Team 101 ROB550 BotLab Report

Hannah Baez, Nithin Donepudi, and Christian Leonard

***Abstract*—The goal of the ROB 550 BotLab project is to use knowledge gained during lecture and during lab time to apply acting, perception, and reasoning capabilities to an MBot-Mini. The robot will need to complete the deliverables for all three phases. Our robot has successfully completed the acting and perception portions of this lab, as well as the A* component of reasoning. Although we implemented an exploration component of reasoning, our robot was not able to explore the map.**

I. Introduction

This lab utilizes three main elements of robotics: acting, perception, and reasoning.

During the *Acting* portion of this lab, the goal is to implement a motion controller that utilizes odometry and IMU data. Odometry is defined as the use of data from motion sensors to estimate change in position over time. It is used in robotics by wheeled robots to estimate their position relative to a starting location. An inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers. These topics are is further discussed in the section *Motion and Odometry*.

During the *Perception* portion of this lab, the goal is to give our robot the ability to not only identify obstacles in the environment as it moves through space, but also the ability to keep track of its own location in relation to the obstacles the robot encounters. This is done by implementing what is discussed in the section *SLAM*.

During the *Reasoning* portion of this lab, the goal is to use A* (a path-planning algorithm, widely used for its efficiency) to successfully explore and navigate through an unknown maze. This is discussed in the section *Planning and Exploration*

This lab is completed using a MBot-Mini. The MBot-Mini consists of a Beagle Bone Blue, a Raspberry Pi, as well as a collection of Mobilebot and Botlab files. Motor control is handled by the Beagle Bone Blue, and commands are sent to the Raspberry Pi through an internet connection. The Beagle Bone Blue and the Raspberry Pi are connected through a Mobile Robotics Cape. The MBot-Mini has two wheels and a rear caster, each wheel motor has a magnetic wheel encoder. This robot is also equipped with a scanning 2D Lidar, and a MEMS 3-axis IMU.

II. Methodology

A. Motion and Odometry

The goal of this section is to give our robot "acting" capabilities. There are six elements to "acting" implemented in this lab, wheel speed characterizations for open loop controller development, open loop controller development with PID, odometry, gyro sensor fusion, a robot frame velocity controller, and a motion controller.

1) Characterize Wheel Speed for a Open Loop Controller: In order to characterize the wheel speed for an open loop controller, it was necessary to first define some required motor parameters. These motor parameters are robot specific. This was done by adding to the compiler #defines in `mb_defs.c`. The parameters defined include, wheel diameter, gear ratio, and encoder resolution. We then modified the `measure_motor_params.c` template code to be able to measure the loaded steady state wheel speed vs. input pulse-width modulation (PWM) measurements. This data was then output to a csv file and graphed using google sheets, this graph provides us with the slope and intercept values we use in the controller discussed below.

2) Open Loop and PID Controller: In `mb_controller.c`, using our open loop calibration, we built a controller that controls each wheel's velocity. We then built a second controller to control each wheel's speed using a PID controller. A Proportional-Integral-Derivative(PID) controller is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control. When building the PID controller it was necessary to tune it's Kp, Ki, and Kd values. In order to test that the PID controller had, in fact, been appropriately tuned, we had to compare the graph we acquired from running the script `plot_step.py` with the data logged by the LCM-Logger. The goal was to have the PWM Command (pwm cmd) as close to the Velocity Command (vel cmd) as possible, and to have both those values tied closely to the actually velocity as possible. The PID controller was considered valid when the velocity and pwm cmd overlapped as much as possible while experiencing as little noise as possible. For our PID we have used the "all things" controller design that was discussed in the lecture, we implemented a PD controller and added an Integrator and a feed-forward controller to complement it. We also added low-pass filters to all the necessary inputs, and we saturate the integrator and the PD controller so that they do not release gains beyond what is acceptable. All of our gains are loaded from an external .config file stored in the bin. The data flow in the filter is as follows, velocity is passed through a low-pass filter, this is then provided as input to the calibrator(the feed-forward controller), the integrator, and the PD controller, we use the sum of the outputs from these controllers to generate a PWM commands upon which we place another low-pass filter. The left command and right command are passed to respective motors, the encoders return feedback values which are used to generate error values. We also use a moving average filter over 30 values to get a curve with less noise.

3) Odometry: The odometry for the robot is calculated using the following equations:

$$\Delta\Theta = (\Delta_{SR} - \Delta_{SL})/b$$

$$\Delta d = (\Delta_{SR} + \Delta_{SL})/2$$

$\Delta\Theta$ defines the rotational change in the robot after it has moved from a point A to a second point B. Δd defines the distance between the center of the robot after it has moved from a point A to a second point B. The overall change in X and Y position was calculated using the following equations:

$$\Delta X = \Delta d * cos(\Theta + \Delta\Theta/2)$$

$$\Delta Y = \Delta d * sin(\Theta + \Delta\Theta/2)$$

These equations were implemented as functions in `mb_odometry.c`. These functions were used to calculate the robots position and orientation and enable dead reckoning with wheel encoders only. Dead reckoning is the process of calculating current position of some moving object by using a previously determined position, or fix, and then incorporating estimates of speed, heading direction, and course over elapsed time. We validated our odometry model using dead reckoning, we used UMBark (a particular method of dead reckoning where the idea is to drive around in a square), and we would make the corrections as necessary, incorrect baseline causes errors in turning, but not in straight line motion, Unequal wheel diameters would cause errors in straight line motion but would be fine while turning. No correction parameters were used in odometry.

4) Gyro Sensor Fusion: For this section we fused the gyro sensor with the odometry data. The gyro sensor data is gathered from the Inertial Measurement Unit(IMU) on the robot. We implemented the "gyrodometry" algorithm discussed in lecture to estimate the heading of the Mbot-Mini. The equation used to implement "gyrodometry" is as follows:

Algorithm 1 Gyrodometry

```
                $\Delta_{G-O} = \Delta\Theta_{gyro} - \Delta\Theta_{odo}$
if $|\Delta_{G-O}| > threshold$ then
    $\Theta_i = \Theta_{i-1} + \Delta\Theta_{gyro}$ else
    $\Theta_i = \Theta_{i-1} + \Delta\Theta_{odo}$
```

In these equations G and gyro represent gyrodometry, O and Odo represent odometry, the threshold = 0.001, Θ_i is a new theta, and Θ_{i-1} is the most recent old theta. Our implementation was tested by manually moving the robot by known distances and by turning it known angles.

5) Robot Frame Velocity Controller: In order to implement our final robot frame velocity controller, we designed it to take inputs of forward velocity (m/s) and turning velocity (rad/s) from an LCM message of type mbot_motor_command_t on the LCM channel MBOT_MOTOR_COMMAND_CHANNEL. Our controller used the following equation:

$$V = (V_R + V_L)/2$$

$$w = (V_R + V_L)/b$$

Here V represents forward velocity, ω represents angular velocity, V_r represents the right wheel velocity, V_l represents the left wheel velocity, and b represents the wheel base diameter. We then use these equations to find V_l and V_r and apply a moving average filter that takes in 30 readings. Our controller uses the equations above to calculates the left and right wheel speeds to send as commands to the wheel speed controllers.

6) Motion Controller: In order to implement our motion controller, we used the template given to us in botlab under the name drive_square.cpp and implemented our controller in `motion_controller.cpp` which ran on the RPi. We set our way-points of type pose_xyt_t as necessary and coded the straight maneuver controllers and turn controllers respectively. Our motion controller takes the robot_path_t as an input on the channel CONTROLLER_PATH and execute the trajectory until the robot reaches the final way-point. In the straight maneuver controller, we provide a transnational and an angular velocity to help keep the bot on the specified heading, in the turn controller we provide only an angular velocity. We provide minimum and maximum thresholds for the transnational and angular velocities and set them to these defaults if either the maximum is exceeded or the minimum is preceded when calculated using the proportionality constants we provide. We use these proportionality constants to come to a steady halt and avoid face-planting. This controller takes the form of an RT controller where it calls the turn maneuver controller and then the straight maneuver controller, and then re-starts in the state machine.

7) Combined Implementation: To run our combined implementation we modified drive_square.cpp to drive the maze set up in the lab. We used measurements provided in the lab document to create the way-points. We ran the robot at a slow speed of 0.2m/s pi/4 rad/s and a high speed of 0.8m/s, pi rad/s.

B. Simultaneous Localization and Mapping (SLAM)

Simultaneous Localization and Mapping (SLAM) makes up Part 2 of this lab project. The components that make up SLAM include mapping, Monte Carlo localization, as well as the action model, sensor model, and particle filter.

1) Mapping: The first step in implementing the SLAM function is to be able to define an occupancy grid. An occupancy grid describes the space around the robot via the probability that each cell of the grid is occupied or free. The occupancy grid is a matrix of numbers that holds the probability of each cell. For the purposes of this project, we used a grid with that maps to 5cmx5cm squares for each cell, with each cell's log-odds values are integers in the range [-127,127]. The occupancy grid was built in the `slam/mapping.cpp` and `slam/mapping.hpp` files.

To measure the probability of each cell, the robot uses ray casting – a method of using light rays to determine the distance of obstacles from the source of that light ray. In this instance, the light rays are cast from a Light Detection and Ranging sensor, or LiDar sensor. LiDar allows for many lasers, or light rays, to be broadcast from a single point in a 360 degree field of view. LiDars are often used as a method of measuring distances in conjunction with ray casting.

To implement ray casting effectively, we implemented **Breshenham's Algorithm:**

Algorithm 2 Breshenham's Algorithm

```
dx = | $x_1 - x_0$ | $dy =| y_1 - y_0 \mid sx = x_0 < x_1?1 : -1sy = y_0 < y_1?1 : -1err = dx - dyx = x_0y = y_0$
while $(x \neq x_1||y \neq y_1)$ do
    update cell odds of x and y $error_2 = 2 * errorif(error_2 \geq -dy)error = error - dyx = x + sx$
    if $(error_2 \leq dx)error = error + dxy = y + sy$
```

From there, we ran SLAM on the provided obstacle_slam_10mx10m_5cm.log file in `--mapping-only` mode, meaning the program only executes the mapping functionality. We used the ground-truth pose provided in that log file to construct an occupancy grid of said file.

2) Monte Carlo Localization: Monte Carlo Localization (MCL) focuses more on SLAM's localization functionality. The MCL itself is a localization algorithm made up of three parts: the action model, the sensor model, and the particle filter. For this section, each instance of SLAM is run in `--localization-only` mode, a function that isolates the localization properties of SLAM, ignoring the previously described mapping functions.

a) Action Model: The function of the action model is to predict the future position of the robot. The action model uses odometry and feedback from the encoders to calculate a best guess of where the robot may travel to next. We implemented the Odometry Action Model algorithm in the `slam/action_model.cpp` and `slam/action_model.hpp` files. Our action model used $K1 = 0.01$ and $K2 = 0.01$ as our uncertainty parameters, we found these through trial and error.

Algorithm 3 Odometry Action Model

```
procedure SAMPLE_MOTION_MODEL_ODOMETRY(u$_t, x_{t-1}$)
    $\delta_{rot1} = a\tan 2(y' - y, x' - x) - \Theta\delta_trans = \sqrt{(x - x')^2 + (y - y')^2}\delta_{rot2} = \Theta' - \Theta - \delta_{rot1}$
    $\delta_{rot1} = \delta_{rot1} - \textbf{sample}(\alpha_1\delta^2_{rot1} + \alpha_2\delta^2_{trans})$
    $\delta_{trans} = \delta_{trans} - \textbf{sample}(\alpha_3\delta^2_{trans} + \alpha_4\delta^2_{rot1} + \alpha_2\delta^2_{trans})$
    $\delta_{rot2} = \delta_{rot2} - \textbf{sample}(\alpha_1\delta^2_{rot2} + \alpha_2\delta^2_{trans})$
    x' = x + $\delta_{trans}\cos\Theta + \delta_{rot1}y' = y + \delta_{trans}\sin\Theta + \delta_{rot1}\Theta' = \Theta + \delta_{rot1} + \delta_{rot2}$
    return x$_t = (x', y', \Theta')$
```

b) Sensor Model: The sensor model is the method by which the robot keeps track of the cells around it and whether or not those cells are free. It relies on ray casting to measure the cells and obstacles nearby to update the occupancy grid as the robot maneuvers through the environment. We utilized the Simplified Likelihood Field algorithm for the sensor model, implemented in the `slam/sensor_model.cpp` and `slam/sensor_model.hpp` files.

Algorithm 4 Likelihood Field

```
procedure LIKELIHOOD_FIELD_RANGE_FINDER(z$_t, x_t, m$)
q = 1 for $a$ do
 ∟ 1
1 k  if $z_t^k \neq z_{max}$ then
    $x_{z_t^k} = x + x_{k,sens}\cos\Theta - y_{k,sens}\sin\Theta + z_t^k\cos\Theta + \Theta_{k,sens}$
    $y_{z_t^k} = y + y_{k,sens}\cos\Theta - x_{k,sens}\sin\Theta + z_t^k\sin\Theta + \Theta_{k,sens}$
    $dist = min\left(_{x',y'}\sqrt{(x_{z_t^k} - x')^2 + (y_{z_t^k} - y')^2|}\right)\langle x', y', \rangle$
     occupied in m
    q = q * ($z_{hit} * \textbf{prob}(dist, \sigma_{hit}) + \frac{z_{rand}}{z_{max}}$returnq
```

c) Particle Filter: A particle filter uses a set of particles to represent noise in the environment as the robot moves. Similar to the action model, the particle filter is used to estimate the future position of the robot, but differs due to the method of future estimations. The particle filter re-samples the distribution of the particles, updating the weight of each particle during every re-sampling. Particle re-sampling requires that the particle filter draw a new set of particles each time, but favoring the past set of particles' whose weights are higher. The higher the particle's weight, the more likely it will be re-implemented in the new set of particles. The particles' whose weights are below a certain threshold are likely to dropped in favor of a new, random particle.

For this project, we used Low Variance Resampling for our particle filter.

Algorithm 5 Low Variance Resampling

```
$X_t = \varnothing r = rand(0; M^{-1}) c = w_t^{[1]} i = 1$
for $m = 1toM$ do
    
$U = r + (m-1) * M^{-1}$ while $U > c$ do
$i = i + 1$
$c = c + w_t^{[1]}$

$X_t = X_t + x_t^{[i]}$
return $X_t$
```

This was implemented in the `slam/particle_filter.cpp` and `slam/paricle_filter.hpp` files.

3) Combined Implementation: To demonstrate our SLAM system, we began by mapping the maze provided by the lab. To drive through the maze we used click to drive in botgui. We then saved the map and the .log file for the map and log files created.

C. Planning and Exploration

1) A Path Planning:* The obstacle distance grid identifies obstacles and adds a buffer zone to every obstacle to ensure we do not get to close to any one of them. To draw our obstacle distance grid, we implemented code from the live coding session. We then ran and passed all three tests. This was called in our A*

Algorithm 6 A*

```
Input: starting position, goal(n), h(n), expand(n) Output: path if goal(start) = true then
    return makePath(start)
open ← $startclosed \leftarrow \varnothing$while
ort(open) n ← $open.pop() kids \leftarrow expand(n)$for all kid in kids do
    k
id.f ← $(n.g + 1) + h(kid)$if goal(kid) = true then
    return makePath(kid)
if kid $\cap closed = \varnothing$ then
    $open \leftarrow kid$
return $\varnothing$
```

2) Map Exploration: Note: We were unable to complete the code for this portion of the lab, we ran out of time, but we did develop a strategy that given more time, we would have implemented. The frontiers and localization strategies are listed below.

Finding frontiers: The first part of exploration is to find the frontiers. Our strategy to find the frontiers starts with the algorithm provided in the template lab code. Frontiers are the borders between free space and unexplored space. We start by adding all the frontiers to an array of frontiers. Then we find the frontier closest to our position by looking at the cells in each frontier to find the one closest to us. We then pull that frontier from the list and find the center cell of the frontier. We then plan and execute a path to that frontier. Then repeat the process again. We continue this until the array of frontiers is empty. The center of each frontier that we have driven to is saved into a path and we use this path to drive back to the start position. To complete this chain of events we utilize the state machine in `exploration.cpp`

Localization: The second part of map exploration is localization. We spread a large number of particles, for example 3,000 particles over the entire free space. We make minor movements, for example, rotating in position, in order to get a better idea of our current location. By rotating in place we will be safe from hitting any obstacles. These particles will form clusters in various regions of the map. As the robot begins to move through the space, these clusters will eventually form one single cluster giving us our actual position in the maze.

III. Results

A. Motion and Odometry

1.1.1 Motor Calibrations

Motor calibrations for Robot 1:

- Left Motor Polarity: -1
- Right Motor Polarity: -1
- Left Encoder Polarity: -1

- Left Motor Polarity: 1
- Max Forward Velocity: 0.8 m/s
- Max Turning Velocity: 2.5 m/s

Motor calibrations for Robot 2:

- Left Motor Polarity: -1
- Right Motor Polarity: 1
- Left Encoder Polarity: -1
- Left Motor Polarity: 1
- Max Forward Velocity: 0.8 m/s
- Max Turning Velocity: 2.5 m/s

Motor calibrations for Robot 3.

- Left Motor Polarity: 1
- Right Motor Polarity: 1
- Left Encoder Polarity: -1
- Left Motor Polarity: -1
- Max Forward Velocity: 0.8 m/s
- Max Turning Velocity: 2.5 m/s

1.1.2 Motor Speeds vs Correlating PWM Values

The following is a chart displays the motor speeds vs the correlating PWM values as the robot moves forward.

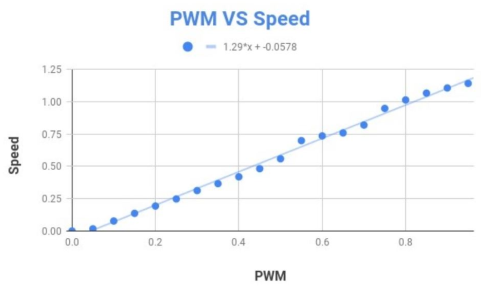
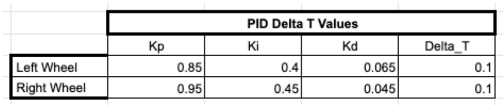


Fig. 1: Motor Speed vs PWM for the loaded wheels

1.2 Parameters for controller

The following values relate to the parameters of our PID controller:



	PID Delta T Values			
	Kp	Ki	Kd	Delta_T
Left Wheel	0.85	0.4	0.065	0.1
Right Wheel	0.95	0.45	0.045	0.1

Fig. 2: PID Paramaters for the PID Controller

0.5

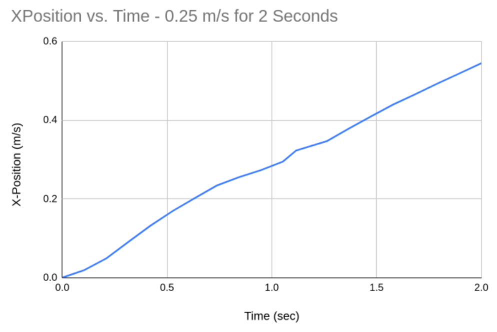


Fig. 3: 0.25 m/s for 2s

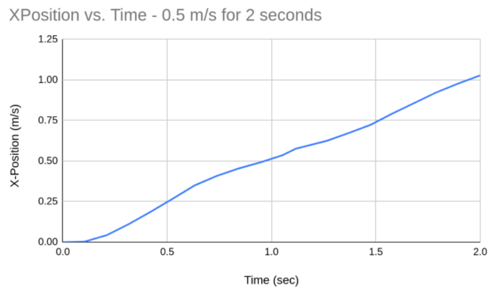


Fig. 4: 0.5 m/s for 2s

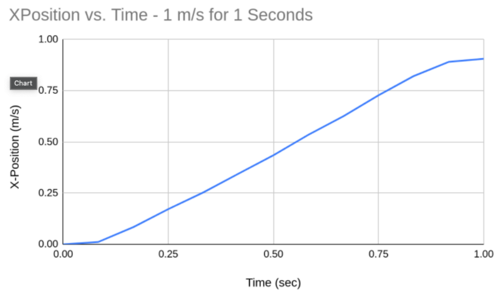


Fig. 5: 1.0 m/s for 1s

1.5.1 Robot x Position vs. Time - for the following inputs:

1.5.2 Robot Frame Heading vs Time - for the following step inputs:

0.5

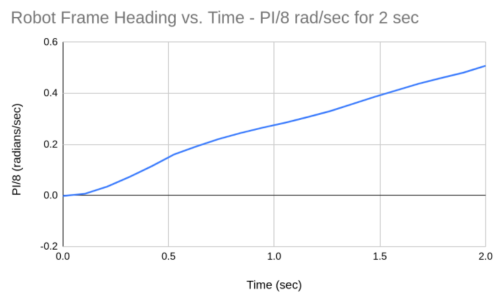


Fig. 6: π/8 rad/s for 2s

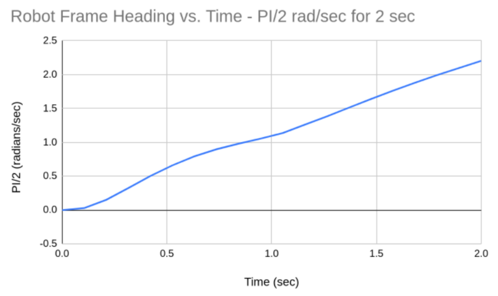


Fig. 7: π/2 rad/s for 2s

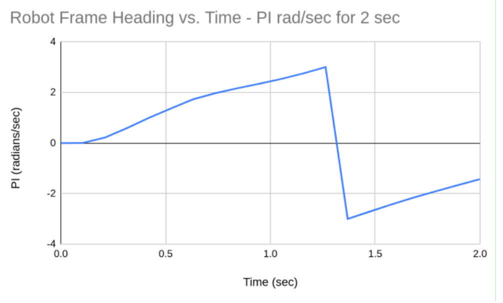


Fig. 8: π rad/s for 1s

2) *Motion Controller:*

The following is the Dead Reckoning Estimated Pose as the robot is commanded to drive 1m square 4x

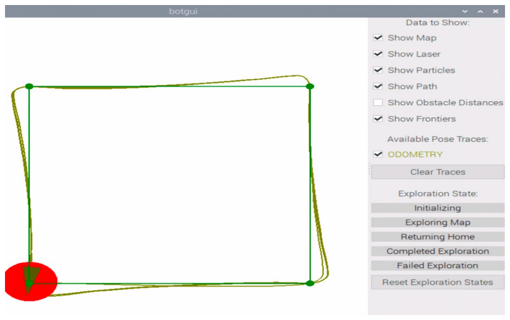


Fig. 9: Dead Reckoning Estimated Pose

Robot's linear and rotational velocity as it drives one loop around the square

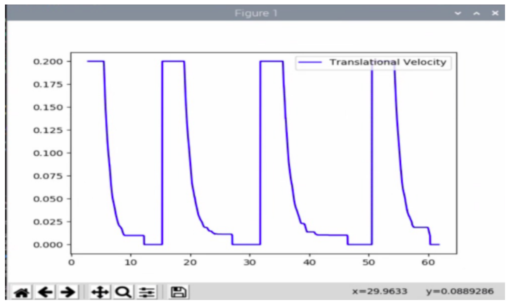


Fig. 10: Linear Velocity of the Robot

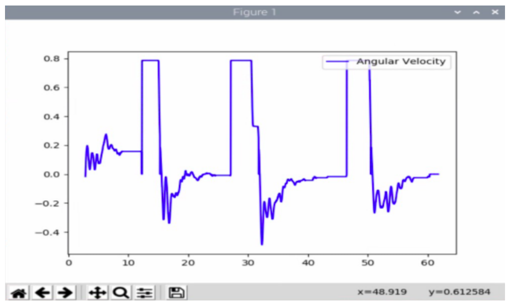


Fig. 11: Angular Velocity of the Robot

B. Simultaneous Localization and Mapping (SLAM)

1) *Mapping:* The following is a map from the log file obstacle_slam_10mx10m_5cm.log

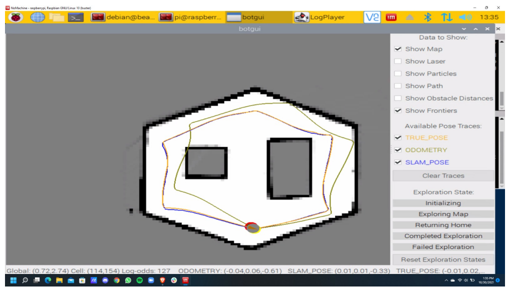
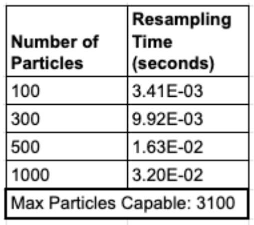


Fig. 12: Map of 'obstacle_slam_10mx10m_5cm.log'

2) *Monte Carlo Localization:*

a) *Sensor Model:*

The following is a table of the time it takes to update the particle filter for 100, 300, 500 and 1000 particles. The table also features the nearest estimate as to the maximum number of particles the filter can generate.



Number of Particles	Resampling Time (seconds)
100	3.41E-03
300	9.92E-03
500	1.63E-02
1000	3.20E-02
Max Particles Capable: 3100	

Fig. 13: Particle Filter Update Rates and Estimate

b) Particle Filter: The following image is a merged set of screenshots with each set of 300 particles coming from in individual photo of the progression of the particle filter on the log file. Photos were taken at each midpoint and turn along the 1x1m square.

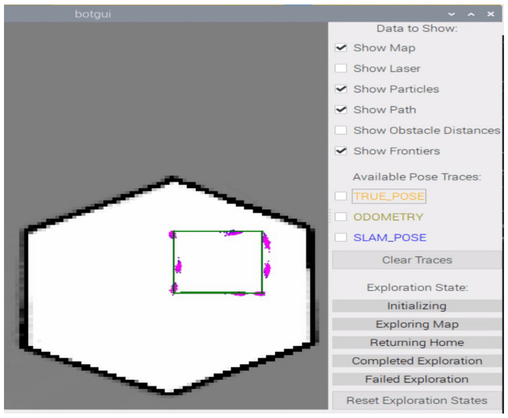


Fig. 14: 300 Particles Throughout Different Stages of 'drive_square_10mx10m_5cm.log' Playback

The following is a block diagram of how the different functions of SLAM work together

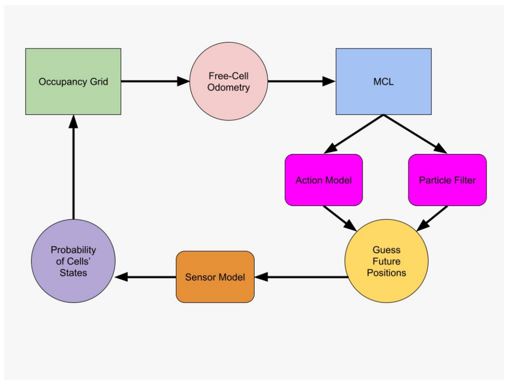


Fig. 15: SLAM Block Diagram

The following is a plot showing how the pose error differs between the SLAM and Odometry poses

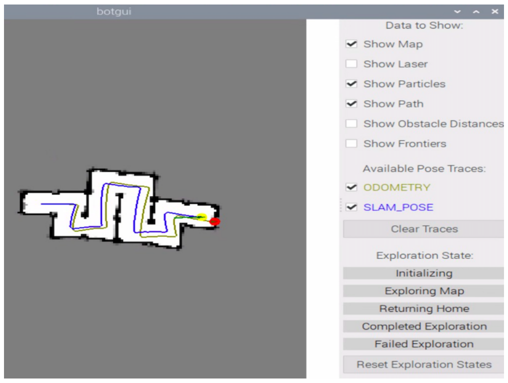
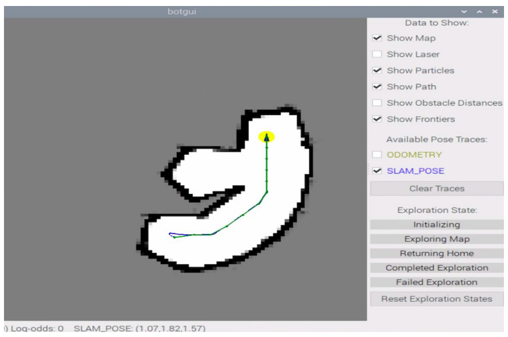


Fig. 16: Pose Error Between SLAM and Odometry Poses

c) SLAM RMS Error: Comparing the estimated poses from our SLAM system against the ground-truth poses in `obstacle_slam_10mx10m_5cm.log`., we can estimate the accuracy of your system using RMS Error, which in our case is 0.0196652.

C. Planning and Exploration

1) A Path Planning:*
The following figure shows a for the robot to take with the actual path driven by the robot overlayed on top



```
) Log-odds: 0    SLAM_POSE: (1.07,1.82,1.57)
```

Fig. 17: A* Planned Path vs Actual Path Driven

Statistics on your path planning execution times for each of the astar test files

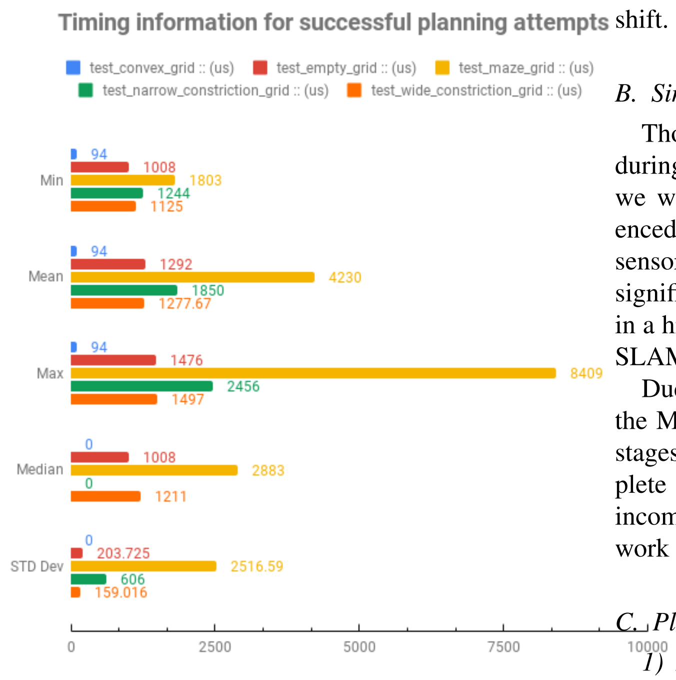


Fig. 18: Timing Statistics for A* Program Executions

IV. Discussion

A. Motion and Odometry

1) Open Loop and PID Controller: Although it took a great amount of time and trials to accurately tune the PID, the final result seems to provide accurate correlation between the command and robot performance.

2) Odometry: There is a hardware defect with the main robot's encoders that results in an error that only impacts the robot's ability to turn. When driving straight, the encoders are highly accurate, as expected. However, when turning 90 degrees, though the expected value of the encoder should be somewhere near 1.57, the encoder value on the robot reads roughly 1. Also, while rotating, only the right wheel of the robot turns. This results in a slight shift to the right while rotating.

3) Gyro Sensor Fusion: Since the odometry was off, in the gyrodometry, we compensate for the odometry by setting the threshold very low and running purely off the gyro sensor. The resulting gyrodometry works perfectly.

4) Robot Frame Velocity Controller: Due to time constraits after being hung up on a hardware issue, it was recommended that we partially skip this section to make up for lost time. Because of this, we use only a moving-average filter without adding the additional controller.

5) Motion Controller: The straight maneuver controller compensates for the error that occurs due to the turning issue within odometry. This results in the robot coming back to the intended path despite the original shift.

B. Simultaneous Localization and Mapping (SLAM)

Though we encountered a great deal of resistance during the Motion and Odometry phase of the project, we were able to make up for it in SLAM. We experienced many issues while coding each the action model, sensor model, and particle filter – these slowed us down significantly – but eventually fixed each of these to result in a high-speed, accurate SLAM model. Each element of SLAM is now completely functional.

Due to the combination of complications faced in both the Motion Odometry section, as well as the beginning stages of SLAM, we unfortunately were unable to complete A* to the extent we would have liked. Though incomplete, the sections of the project that we did get to work very well in combination.

C. Planning and Exploration

1) A Path Planning:* Our A* path planning implementation successfully completed all 6 tests required. The success times for all 6 tests were acceptable, accept for the `test_maze_grid` where the max time is extremely high.

2) Map Exploration and Localization: As stated in the methodology, we were unable to complete the code for this portion of the lab, we ran out of time. We ran out of time due to the hardware and coding issues stated above. Our code was often correct but hindered by a small hidden syntax error or a hardware error that would carry on for days. Although despite these issues, we did develop a strategy that given more time, we would have implemented. We have described the strategies we would have implemented in the methodology section of this report. We have no results to discuss for this portion.

References

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2006. [Online]. Available: http://www.probabilistic-robotics.org/

[2] M. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Wiley, 2005. [Online]. Available: https://books.google.com/books?id=wGapQAAACAAJ