Using a Particle Filter to Perform SLAM with Camera, Odometery, and Lidar Data from THOR Robot

William Argus[1]
Department of Electrical and Computer Engineering, UCSD
wargus@eng.ucsd.edu

I. INTRODUCTION

Simultaneous localization and mapping is a problem that involves determining the state and environment of a robot at the same time. This presents a challenge because typically mapping the environment requires knowledge of the state and mapping the state requires knowledge of the environment. The approach employed by this author in this paper is to implement a particle filter, which essentially means that rather than having a definite state that the robot is in, the state is a combination of many different states, each with their own probability of being the correct state.

The steps to achieve the desired result were to first establish the ability to map the environment from the initial state accurately. The next step taken was using the known odomoetry measurements from the robot, to map out its environment, and then splitting the state into many particles, which were evaluated with the best probability particle selected as the state at any given time. The evaluation of particles is an important part of this technique, as without a very good way to determine which particle is indeed the most likely, and therefore the best, the whole solution falls apart. This is discussed in the Technical Approach section of this paper. Results are presented and discussed at the end.

II. PROBLEM FORMULATION

A. Localization Problem

In order to solve the localization problem in this project, the concept of a Markov assumptions in robotics must be discussed. This refers to the concept that the state of a robot x_{t+1} depends only on the the previous input u_t and state x_t. Hence the motion model of a robot with Markov assumptions can be described in equation 1, where w_t is noise.

$$x_{t+1} = f(x_t, u_t, w_t) \tag{1}$$

This means that the given a map and a sequence of control inputs, the robot state can be determined as long as there is a map and history of the control inputs. It will be discussed in the following sections, however, that this problem is formulated such that there is no map of the environment, so this problem is not formulated as a simple problem of solely localizing a robot under Markov assumptions.

B. Mapping Problem

There is no available map of the robot's environment. Therefore this means that mapping is also included as a part of this problem. In order to map the environment, the probability density function of a cell on the map can be determined by tracking the accumulating cell log odds, determined as seen in equation 2.

$$\lambda(m_i|Z_{0:t}, x_{0:t}) = \lambda(m_i) + \sum_{s=0}^{t} log[g_h(z_s|m_i, x_s)] \tag{2}$$

This equation basically states that the current log odds of a given cell are the sum of the current log odds and the log odds added during all past time instances. In order to generate a map, in general, it is required to know the location of the robot whose sensors are giving the measurements used to create the map. Hence, because location of the robot is not known, this is not a simple mapping problem.

C. SLAM Problem

As discussed previously, neither the robots location, nor the map of the environment is known, so the problem is formulated as a SLAM, (simultaneous localization and mapping) problem. The general solution to SLAM uses the fact that since the robot is assumed to be Markov, the observation and motion models can be treated as separate probability density functions, seen in Equation 3.

$$p(x_{0:T}, m, z_{0:T}, u_{0:T-1}) = p_{0|0}(X_0, m) \prod_{t=0}^{T} p_h(z_T|x_t, m) \prod_{t=1}^{T} p_f(x_t|x_{t-1}, u_{t-1}) \tag{3}$$

Here, x is the robot state, m is the map of the environment, z is the observations, and u is the control input. The inputs to the SLAM problem are the robot's observations, z, and the control input, sometimes put through a motion model and viewed as odometry, is u. The outputs are the robots state at all time instances, x, and the map of the environment, m

The goal of this SLAM problem is to solve the MLE formulation of equation 3 for the outputs, seen in equation 4.

$$max_{x_{0:T},m} \sum_{t=0}^{T} log[p_h(z_T|x_t, m)] + \sum_{t=1}^{T} log[p_f(x_t|x_{t-1}, u_{t-1})] \tag{4}$$

III. Technical Approach

This section details the approach used to solve the problems listed in the problem formulations. It will describe first the technical approach and supporting mathematical equations used to solve this problem and then the specific approach and techniques used within the code to get the desired result.

A. Mapping

1) Mathematical Approach to Mapping: In order to use the provided lidar data, it was necessary to use linear transformations to move the data from the reference frame of the lidar sensor to the world reference frame, as that was the frame the robot state and map were stored in. The first step in this was to take the lidar data, given as distances measured from the sensor at discrete angles, and change it from polar coordinates to cartesian coordinates, as the rest of the project was done in cartestian coordinates. This was accomplished with equations 5 and 6.

$$x = r\cos(\phi) \tag{5}$$

$$y = r\sin(\phi) \tag{6}$$

With the lidar scans now represented as points in cartesian coordinates, the next step was to transform from the lidar frame of reference to the head frame of reference. The equation for a transformation matrix is seen below, in equation 7, for transforming from a reference frame B to a reference frame A.

$$\begin{bmatrix} S_A \\ 1 \end{bmatrix} = \begin{bmatrix} R & p \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} S_B \\ 1 \end{bmatrix} \tag{7}$$

Where S_A is in this case, the x, y, and z coordinates in reference frame A, S_B is the x, y, and z coordinates in reference frame B, R is the rotation matrix, and p is the translation matrix; in other words, the amount in each direction that reference frame B is offset from reference frame A after they are aligned via rotation. In the case of the lidar to head transformation, the lidar sensor was $0.15meters$ above the head, so p can be described below.

$$\begin{bmatrix} 0 \\ 0 \\ 0.15 \end{bmatrix} \tag{8}$$

This makes the lidar to head transformation matrix the matrix seen below, in equation 9, where R is the identity matrix, there is no rotation between the reference frames. The exact math supporting this will be clear when calculating the transformation from head to body below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0.15 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

The next step is to find the transformation that can be used to transform the lidar coordinates from the head reference frame to the body reference frame. To do this, a rotation matrix must be used, as the head rotates relative to the body in the neck angle (yaw), and head angle (pitch). The rotation matrix equation is shown below, in equation 10.

$$R = R_z(\psi)R_y(\theta)R_x(\phi) =$$

$$R_z(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & -\cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \tag{11}$$

$$R_y(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \tag{12}$$

By setting the pitch, or head angle equal to ϕ and the yaw, or neck angle equal to θ, the rotation matrix can be calculated and combined with the translation matrix p, shown below, to get a transformation matrix. It should be noted that since the yaw and pitch angles of the head are subject to change, there is not an explicit transformation to go from head reference frame to body reference frame, as it must be calculated again and again at every time step; therefore no distinct head to body transformation matrix is shown in this report.

$$p = \begin{bmatrix} 0 \\ 0 \\ 0.33 \end{bmatrix} \tag{13}$$

The final transformation required is the body to world transformation. This transformation depends on the robot's state in the world frame and must be calculated at every time-step. therefore no explicit matrix for this transformation. The transformation is calculated as follows.

$$p = \begin{bmatrix} x \\ y \\ 0.93 \end{bmatrix} \tag{14}$$

And $R = R_z(\psi)$, where ψ equals the rotation angle θ in the robot's state

Once the lidar scans were converted into the world reference frame, the next step was to map the scans in order to update the log odds map with the free and occupied space from the current scan. This was achieved using the openCV library function drawContours. The specifics of how this function worked will be discussed in the next section. Once the log odds map is updated, the mapping step concluded. It should be noted that this mapping procedure will be performed for every lidar scan, once the prediction step has

been performed, and an update step is also required after the mapping step. It is these two steps that account for the change in state of the robot, as the mapping step merely updates the log odds map with the current lidar scan from the current state. The prediction and update steps will be discussed later in this report.

2) Python Implementation of Mapping: This section will first detail all the functions required for mapping the lidar scans in the Python program. Once this has concluded, there will be a short description of the main function required to perform mapping. Additionally, the code is heavily commented, and the Readme file included is very comprehensive, so the hope of this author is that the code will not be hard to understand.

The first function, lidar2cartesian, is passed a time index. It then creates an array, $angles$ corresponding to the angle of each lidar distance measurement. It then creates an array, $ranges$, which pulls all of the lidar scan data from the imported lidar data. The angles in $angles$ correspond, cell to cell, with the distance measurements in $ranges$. $Ranges$ is then searched through and all indices with acceptable distance measurements (not less than 0.1 meters, which implies hitting the robot, and not more than 30 meters, which is the limit of the sensor, so any measurement greater than 30 meters must be erroneous). These indices are then the only ones kept in the arrays $ranges$ and $angles$. Next arrays of the x and y Cartesian coordinates of these scans are created by applying $ranges$ and $angles$ to equations 5 and 6. The x and y coordinates are then put into an array together, with each x-y pair sharing a column. A row of zeros and a row of 1's below that are included to the bottom of the array in order to aid in the transformation to the head reference frame.

The second function, lidar2head takes the lidar state as input. Since the transformation matrix is the same for all time idices to go from lidar frame to head frame (discussed in previous section), it is written out in this function, and then applied to all the lidar scan coordinates via matrix multiplication for efficiency, and an array of lidar scan points in the head frame of reference is returned.

The third function, findR, is intended simply to find the rotation matrix when passed the ψ, θ, and ϕ angles. Each matrix corresponding to the 3 aforementioned angles is written out in the function, the same as listed above in equations 10 through 12. The fist 2 matrices are multiplied and their result is stored in a temporary array, which is then multiplied with the final matrix to give the rotation matrix, which is returned by the function.

The fourth function, head2body, is intended to be passed a lidar scan in the head reference frame and the current time index, and return the lidar scan in the body reference frame. First it finds the closest time in the joint dataset to the time index that was passed in, as the goal is to find the rotation of the head to the current time. Once the closest value is found, it is checked to make sure it is equal to or less than the current time and decreased one time index if it is not. After this, the neck and head angles are pulled from the joint data using the index closest to current time, that was just found. These angles are then passed into the findR function along with a 0 angle for θ next, the transformation matrix to go from body frame of reference to the world frame of reference using the rotation matrix and the translation matrix discussed in the previous section. This transformation is then applied to the lidar scan and the lidar scan in the body frame of reference is returned.

The fifth function, body2world, is passed the current time index, the lidar scans in the body frame, and the current robot state. This function uses the x and y position of the robot state as well as the constant z term as discussed previously, to create the translation matrix. It also passes the θ angle from the robot state into the findR function as the angle ψ, as theta is actually a rotation about the z-axis, along with zeros for the other two angles to get the rotation matrix and create the translation matrix. This matrix is then multiplied by the lidar scans to convert the scans to the world frame via efficient matrix multiplication. The array of lidar scans in the world frame is then returned.

The sixth function, meters2cells, is passed x and y coordinates and the arrays containing the x-y coordinates for each cell. It uses these arrays to find the cell that has the minimum difference between the x and then y coordinates and returns the x-y coordinates of the cell that original x-y coordinates belong in.

The seventh function, meters2cellsVector does the exact same thing as the previous function, except it is an improved version that can handle being passed a vector of x-y coordinates, finding the coordinates of the cell with the closest x-y coordinates for each of the coordinate pairs, and returning these cell coordinates in a vector. In effect, it can convert an entire array of lidar scans in meters into cell coordinates, which is what it was created for.

The eighth function, cells2meters, is designed to convert cell coordinates passed to it into meter coordinates and return meter the coordinates. It was never used in this program, but is left in the code in case another user wishes to use this code and finds themselves in need of a function that does this.

The ninth function robotState2Cells, takes as input a robot state in meters. It takes out the x-y coordinates, passes them to the meters2cells function, and then puts the cell coordinates back into the robot state array with the pose angle and returns the entire robot state array.

The tenth function, robotUpdateSate is designed to update the robot state to the next state using the delta pose data. Note that this function is only used in dead reckoning. This fucntion is not explicitly used in the mapping step, but in order to make this code explanation easy to read, with all the functions in one place, it is included here. This function takes the current time index and robot state as inputs. It uses

the time index to pull the delta pose data for the x, y, and θ from the lidar data. It then assembles this into a delta state array and adds it to the robot state. It then returns the robot state array.

The eleventh function, MappingSmart, does the majority of the heavy lifting when it comes to updating the log odds map. It is passed the current time index and robot state. It converts the robot state to cells, and then gets the lidar scan data for the given time index and runs it through the aforementioned transformation functions in order to get the lidar scans in the world reference frame. It converts all these lidar scans in meters to cell coordinates with the function meters2cellsVector. It then makes these lidar scan cell coordiantes into type int32. It creates a psudeo map the same size as the log odds map. It then calculates the log odds value given the lidar trust (set to 0.9). It converts the cells corresponding to the lidar scan into an array with an x column and a y column, with the robot state being the last pair of coordinates, as cv2.drawContours requires a closed contour to fill in all the space between the robot and obstacles as free space. This array is added to $contours$, which is a list of arrays, the required format for the drawContours function. This function was used as it performs much faster than the bresenham algorithm, and high speed was necessary, as without it, the SLAM algorithm would take far too long to run. The function cv2.drawContours was called with the lidar scan coordinates passed as the contours, and the last argument being -1, meaning the function would apply the value of the second to last argument, -logProb, to all the cells in the area surrounded by the contours, hence all the free space, and the lidar scan points. This value being assigned to the scan points was not a concern, as in the next line, cv2.drawContours was called again, and passed the lidar scan coordinates as contours, but this time with the last argument being 1, meaning that only the contour points, the lidar scan points, would have the second to last argument, +logProb, applied to them. This psudeo map is then added to the map of log odds in order to update the log odds.

The twelfth function, updateParticles, will be discussed in the update step as it specifically pertains to that step.

The 22 lines of the main function are adapted from the file given as part of the project, load-data.py. First a dictionary called j0 is created by calling the get-joint function from the load-data file. This function creates entries with the key names 'ts' and 'head-angles', and imports the data for all the time stamps that correspond to the head and neck and angles stored in 'head-angles', respectively. Then a dictionary called l0 is created and uses the get-lidar function to create a list with an entry for each time index, and at each list item, a dictionary with entries called 'lidar', 'scan', and 'delta-pose', to which it loads all of the corresponding lidar and delta-pose data. Next the dictionary MAP is created, with entries for map resolution, minimum and maximum x and y coordinates in meters, number of cells in the x and y dimensions the map, and a map of all zeros the size of the aforementioned cells. Next two arrays are created. In the first one, the indices correspond to the x coordinates of cells, and the values are each index correspond to the x coordinates of that cell in meters. In the second one, the exact same thing is done, except for y cells and y coordinates. These are useful when converting from meters to cells and from cells to meters.

The following code is written by the student, but contains some techniques for vectorizing operations seen in the load-data and p2-utils files.

First, the map is reinitialized to all zeros to ensure it is empty. Then the robot state in meters is declared, with all zeros, as the robot's first state is at the origin of the world frame. This state is converted to cell coordinates and then stored, as all robot states are stored for viewing on the occupancy grid once it is created. Next, some variables are initialized outside of the loop that loops through all lidar scans in order to make the program run faster. Lidartrust, and the number of particles are declared, as well as an array for storing particles for use in the prediction and update step. Next, the max weighted particle is selected (all particles are at the origin with equal weight so it doesn't matter), and set as the robot state. This is more of a symbolic step, as it doesn't actually change anything at this point in the program. Next the function MappingSmart is called and passed time index 0, and the initial robot state at the origin to map the initial scan. After this is completed a few more variables are initialized to count the number of particle resamples in the update step, control how often a lidar scan is used (ie use every scan, every other scam, etc), and finally, a variable to control how often pictures of the SLAM in progress are saved. After this, the mapping step is performed every loop as the for loop goes through all the lidar scans, simply by calling MappingSmart, once the prediction step has been completed and the highest weight particle has been assigned as the current robot state.

B. Prediction

Basic movement in the case of dead reckoning is simple due to the nature of the program. In order to move the robot in this case, it is simply required to call the robotUpdateState function discussed earlier and add the delta pose and the current time index to the robot's current state. The prediction step relies on equation 15 to approximate the pdf with particles from which samples are drawn.

$$p_{t+1|t}(x) = \sum_{k=1}^{N_{t+1|t}} \alpha_{t+1|t}^{(k)} \delta(x; u_{t+1|t}^{(k)}) \tag{15}$$

In the case of using particles in the particle filter, this becomes somewhat more difficult. This function for doing this is called updateParticles, and is the only function that has not been discussed in detail. This function is passed in the array containing the states and weights of all the

particles, as well as the current time index and the total number of particles. The first part of this function works the same way as the robotUpdateState function, getting the values of x, y, and θ for the current time index and creating a delta pose matrix. The only difference in the first part of this function is that this delta pose matrix is then added to every single particle in the array, not just the 1 robot state like in robotUpdateState. The second part is the more unique and important part of this function. The second part of the function generates random Gaussian noise for x, y, and theta, different for each particle, and adds it to the particles to disperse them, representing all the possible ways the robot could have actually moved, due to the odometry data not being perfectly accurate.

The student spent a lot of time tuning these noise parameters, but in the end realized there was an even better approach to solve this problem that would improve runtime and performance. The noise is there simply to account for the robot moving in ways not quite accounted for in the odometry. This means that there really is only noise in the robots state compared to the odometry when it moves. By using an if statement to check if the robot has moved in the current time index, noise is only added when the robot moves. This keeps particles from dispersing when the robot is standing still - which would happen if noise was added every time, and was causing a lot of problems before the if statement was added in. The if statement only checks x and y movement, because, by the physical design of the robot, it can only change orientation angle by taking steps, hence it will move, which will be reflected in the odometry when it wants to change orientation. By only adding noise during movement, the robot is able to build a strong map while standing still, due to the absence of unnecessary noise, which in turn helps it update to a very accurate real world position when it does move by virtue of the strong existing map. Different values for the variance on the zero mean noise were used and left in the program intentionally so another user would be able to see what has already been tried and what worked and what didn't. The tuning of the noise parameters will be discussed with the results in the results section.

The prediction step also includes resampling of the particles if necessary. The student found that the stratified resampling algorithm, presented in the lecture slides, which guarantees at least 1 high weight particle will be chosen, was the best choice. The pseudo code for this algorithm implemented in the project is displayed below, as it was determined by the student that this was most intuitive way to present the implementation of the algorithm.

In the project code, the updating particle state part of the prediction step and the updating weights part of split at the beginning and end of the loop. It should be noted that the particle re-sampling could be moved to the beginning of the loop to bring these two operations together, but it would have no effect on how the code actually ran so the student elected to leave it, as checking for an update on the particles on the first iteration of the loop would have been pointless, as they all would have the same weight.

```
If n-effective is less than or equal to n-threshold:
    Increment count resample
    Initialize variables for stratified resampling algorithm
    Create an array to store new particles

    For the range of all particles:
        This is the stratified resampling algorithm in slides
        Set value of b
        While b > c:
            Increment j
            Add particle weight to c
        Assign x value to new particle
        Assign y value to new particle
        Assign theta value to new particle
        Assign weight to particle
    Set new particles as current particles
```

Fig. 1. Pseudo Code

1) :

C. Update

In the update step, the particles weights are evaluated and updated if necessary. This is done in the main loop, under the title "Update the correlations". First an empty array to store all the particles correlation values is declared. Next, the log odds map is converted into a binary map that zero everywhere except of the positive cells, which are labeled 1. The purpose of this is to be able to easily compare the current lidar scan from the perspective of any given particle to the map of occupied positions, to see how well it aligns with the current map. Next the current lidar scan is taken and transformed all the way to the body; the final body to world transformation must be done inside the loop of particles as that transformation depends on the state of each particle. Then all the particles are looped through, with the current particle selected and the final body to world transformation done on the lidar data. Then the points of the lidar scan are converted into cell coordinates from meters, and placed in an array of x-y pairs. Finally, using these cell coordinates, the locations of the cell coordinates of the lidar scans on the binary map are summed, and since occupied cells are 1 and unoccupied are 0, this gives the number of matches between the known occupancy grid and lidar scan. This is stored as the correlation value for that particle in the array particle correlations. Note that the cell coordinates are fed into the map y-x instead of x-y, as the numpy array seemed to flip x and y in this occasion, so this was the solution and gave very good results. Note the array particleCorrelationsget stores the correlations for all particles at all times, which is very useful in debugging. Once all particles have been looped through, the softmax of the array of correlations for the current set of particles is taken to create the new weights, which are then applied to the particles. In essence, this rescales the particles based on the observation likelihood.

D. High Level Program Description

This section serves as a high level description of the main loop in the program, in essence describing the full SLAM

procedure. Note that this references the code by lines, as this is the easiest way to speak about specific sections of code.

First, all relevant data is imported and all necessary variables are initialized. Second, after entering the loop going through all lidar scans, whether or not to save the current map and robot state is checked and executed if necessary. Note that this is not part of the main SLAM algorithm, so it wasn't described above, but is commented in the code and found in the code's Readme file. Third, all particles are updated with delta-pose data and noise if necessary. Fourth, after checking if this was a scan to be used (in the end, every scan ended up being used), the particle correlations were found and weights were updated. Fifth, the max weighted particle was chosen as the robot state and the map was updated using this state. Sixth, the particles were re-sampled, if necessary. After the loop, the log odds map and occupancy grid map were displayed and saved. Note that this is not part of the main SLAM algorithm, so it wasn't described above, but is commented in the code and found in the code's Readme file.

IV. Results

For ease of display in this document, all the results will be broken into several sections, where the picture is shown first, then the section following will discuss these results. Also note that the orientation of the maps is because the use of the openCV library to save the images causes the x and y coordinates to be flipped. The result is still the same however.

V. Dead Reckoning Results

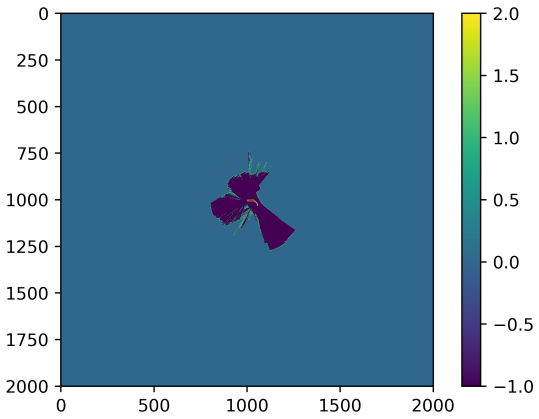

Fig. 2. Map 0, Dead Reckoning Result

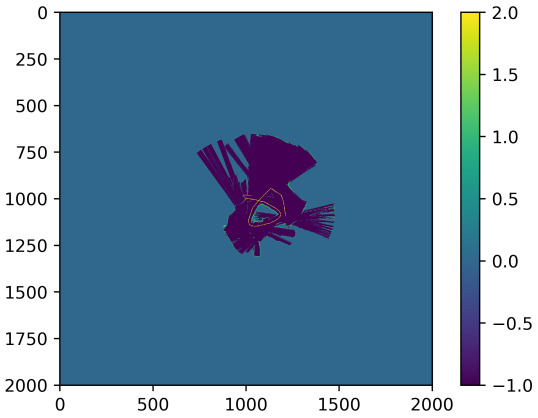

Fig. 3. Map 1, Dead Reckoning Result

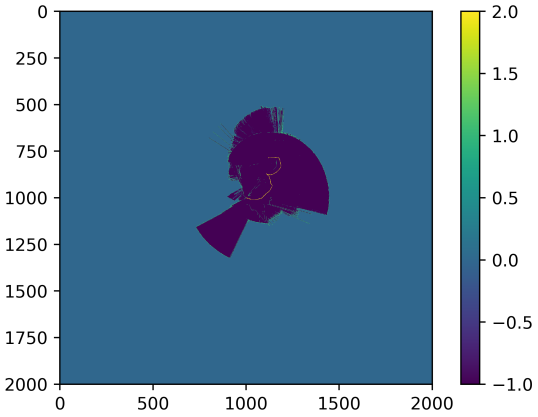

Fig. 4. Map 2, Dead Reckoning Result

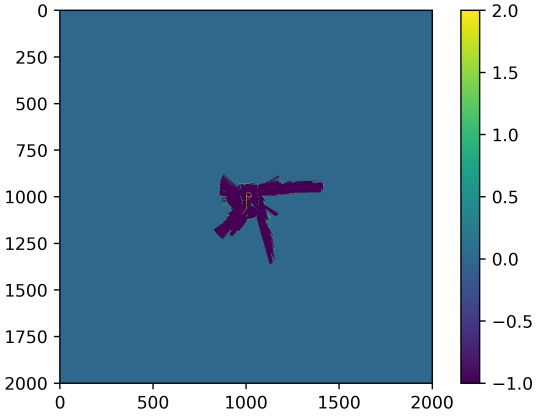

Fig. 5. Map 3, Dead Reckoning Result

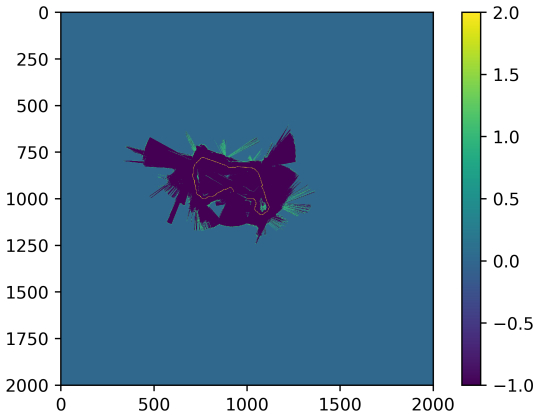

Fig. 6. Map 4, Dead Reckoning Result

VI. Final Occupancy Grid Maps

Note that these maps are quite large as the map was set to a sufficiently large size so as to not worry about the robot running off the map. As this program took some time to run, the student only wanted to have to run it once, hence the large maps as insurance. If the reader needs to zoom in on the figure, to do so, please do zoom in on this pdf document. Also note, the student's latex compiler timed out when trying to add any more images. Please see the videos and photos attached in the folder for the full progress over time, as the student did them for all maps, but just can't display them all in the report. Please also see the images of grid occupancy maps obtained with different levels of noise included in the folder.

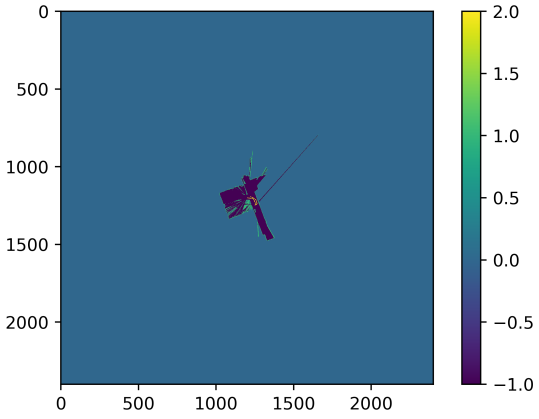

Fig. 7. Map 0, final occupancy grid

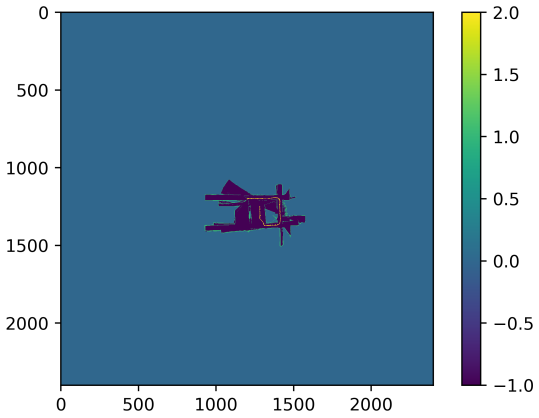

Fig. 8. Map 1, final occupancy grid

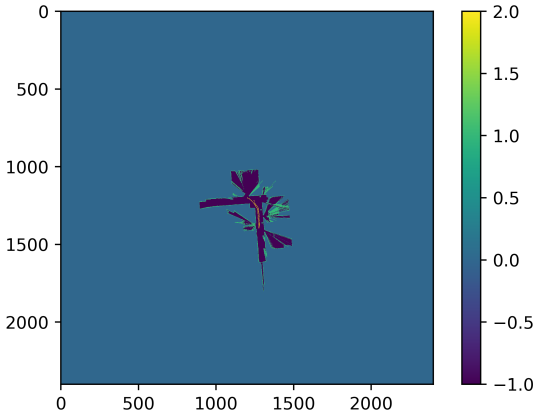

Fig. 9. Map 2, final occupancy grid

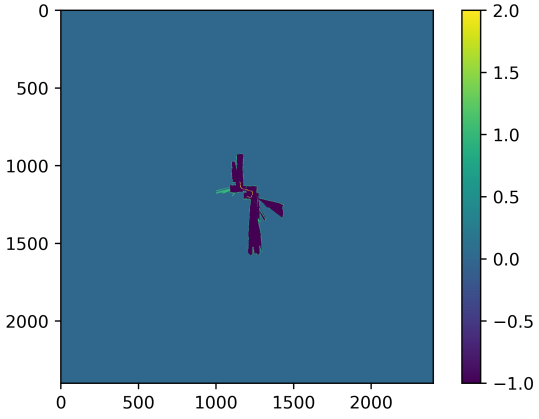

Fig. 10. Map 3, final occupancy grid

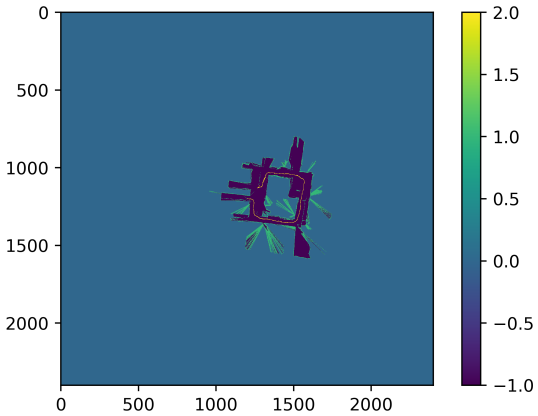

Fig. 11. Map 4, final occupancy grid

VII. Occupancy Grid, Lidar 0 Shown over Time

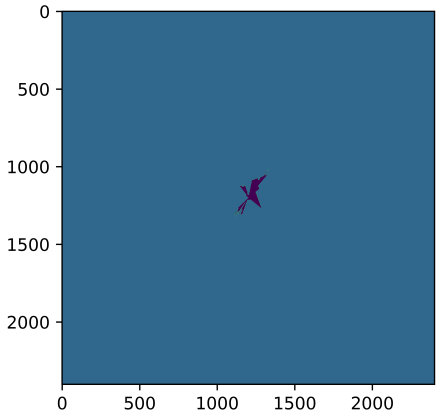

Fig. 12. Map 0, occupancy grid progress 1

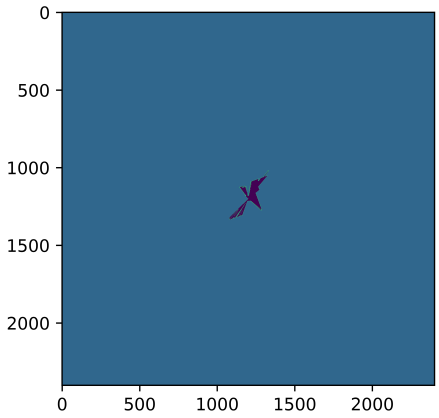

Fig. 13. Map 0, occupancy grid progress 2

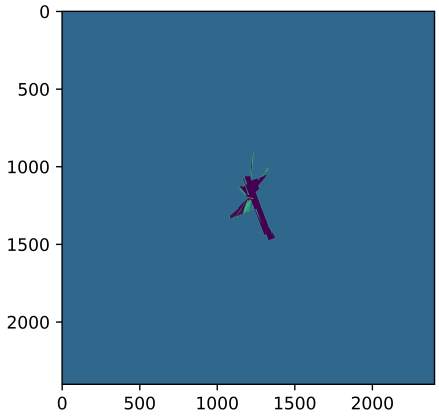

Fig. 14. Map 0, occupancy grid progress 3

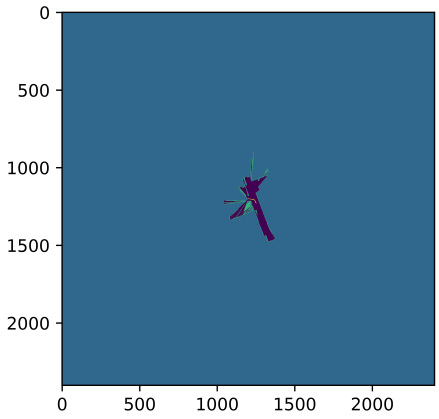

Fig. 15. Map 0, occupancy grid progress 4

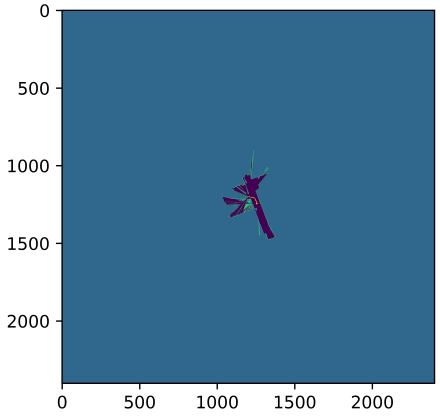

Fig. 16. Map 0, occupancy grid progress 5

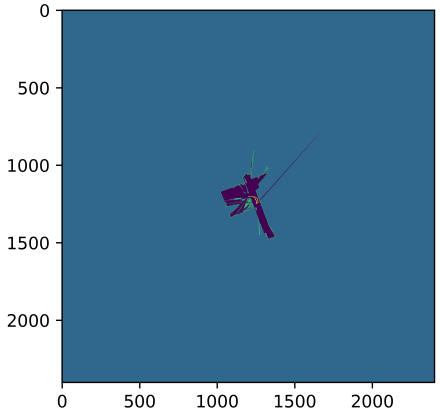

Fig. 17. Map 0, occupancy grid progress 6

VIII. Occupancy Grid, Lidar 1 Shown over Time

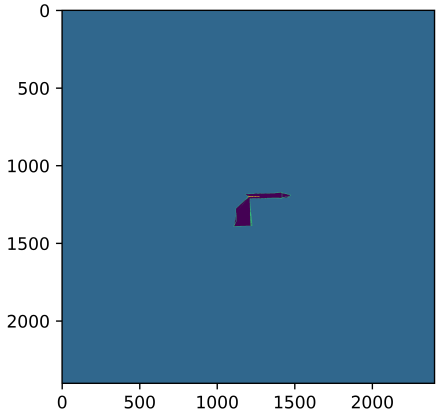

Fig. 18. Map 1, occupancy grid progress 1

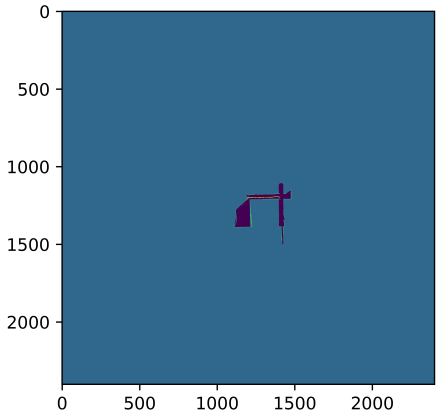

Fig. 19. Map 1, occupancy grid progress 2

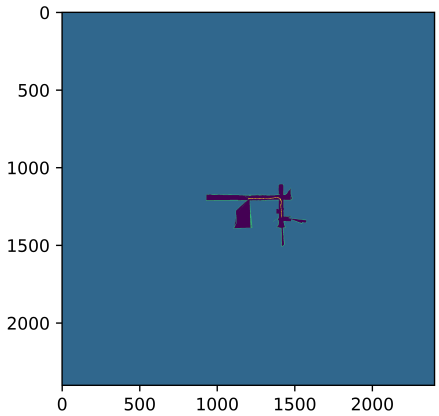

Fig. 20. Map 1, occupancy grid progress 3

IX. Occupancy Grid, Lidar 2 Shown over Time

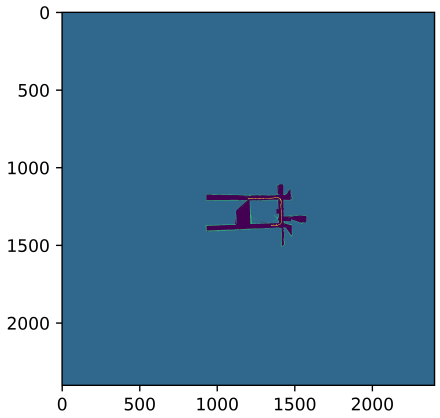

Fig. 21. Map 1, occupancy grid progress 4

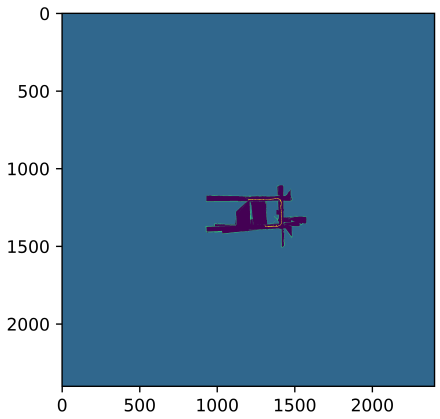

Fig. 22. Map 1, occupancy grid progress 5

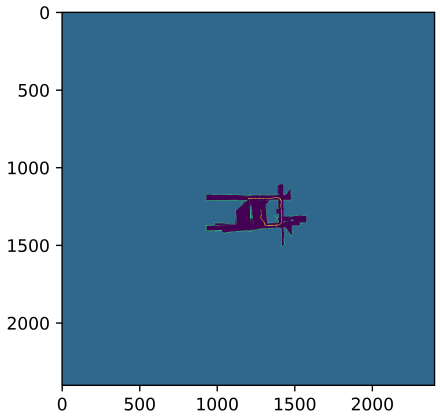

Fig. 23. Map 1, occupancy grid progress 6

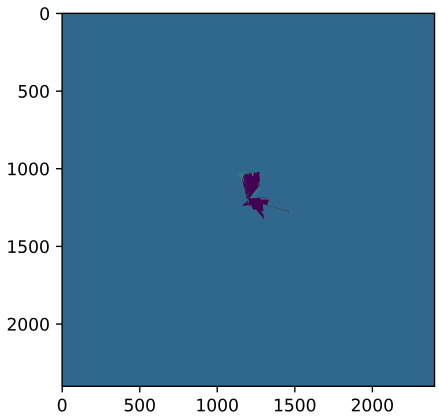

Fig. 24. Map 2, occupancy grid progress 1

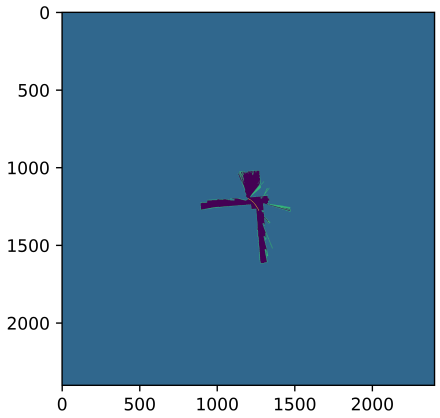

Fig. 25. Map 2, occupancy grid progress 2

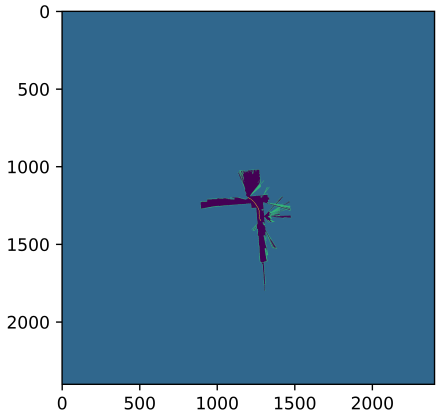

Fig. 26. Map 2, occupancy grid progress 3

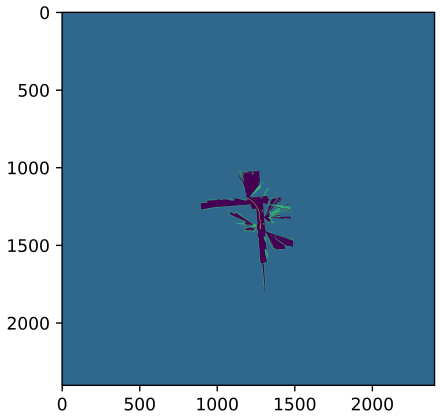

Fig. 27. Map 2, occupancy grid progress 4

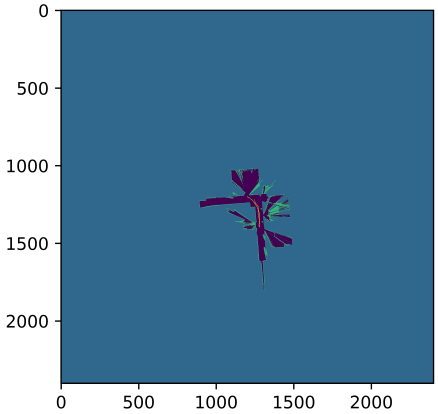

Fig. 28. Map 2, occupancy grid progress 5

X. Occupancy Grid, Lidar 3 Shown over Time

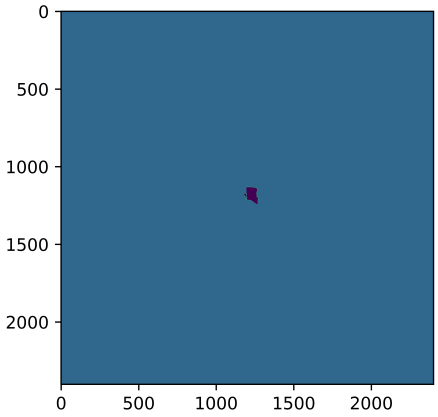

Fig. 29. Map 3, occupancy grid progress 1

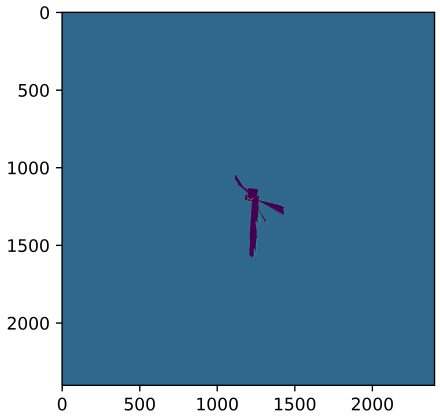

Fig. 30. Map 3, occupancy grid progress 2

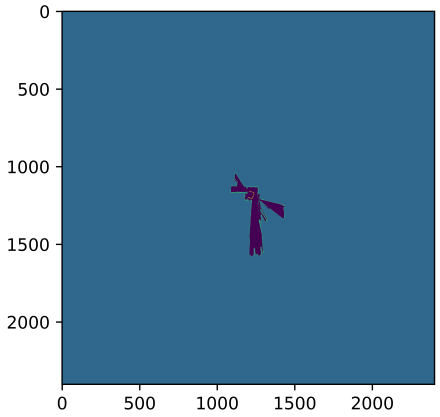

Fig. 31. Map 3, occupancy grid progress 3

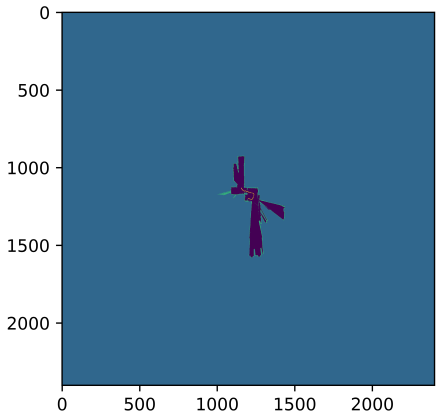

Fig. 32. Map 3, occupancy grid progress 4

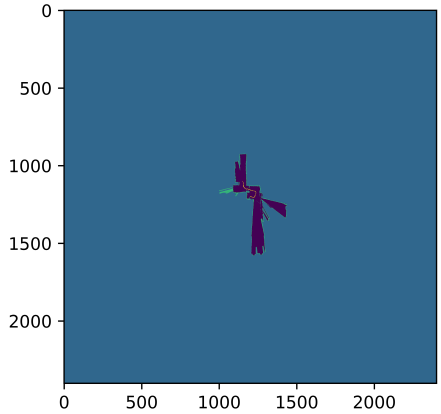

Fig. 33. Map 3, occupancy grid progress 5

XI. Results Discussion

The tuning of noise was done mainly using maps 1 and 4, as map 1 could run fast as an initial check, and map 4 was complex and would serve to further check noise that worked well in map 1. The noise was tuned using visual checks of the resulting occupancy grid maps, as well as looking through the saved correlation data for all particles at all time-steps. Generally, maintaining a high correlation throughout the entire run of the program was an excellent indicator of a good choice for the variance on the noise. The results achieved here are very good. In general, all major lines of the map line up, and and robot's path generally returns to the start point on maps that it is supposed to. Starting off with the dead reckoning, clearly the robot's trajectory was not accurately defined by the sum of the delta pose's. This not accurate trajectory lead to the creation of maps that were indistinguishable and useless for all intents and purposes. Clearly SLAM was needed in this case to get accurate localization. Because the robot's sensor data depends on its location, the poor localization of the robot in dead reckoning resulted in poor maps. The most common characteristic was errors in rotations leading to maps that had rotated hallways and other features. Comparing the final results with dead reckoning, it is clear that the SLAM algorithm implemented did what it was intended to do. The robot's path in the final occupancy grid maps is clear, and the environment is mapped very accurately. This is due to the robot having the correct location after implementing the particle filter due to the program always choosing the robot state to be particle with the maximum likelihood according to the current known map and current lidar scans. This accurate location allowed the lidar scans to be interpreted accurately, and thus improve the map, instead of making it worse, which is what happened in dead reckoning.

Before implementing the piece of code described earlier where the prediction step only adds noise to the particles' positions when the robot is moving, the map was not much

better than the dead reckoning. This is because with low noise it would be similar to dead reckoning, and with high noise, the robot state would shift when the robot was staying still, and it simply wasn't computationally possible to maintain enough particles to keep the robot state in the correct location. Included in the folder with code are some grid occupancy mappings with different noise parameters used than the ones for the final maps, but still using the trick where noise is only added when the robot moves. These maps are passable and not bad, but in general not quite as good as the final maps shown here. What this does show, however, is the the algorithm, mainly on account of only adding noise when the robot moves, is fairly robust at different levels of added particle noise, which means three things. First, not a lot of time had to be spent tuning the algorithm's noise. Second, the algorithm can readily be used on other data-sets not tested here, as it won't require any noise tuning due to its robustness. Third, the algorithm could be ported over to a slightly different robot, and still likely work well, because though the robot likely has a slightly different amount of error associated with its odometry, the algorithm would be able to handle the difference without the user having to spend time re-tuning the algorithm.

One area where this algorithm did struggle a little is the addition of walking humans to the environment. This presents a challenge as the obstacles are now moving and the algorithm must be able to account for this without having its location or its map affected. In general the best practice to dealing with moving obstacles was to run every single lidar scan, as it would give the algorithm the longest amount of time to adjust to a moving obstacle in order to not have a rapid change in location or map.

Overall, it can be seen from the results that the program works well overall, and is able to accurately localize the robot and map the environment. Though the moving obstacles did present a challenge, it was ultimately overcome, with the end result being very accurate maps of the robot's environment.

XII. Discussion with Fellow Students

This student regularly studies in a group with other students including Roumen Guha, Stephen West, and Maria Fatima and it is acknowledged here that this student discussed this project with these students in capacities such as: the math behind the frame transformations, and the math seen in the lecture slides pertaining to the particle filter resampling.

References

[1] Atanasov, Nikolay. "ECE 276A." UCSD ECE276A: Sensing Estimation in Robotics (Winter 2020), Github, 1 Jan. 2020, natanaso.github.io/ece276a/.