Botlab: Autonomous Exploration and Navigation

Gustavo Camargo, *Masters Candidate* Sanika Kharkar, *Masters Candidate* Theodore Nowak, *Masters Candidate*

***Abstract*—Mobile robots are quickly making inroads in every aspect of human life. From small toys, to house vacuums, autonomous vehicles, civil and military drones and the mars rover; they all share one distinctive feature: the ability to move around the environment. In this paper we explore this critical capability by deploying a small wheeled robot on a mission to autonomously explore and escape from a maze. In doing so we study the building blocks of autonomous navigation, namely: 1) Localization, 2) Mapping, 3) Path planning and 4) Motion Control. We also implement a visual interface that is indispensable to understand the robots' belief of the state of the world in real time. As we provide the robot with these fundamental capabilities we discuss both the theoretical basis as well as the practical complexities of each.**

I. Introduction

IN this project we have implemented a robot capable of exploring a maze autonomously and ultimately escaping from it. Automation has become the main focus of all industries in the past few decades. Unmanned vehicles are being sent out to explore extraterrestrial lands as well as regions on the Earth inaccessible to human beings. These vehicles rely on autonomous navigation and mapping to collect data that is utilized to benefit mankind.

The efficiency of autonomous navigation depends on the mechanical structure of the vehicle, the traversibility of the terrain, the algorithms implemented for mapping, localization, obstacle avoidance, exploration, etc. A balance has to be achieved between all these factors in order to get optimal performance from the vehicle.

It is with these objectives in mind that we undertook this project. We have started off by assuming a planar environment that is easily traversible and tackled autonomous navigation using the Maebot platform. Our approach includes estimating the location of the robot based on odometry and laser scan data, motion using a PD controller and path planning using the A* algorithm.

II. Methodology

This section describes the various methods and models used for the different tasks in Botlab.

A. Odometry

Our wheeled robot needs to move around the environment in order to accomplish its goal to explore and exit the maze. In order to move reliably it first needs to be able to estimate its position at any point in time. One method to do this is to keep track of the robot's relative displacements with respect to a known initial position. If we assume no slippage occurs between the wheels and the floor, the displacement of the robot can be expressed as a function of the rotation of the wheels. By decomposing the displacement of the robot into a sequence of sufficiently small movements, each of them can be accurately represented as the robot moving along an arch. Furthermore each small step can be decomposed as a movement in a straight line plus a small change in the orientation of the robot. Figure 1 shows the parameters that allow for the decomposition of movement into the equations shown in Algorithm 1.

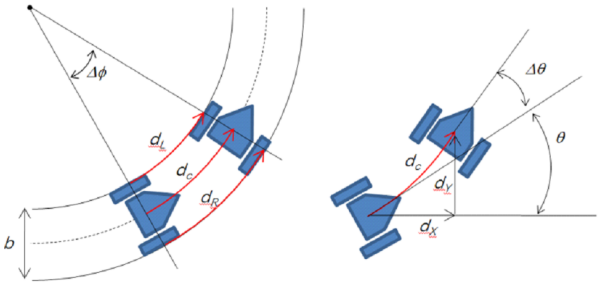

Fig. 1: Geometric analysis of a 2-wheel differential drive robot.

Algorithm 1 Odometry equations of displacement

$$\Delta\Theta = \frac{d_R - d_L}{b}$$
$$d_C = \frac{d_R + d_L}{2}$$
$$X_{t+\Delta t} = X_t + d_C * cos(\Theta)$$
$$Y_{t+\Delta t} = Y_t + d_C * sin(\Theta)$$
$$\Theta_{t+\Delta t} = \Theta_t + \Delta\Theta$$

The time-stamped wheel rotation is the minimum information needed to estimate displacement using odometry. However additional information provided by the

Gyroscope can be used to improve the estimated displacement and position. Specifically we can use the Gyroscope's measurement of delta theta of the robot instead of the that derived in equation (1) from the difference between the displacements of the left and right wheels. Gyroscopes measurements however accumulate error over time (drift) and it is necessary to correct for it following the procedure explained in [3]. To do this we first note that the Gyroscope orientation values are measured with respect to an internal frame of reference and need to be expressed in the global frame of reference before using them. Because the gyro frame is only rotated about the Z axis with respect to the global frame we only need to subtract the initial orientation reading of the gyro from any subsequent measurement. We then measure the drift in the Gyroscope by running it without movement for 10 seconds. Figure 2 shows how despite the robot not moving the rotation output continuously varies. In our case we are interested in the drift about the Z axis which was equal to 7.4 degrees per second (Slope_drift). We use the slope of the Z drift line multiplied by the amount of time since the robot started to move to offset the cumulative drift error over that same period of time. Lastly we convert the corrected Gyroscope orientation value to SI units by using the conversion factor (K_gyro) included in the IMU specification. Finally a delta theta can be calculated for any time interval using these corrected Gyroscope inputs and the equations summarized in Algorithm 2.

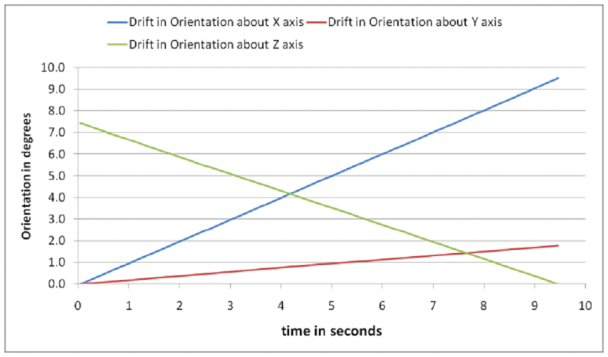

Fig. 2: Drift in Gyroscope orientation measurement.

Algorithm 2 Gyrodometry Equations

$\Theta_G = \frac{(\Theta_{Gyro} - \Theta_{Gyro(t=0)}) - (Slope_{drift}\Delta t)}{K_{Gyro}}$

$\Delta\Theta_G = \Theta_{G(t+1)} - \Theta_{Gt}$

With this correction in place, the gyroscope instant reading is considered fairly accurate however the correction is not perfect and it will continue to accumulate error over time. Because of this we do not use the Gyroscope heading continuously but instead only when we detect large discrepancies between the Gyroscope delta in heading and the Odometry delta in heading (possibly due to wheel slippage or similar events) in which case the Gyroscope delta in heading will be more accurate than the differential drive estimation.

B. PD Control

A PD controller was used to control the orientation and speed of the Maebot. Integral control was not used because insignificant amounts of drift were found during testing. This is in part because pose commands were given frequently and with differing poses such that the controller was not forced to attempt to closely follow a single velocity or orientation over long periods, and thus accumulate offset error. The controller was tuned by observation. The proportional gain was set by increasing the gain until it overshot the target angle, and oscilated a small amount to return to it. Then the differential gain was raised until this oscillation was removed. The velocity controller was designed via the following equations. Note that the change in velocity is encouraged to be 0, while the desired velocity is 5 cm/s. The PD controller thus fights itself, and ramps up and down the velocity output accordingly. The controller was designed so that it never went backwards. The case is similar for the orientation controller.

Algorithm 3 Velocity and Orientation PD Controllers

$P_{vel} = k_{pv}(v_{des} - v)$

$D_{vel} = k_{dv}(-\alpha)$

$P_{ori} = k_{p\theta}(\theta_{des} - \theta)$

$D_{ori} = k_{d\theta}(-\delta\theta)$

Values for k_{pv}, k_{dv}, $k_{p\theta}$, and $k_{d\theta}$ were 200, 10, 5, and .5 respectively. The controller values (P_{vel}, D_{vel}, P_{ori}, and D_{ori}) were then summed for velocity and orientation and then proportionally weighted to yield the output motor commands. The weighting is given below.

Algorithm 4 Motor Commands

$c_{bias} = (v_{max} * v_{pid} - .5(|o_{pid}|)$

$c_{right} = .5o_{pid} + c_{bias}$

$c_{left} = -.5o_{pid} + c_{bias}$

In the above, v_{pid} and o_{pid} are the values output by the two PD controllers and v_{max} is the maximum allowed velocity. Note that these equations are precisely constructed so that the overall velocity will slow down to zero when turning more aggressively. During our tests and execution v_{max} was set to .5 (50%) to allow for greater control and precision navigation.

C. Simultaneous Localization and Mapping (SLAM)

1) Mapping-Occupancy Grid: To arrive at a target location, mobile robots need to navigate the environment without colliding with obstacles. While wheel encoders allow us to estimate the robot's position, they provide no information on the obstacles around it. For this reason we need additional sensors that provide such information. One such sensor is a laser scanner capable of measuring the distance to the nearest object in a large number of directions around the robot. This representation of free and occupied space surrounding the robot is often called a local map. As the robot moves to a new position it can create a new local map in that position. Furthermore, if we express all of the local maps in the same frame of reference, in the aggregate they form a map of the entire environment that we have visited which is often called a global map. Figure 3 illustrates the evolution from the initial local map on the left to the final global map on the right.

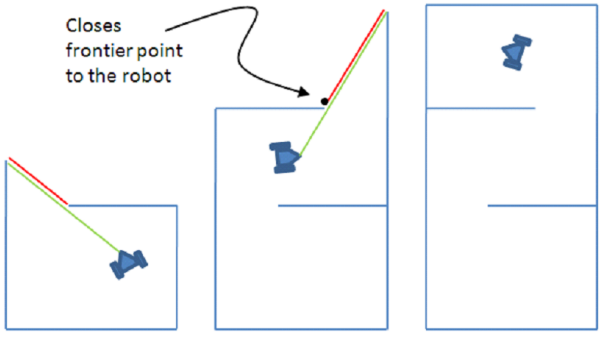

Fig. 3: Local and Global Maps. Red lines represent frontiers.

In addition to enabling the robot to avoid collisions, the global map defines connectivity of free space on the visited environment and can be used to determine if a robot of known size can navigate from one point of the map to another. Nevertheless the estimated position of the robot and the obstacles have errors which can lead to unexpected collision. The likelihood of such collisions is reduced when the robot moves further away from the estimated position of the obstacles; this intuitively corresponds to 'driving in the middle of the hall' instead of close to its walls. We incorporate this concept into our model by assigning a collision risk to each cell that is larger when the cell is close to an obstacle and lower when it is further away. The robot can then use this information when planning how to move in the environment to balance the shortest path with the safest path to the target location.

Our implementation of Mapping uses two global maps in the same frame of reference and each capable of representing an area of 10 meters by 10 meters which is sufficient for all out test cases. Although the real world is continuous, we use a discrete representation of it in the form of a 2D matrix where each cell represents a space of 5cm x 5cm in the real world. This abstractions makes the problem tractable.

The first map is called an occupancy grid and holds the likelihood of a cell being occupied by an obstacle or free. This is continuously calculated with the information received from the laser scans. If the cells of our map were infinitely small, our position estimate had zero error, the laser instrument had zero error and there was no numerical error; then we would expect that a given cell in this map would always be reported as either free or obstacle. Nevertheless because none of these conditions are true, a given cell can be reported sometimes as being occupied and sometimes as being free. Hence the map holds the cumulative odds of any cell being free or occupied by an obstacle.

We implement the collision risk concept described above with the use of a second map called an obstacle distance grid. In this map the risk of collision is represented by an real number, where higher numbers mean higher risk of collision. To build this map we start by looking at the occupancy grid map and assigning very high values (above 5,000) to any cells that are equally or more likely occupied than free. For the purpose of navigation these represent non-navigable cells occupied by obstacles. The remaining cells represent free space and are assigned lower collision risks such that the further from any obstacles the lower the risk value is. We do this by recursively coating the obstacles with layers of cells of lower risk values calculated according to the equation below. The result of this process is shown in Figure 9. This risk value is used by our path planning algorithm. The selection of the gradient of descent in the free space defines how much the robot will prefer going on a safer yet longer route than on a shorter but riskier path. If we want to make the robot behave safer, we increase this gradient (increase the constant K), and if we want to make it find the shortest path we make this gradient zero (K=1). We found the best balance results with a K value of 1.6.

$$CollisionRisk\_n = \frac{CollisionRisk\_n - 1}{K^n} + 1 \quad K = 1.6 \tag{1}$$

2) Action Model: The *pose* of a robot in a planar environment comprises its two-dimensional planar coordinates relative to an external coordinate frame, along with its angular orientation. Denoting the former as x and y, and the latter by θ, the pose of the robot is

described by the following vector:

$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$$

The action model is given by the conditional probability distribution $p(x_t|u_t, x_{t-1})$, where x_t and x_{t-1} are robot poses at times t and $t-1$ respectively, and u_t is the the motion command. We used the odometry motion model for this lab. In this model, the motion information u_t is given by:

$$u_t = \begin{pmatrix} \bar{x}_{t-1} \\ \bar{x}_t \end{pmatrix} \tag{2}$$

where $\bar{x}_{t-1}$ and $\bar{x}_t$ are the poses reported from the odometry data at times $t-1$ and t respectively, i.e $\bar{x}_{t-1} = (\bar{x}, \bar{y}, \bar{\theta})$ and $\bar{x}_t = (\bar{x}', \bar{y}', \bar{\theta}')$.To extract relative odometry, u_t is transformed into a sequence of three steps: a rotation δ_{rot1}, followed by a translation δ_{trans} and another rotation δ_{rot2}.

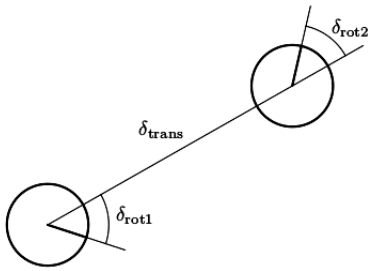

Fig. 4: Transition from $\bar{x}_{t-1}$ to $\bar{x}_t$ broken into three steps

The action model takes as input x_{t-1} (the robot pose at time $t-1$) and the motion information u_t to generate an estimate of the robot pose at time t which is denoted by x_t. This is done by introducing a Gaussian noise factor in each of δ_{rot1}, δ_{trans} and δ_{rot2}. The following is the algorithm [1] used for the same:

Algorithm 5 Action Model(u_t, x_{t-1})

1: $\delta_{rot1} = atan2(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
2: $\delta_{trans} = \sqrt{(\bar{x}' - \bar{x})^2 + (\bar{y}' - \bar{y})^2}$
3: $\delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1}$
4: $\hat{\delta}_{rot1} = \delta_{rot1} - \epsilon_{\alpha_1 \delta^2_{rot1} + \alpha_2 \delta^2_{trans}}$
5: $\hat{\delta}_{trans} = \delta_{trans} - \epsilon_{\alpha_3 \delta^2_{trans} + \alpha_4 \delta^2_{rot1} + \alpha_4 \delta^2_{rot2}}$
6: $\hat{\delta}_{rot2} = \delta_{rot2} - \epsilon_{\alpha_1 \delta^2_{rot2} + \alpha_2 \delta^2_{trans}}$
7: $x' = x + \hat{\delta}_{trans} cos(\theta + \hat{\delta}_{rot1})$
8: $y' = y + \hat{\delta}_{trans} sin(\theta + \hat{\delta}_{rot1})$
9: $\theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$

where α_1, α_2, α_3, α_4 are robot specific error parameters and ϵ_{σ^2} denotes Gaussian noise with zero mean and σ^2 variance. These iterations are carried out at each time instance to give an estimate of the robot pose based on the previous pose and odometry data.

3) Sensor Model: Sensor models account for the inherent uncertainty in the robots sensors. The sensor model is defined as a conditional probability distribution $p(z_t|x_t, m)$, where x_t is the robot pose, z_t is the measurement at time t, and m is the map of the environment. The sensor used for the Botlab was a 360^o laser scanner from RoboPeak. The number of scans per cycle is denoted by K which is ideally equal to 360. The measurement at time t is denoted by:

$$z_t = \{z_t^1, z_t^2, ..., z_t^K\} \tag{3}$$

We approximate $p(z_t|x_t, m)$ by the sum of the individual measurement likelihoods:

$$p(z_t|x_t, m) = \sum_{i=1}^{K} p(z_t^k|x_t, m) \tag{4}$$

where z_t^k is an individual range value given by the laser scanner.

We implement a modified form of the Likelihood Field Model [1] in our sensor model, which uses the occupancy grid computed in II-C1 as the map. According to this map, the global co-ordinate space is divided into cells of dimension $0.05m$ x $0.05m$. We assume that the sensor position in the global co-ordinate system is the same as the robot pose at time t which is $x_t = (x\ y\ \theta)^T$. Any range value z_t^k which is greater than or equal to the maximum range of the sensor ($5m$ for our sensor) is considered invalid. We then project the end points of the valid ranges z_t^k onto the global co-ordinate space and check if that cell is occupied on the map. If it is indeed occupied, we assign maximum probability to that measurement. If it is not occupied then we check one cell each, ahead and behind it in the direction of the laser ray and assign it a probability value from a very narrow Gaussian distribution centered on the projected cell. The end result assigns higher weight to laser scans that actually terminate on occupied locations in the map and a very low weight to ones that are off by one cell. We then add these probabilities for every range value in a scan to give the cumulative weight of that scan. The algorithm is described below:

The value 0.0707 added or subtracted from the range value ensures that the computation is done for one cell ahead and one cell behind the projected cell respectively. $prob(dist, \sigma^2_{hit})$ computes the probability of $dist$ under a Gaussian distribution with zero mean and a very low variance of σ^2_{hit}.

Algorithm 6 Sensor Model(z_t, x_t, m)

$q = 1$
for all k **do**
  if $z_k^t \neq z_{max}$ **then**
    $x_{z_t^k} = x + z_t^k cos(\theta + \theta_{k,sens})$
    $y_{z_t^k} = y + z_t^k sin(\theta + \theta_{k,sens})$
    $x1_{z_t^k} = x + (z_t^k + 0.0707) cos(\theta + \theta_{k,sens})$
    $y1_{z_t^k} = y + (z_t^k + 0.0707) sin(\theta + \theta_{k,sens})$
    $x2_{z_t^k} = x + (z_t^k - 0.0707) cos(\theta + \theta_{k,sens})$
    $y2_{z_t^k} = y + (z_t^k - 0.0707) sin(\theta + \theta_{k,sens})$
    if $(x_{z_t^k}, y_{z_t^k})$ is occupied in map **then**
      $dist = 0$
    else if $(x1_{z_t^k}, y1_{z_t^k}))$ is occupied in map **then**
      if $(x2_{z_t^k}, y2_{z_t^k}))$ is occupied in map **then**
        $dist = min(((x_{z_t^k} - xi_{z_t^k})^2 + (y_{z_t^k} - yi_{z_t^k})^2)^{0.5})$
        $where\ i = 1, 2$
    else if $(x2_{z_t^k}, y2_{z_t^k}))$ is occupied in map **then**
      $dist = ((x_{z_t^k} - x2_{z_t^k})^2 + (y_{z_t^k} - y2_{z_t^k}))^{0.5}$
    $q = q + prob(dist, \sigma_{hit}^2)$
$return\ q$

4) Particle Filter: The Particle Filter recursively determines the current pose of the Maebot. It does this by determining the pose and probability of said pose for N number of guesses. Given the previous description, "guesses" are particles, "probabilities" are weights, and the poses are pose. Each of these particle's poses is generated via the Action Model described in II-C2. The Sensor Model, described in II-C3, calculates the pose for each of the proposed particles. Because the starting position is known, the particles of the Particle Filter are initialized with the starting pose, and with equal weight summing to one. For each iteration, the particles are first redistributed based on the weight of each particle. For example if you had ten particles, and one of them had a weight of .5, then five of the redistributed particles should be particles of that pose. After which, the poses and weights computed by the Action Model and Sensor Model are applied to the particles, and the Particle Filter then re-normalizes the weights to sum to one. The average of the highest weighted particles is then returned as the estimated pose. The process thus repeats, ad-infinitum [1].

D. Planning and Exploration

1) A Planning:* The A* path planning algorithm uses a map of weights and the estimated current position to determine the most optimum path to take. It then returns a path of poses which is sent to the motion controller, which physically maneuvers the robot to those poses.

Algorithm 7 Particle Filter Pseudo-Code

#Particles= N, Pose= P_i, Weight= W_i, $1 \leq i \leq N$
$P_i = P_0$, $W_i = \frac{1}{N}$
while true **do**
  Redistribute Particles
  Compute Particle Poses
  Compute Particle Weights
  Normalize Particles
  Return Estimated Pose

The A* path is cleverly computed by looping over each, consecutive lowest cost position on the weighted map, based on the maps weights and a heuristic equation, and by searching it's neighbors. For each accessible neighbor that has not been searched or visited before, the A* planner adds the position to a list of positions which is automatically sorted to place the lowest cost, of all previously searched, unvisited positions at the top. The algorithm then explores this top position, searches its neighbors, and thus the planner loops until it sees that it has found the goal position. It then backtracks each of the positions it took to get to the goal pose, which defines our path to take.

To accomplish this task, a new data type called a "node" was defined to store a weighted map pose, a world pose, a total cost, and a parent node. The heuristic for the A* was a simple Pythagorean Distance of the current position to the known goal position. Thus, the algorithm was encouraged to explore nodes closer to the end goal first. The key to a successful A* is the weighted map. Our weighted map recursively applied 5 to each position, every weight grid step from a known wall.

$$C_{i+1} = \frac{C_i}{1.6^i} \tag{5}$$

Where C_i is the cost of the current node i distance from the known wall. This exponential drop off created a very steep channel of weights between walls, such that the A* would be highly discouraged to plot paths anywhere close to walls.

2) Map Exploration and Escape: Exploration is by definition searching for the unknown. We use this concept to explore a map by first identifying the unknown areas and then exploring them. Unknown areas of the map are areas that the robot has not sensed or 'seen' and thus for which we have no information on whether they are free space or obstacles. This information is available to us through the global map. We note that an unknown area in the map is always bounded by either the edge of the map, or a known area. If there is free space adjacent to the unknown area we call it a frontier (see red lines in Figure 3 and we note that we can navigate to it and explore it. Also note that as the robot moves new areas of

the map become known and the frontiers are constantly being redefined.

Considering this, and in order to explore the environment it suffices then to iteratively drive the robot to any of the known frontiers, then check what frontiers remain unexplored and repeat these steps until no more frontier exist at which point we have fully explored the map. If the robot can navigate through all frontiers the map is fully explorable while if the robot cannot navigate through one or more frontiers, the map may or may not be fully explorable. The sequence in which frontiers are visited determines the efficiency of the algorithm to explore the map. However an optimal solution is not possible because at any point we only have partial information of the map, meaning we do not know the full global map until we have finished the exploration. We therefore use a heuristic to explore the map and decide to always visit the frontier that has the lowest A-star cost to visit. This approach balances both physical proximity and safety of the path to follow.

Once the full map is explored the problem statement directs the robot to visit the home position, then a key position and then a treasure position and guarantees that in that trajectory an exit door will open. This exit door, by merit of connecting with the unknown, will define at least one new frontier that we will visit later to exit the maze. We direct the robot to visit home, key and treasure positions without regard to any frontiers as required in the problem statement. After that we direct the robot to any new frontiers which will guide him through the exit of the maze.

E. Visualization

In order to be able to test the accuracy of each algorithm implemented and to compare it against the ground truth, we made a Graphical User Interface (GUI), which would read data from the Maebot and visualize it on the screen in real time. Vx [2], a visualization library designed for robotics applications was used for this. This library provides a wrapper around OpenGL to make GUIs quick and easy to create. The Vx objects were added to the world buffer and were visible on the GUI after swapping the buffer. This being a continuous process, the motion of the robot in the real world was reflected on the screen.

Various visualizations were implemented to test different parts of the algorithm. The robot pose at every time instance was mapped on the screen along with the path it followed to reach that point in order compare the odometry pose estimations and the SLAM pose estimations against the true poses from the Optitrack system. The occupancy grid was drawn as a gray-scale image in order to test the mapping algorithm. The particles generated from a combination of the action model, sensor model and the particle filter were plotted as arrows and were used to tune the error parameters for SLAM. The inflated map was tested by plotting the distance grid as a color image with obstacles drawn in black, close proximity of obstacles in red and free space in white. The frontiers and the calculated least cost path to reach a location were also drawn in order to check and tune the A* Planning and the Exploration algorithms.

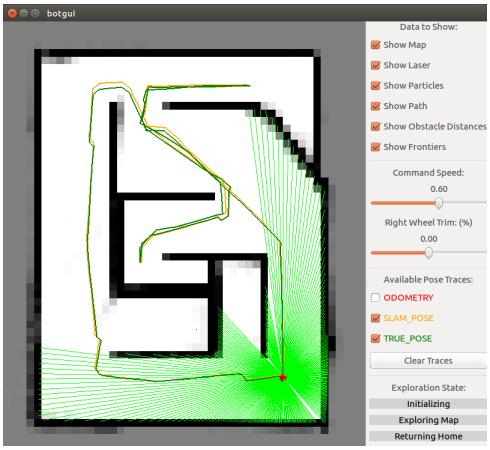

Fig. 5: Graphical User Interface

III. Experimental Results

The following section describes the results of various experiments carried out on the Maebot in order to test the performance of the algorithms implemented for maze exploration and escape, and also discusses the effectiveness or lack thereof of via said results.

A. Odometry

In order to test the performance of our odometry position estimate we drive a predefined route and compare the position estimates vs. ground truth values. To complete this test in a practical way we marked the initial position in the floor and travelled a closed loop of 2.4 meters using the graphical user interface open loop drive commands. The course had the shape of the number eight and it could be traversed twice in approximately 80 seconds, once driving forward and once driving backwards. At the end of the loop, the robot was stopped precisely on top of the starting point, i.e. ground truth position x=y=0 (+/- 0.5cm). Figure 6 shows the x and y odometry and gyrodometry position results at the end of a typical run. These values should have been zero hence they directly represent the error in the position estimate. In this case the Odometry error was in the range of 7 to 12cm while the both the Gyroscope and the Gyrodometry errors were approximately 1cm. We observe that Gyrodometry improves the Odometry

position estimate by a factor of 10x in this example (using Gyroscope data when Odometry differs by more than 0.10 radians). Similar improvement in the range of 5x to 10x were consistent in all runs.

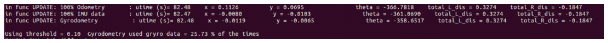

Fig. 6: Position estimates using 100% Odometry, using 100% Gryoscope orientation data and using the Grydometry approach with a treshold of 0.1rad.

Odometry is a valuable method to estimate position relative to a previous known position. Its open loop nature makes it subject to accumulating errors over time even if gyroscope data is used. Therefore it is unlikely that a mobile robot can get very far only counting on Odometry and thus it becomes necessary to have a feedback signal to better estimate position.

B. PD Control

The PD controller was slightly underdamped as oscillations would at times occur depending on the Maebot which was used. An example of such oscillations can be seen during the drive square test in Figure 7 below. Note how only the first turn experienced oscillations. The other two were mostly steady and smooth.

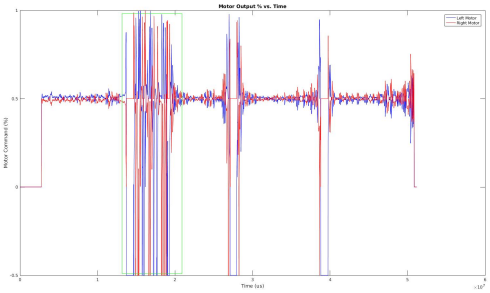

Fig. 7: Underdamped controller oscillations in green occurring after turning during a drive square test. Map from sample shown in Figure 8.

Such oscillations gave us some problems while navigating the final maze, as continuous high speed turning tended to rotate and shift the map estimated by SLAM. While the controller could have been tuned to remove the oscillations, working with multiple Maebots meant that the controller would need to be tuned for each one, as each performed slightly different. In the end, we were able to tune the controller to a generally acceptable setting, and then apply it with moderate success across all Maebots.

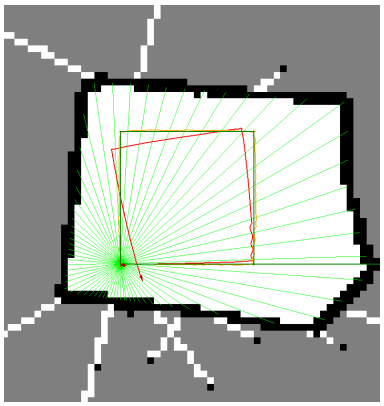

Fig. 8: Green - Commanded Path; Red - Odometry Data (without IMU); Yellow - SLAM data

C. SLAM

1) Mapping-Occupancy Grid: A discrete map of the environment proved to be an effective way to make the mapping problem tractable. The grid resolution needs to be carefully selected to balance the tradeoff in accuracy with the gain in tractability. The obstacle distance grid containing the collision risk associated with each cell in the map proved to be an effective way to enable quick planning while meeting the requirement of navigating safely. Furthermore the gradient of descent used to define how the risk value lower as the robot is further from the wall can be adjusted to strike the desired balance between shortest route and safest route.

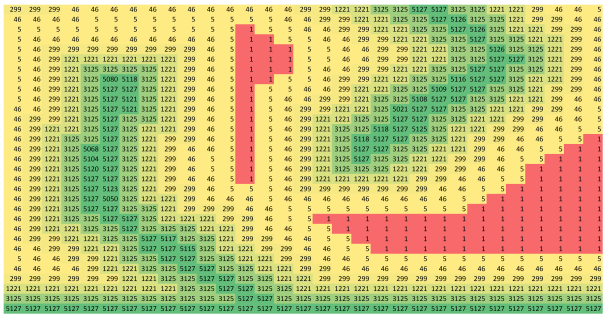

Fig. 9: Part of the obstacle distance grid. High values represent higher risk of collision while lower values represent lower risk of collision. A treshold is defined aboce which the cell is considered too risky and hence non-navigable.

2) Localization: The error parameters for the Action Model described in II-C2, α_1, α_2, α_3, α_4 were set by comparing the particle cloud visualized in the GUI with the ideal particle distribution [1] as well as comparing the estimated SLAM pose with the true pose from the log files. The values for the parameters were set as follows:

TABLE I: Action Model Error Parameters

Error Parameter	Value
α_1	0.05
α_2	0.05
α_3	0.001
α_4	0.0001

The variance value for the Sensor model described in II-C3 was set to 0.0009, which gave a probability of ≈ 0.05 for locations that were off by one cell, thus generating a very narrow Gaussian curve around the projected cell.

The Localization thus implemented by merging the Action Model, Sensor model and Particle Filter was compared against the Staff SLAM estimation as well as the Ground Truth generated from the Optitrack system. The following image plots the Staff, SLAM and True poses for the log file 'challenge.log' with only localization:

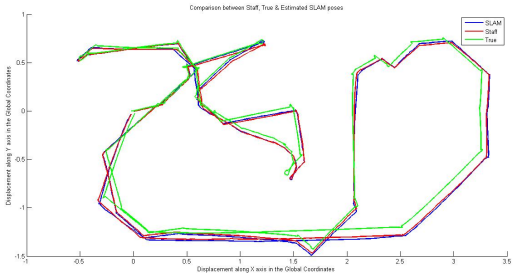

Fig. 10: Localization pose VS Staff pose VS True pose

The following were the errors in the estimated localization poses with respect to (w.r.t.) the Staff implementation and the True pose:

TABLE II: Error in Localization w.r.t. True & Staff poses

Comparison		x (meters)	y (meters)	theta (radians)
TRUE	max	0.18	0.08	6.22
	mean	-0.0248	0.0304	0.2768
	std	0.0567	0.0444	1.6376
STAFF	max	0.057	0.026	6.16
	mean	0.0123	-0.0089	0.0141
	std	0.0193	0.0132	0.7809

So far we had run mapping and localization as two separate processes. Now we combined the two to give the complete SLAM algorithm. The following image plots the Staff, SLAM and True poses for the log file 'challenge.log' with the mapping and localization combined:

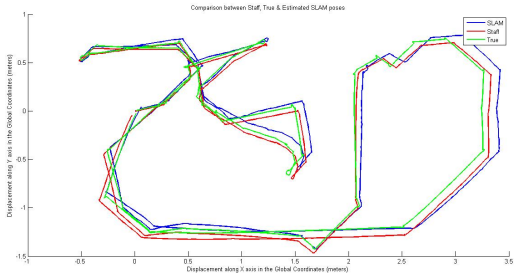

Fig. 11: SLAM pose VS Staff pose VS True pose

The following were the errors in the estimated final SLAM poses with respect to the Staff implementation and the True pose:

TABLE III: Error in final SLAM w.r.t. True & Staff poses

Comparison		x (meters)	y (meters)	theta (radians)
TRUE	max	0.19	0.14	6.21
	mean	0.0526	0.0228	0.3563
	std	0.0641	0.044	1.6329
STAFF	max	0.071	0.123	6.24
	mean	0.0152	0.0623	0.0639
	std	0.0297	0.0328	1.0362

The final map for 'challenge.log' was generated and is displayed below:

Fig. 12: Final Map

The error parameters of the Action model had to be tuned many times in the duration of the project. Also the case of backward motion had to be taken into account and the equation for δ_{rot1}, δ_{trans} and δ_{rot2} modified accordingly. As opposed to the original Likelihood Field Model which searches for occupied locations on the entire map for the projected end point of each laser ray in each scan, we opt for the more computationally efficient alternative of checking only one cell ahead and one cell behind the projected cell, since any farther than that would have a probability tending to zero. Also, we

add the probability values instead of multiplying them since the multiplication of even a few 0.05 probabilities generates an infinitesimal value which is read by the system as a zero. The probabilities thus assigned reflect the extent of match between the estimated pose and the map generated from the laser scans and are used as weights for the particles in the Particle Filter.

With this model we were still getting a rotation in our map due to small errors in the SLAM theta estimates. In order to prevent this, we set a threshold on the difference in angle between two instances of map update and simply chose to not update the map if the difference exceeded the threshold. This was just a temporary fix and a more robust way to fix this can be thought of.

D. Planning and Exploration

1) A Planning:* The A* algorithm was tested by navigating a small scale maze, and recording all sensor, SLAM, A*, and map data for later analysis. In Figure 13 shown below, the A* pose outputs for the many planned paths involved in exploring a maze are shown with the actual SLAM estimated poses in the context of the obstacle map. It is clear that combined with a sufficiently steep weight map, as aforementioned, the A* algorithm implemented was able to navigate effectively and at a safe distance from the maze walls.

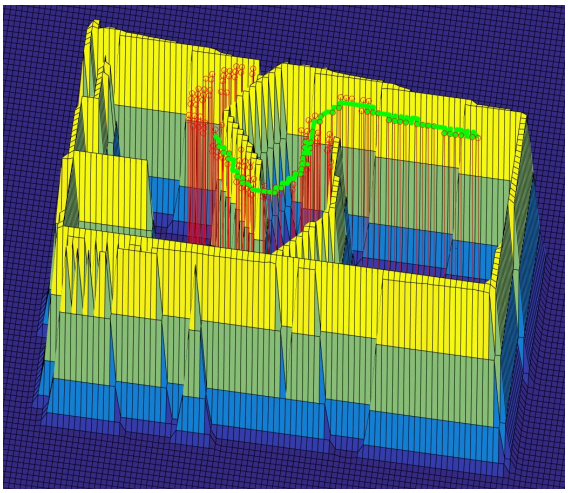

Fig. 13: Shown by red circles is all of the various paths output by A* during this exploration. The actual path of the robot via SLAM is given by the thick green line. Weights of the obstacle map are colour coated. Note that although there are 4 gradations in colour, there are 6 steps before the values are bottomed out at 1.

The speed of the A* algorithm is only extrapolated here from output pose data. To extrapolate the speed of the A* algorithm, the time of generation between two exploration states was found. Each exploration state conducts many A* searches. In the case analyzed, 33. By dividing the time between these explorations by the number of A* searches it was found that each A* search took around 3000 microseconds. This is very short as compared to those mentioned by Collin. This can be attributed to both an efficiently written A* algorithm and sufficiently steep walls, such that the A* had to do less iterations before it arrived at the goal pose.

Neither the speed nor effectiveness of the algorithm could be tested in comparison to Collin's test code, because the A* algorithm implemented in this paper differed in structure from his. Despite this, the tests described above and their ajoining figures aim to persuade the reader that these cases were not necessary when validating the effectiveness of the algorithm.

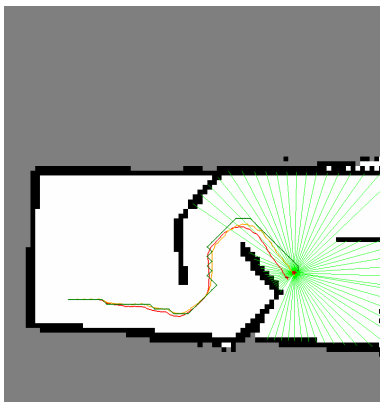

Fig. 14: A* path shown in dark green in the GUI output. SLAM pose is shown in yellow and odometry pose in red.

2) Map Exploration & Escape: Our problem definition is sequential in nature as the challenge requires the completion of one stage before proceeding to the next. Secifically the following states will happen in sequence: map exploration, travel to home position, travel to key position, travel to treasure position and lastly escape the maze. The transitions from one state to the next are well defined by either success or failure to complete the task. For example in exploration: no more frontiers exist marks a successful end while only unreachable frontiers remain would mark a failed end. Figure 15 shows summarizes the different states. Our implementation leverages the Finite State Machine proposed in the problem definition as an appropriate computational model to represent this problem.

Exploration is the first task that requires effective reasoning to iteratively select the frontier to be visited until the full map is explored. The cost of visiting a frontier is impacted by the our choice of the specific point that we want to visit, whether a point part of the frontier itself or a point near the frontier. Our implementation

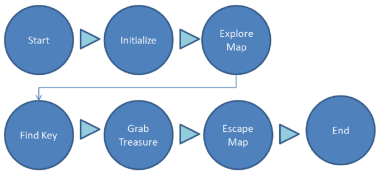

Fig. 15: Finite State Machine diagram.

uses the closest point to our current position, that is part of the frontier and that the robot can navigate to; Figure 3 shows an example of how the point is defined. This approach is simple to implement and proved to be robust driving the robot around the maze to explore the full map in the demonstration runs. Nevertheless we discovered during the demonstration runs that it does not handle well the case where a frontier exists parallel and adjacent to a wall. This happens because the frontier falls entirely in a high collision risk non-navigable area of the map. Should a different application require a guarantee that frontiers against walls are fully explored the algorithm can be modified by picking the closest navigable point to the frontier.

Once exploration was complete the remainder of the states were straight forward to complete using the SLAM and the PID to drive the robot to the desired location with the required precision. During the drive home, drive to key and drive to treasure states, the robot was simply given a navigable path to follow until it arrived. During the escape state the robot simply looked for frontiers defined by the newly open exit and drove to it which automatically ensured that it would crossed the exit line.

During demonstration runs our implementation fully explored the map, moved the required positions, found the exit and used it to escape. In the rare instances when a frontier against a wall was left unexplored, it typically got explored during the subsequent movements of the robot. Should a different application require a strong guarantee that frontiers against walls are fully explored the algorithm can be modified by picking the closest navigable point to the frontier.

IV. Conclusion and Future Work

Ultimately given a multifaceted and probabilistic framework, the end goal is the main objective and our algorithms and code were able to attain it. That being said there were still flaws in various parts of our project which could have been remedied. The motion controller could have been extended to integral control, and more finely tuned. With less jitter we could have moved faster, and thus updated SLAM more quickly. Additionally the planning, exploration, and state machine code could have been further optimized, faster, and with less logical flaws and bugs. All in all, this project could have been executed in many different, and more optimal ways. But in the end, each subsystem was able to balance out the flaws of another, and thus lead to a successful exploration, navigation, and escape of the maze.

References

[1] Dieter Fox, Sebastian Thrun, & Wolfram Burgard. (2005). Probabilistic Robotics. The MIT Press, Cambridge.
[2] https:/april.eecs.umich.edu/courses/eecs467_w14/wiki/index.php /Vx_Introduction
[3] Borenstein J, Feng L, ”Gyrodometry: A New Method for Combining Data from Gyros and Odometry in Mobile Robots” 1996.