ROB 550 : BotLab Project Report

John Brooks Kevin Choi and Ripudaman Singh Arora

Abstract— **The BotLab project was an introduction to odometry and SLAM in two dimensions. We utilized the two wheeled autonomous robot platform, the MaeBot. Dur ng this lab we implemented a basic encoder based odometry system, a more advanced Gyrodometry system, a PID controller, and A* path planning. We utilized an off the shelf SLAM implementation. In the end our robot was able to autonomously navigate through a novel environment with no prior knowledge, and no human input.**

I. INTRODUCTION

The BotLab project was an introduction to the basics of odometry, SLAM, and path-finding. It focused on the simple two wheeled autonomous platform, the MaeBot. The Maebot is a simple robot which packs a wide array of sensors which allow it to autonomously navigate environments. The project was divided into two major parts. In the first part, we focused on implementing basic odometry for a two wheeled robot. We enhanced this odometry with the use of a gyroscope to implement Gyrodometry. Once we had odometry we implemented a PID speed controller and calibrated our odometry readings. In the second part, of the BotLab project we utilized the Breezy-SLAM library to enable simultaneous location and mapping capabilities using the LIDAR range finder on the MaeBot. We implemented the A* search algorithm to plan paths through the maps generated by Breezy-SLAM. Using these paths we implemented a simple way-point following algorithm which allowed the MaeBot to navigate a simple maze like environment.

II. METHODS & MATRERIALS

A. BotLab - Part I

1) Sec 1.1 : Setting up the Maebot: Wireless communication between the Maebot and laptop enables the user to issue manual commands to the robot, observe sensor data, and implement processor intensive algorithms on the laptop rather than the Maebot. We use the LCM toolset and libraries shuttle data between the two clients. We create instances of Procman deputy on both client nodes to receive messages and execute commands, and an instance of Procman sheriff on the laptop to issue start and stop commands to the deputies. Setting up a bot-lcm-tunnel between the deputies enables direct communication without having to broadcast LCM messages to the entire network. LCM messages from the Maebot contain sensor data, which is used for mapping and path planning, and outgoing messages from the laptop relay guidance commands to the Maebot.

2) Sec 1.2 : 2-wheeled robot odometry: The Maebot uses odometry to locate itself with respect to the world frame, where measurements from the wheel encoders are translated into location and heading. Odometry is generally used as a rough estimate of the robots position in the world. It is often susceptible to error and so other techniques are needed to provide a more accurate estimate. Successive ticks of the encoders are translated into the distance traveled by left (D_L) and right(D_R) wheels for a given time step, from which the change in heading $\Delta\theta$ and ΔD position are calculated.

$$\Delta D_{L,R} = \frac{(encoder_{L,R} - encoder_{n-1}) \cdot \pi \cdot \phi_{L,R}}{gear\ ratio\ \cdot ticks\ per\ revolution} \quad (mm)$$

$$\Delta\theta_{wheel} = \frac{\Delta D_R - \Delta D_L}{Axle\ Length} \quad (rad)$$

$$\Delta D_{avg} = \frac{\Delta D_R + \Delta D_L}{2} \quad (mm)$$

The global x and y location and heading θ are calculated from the previous position and heading θ'_{global}.

$$\theta_{global} = \theta'_{global} + \Delta\theta_{wheel} \quad (rad)$$

$$x_{global} = x'_{global} + \Delta D_{avg} \cdot cos(\theta_{global}) \quad (mm)$$

$$y_{global} = y'_{global} + \Delta D_{avg} \cdot sin(\theta_{global}) \quad (mm)$$

3) Sec 1.3 : Maebot gyrodomerty: Using the angular velocity from the Maebot's gyroscope to calculate heading is appropriate during large wheel slip conditions or when the Maebot is picked up and rotated. Gyrodometry is one technique used to refine the position estimate given from pure odometry (described above). The Maebot obtains angular velocity from the gyro and translates it into the instantaneous change in heading using the following calculations:

$$dt_{gyro} = t_{gyro} - t'_{gyro} \quad (sec)$$

$$\Delta\theta_{gyro} = \omega_{gyro} \cdot dt_{gyro} \quad (rad)$$

Our algorithm compares the differences between $\Delta\theta_{gyro}$ and $\Delta\theta_{wheel}$ II-A.2. If the absolute difference is larger than a $gyro_{threshold}$ (0.01 rad) the Maebot switches over gyroscope's measurement for calculating θ_{global}, where θ'_{global} is the previously calculated odometric heading. For a given time step dt_{gyro}:

```
if $|\Delta\theta_{wheel} - \Delta\theta_{gyro}| > threshold_{gyro}$
then $\theta_{global} = \theta'_{global} + \Delta\theta_{gyro}$
else $\theta_{global} = \theta'_{global} + \Delta\theta_{wheel}$
```

The gyroscopic measurement of heading cannot be solely relied upon as there is drift over time. To calibrate the $gyro_{threshold}$ the Maebot is set to turn on the ground at a certain speed and the threshold is changed so that the Maebot relies on the odometry reading calculated by the encoder. In our implementation, the LCM message will indicate if the Maebot is using gyrodometry. Throughout our experiments did not notice gyrodometry to be useful as the robot rarely indicated that gyrodometry was in use; we accelerated slowly and did not see slip conditions nor did we handle the robot while it was traversing its path.

4) Sec 2.1 & 2.2 : Implementing & testing PID controllers for robot wheel movements: With an open loop controller, the Maebot cannot travel in a straight line over travel long distances nor correctly turn to a desired heading because the left and right motors exhibit different dynamics and the wheels see different ground conditions, thus a closed loop controller is required to regulate wheel movements. We implement a PID controller (see Fig. 1) that tracks to a desired set-point linear velocity for each wheel. A trim value is added to the motor input μ_0 for a nominal input power, such that the PID controller is adding or subtracting from the trim value.

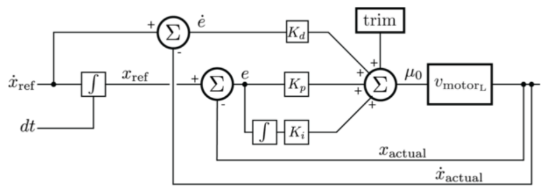


Fig. 1. PID controller for velocity

We begin the tuning process by setting the K_i and K_p terms to zero so that a initial K_d term, which affects the velocity error, can be found; we evaluate the K_d term based on the Maebot's speed given a desired velocity. At this point the Maebot still has issues driving in a straight line so we must add in the K_p term which helps with small corrections in heading as it veers off the straight line path. Optimizing the K_i term allows our Maebot to correct long term heading errors which results in ± 6 cm error to the left or right of the straight line path over a traveled distance of one meter, as seen in our video [1]. Our final controller gains are:

$$K_p = 0.003,\ K_i = 0.00001,\ K_i = 0.00045$$

Changing heading direction is accomplished by multiplying the left and right motor inputs μ_0 by 1 or -1 depending on the turning direction. For example, to turn right the motor input for the right motor is multiplied by -1 so that the right wheel spins counter-clockwise as the left wheel spins clockwise. The whole of our PID script acts on commands given by ground control (ie. move forward 1 m, turn 90 ° clockwise) and is parametrized by desired distance, desired heading, angular velocity, and linear velocity; these commands are then fed through the actual PID controller as the reference input velocity $\dot{x}_{ref}$.

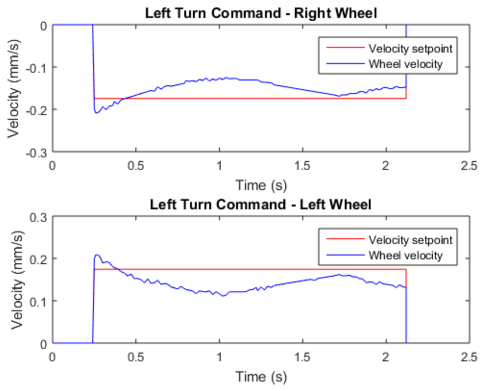


Fig. 3. Plot showing step response to left turn input command

The controller's response to step inputs are seen in Fig. 2 and 3 as well as in our video [1].

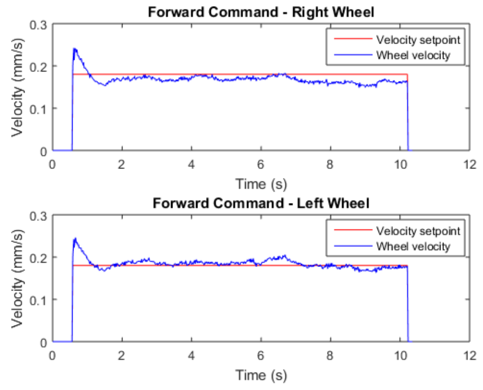


Fig. 2. Plot showing step response to forward input command

5) Sec 2.3 : Odometry calibration: Errors in wheel base and diameter measurements affect the accuracy of the odometry but we can characterize correction factors using experiments described in [[3]]. Type A and B errors can be derived from odometric and ground truth end point position of the robot after it has driven in a counter clockwise and clock wise square multiple times. Type A error represents error in the wheel base measurement which reduces or increases turning in both directions; for example, a larger wheel base will cause the Maebot under rotate. Type B error represents error in wheel diameter measurements and will cause over rotation in one direction and under rotation in the other. We use E_b to characterize Type A error and E_d to characterize Type B error, and assume in our calculations that Type A error only affects turning and Type B only affects straight line motion.

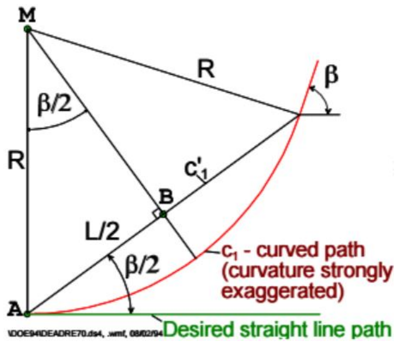

M
R
β
β/2
R
C'1
B
L/2
β/2
C1 - curved path (curvature strongly exaggerated)
A
Desired straight line path

Fig. 4. Geometric relations for finding radius of curvature [3]

To characterize our errors, we drive the Maebot around a one meter by one meter square in the clockwise and counter clockwise direction then calculate the center of gravity of our error clusters:

$$\epsilon_x = x_{meas} - x_{odo}, \quad \epsilon_y = y_{meas} - y_{odo} \quad \epsilon_\theta = \theta_{meas} - \theta_{odo}$$

$$X_{c.g.cww/cw} = \frac{1}{n}\sum_{i=1}^{n} \epsilon_{x_{i,cw/ccw}}$$

Rotational error imparted during a turn is denoted by α and rotational error imparted during straight line motion is β, with the following relationship to the endpoint in the x-direction:

$$-2L\alpha - 2L\beta = X_{c.g.cw}$$

We can use either x or y coordinates of the end point to calculate β:

$$\beta = \frac{X_{c.g.cw} - X_{c.g.ccw}}{-4L}\frac{180^\circ}{\pi}$$

$$R = \frac{L/2}{sin(\beta/2)}$$

Type B error is the relationship ration between the two wheels which causes the Maebot to turn a radius R during straight line motion.

$$E_d = \frac{D_R}{D_L}\frac{R + b/2}{R - b/2}$$

Now we can compute actual wheel base and Type A error:

$$\alpha = \frac{X_{c.g.cw} + X_{c.g.ccw}}{-4L}\frac{180^\circ}{\pi}$$

$$b_{actual} = \frac{90^\circ}{90^\circ - \alpha} b_{nominal}$$

$$E_b = \frac{90^\circ}{90^\circ - \alpha}$$

The correction factors that are applied to the calculations of ΔD_R and ΔD_L:

$$c_L = \frac{2}{E_d + 1} \quad c_R = \frac{2}{(1/E_d) + 1}$$

$$\Delta D_{R,corrected} = \Delta D_R * c_R$$

$$\Delta D_{L,corrected} = \Delta D_L * c_L$$

Our calculated values for calibration is as follows:

$$E_d = 0.99 \quad E_b = 1.04 \quad c_L = 1.01 \quad c_R = 0.99$$

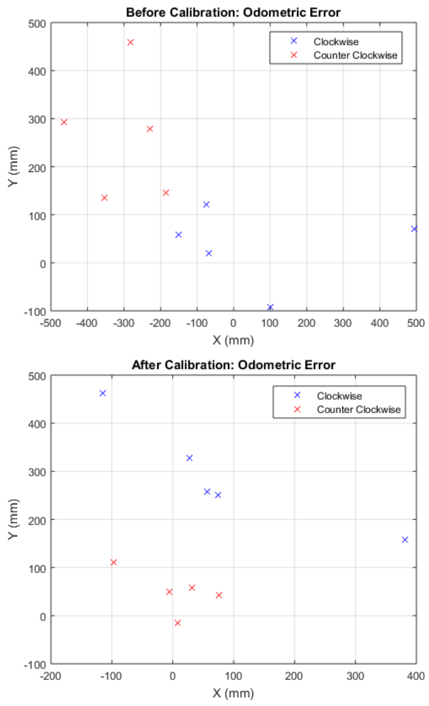


Fig. 5. Odometric measurement error after Maebot traverses a 1M by 1M square. There is lower error after calibration following methods listed in [3]

Trial	1	2	3	4	5
Before Calibration					
$\epsilon_{\theta,CW}$	-0.14	-0.14	0.01	-0.15	0.10
$\epsilon_{\theta,CCW}$	0.31	0.60	0.42	-1.02	0.12
After Calibration					
$\epsilon_{\theta,CW}$	-0.73	-0.91	0.13	0.42	-0.01
$\epsilon_{\theta,CCW}$	0.01	0.05	-0.14	-0.10	-0.02

Fig. 6. Angular odometric error in radians, before and after calibration

B. BotLab - Part II

1) Sec 3.1 : LIDAR initialization and plotting on Ground-station: Mentioned previously, the goal of this lab is to move the MaeBot autonomously through an unknown, indoor and outdoor, region. For complete autonomous movement we require on-the-fly map generation and bot localization. The most common sensor used in SLAM and remote sensing is LIDAR (Light Detection and Ranging). It illuminates the region by targeting laser and then determines the range in a particular direction by analyzing the reflected beam. The MaeBot is equipped with a RoboPeak LIDAR 'RPLIDAR A1M1' namely. Though not the best, RPLIDAR maps a 2D point cloud with a maximum range of 6 meters and 360°at 2000 samples per second, providing enough data for map generation.

The RPLIDAR is provided with its own development kit and can be initialized directly from the ground-station by starting the LIDAR drivers on the *bot-procman-sheriff*. During initialization it calls the RPLIDAR constructor to initialize basic parameters. Once initialized the LIDAR returns data samples as $(r[k], \theta[k])$ where $k = 1, ..., n$, n is the number of data samples in one scan and $r[k]$ is the range in meters in direction $\theta[k]$. For this experiment: *n* = *360* and a scan rate of roughly 8Hz. The communication is via a LCM channel(*RPLIDAR_LASER*) defined in the *groundstation*. The LCM packets are decoded and then plotted on the *groundstation* polar graph. The 2D point cloud is updated continuously at the scan rate and is displayed on the polar grid with respect to the bot at the origin.

In the implementation of LIDAR we noticed that the raw LIDAR data need to be pre-processed. The LIDAR range was in meters and angle in radians, whereas SLAM required parameters in millimeters and degrees respectively. When the data was pre-processed before being passed as SLAM arguments, the LIDAR update in the *groundstation* slowed down drastically. To solve this problem we converted the raw data directly in the SLAM arguments. We believe accessing data outside the argument and then the conversion took time which slowed the LIDAR update on the *groundstation*.

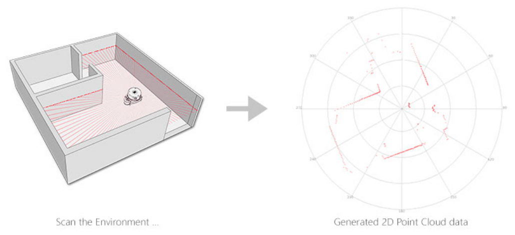


Fig. 7. LIDAR mapping of a 3D region into a 2D point cloud

2) Sec 3.2 : Simultaneous Localization and Mapping - (SLAM) : Once the 2D point cloud of the unknown environment is available, we need to construct a map from the data and simultaneously track the bot's position. Thus the SLAM problem can be defined probabilistically as if given a set of sensor observation o_t over a discrete set of time steps t, we must compute the the probabilistic estimate of the bot's position x_t and map of the unknown environment m_t. This can be formulated as to compute:

$$P(x_t, m_t | o_{1:t})$$

There are various implementations to solve the SLAM problem that are classified according to the use of calculation methods [4], [5], [6] and sensors used [7], [8]. In our experiment, we solve this problem by using BreezySLAM, a simple, efficient and open-source package for SLAM. It is an implementation of CoreSLAM [9] that is a fast but not so robust algorithm for SLAM.

In the implementation, the SLAM class inherits the Random Mutation Hill Climbing(RMHC) [10] class and its constructor initializes the map parameters and also the LIDAR parameters(RPLIDAR). When a LIDAR update is received, the LIDAR handler will update LIDAR data $(r[k](m), \theta[k](rad))$ and thus the current odometry data(position(mm) and orientation(deg) in world co-ordinate system. To determine the current odometry, we must update the map followed by the position of the bot in the map. This is summarized in the algorithm 1.

Algorithm 1 LIDAR and BreezySLAM

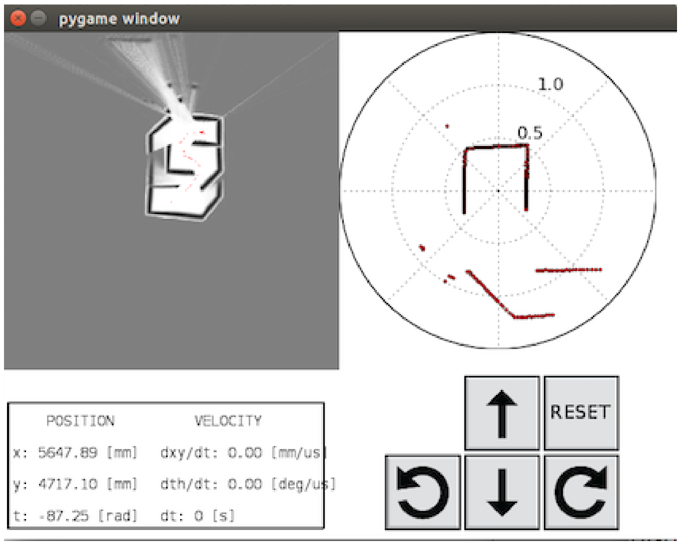
```
 1: Initialize LIDAR parameters
 2: Initialize map parameters
 3: Create RMHC SLAM object
 4: while True do
 5:     if LIDAR updates then
 6:         LIDAR.read() $(m, rad)$
 7:         Generate Map by mapping new cloud
              from LIDAR into pixels
 8:         Update bot position using new
              map and Velocities
 9:         Get new bot position in world co-ordinate system
              $[x(\text{mm}), y(\text{mm}), theta(\text{deg})]$
10:         Convert pixel map to
    $datamatrix(numpyarray)$
11:         return $[y(\text{mm}), x(\text{mm}), theta(\text{deg})]$
12:     end if
13:     Update current map with new map
14:     Plot bot position and trajectory on map
15: end while
```

Do note the change in the x and y positions in the output returned by SLAM.

3) Sec 3.3 : Map integration with Groundstation: Once SLAM generates maps for each LIDAR update, we can update the visualization of the map on the ground station. We must note that the initial bot position on the generated map is in the center of the map. The generated plot shows the recently updated map with respect to the current position of the bot and a trace from its original position. Fig. 8 shows an image of the updated *groundstation*.

4) Sec 4.1: Path planning and real-time testing - A algorithm:* In the previous section II-B.1 we discussed map generation. Once the map is generated, the bot must generate a path to move in the map. This can be done in multiple ways [9] but as this bot will be taking part in a competition we want the simplest and fastest/shortest path possible. This process of generating a path depending on various requirements is referred to a path-planning. Here we consider the fastest as the shortest path because the map has a uniform graph i.e. movements in all directions weigh the same. To determine a path we need a $start$ point, end point and $validmovements$ in the map traversal. We assume that a point determines the center of the selected block.

Resolution: Each block is an element of the map, whose size is decided upon the resolution of the generated map considered for path planning. There is trade-off between path smoothness and computation with resolution. One can infer that high resolution means many pixels that corresponds to



```
    POSITION              VELOCITY
x: 5647.89 [mm]   dxy/dt: 0.00 [mm/us]
y: 4717.10 [mm]   dth/dt: 0.00 [deg/us]
t: -87.25 [rad]   dt: 0 [s]
```

Fig. 8. Updated groundstation with LIDAR point cloud and SLAM map

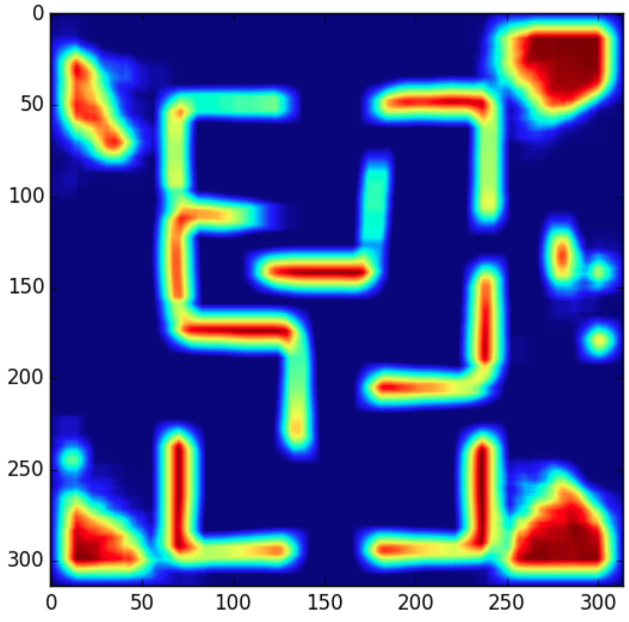


Fig. 9. Filtered map of test map 1 after convolution with circular filter

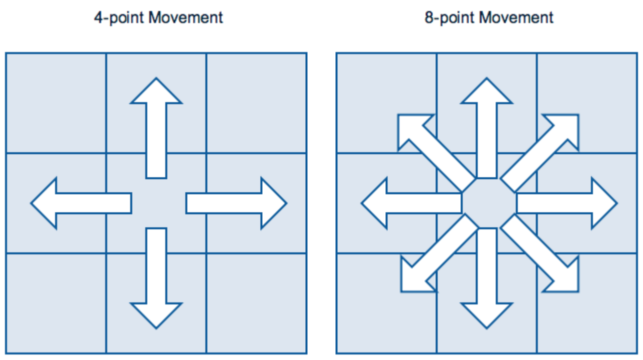


Fig. 10. Movements in a graph

computing path over many pixels, thus increasing computations but generating a smoother path. For this experiment we have decided the map resolution as 1 cm : 1 pixel, that means each block is a square element of 1 cm X 1 cm.

Occupancy map: Before any path-planning and movement, we need to determine the blocks in the graph that are traversable. To generate this Occupancy grid we used a 2D circular filter with the diameter greater than the bot. The idea was to generate a map such that it provides a buffer for the bot in deciding the path through which the bot must be able to pass. Initially we used a square filter as the indoor map would be block-wise with rectangular wall edges but this would create issues due to curved trees in outdoor environments thus moved to using a circular filter. The Occupancy grid is created over a map which is pre-processed by thresholding at a VALID_THRESHOLD. We choose the threshold as 120 because its close to the average pixel value. The pre-processed map is convolved with the filter to generate the Occupancy grid 9 where the darker blue regions are travesable and red regions correspond to the higher probability of obstacles.

We faced a problem while using the *convolution2d* function in scipy. Initially we used the default mode i.e. *full*. This mode returns a linear convolution of the two input matrices. Thus as the origin was shifted, the resultant map generated was also shifted, thus causing miss guided or failed/no paths. We shifted the origin to the center of the filter by using the 'same' mode and default boundary conditions, the $A*$ returned good traversable paths.

Bot Movements: In a graph, the most common movements are 4-point and 8-point movements as shown in the figure 10. In our $A*$ algorithm we consider the *Chebyshev distance* i.e. all 8-point movements have same weight = 1.

Search Algorithm: Followed by deciding basic factors for a graph search algorithm we need to determine the best-fit algorithm for path finding for our experiment. As mentioned previously we need to find the shortest path between two points. The most commonly used shortest-path algorithm is Dijkstra's algorithm but this algorithm searches over the entire graph and determines the shortest path. In our experiment, the start and end point are given, thus we can reduce computation by reducing the search area. This is a suitable condition to use $A*$ graph search algorithm.

$A*$ - *Cost Function*: $A*$ combines the approach of a Greedy Best-First-Search with Dijkstra's. Consider the cost of a path from any point n on the graph as $g(n)$ and $h(n)$ the *heuristic cost* that determines the closeness of the point from the end point. Thus $g(n)$ favors points closer to the start point and $h(n)$ favors points closer to the end point and the resultant cost function used in $A*$ is f(n) = g(n) + h(n). $h(n)$ can be considered in multiple ways depending on the movement scheme considered. We choose our *heuristic function* as the Euclidean distance as we planned to allow movement of the bot in all directions, if time permits. Thus our heuristic function is:

As we could not incorporate movement in all angles, this

Algorithm 2 Heuristic function

```
function heuristic(present_node, end_node):
        dx = present_node.x - end_node.x
        dy = present_node.y - end_node.y
        return $\sqrt[2]{(dx^2 + dy^2)}$
```

function works as the Absolute distance function where we just return the absolute Chebyshev distance between two nodes. Replacing the present function with this approach could make the algorithm faster.

Tanking into account all the above discussed ideas, the final *A** algorithm,

A algorithm*: Each of the team members participated in a friendly hack-a-thon to implement *A** and the fastest one was chosen. Our implementation takes the present map, start and end point as inputs. First we generate the occupancy grid as described above. Our *A** algorithm runs on-the-fly, thus this function generates the best path from the available map and starts from the last end point. Before the path is generated we check where the start point is a VALID_PIXEL i.e. whether it is a pixel in an obstacle region. If it is then we run *Breath-First-Search* to search for the closest VALID_PIXEL and run *A**. This is acceptable given the time constraints because the distance we have to search to get out of a perceived obstacle should be small.

In *A**, we maintain an OPEN and a CLOSED list. OPEN list contains nodes that need to be examined and CLOSED list contains the nodes that have been examined. Initially the OPEN list contains just the start point and the CLOSED list is empty. The neighbors of each OPEN list node are checked for valid movements. A valid movement is when the neighbor is a VALID_PIXEL and is not in the CLOSED list. Following which we compute the cost function as mentioned above and create a priority queue based on the weights of each node and store the parent node and the path cost *g(n)* for the present node. This continues till the end point isn't reached.

Once the weighted path is available, we must traverse the lowest weighted path back to the start point. One must note that this is the reverse path and we must reverse this to determine the actual path. The algorithm describing our *A** implementation is as follows:

Algorithm 3 A* Algorithm

```
astar(map, start, end)
    Generate 2D circular filter
    Threshold input map
 3: Convolving filter and map thresholding
    if start not VALID then
        Find VALID start using Breath-First-Search
 6: end if
    Initialize priority queue Q()
    Q.add(0, (0, start))
 9: Initialize CLOSED & PARENT list
    while not Q.isempty() do
        present = Q.get()
12:     if present == end then
            PathFound = True      break
            for neighbor in present.neighbors() do
15:             if neighbor VALID and
           not in CLOSED then
                    h(n) = heuristic(neighbor,end)
                    g(neighbor) = g(present) + 1
18:                 Q.add(f(n),(g(neighbor), neighbor))
                    CLOSED.add(neighbor)
                    PARENT[neighbor] = present
21:             end if
            end for
        end if
24: end while
    if PathFound then
        current = end
27:     Path = end
        while current != start do
            current = PARENT[current]
30:         Path.add(current) return Path.reverse()
        end while
    else
33:     print "Path not found"
    end if
```

5) Sec 4.2: Path traversal: Guidance: In the previous section we discussed path planning and generated a path using *A** algorithm. The algorithm will return as list of points that the bot must traverse in the map respectively. Thus we need a guiding function that translates the path between nodes as a direction and corresponding distance the bot must traverse. These commands are issued by a file provided as *guidance.py*. We will discuss its details further in the report.

Before we guide the bot using *guidance.py*, one must make sure that the world co-ordinate axis are transformed adequately to replicate bot movement commands by the program. In our experiment, the bot faces the *-x* axis with the *+y* axis to its right and the angle increase positively i.e.$+\theta$ from *-x* to *+y* (counter clockwise) and the increases negatively i.e. $-\theta$ from *-x* to *-y* (clockwise). Following which we must notice that the map co-ordinate axis is also different with *+x* to the south and *+y* to the east. Thus we have introduced two functions *world_to_astar* and *astar_to_world* in the *groundstation* to do the job, where *world_to_astar* converts the odometry measurements into the map co-ordinate system and *astar_to_world* converts the map co-ordinate outputs into the world co-ordinate system.

Due to these co-ordinate axis changes, we ran into a number of problems with the given *guidance* file and thus chose to implement our own simplified version. This program assumes the same states as those defined in *guidance* but it will move to just the next valid position provided by *A** algorithm rather than cruising through a number of points in one go. Moreover, we also had to add a global timeout for the map to get updated. This provided us with higher accuracy at the cost of time. The *DISCTANCE TOLERANCE* was set to 5cm and *ANGULAR TOLERANCE* to 1°.

The initial trial runs were performed on a simple map with just one edge. The bot was instructed to move 1 meter in the forward direction such that the shortest path was around the edge. Once the bot performed this task perfectly, we tested the bot in the practice maze provided in lab before the competition. The results were as expected, the bot moved in the outdoor and indoor environments autonomously with good accuracy in reaching within 10cm of the target but, as the movement was restricted to just 5cm and a global timeout of 5s the bot took around 6 minutes to clear the indoor environment.

III. RESULTS & ANALYSIS

A. *Section 1: Odometry*

The results of our calculated odometric measurements are shown in Fig. 11. This method of calculating the position of the Maebot is susceptible to errors if the wheels slip, which can be caused by abrupt starts or stops or bumps on the floor, as well as incorrect measurements of the axle length and wheel diameters.

B. *Section 2: PID and Calibration*

After calibration we noticed improvements in the calculated XY odometric measurement for counter clockwise traversal. The grouping of clockwise traversal error is tighter however there are too many outliers to conclude that there is an improvement and we require more samples.

C. *Section 3: LIDAR and SLAM*

The updated *groundstation* (Figure 8) shows the LIDAR 2D point-cloud on the polar co-ordinate graph with respect to the bot's orientation at the origin. On the left edge of the *groundstation* we can see the map being updated by SLAM along with the position of the bot and the path it traversed. We have also provided a video of the map being generated when the bot follows the *A* path* autonomously through the test map.

D. *Section 4: A* and path traversal*

We tested our *A** algorithm on the maps that were provided. From the figure 12 we observe that *A** was successful in generating paths in all of the test images. The first row provides the filtered images that have been used to generate the path more effectively. One can notice the rounding of the edges in the maps that shows the use of a circular filter. The second row provides the paths shown in red on the original map. We can see that *A** find shortest paths through the indoor and outdoor environment successfully.

In one of the maps you can see that the region outside the central area where the bot traversed in red. This shows that with filtering the region that is not well developed in the mapping is considered to be non-traversable and the algorithm will take care of it.

Moreover when odometry was compared with SLAM, we observed that:

- MSE_{odo} = 93 mm
- MSE_{SLAM} = 70 mm

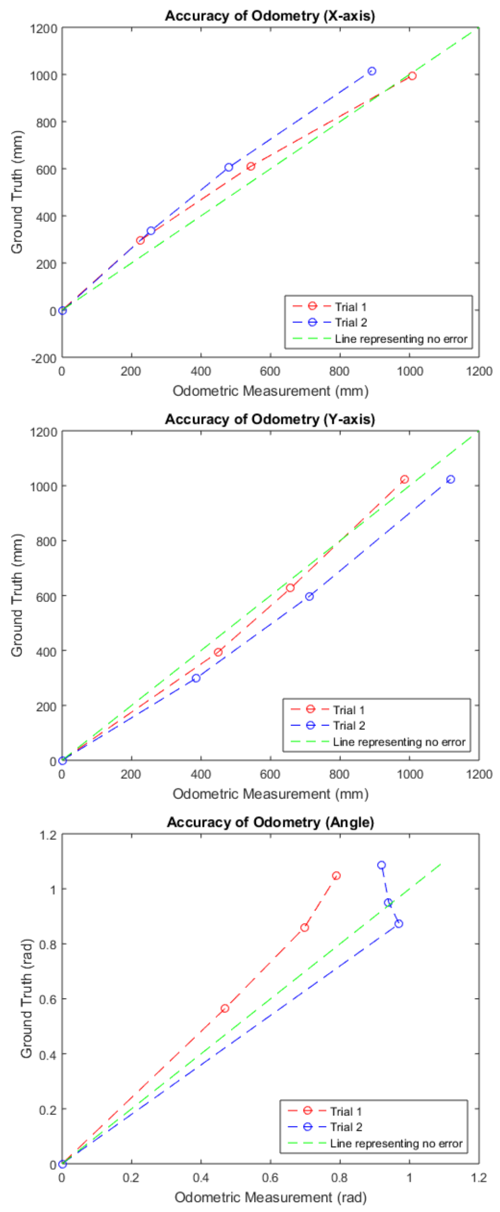


Fig. 11. Plot showing an open-loop driving sequence comparing ground truth (distance traveled according to camera; ground truth angle as measured by LIDAR SLAM) and open loop odometry estimate

which is as expected but there isn't a significant difference. The observations were made by averaging five samples per run over the course of two trails, each driving the robot diagonally in the 1 meter square as this provides variation in x and y simultaneously.

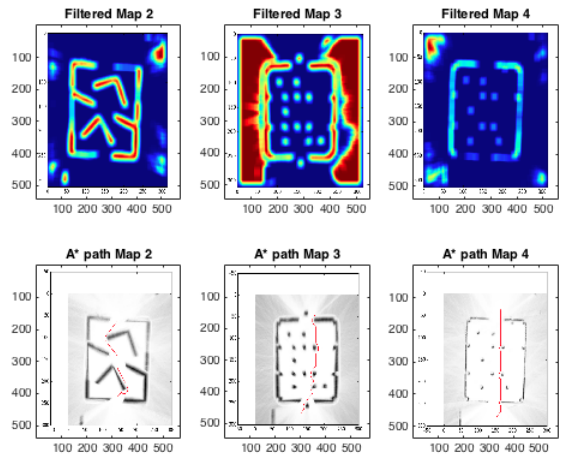


Fig. 12. Paths generated by A* algorithm on maps provided for testing.

IV. DISCUSSION

There were many takeaways from this project. We ran into many issues while working on this project. Our biggest issue was the speed of our robot, even factoring in our choice to map the environment online a 19 minute maze solving time is significantly slower than any other solution. There were several problems with our solution, first, we spent a large amount of time waiting. Because we only moved in 5cm increments there was not a lot of time for the SLAM algorithm to update the map. This let to us having to wait for the LIDAR to receive updates and the SLAM algorithm to update our location.

In addition to the shortcomings of our planning algorithm we also suffered slightly in our guidance function. We ran into many problems translating the SLAM coordinate system into the A* planning coordinates and then back to the SLAM coordinate system. Because of this we chose to implement our own guidance which made many simplifying assumptions. These simplifying assumptions made it so we could only navigate one point at a time and required us to have a global timeout for each movement. This timeout effectively limited the distance robot could move in one step. These drawbacks combined with slight mis-tuning of our PID controllers meant that we were consistently waiting for our guidance function to timeout.

Another area we had some fall backs was with our gyrodometry. During our implementation we failed to correctly handle the gyroscope bias. This caused us to have a larger than necessary theta difference threshold which made us more susceptible to wheel slip during turns. While this was not a major problem for localization, because SLAM can utilize the map and LIDAR readings to help estimate global theta, it was a major problem when trying to turn to a correct heading in between SLAM updates. This mostly occurred while using the guidance system to follow the way-points generated by our A* path planner.

Despite some of the drawbacks of our solution we did have many parts of the project that worked well. Our calibration worked fairly well for the robot that we tested it on. The calibration did not work as well when we had to switch robots later. Another part of the project that worked well was our A* path planning. We utilized numpy built in convolutions to allow our path planning to work on the raw data with high speed. Our implementation was able to compute on a 3 meter map (300 x 300 pixels) in 0.1 seconds. The only time our A* falls down is when it cannot find a solution, when this is the case it has to iterate over all of the pixels (90000 in our tests and up to 640000 in the competition environment).

Overall our solution functioned well in several different test environments. There are many optimization that we could have added to our final algorithm. Beyond the obvious improvements to our guidance and the SLAM algorithm we could implement an hierarchical search step to our path planning. This would allow us to search much larger grids much faster both when there is a path and when no path can be found. Another optimization would be dynamic thresholding of the map. Currently if the robot cannot find a path it simply waits for more LIDAR data and hopes that a path becomes available. If instead the robot could adjust what is considered and ”acceptable” probability of an obstacle it may be able to find a path in more complicated environments.

REFERENCES

[1] Optimized PID controller video: `https://www.youtube.com/watch?v=O6eXRlEedg0`
[2] Team 10 Maebot SLAM: `https://www.youtube.com/watch?v=o42CBWYbugk`
[3] Borenstein, Johann, and Liqiang Feng. ”Measurement and correction of systematic odometry errors in mobile robots.” Robotics and Automation, IEEE Transactions on 12.6 (1996): 869-880.
[4] Huang, Shoudong, and Gamini Dissanayake. ”Convergence and consistency analysis for extended Kalman filter based SLAM.” Robotics, IEEE Transactions on 23.5 (2007): 1036-1049.
[5] Dissanayake, M. W. M. G., et al. ”A solution to the simultaneous localization and map building (SLAM) problem.” Robotics and Automation, IEEE Transactions on 17.3 (2001): 229-241.
[6] Grisetti, Giorgio, Cyrill Stachniss, and Wolfram Burgard. ”Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling.” Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on. IEEE, 2005.
[7] Newman, Paul, David Cole, and Kin Ho. ”Outdoor SLAM using visual appearance and laser ranging.” Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on. IEEE, 2006.
[8] Agrawal, Motilal, and Kurt Konolige. ”Real-time localization in outdoor environments using stereo vision and inexpensive gps.” Pattern Recognition, 2006. ICPR 2006. 18th International Conference on. Vol. 3. IEEE, 2006.
[9] Van Vuren, T., and G. R. M. Jansen. ”Recent developments in path finding algorithms: a review.” Transportation planning and technology 12.1 (1988): 57-71.
[10] Bajracharya, Suraj. ”BreezySLAM: A Simple, efficient, cross-platform Python package for Simultaneous Localization and Mapping (thesis).” (2014).