CSCI 5551
Robot Navigation and Mapping
Project Final Report

Jacob Medchill - medch004@umn.edu
Chung-En Yu - yu000625@umn.edu
Matthew Basken - baske024@umn.edu

1 Abstract

This project developed a SLAM (Simultaneous Localization and Mapping) implementation for the Turtlebot3 Burger. The three functions of the package, mapping, localization, and movement, work completely together to produce a useful and applicable result. The project demonstration and testing was done almost entirely using a simulation, where an environment was created for the robot to explore. The robot was spawned with no knowledge of its surroundings. Using this package, it plotted its surroundings while randomly exploring to create a complete map of the unknown environment. When testing the package, a complete map was successfully created, but the time increased exponentially as the environments got more complex.

2 Introduction

This project is fully operated on ROS, a high performance platform, and can be divided into three main parts: mapping, localization, and exploring. Firstly, in order to map the environment, the mapping node fetches the sensor data from **/tf** and **/scan** topics, it calculates a coordinate representation in a global frame of the given current location and position of the robot as well as the distance measurement, then it plots the coordinates and the robot's position on the "Rviz" application. The localization node would calculate the estimated position of the robot based on the initial robot position and give nine landmarks' positions as a priori. Moreover, the localization node was implemented with the "**Extended Kalman Filter**" technique, so it would be able to approach more accurate robot position estimates. Lastly, the laser scanner is used to detect obstacles for the movement algorithm.

The robot model is "**turtlebot3** - **burger**", and it has the basic sensors we need in this project, including laser and odometer. The result would show the simulation of the random exploring robot with SLAM based on the gazebo simulation software.

3 Approach

3.1 Mapping

The input from the sensors on the robot were to be collected over time and all plotted together in a way that shows the shape of the obstacles and boundaries that the robot encounters. The applications of this algorithm are plenty. It could be used to bug test navigation algorithms, the mapping of the sensor inputs could be further processed to create a simplified map for the robot with simple geometry, or it could be simply used to see how much of an area the robot has discovered.

3.1.1 Sensor Data Transformation

The information used to generate the map of the sensor data was from the **/tf** topic and the **/scan** topic. The **/tf** topic was used to obtain the current coordinate location of the robot as well as the current orientation of the robot. The **/scan** topic was used to obtain the distance measurements from the lidar sensor. In order to convert these distance measurements to coordinates for the map, they must be converted into a global frame. The global frame used was

defined by the coordinate system that the **/tf** topic abided by. The conversion that was used relied, unsurprisingly, on trigonometry. The lidar data points are taken from **/scan** and transformed to the global frame by using the robot's displacement and orientation. The orientation was calculated using the quaternion given by the **/tf** topic. This quaternion was converted into the robot's current yaw from the global frame. One final note about the conversion is about how the angle of a given distance measurement was determined. It was found that lidar readings were stored in an array and that the angle of increment between each of the readings is one degree. Therefore, the index into the array of measurements is also equal to the angle that it was measured at. The equations used to calculate the x and y coordinates of a given lidar reading are below.

$$rawX = ranges[i] * cos(i * \pi/180 + curYaw) + curX$$
$$rawY = ranges[i] * sin(i * \pi/180 + curYaw) + curY$$

The value ranges[i] is the access into the array which stores the lidar readings, curYaw is the calculated yaw of the robot from the global frame, curX and curY are the x and y displacement from the global frame, and rawX and rawY are the coordinates to be plotted of the lidar measured distance.

3.1.2 ROS Topic Synchronization

A peculiar result was found when angularly accelerating the robot. New points plotted to the map would appear to rotate about the robot center as if someone had placed their hand on a wet painting, rotated it about the center of their palm, and smeared the painting. The "smearing" effect would stop when the robot reached its maximum angular velocity and stopped angularly accelerating. It was assumed that the cause of this distortion was that the readings used to calculate the points to be plotted were generally a certain amount of time offset from each other. In other words, the data used from the **/tf** and **/scan** topic generally had a certain time offset from each other. The solution to this problem was to store all of the messages from the callback queue and compare their timestamps and only use the pair of messages from **/tf** and **/scan** which had the smallest difference in their timestamp. This was implemented by having a while loop which did the work of adding new points to the map and at the end of that while loop, there was a *ros::spinOnce()* command. In each of the callback functions for **/tf** and **/scan**, the messages from **/tf** were added to a vector, and the messages from **/scan** were added to a vector. At the beginning of this while loop in the main function, there is a double nested for loop which compares the time stamps of each message and determines the pair which has the smallest difference in time. This is the pair which is used to calculate the x and y coordinates which are added to the map, the quaternion value used for the robot orientation, and the robot's current location. After the messages are used, the vectors are cleared and *ros::spinOnce()* is called again.

3.1.3 Avoiding Overloading Rviz

ROS is able to run very quickly and the map is filled surprisingly fast. If the program is allowed to run unrestricted for as little as 30 seconds, Rviz becomes bogged down with all of the

points. Rviz updates slowly with new points and responses from interaction with the Rviz window are very delayed. It is clear that the robot collects all of the necessary data, while located in a particular location, from one or two readings from its lidar. Additional data points added on top or very near to these points act only to slow down Rviz without giving additional insight to an observer. This issue needed a solution in order to allow the program to run unmonitored and without stopping it. The solution employed came from the joint use of coordinate rounding and a quadtree data structure. This implementation essentially ensures that points that are close to each other are not both plotted into Rviz.

3.1.3.1 Coordinate Rounding

The above calculation for converting sensor data into coordinates for mapping creates a value with a high accuracy so it is highly unlikely that the exact same point will ever be generated twice. This means that it is not enough to simply not plot the same point more than once. One way to solve this problem is by rounding the coordinates that are calculated. One way to visualize this is to think of the coordinate space in Rviz to be made of pixels that can either be on or off. When coordinates are calculated, they are rounded in such a way that all points within a certain region become equivalent. The equation for rounding the calculated coordinates in this manor is shown below.

$$roundX = ((int)(rawX/width)) * width$$
$$roundY = ((int)(rawY/width)) * width$$

The values rawX and rawY are equivalent to those calculated above in 3.1.1, width is the width of a pixel, and (int) is a use of C++ type casting. This equation essentially rounds down the raw coordinate value to the integer multiple of width that is below the raw value.

3.1.3.2 Quadtree

Coordinate rounding was used to enable the use of a quadtree (this is explained later). A quadtree is a data structure which is a collection of nodes where every node has four child nodes and stores an x and y coordinate. It is similar to a binary search tree in that the value of a node determines which child slot of its parent it will be stored in. A parent can be thought of as having a North West (nw), South West (sw), South East (se), and North East (ne) child slot. Consider a cartesian plane and imagine a node is created for the point (0,0). A new point would be placed in its nw slot if its x coordinate is less than its parent's ($x < 0$) and if its y coordinate is greater than its parent's ($y > 0$). This logic can be applied to understand the sw, se, and ne slots. A potential new point can "collide" with an existing node if their coordinates are equal. If the coordinates added to the tree were unrounded then this would hardly ever happen. To increase the frequency of collisions, coordinate rounding is used. The amount of coordinate rounding needed to avoid overloading Rviz still produces an acceptable "resolution" for the produced map. The reason a quad tree was used to store the points instead of simply storing all of the points in an array is that the speed of detecting a collision with a quadtree is much greater than for an array, just as with a binary search tree. This quadtree and coordinate rounding combo is used to ensure that points are not unnecessarily added to Rviz. If a point is near enough to

another one already in the quadtree then it won't be added and so the point will not be plotted on the map.

3.2 Localization

Every sensor in the real-world inevitably has some noise within the data so when we simulate our system in gazebo, we manually add some random noise on both the odometer and robot's speed (input) to make the algorithms developed in the gazebo more practical in the real-world. In order to deal with the noisy data, we used one of the most common techniques, called "**Extended Kalman Filter**" to optimize the noisy data so that we are able to utilize it in our system. With this technique, we can localize the robot's position without GPS, which simulates scenarios where GPS might not work or indoor environments. Furthermore, we utilized nine known landmarks (know their position as a priori), to update our estimated robot position.

3.2.1 Extended Kalman Filter with Localization

Extended Kalman Filter can deal with non-linear data by linearizing it, then estimating the linearized state function, and updating the estimated values with "**filtered"** measurement. With such, we can have more accurate sensor data, greatly reduce noise, and at the same time update the robot's position. In addition, it can prevent errors from propagating through the iteration of estimate and update; otherwise, errors are quickly becoming significant during the process. The following sections show how we implemented our robot with the Extended Kalman Filter.

3.2.1.1 Propagate Estimation

Our state function for this system is shown below, where x, y represent the robot's position coordinates and φ is the orientation of the robot, *dt* is the small amount of time of the robot, and k represents one timestep, which would increase while iteratively updating. V and ω are linear speed and angular speed of the robot, respectively, and n_v, n_ω are the noise of the speed respectively, where we add them manually.

$$x_{k+1} = x_k + dt * (v + n_v) * cos(\varphi)$$
$$y_{k+1} = y_k + dt * (v + n_v) * cos(\varphi)$$
$$\varphi_{k+1} = \varphi_k + dt * (\omega + n_\omega)$$

We generalize these three equations as state function $\mathbf{x_k}$, and (v, ω) is the control input, so we set it as $\mathbf{u_k}$, and (n_v, n_ω) as **n**. Thus, we may write the state function as $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k, \mathbf{n}_k)$; besides, our estimate of state should be without noise, so the **estimate of state function** notes as $\hat{\mathbf{x}}_{k+1|k} = f(\hat{\mathbf{x}}_{k|k}, \mathbf{u}_k, 0)$. Also, since this state function is non-linear, we used jacobian matrix to linearize it, which we can get the parameters (**ϕ**, **G**) for our **propagation covariance (P)** that we used later in updating the estimation. The parameters, **ϕ**, is the partial differential equation

of state function with respect to the state **x**; **G**, is the partial differential equation of state function with respect to the state noise **n.**

$$\boldsymbol{\phi}_k = \nabla_{\mathbf{x}_k} f(\hat{\mathbf{x}}_{k|k},\ \mathbf{u}_k,\ 0)$$

$$G_k = \nabla_{\mathbf{n}_k} f(\hat{\mathbf{x}}_{k|k},\ \mathbf{u}_k,\ 0)$$

$$P_{k+1|k} = \boldsymbol{\phi}_k P_{k|k} \boldsymbol{\phi}_k^{\ T} + G_k Q_k G_k^{\ T}$$

3.2.1.2 Update Estimation

After the definition of basic state function and its covariance, now we may fetch our measurement, which is the odometer data that we note it as $\mathbf{z}_{k+1} = h(\mathbf{x}_{k+1}) + \mathbf{n}_{z_{k+1}}$, where $h(\mathbf{x}_{k+1})$ is the measurement function of $\mathbf{x}_{k+1}$. Same as estimate of state function, we can get estimate of measurement function as $\hat{\mathbf{z}}_{k+1|k} = h(\hat{\mathbf{x}}_{k+1|k})$. As we used the known landmarks position to localize the robot, plus we only use the odometer as sensor, so the only measurement we have is the distance between landmarks and the robot. The measurement function $\mathbf{z}_{k+1} = h(\mathbf{x}_{k+1}) + \mathbf{n}_{z_{k+1}}$ shows below, where (x_L, y_L) is the known landmark's position, and $(x_{R_{k+1}}, y_{R_{k+1}})$ is a robot position. Moreover, the estimated measurement function $\hat{\mathbf{z}}_{k+1|k}$ is also non-linear, so we have to linearize it with respect to state $\mathbf{x}_{k+1}$, so we can get **H**.

$$z = \sqrt{(x_L - x_{R_{k+1}})^2 + (y_L - y_{R_{k+1}})^2} + n_{z_{k+1}}$$

$$H_{k+1} = \nabla_{\mathbf{x}_{k+1}} h(\hat{\mathbf{x}}_{k+1|k}) = \left[-\frac{x_L - \hat{x}}{||\, x_L - \hat{x}\,||} \quad -\frac{y_L - \hat{y}}{||\, y_L - \hat{y}\,||} \quad 0 \right]$$

With measurement, we are able to get other parameters for updating estimation that are listed below. $r_{k+1|k}$ is the residual of the measurement and its estimate. R is the variance of measurement noise.

$$r_{k+1|k} = z_{k+1} - \hat{z}_{k+1|k}$$

$$S_{k+1|k} = H_{k+1} P_{k+1|k} H_{k+1}^{\ T} + R_{k+1}$$

$$K_{k+1|k} = P_{k+1|k} H_{k+1}^{\ T} S_{k+1|k}^{\ -1}$$

Finally, with these parameters, we are able to update our estimation of the state ($\hat{\mathbf{x}}$) and covariance (**P**).

$$\hat{\mathbf{x}}_{k+1|k} = \hat{\mathbf{x}}_{k+1|k} + K_{k+1|k} r_{k+1|k}$$

$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1|k} H_{k+1} P_{k+1|k}$$

3.2.2 Code Implementation

It is important to carefully look into the matrix shape of these parameters, so that when the equations using "@" as "matrices multiplication" could process the values properly. Furthermore, for every timestep, we update the estimation through all nine landmarks, so that the robot should be able to have a more accurate estimated position. The list below is the shape of the parameters, which makes it more clear about how the data is stored in the parameters.

Parameters	**Shape**	**Parameters**	**Shape**
P	3x3	z (distance)	1
Q	3x3	H	1x3
Phi G	3x3 (Jacobian)	r	1
u (v, omega)	1x2	R	1
x (x, y , orientation)	1x3	S	1
		K	1x3

3.3 Movement

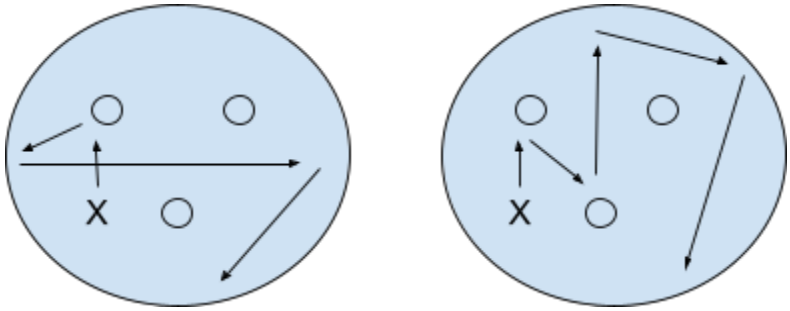
In order to plot the entirety of the unknown environment, the robot had to continuously and randomly explore the map. In this project, the robot is initialized with no information besides its immediate surroundings. There are multiple ways to explore from this point, one of the most effective being rapidly-exploring random trees (rrt). This package used a simpler random movement algorithm that is less efficient than rrt, but effectively is the same.

3.3.1 Exploration Algorithm

This algorithm has two cases: an open path and an obstacle impeding the path. If the path is open, the robot moves forward until an obstacle is sensed. This switches to the second case, where the robot chooses a random direction to turn and travel next. This basic process repeats itself so that the entire environment is explored.

The function uses the laser scanner in order to detect the obstacles. An obstacle is detected" if the laser reads an object within .3 meters and within 45° of the forward facing direction. Once detected, the robot turns a random angle between 0° and 360° and checks the sensor again. This continues until there is no obstacle directly in front of the robot and the robot can freely move forward.

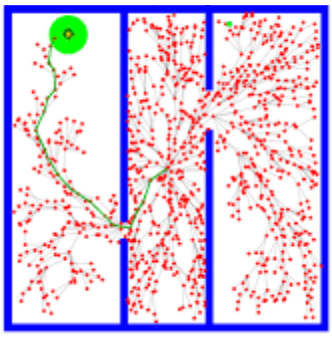
Examples of random path generation can be seen in the figure below. The two scenarios represent simulated worlds where the X is the initial spawn of the robot and the inner circles are obstacles. As the robots are initialized with the same pose, they both start in the same case in the algorithm and move forward. At first detection of an obstacle, rotation of the robots are randomized and the paths diverge. Each situation is completely random, but eventually produces the same result.



3.3.2 Future Implementation

While the current exploration algorithm accomplishes the requirements needed in this project, it could be improved. Theoretically, even though the probability of this occurring is barely above zero, the robot could randomly generate the same two angles essentially bouncing it back and forth between two points. This situation is highly unlikely, but it represents the fundamental inefficiency of this algorithm.

One solution to this problem is using an rrt algorithm. As seen in the image below, the algorithm creates branches of nodes from a source. The branch gets built by randomly sampling new nodes from its surroundings to see if it is possible to reach. If possible, the branch continues. The most useful case for this algorithm is finding a destination point. In this case, once a branch reaches the end, the nodes are backtracked to give the most efficient path to the destination. A version of this algorithm was attempted for this project, but was unable to be implemented. Since there was no destination and the whole map wasn't given, a localized version of the algorithm was necessary. However, documentation was limited and it was outside the scope of the project to complete.



Additionally, merging the mapping and movement functions would be useful in creating a more efficient movement algorithm. The problem with an independent algorithm is that there is no knowledge of the existing surroundings including the past traveled locations, boundaries of the map, etc. All of this data would be crucial in creating a “smart” algorithm that could explore the environment in a shorter amount of time.

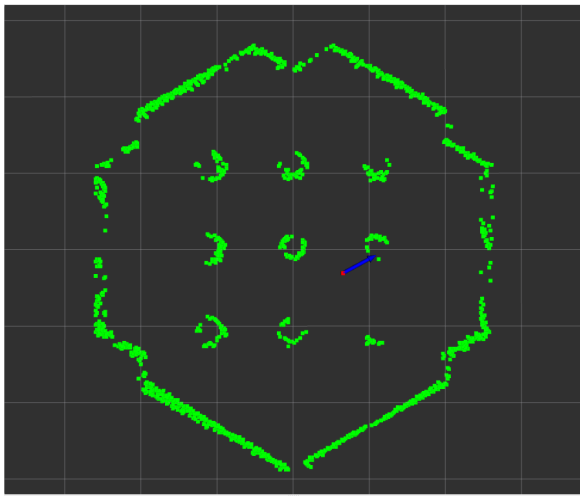
4 Results

4.1 Mapping

The mapping program used is reliable and possesses a high enough accuracy and speed to rely only on the Rviz window for basic operation. Shortcomings certainly do exist and are likely due to incorrect implementation of the quadtree data structure or the coordinate rounding. Overall, the mapping program successfully collects data from the sensors and **/tf** topic and produces a map which clearly provides insight on the robot’s surrounding environment.

4.1.1 Map Accuracy

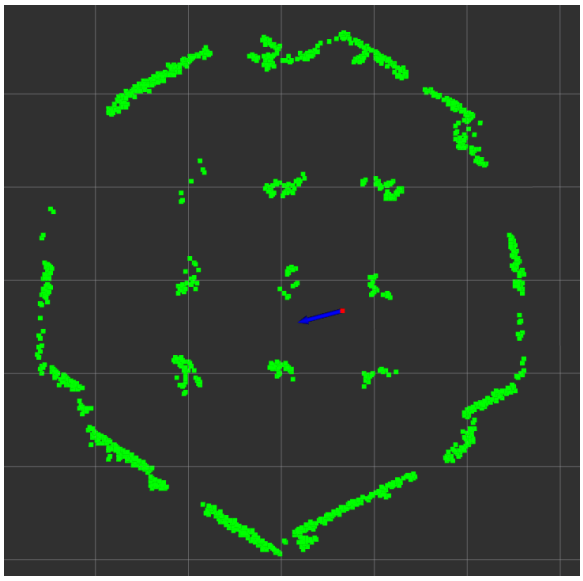
The points plotted by the program are reasonably accurate. The shape of large boundaries are clear and understandable. Certain regions of the map do contain clusters of points and there is a “thickness” to walls. Areas of the map are left without points even after the robot has had the opportunity to scan them, but they are usually small compared to the total size of the map. Refer to the image below for further detail.



As can be seen, all of the pillars contain these bald spots of points to varying degrees. The left and right walls also contain considerable bald spots.

4.1.2 Smearing From Angular Velocity

The following map was produced by setting the robot's angular velocity to the maximum allowed by the turtlebot3 teleoperation node, starting the mapping node, then reversing the angular velocity to the maximum in the opposite direction allowed by the turtlebot3 teleoperation node. Linear velocity was held at zero and no change in position occurred.



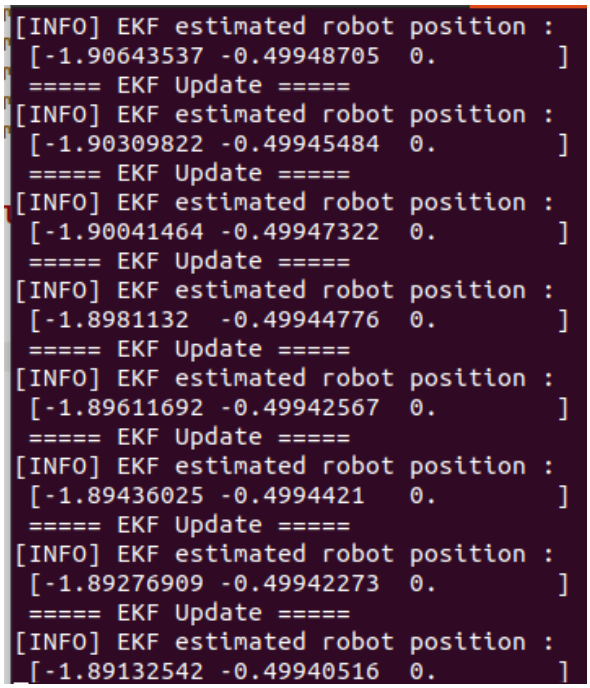
A small amount of the smearing effect is visible in this map compared to the previous map, but in general, the accuracy of the map is still respectable.

4.1.3 Mapping Speed

The program is set up to operate its main while loop at a maximum frequency of 10 hertz. This limitation is primarily implemented to decrease the maximum computing load of the program. With a maximum frequency of 10 hertz, a user is able to maneuver the robot from the Rviz window alone. If this limitation is removed, the Rviz appears to operate at the same speed which suggests that the while loop runs at a frequency near or less than 10 hertz.

4.2 Localization

The localization algorithm is independent of the other processes (mapping and movement). As long as the odometer is functioning well, the localization algorithm is able to calculate the position of the robot no matter how the robot moves. From the figure below, when we activate the localization node in another terminal, it will show up the estimated robot position.



From the figure, we can see that the estimated position is gradually decreasing due to the errors (noises) propagating to the future state, although it is subtle for the current timestep, it will eventually grow larger, then we will lose the accuracy of the robot position. And the orientation of the robot is not updated properly, it might be caused by insufficient measurement so that it can not obtain the correct value.

4.3 Movement

The final implementation of the movement algorithm is a useful random exploration technique for a relatively simple and small environment. It successfully generates a random

path that will eventually reach the bounds of the map. Therefore, a map of an unknown area is created and by definition the project goal is completed.

However, the effectiveness of the algorithm comes into question when there are larger, more complicated maps. The time increases exponentially as the size grows, the obstacles increase, and the boundaries become more complex. If time was not an issue, this drawback is meaningless, but for a practical application a more efficient algorithm would be necessary. Additionally, the simulation could be sped up with greater computational power which could decrease the total time.

In the test cases, the original worlds from the turtlebot3_gazebo were used. Due to the algorithm moving straight until it hits an obstacle, the turtlebot3_world world could be mapped very quickly. The robot spawned with an open path from the start. In other worlds, turtlebot3_house for example, the algorithm was put to the test. The robot was able to explore a room/open space relatively quickly, but had a hard time getting the perfect angle to make it through a door or small gap in a wall. Once it made it through, though, the new space was quickly mapped. Overall, the algorithm used was most successful in smaller, more open worlds.

5 Conclusion

This project recreates and independently builds basic implementations of SLAM and demonstrates simultaneous movement, mapping, and position estimation of the robot. It is a highly potential research field that can be useful for projects like Mars exploration, autonomous cars, etc, which could be done by more advanced algorithms and add on machine learning. Another highly potential research field that can be related to this project is deploying multiple robots for exploration and SLAM, which focuses on the cooperation between robots and aims to make the system work more efficiently.

Overall, the mapping node performed well. It was able to quickly and mostly accurately map the surrounding area of the robot in a way that does not overload rviz while allowing the node to run unmonitored. The robot's position and orientation within the map is accurately modeled and quickly updated as well. Smearing from high angular acceleration is also quite low due to the successful improvement of topic synchronization. This node works well enough that a user could teleoperate the robot and rely only on the output of Rviz to guide them due this node's speed and accuracy. Despite this success, the mapping node has some obvious setbacks. Areas of bald spots are visible and inconsistent, smearing is still evident, and a larger area with more complexity could possibly overwhelm this node as it appears to operate at a frequency less than 10 Hertz. Plenty of improvements could be made. Averaging of sensor information could improve the accuracy of the mapping and even decrease smearing. A dynamic resolution could be employed to improve detail in more complex areas and save on memory in areas of simpler geometry. Finally, the mapping node could calculate its position from either the localization node or from its own speed in order to create a more realistic and practical algorithm.

The localization algorithm works well at the beginning, but later when the robot starts exploring, the algorithm cannot show the estimated position immediately; plus, the errors grow larger after each huge amount of iteration. Another flaw is that the algorithm does not update the orientation of the robot, which was unexpected from the theories. However, besides these two flaws, the algorithm is easy to add to other measurements, without modifying the whole algorithm structure. It makes it easier for programmers to implement different sensors to approach more precise position estimation.

Due to the errors that we found in the results of the localization, we believe that one way to solve the problem is to calculate more measurements from other sensors, then we might observe more estimated robot positions, then we might be able to integrate them together and have a more accurate estimation. With such, it will need to compute the data heavily, so using another common localization technique, i.g. and "**Particle Filter**" might be able to save more time than using "**Extended Kalman Filter**". Another alternative is to change the technique we applied to the localization, i.g. "**Unscented Kalman Filter**" might be able to gain more accurate results for this project.

While the movement algorithm accomplished its goal for the scope of this project, this area of the package has the biggest room for improvement. A new algorithm could be implemented that uses memory, nodes, improved usage of sensors including LiDAR, and many other tools that weren't able to be built into this project. The drawback of the current movement technique is the time required for a complete exploration. On any larger scale, these other tools are a necessity in order to fully explore a map in a reasonable amount of time. Besides improving the current algorithm, the standalone movement function could be used for many other applications. If more of the environment was known, the possibilities for the algorithm are virtually endless.