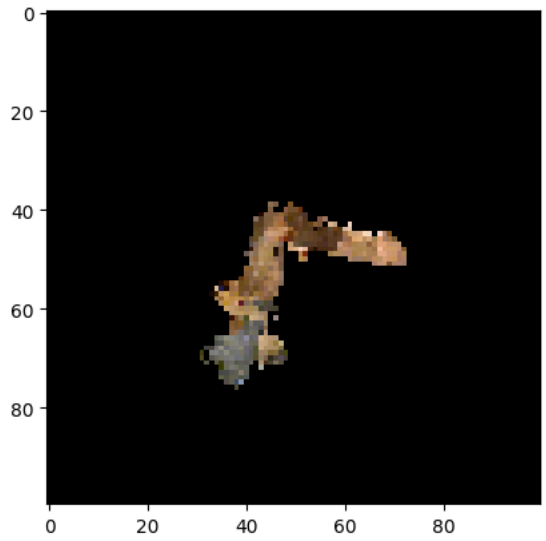
(Dataset 21)



(Dataset 20)



(Dataset 21)



Analysis: based on the above images it is clear that the graph optimization did a good job at correcting some of the compounding errors in the odometry estimation. This is especially clear for the results with dataset 21. Dataset 20. Was not affected as much due to its better initial estimation. If we wanted to further improve the optimization on dataset 20, we would need to use a smaller interval for the loop closure however the current number was chosen for its benefits across both datasets.

Conclusion

Based on the overall results of this project we can see that the ICP scan matching and Encoder/IMU odometry estimates are an affective choice when preforming SLAM on a robot. Although the Scan matching odometry data was not as good as the encoder odometry this can be further improved and does not discount the value of the techniques demonstrated in this project.