BotLab Report

An Dang, Lihan Lian, Shrey Shah
{andang, lihanl, shreyzz}@umich.edu

***Abstract*—In this lab, the aim is to enhance the Mbot's autonomy by integrating PID controller, odometry, computer vision, SLAM, and path planning to the autonomous exploration and manipulation of blocks in the workspace. The report encompasses the development and application of a control model for accurate movement of the bot along with detailed explanation on mapping, action and sensor models. On the sensing front, a Lidar sensor is utilized for pose estimation along with a camera. Furthermore, the work involves formulating algorithms for exploration and facilitating autonomous decision-making by the robot.**

I. Introduction

Autonomous mobile robots represent a complex yet crucial field within advanced technology. Our project focuses on developing a system to autonomously navigate the most cost-effective routes. The accompanying report outlines the programming process to build up an autonomous exploration robot with gripping capabilities and evaluates the effectiveness of this programming through simple task demonstrations. The report highlights the limitations and challenges faced in the application of mobile robotics.

II. Methodology

Our methodology details the algorithms utilized to enable a robot to perform speed control, path following, shortest path generation, exploration, and simultaneous localization and mapping (SLAM). Before summarizing the contributions of this report, it is essential to describe the sensors and robot kinematics on the Mbot, utilized in the algorithms. The Mbot is equipped with an RPLidar, 2 encoders mounted on the motors, and a Bosch IMU for rotational measurements [1], [2]. The Assembled Mbot of our group is shown in Figure 1

III. Motion and Odometry

This section details the implementation of speed control and odometry of the robot. This section describes the method to develop and implement a PID control system on the Mbot based on the odometry and the encoder readings. It primarily uses a gyroscope and IMU to the angular motion. The details are entailed below.

A. Kinematics

A precise description of kinematics of the Mbot will be referenced extensively throughout this report. The

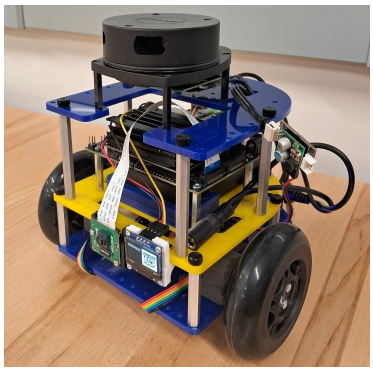

Fig. 1: Assembled Mbot

Mbot kinematics is a classic example of differential drive. Extensive works have already been made on kinematics [3] which we will summarize as the equations below:

$$\omega = (v_R - v_L)/b \tag{1}$$

$$v = (v_R + v_L)/b \tag{2}$$

$$v_x = v\cos(\theta + \omega/2) \tag{3}$$

$$v_y = v\sin(\theta + \omega/2) \tag{4}$$

v_L is the velocity of left wheel, v_R is velocity of right wheel, and the b variable is a parameter which is the baseline of the robot (distance between two wheels).

B. Calibration

For effective utilization of kinematics, calibration is used to estimate the parameters. A calibration procedure is implemented to map the PWM of the motors to velocities on the left and right wheels (feed-forward parameters). Additionally another calibration procedure estimates the baseline variable used in Equation (2) and (1).

C. Odometry

For our initial odometry system, we utilize only the encoders and IMU on the Mbot. Utilizing the encoders, we can measure the change in rotation. Using the change in rotation as wheel velocities, Equations (5) can be used to calculate the change in position of the robot throughout a movement. However, estimating velocities using change of encoder measurements is inherently

noisy and can blow up for measurements in rotation. For the calculation of Equation (1), we instead take the gyroscope measurements for larger rotations known as gyrodometry [4]. For global heading, we utilize the heading given by the Bosch IMU. The angular velocity measured will be useful for PID speed control.

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} \tag{5}$$

D. PID Controller

For speed control, we are given translational and angular velocity v and ω and wish to follow them as accurately as possible. Initially we plug in feed forward values by using Equations (1) and (2), solving for v_L and v_R, mapping to PWM using calibrated parameters.

The feed forward control isn't enough to accurately follow the desired velocities, so we integrate feedback PID control to minimize our observed error i.e. $(v - v_{odo}, \omega - \omega_{odo}) \rightarrow (0, 0)$. Where odo denotes velocities measured with odometry. We also developed a PID for the individual wheels for speed control to minimize wheel speed error $(\omega_{wheel} - \omega_{obs} \rightarrow (0, 0))$. Due to the noise from the encoders, several lowpass filters are employed at several stages to smoothen the values. The value for our controller parameters are shown in Figure 2.

Type	Kp	Ki	Kd
wheel	1.5	1.5	1e-9
translation	0.2	0.01	0.5
rotation	0.3	0.01	0.04

Fig. 2: PID parameters for controller

In order to tune our system, we first tuned the wheel PID controller. Afterwards, we tuned our translational and rotational velocity PID controllers. We tune the PIDs to reduce the oscillations in our plots and ensure that there is no steady state error after convergence. After tuning these PIDs, we notice heavy oscillations in the velocity estimated, so we decide to lowpass our estimated speeds.

E. Path Following

Our PID set points utilize v and w commands which the Motion Controller will supply for path following. We chose Pure Pursuit as it allows us to seamlessly follow the waypoints without stopping [5]. The Pure Pursuit algorithm can be summed up by Algorithm 1. The k_a and k_b are parameters to scale how the Pure Pursuit will output velocities. The general idea is to take the entire path and find the best fit circle between the robot and the path. The best fit circle must have the radius of the look ahead distance l_r which can be tuned. The importance of the algorithm lies in calculating the curvature κ which steers the robot into the path.

Algorithm 1 Pure Pursuit

Input: $p_r \in SE(2)$, $p(t)$, l_r
Output: v, w
circle $\leftarrow makeCircle(p_r, l_r)$
$p_{inter} \leftarrow getIntersection(circle, p(t))$
$circle_{fit} \leftarrow fitConcaveCircle(p_r, p_{inter})$
$r \leftarrow GetRadius(circle_{fit})$
$\kappa \leftarrow \frac{1}{r}$
$w \leftarrow k_b \kappa$
$v \leftarrow k_a \|p_r - p_{inter}\|_2$
$return(v, w)$

IV. SLAM

With odometry from the encoders and IMU, the robot's localization will slowly drift away from the ground truth because there is no global measurement of its position. Using a map will allow the robot to localize itself and decrease the drift caused by odometry. This requires a map of the environment as well solved by SLAM. Mapping is not only needed for localization, and its usefulness extends to 3d reconstruction and exploration. We employ an occupancy-grid model for the mapping and utilize a Particle Filter to perform localization given a map. The Figure 4 shows how each component of the SLAM system interact with each other.

A. Mapping

Occupancy-grid mapping represents the map as a finite rectangle composed of MxN square cells. Each cell is encoded as a Bernoulli random variable $c \sim Ber(p)$. The variable p encodes the belief that the cell is being occupied and is recursively updated by the Lidar scans and the robot pose [6].

The lidar shoots rays from all sides and the receiving signal is composed of pairs of (θ, r) where θ is the angle from the robot and r is the magnitude of ray. This data allows us to creates line segments around the robot where the line segments start at the center of the robot and end at an obstacle. Since our odometry and Lidar data operate at different rates, we interpolate our odometry data to get an appropriate robot pose for a corresponding lidar scan.

For each laser, every cell traversed by the laser (calculated by bresenham algorithm) has its probability of being occupied decreased while the end cell has its probability of being occupied increased. The amount to increase and decrease is denoted as hit odds and miss odds. To prevent numerical instability, we track the log probability in each cell.

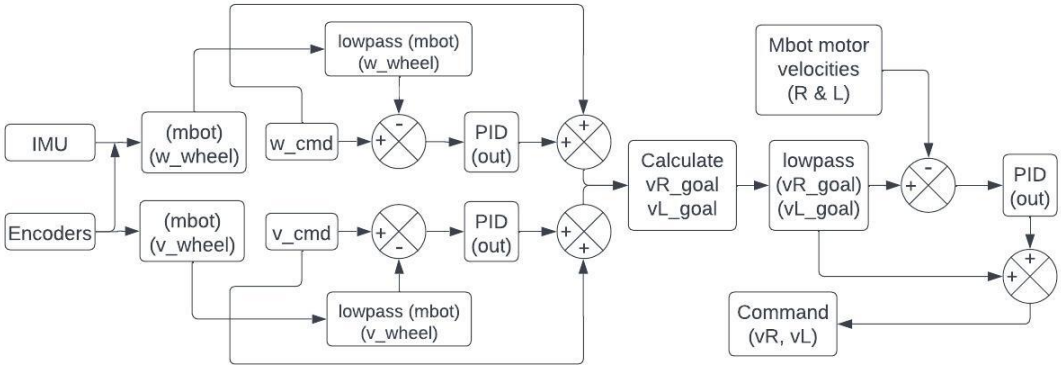

Fig. 3: Enter Caption

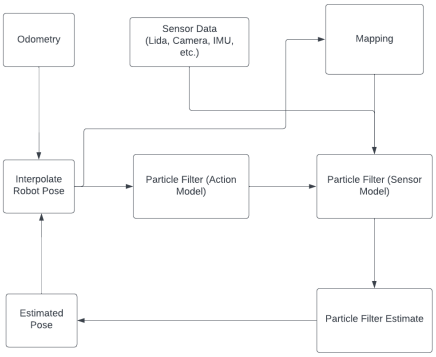

Fig. 4: Block diagram of SLAM system components

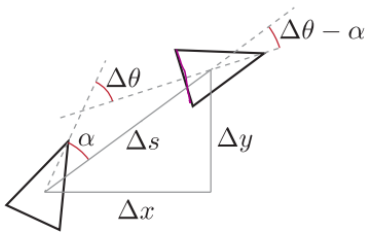

Fig. 5: Action model parameters

B. Particle Filter (Action Model)

The particle filter is a bayesian filter which uses an action model and sensor model to find the max posterior belief of the robot's current position given its previous states and lidar measurements. The action model of the particle filter is a constant motion model with no consideration of the control input of the robot wheels. This model is calculated by taking the previous motion of the robot from p_{t-1} to p_t and applying it again from p_t to p_{t+1} where $p \in SE(2)$ is the pose of the robot. For the particle filter, we represent our action model as a gaussian distribution, having $x_{t+1} = \mathcal{N}(\mu = f(x_t, u_t), \Sigma)$ where $u_t = [\alpha, \Delta s, \Delta\theta - \alpha]$. When the the particle filter is initialized with K particles, the particles are distributed around the robot odometry pose. Afterwards, the action u_t is applied on each particle using the equations below and sampling from the random terms.

- Turn $(\alpha + \epsilon_1)$
- Travel $(\Delta s + \epsilon_2)$
- Turn $(\Delta\theta - \alpha + \epsilon_3)$
- $\epsilon_1 \sim \mathcal{N}(0, k_1|\alpha|)$
- $\epsilon_2 \sim \mathcal{N}(0, k_2|\Delta s|)$
- $\epsilon_3 \sim \mathcal{N}(0, k_1|\Delta\theta - \alpha|)$

α represents the initial rotation followed by translation, Δs and finally the rotation $\Delta\theta - \alpha$ .The final action update model is given below.

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} (\Delta s + \epsilon_2)\cos(\theta_{t-1} + \alpha + \epsilon_1) \\ (\Delta s + \epsilon_2)\sin(\theta_{t-1} + \alpha + \epsilon_1) \\ \Delta\theta + \epsilon_1 + \epsilon_3 \end{bmatrix}$$

C. Particle Filter (Sensor Model)

Delving into the sensor model, we want to utilize the sensor measurements to improve our belief of the robot pose. The sensor model allows us to compare our predicted measurement with the actual measurement and weigh that information in each particle. This is done by updating the weight of the particle using the Lidar scan given. The weighting is done by a fitness score which increases for each Lidar scan of the particle that hits an obstacle. If the Lidar scan does not hit an obstacle, a Breadth-First-Search finds the nearest obstacle to the ray. Its distance is applied to a gaussian density function as a fitness score. The fitness scores are normalized over all particles to become particle weights. Taking the weighted average gives the estimated pose of the robot. The Particle Filter combines both action and sensor model into the following steps:

1) **Initialization:** *InitializeFilterAtPose* sets particles at a given pose with equal weights & orientation noise.

2) **Update Loop:**
   - *ResamplePosteriorDistribution* introduces randomness to simulate motion uncertainty.
   - *ComputeProposalDistribution* applies the action model, predicting new particle positions.
   - *ComputeNormalizedPosterior* reweights particles based on sensor data alignment with the map.
3) **Pose Estimation:** *EstimatePosteriorPose* calculates the estimated pose as a weighted average of the particle distribution.

D. Particle Filter (MCL)

When the Mbot is put inside a closed space that is known, it can localize itself using the same particle filter framework. There is a slight deviation where the initialization is done by uniformly distributing the particles along the free space of the known map. These uniformly distributed particles will eventually converge toward the robot after each iteration of the particle filter. This discovery shows that particle filters are powerful for performing tasks when understanding the probability framework behind the particle filter.

V. Path Planning

Path planning is the task of determining a safe and efficient route for a robot or autonomous vehicle to navigate through an environment. Our path planner will avoid obstacles while finding the optimal path to a goal. Our algorithm of choice is A* where we utilize the occupancy grid mapping and represent it as a node graph to solve a Travelling Salesman problem. The edges of the node graph are calculated using a cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the actual path cost from the start to node n, and $h(n)$ is a heuristic estimate of the cost from node n to the goal.

A. A* Algorithm

In our A* algorithm, we represent the occupancy grid node graph as an 8-connected grid map where our function $h(n)$ is calculated by the manhattan distance from the goal node and the $g(n)$ is the 2-norm distance from the start node. The pseudo code of the algorithm is shown in Algorithm 2.

In the provided A* test cases, we passes 4 out of 6 and here are the statistics for the path planning execution times.

Our A* algorithms passed most test cases, and took some time. One of the reason it failure modes is when no safe path exists resulting in traversal of all nodes, failing in the process. In our real exploration task, the A* algorithm worked well on find a path to the frontier so the implementation of A* is good enough to use for exploration.

Algorithm 2 Algorithm: A*

Input: start, goal(n), h(n), expand(n)
Output: path
if goal(start) == true **then** return $makePath(start)$
$open \leftarrow start$
$closed \leftarrow \emptyset$
while $open \neq \emptyset$ **do**
  $sort(open)$
  $n \leftarrow open.pop()$
  $kids \leftarrow expand(n)$
  for all kid in $kids$ **do**
    $kid.f \leftarrow (n.g + 1) + h(kid)$
    if $goal(kid) == true$ **then** return $makePath(kid)$
    if $kid \cap closed = \emptyset$ **then** $open \leftarrow kid$
  $closed \leftarrow n$
end while
$return\ \emptyset$

Test	Min	Mean	Max	Std
convex	4640	44644	84648	4004
maze	11211	11211	11211	0
narrow	1.38e8	1.97e8	2.56e8	5.90e7
wide	1.37e8	3.64e8	7.03e8	2.44e8

Fig. 6: Execution time for successful attempts (in μs)

Test	Min	Mean	Max	Median	Std
empty	34	41	53	39	6.72
filled	27	40.6	60	37	10.85

Fig. 7: Execution time for failed attempts (in μs)

B. Exploration

The exploration module allows a robot to explore an entire unknown closed space. Using previous algorithms, once a robot is placed, it will perform initial mapping and then explore the map by finding "frontiers" of the current map. A frontier is any portion of the map the robot currently explored that is not closed by occupied cells. The frontiers represent where a robot should explore, utilizing the A* pathfinding algorithm to find the shortest obstacle-avoidant path to the frontier. This will continue until there are no frontiers left. The algorithm 3 gives a simplified pseudocode of the exploration module.

VI. Gripper Design

The lifting task in BotLab necessitated an innovative gripper design for the Mbot. Our team opted for a straightforward yet effective rack and pinion forklift mechanism. This gripper was installed on the Mbot's

Algorithm 3 Frontier Exploration

while $len(fs) > 0$ **do**
  $m \in Int[M][N]$ ▷ Get map from mapping
  $p_r \in SE(2)$
  $fs \leftarrow GetFrontiers(m)$ ▷ frontiers from map
  $f \leftarrow ClosestFrontier(fs, p_r)$
  $p \leftarrow CenterFrontier(f), p \in SE(2)$
  $ps \leftarrow Astar(p, m, p_r), ps \in SE(2)^n$ ▷ get path
  while $p_r \neq ps[len(ps) - 1]$ **do**
    $v_x, w_z \leftarrow PurePursuit(p_r, ps)$
    $Controller(v_x, w_z)$
    $p_r \leftarrow Estimator()$
  end while
end while

fourth layer, which accommodates all the essential motors and connectors, ensuring the gripper's stability. The rack, pinion, and fork were cut from acrylic sheets available in our lab, while the connectors and motor housing were 3D printed. Several design iterations were undertaken to optimize the lift's functionality. The pros of our distinctive design are as follows:

- The mechanism, situated on the fourth layer, ensures unobstructed operation during cube lifting.
- The layer offers ample space for motor placement, allowing for flexible motor positioning as required.
- Our connectors are uniquely designed with adjustable guideways, accommodating positional variations within a certain range, thus removing the need for redesign in case of minor design inaccuracies.

VII. Results

We were able to accurately implement SLAM with exploration with the error of motion being less than 5cm overall. The following section will enlist all our experimental results and improvements made during the entire process.

A. Calibration and Odometry

The calibration was tested on 2 different floor types: **carpet** and **Lab's tile** floor. The motor speed vs PWM command is plotted and a least squares method is utilized to best fit the values. The data is calculated in the form $y = mx + c$ where m represents the slope and c represents the intercept of the equation of line. The average values of all tests cases for **Carpet** are summarized below.

Figure 8 shows that the values are fluctuating at every iteration of calibration. The table below shows the average calibrated values for the carpet along with the variance of each parameter. We can observe that there

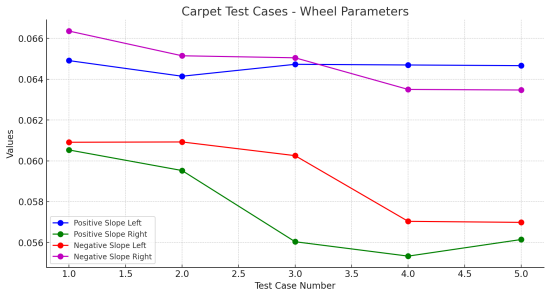

Fig. 8: Variation of parameters on Carpet

is little variance for each parameter which doesn't affect our Mbot as a whole.

+ve 'm'	+ve 'c'	-ve 'm'	-ve 'c'
0.06452	0.08643	0.06103	-0.09597
0.06042	0.09790	0.06539	-0.09204

- Positive Slope: 8.71e-08, 2.45e-07
- Positive Intercept: 1.25e-05, 1.29e-05
- Negative Slope: 2.87e-07, 4.07e-07
- Negative Intercept: 1.23e-05, 2.00e-05

The below graph shows how the PWM command affects the motor speed on the **Tile** based floor. The table shows the avg. of each metric for both wheels followed by the variance. We can notice the floor is more consistent than the carpet. Since the variance is also less when comparing both of them.

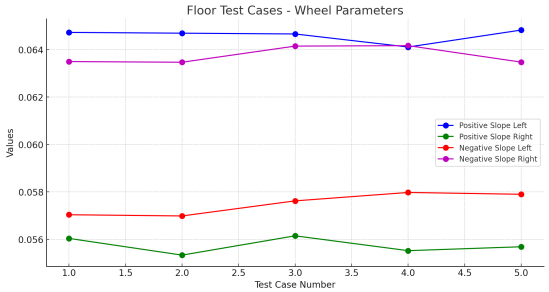

Fig. 9: Variation of parameters on Tile floor

+ve 'm'	+ve 'c'	-ve 'm'	-ve 'c'
0.06460	0.03884	0.05750	-0.05742
0.05574	0.05987	0.06375	-0.04226

- Positive Slope: 6.40e-09, 9.42e-09
- Positive Intercept: 8.89e-06, 1.69e-06
- Negative Slope: 1.76e-08, 1.11e-08
- Negative Intercept: 1.65e-05, 1.30e-05

We can clearly see that most of the values are at least 10 times lower than that of **Carpet's** calibration. The

data shows variance due to uneven surface and a change in friction of the surfaces requiring higher torque over the carpet than floor. The deviations in the floor readings can be associated with the slippage over the shaft itself and the backlash of the shaft itself when locked with the wheel. Carpet, with its variable texture and resistance, introduces more unpredictability in movement, while the smoother floor provides a more consistent environment for robot navigation.

The block Diagram for our PID control system is shown below. we have implemented a total of 3 PID filters (with 3 lowpasses) excluding the ones already present in the smart controller in the mbot autonomy.

After implementing the PID control, we observe that it strives to stabilize towards the target velocity. The blue line represents the target velocity of 0.5 m/s. Due to some errors in our motors, the PWM command for linear velocity fluctuates between 0 and 0.5 m/s frequently. However, it is evident from the estimated velocity graph (orange line) that the PID controller effectively adapts to these fluctuations and ultimately achieves the desired target velocity. Both the plots are shown below.

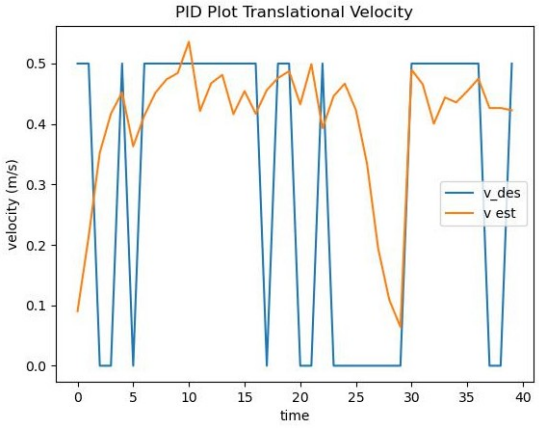

Fig. 10: Translational PID

1) Motion controller: The implementation of PID on the path following works well in our case. From the the figure below, we can infer that using the pure pursuit to follow the path works very well relative to the robot as there is almost no drift in the path during the motion.

The Mbot's linear velocity plot follows a trapezoidal motion where every motion consists of acceleration, constant velocity of 0.3 m/s and deceleration. As for the rotational velocity of the Mbot the trapezoid is stretched due to the short time frame for turning motion. we tuned it such that it tries to turn as fast as possible without any constant velocity upper limit resulting in a triangular plot at every turn rather than proper trapezoid. We can see Pure Pursuit's effect with the reduction of the translational velocity to allow for higher angular velocity

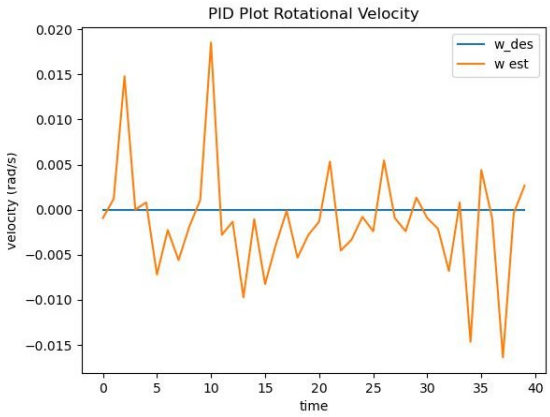

Fig. 11: Rotational PID

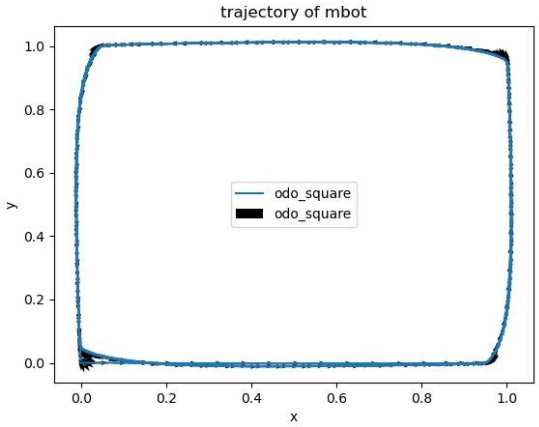

Fig. 12: Dead reckoning estimated pose

for sharp turns. This allows consistent path following as fast as possible as the translation rotation trade off is encoded in PurePursuit.

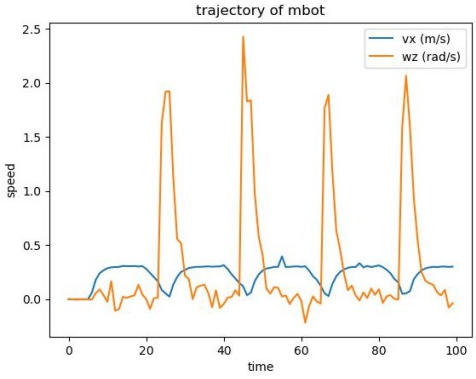

Fig. 13: Linear and rotational velocity

B. Mapping

We benchmark our mapping algorithm by running it on a preset map that the Mbot follows. This is done by just running the Particle Filter and Mapping Module simultaneously which emulates the SLAM algorithm. Doing this on an example map led us to the results in Figure 14. The map is ideally a rectangular maze, but the figure shows a drift toward the bottom of the map which is attributed to the drift in robot heading from the Particle Filter.

Fig. 14: Map of Drive maze log file

C. Localization

After conducting localization tests, we ran the entire SLAM algorithm on the lab's maze without using any pre-existing log files. This experiment revealed notable discrepancies between the odometric readings and the SLAM-derived pose in the map. This discrepancy highlights the drift and positional inaccuracies inherent in relying solely on odometry. For this task, we initialized 1000 particles, which accurately predicted the robot's pose, as showcased by the dense cluster of red dots visible in the image.

Fig. 15: SLAM vs Odometric pose

Conducting speed tests on the algorithm as a function of the particles, we summarized the Table I. As the number of particles increase, the response takes longer

Fig. 16: SLAM pose on lab maze

Particles	100	200	500	1000
Time (ms)	2.73	2.74	10	19
Response (Hz)	365	363	98	50

TABLE I: Particle Filter algorithm speed performance benchmark by varying particles.

and decreases as a function of the particles. A linear relationship is seen and we can estimate that it would take roughly 2500 particles for the particle filter to have 20 Hz speed. Hence, the max number of particles before the speed reduction will affect the robot is around 2500 particles.

From the plotted particles along the Mbot's path using the particle filter, we can see the tight clustering of particles around the true path suggests accurate localization, indicating the filter's effectiveness in handling sensor noise. Resampling behavior can be observed at points of direction change.

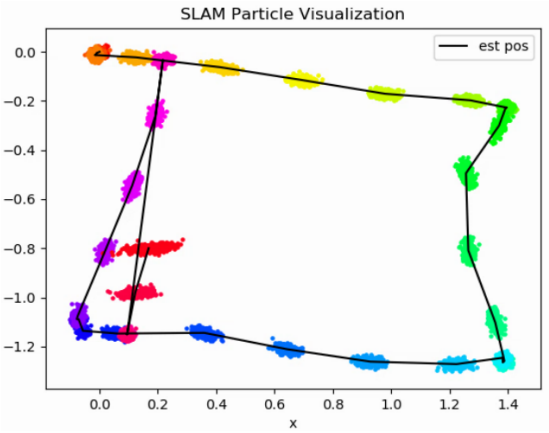

Fig. 17: Intermittent Particle distribution

After developing the complete SLAM system, we compared our SLAM pose and the ground truth from the existing log file. It is evident that there are noticeable deviations from the ground truth. One significant reason for this is that our initial heading is tilted, causing a rotation of the entire SLAM pose. To assess the accuracy

of our system, we calculated the root mean square **(RMS)** error.

- x **error -** 0.19449
- y **error -** 0.24574
- θ **error -** 0.60854

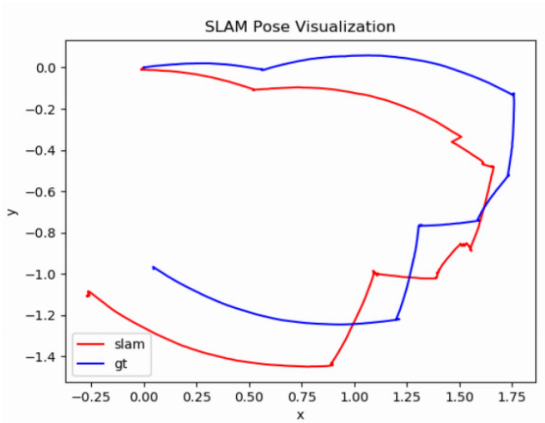

Fig. 18: SLAM pose vs Ground Truth

D. Path Planning

As previously noted, our A* algorithm successfully passed most of our test scenarios. Building on this success, we applied it to navigate the lab's maze. In the visualization, the green dots represent the path as charted by the A* algorithm, marking each planned point. In contrast, the blue line depicts the actual path the robot followed using SLAM. Initially, there's a noticeable but minor deviation between the two paths, likely due to the inherent discrepancies between theoretical models and real-world conditions. However, as the robot progresses, the A* path and the SLAM path align closely. This alignment indicates the effectiveness of the A* algorithm in real-time navigation.

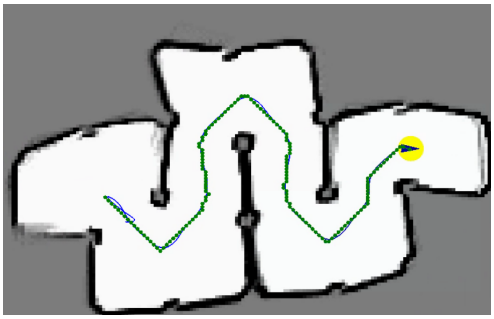

Fig. 19: Path planned by A* in real time

E. Gripper implementation

The rack of the gripper made 250 mm long, enough to lift and go over the 3^{rd} layer of the Mbot without obstruction. The fork lift cut via acrylic was successfully able to lift objects without bending. We moved the 4^{th} layer forward by 20 mm to avoid hitting the rest of the bot. The CAD models of all parts are given below. The guideways are specially designed to enable any orientation possible within a certain range. The final assembled Mbot with gripper design is also shown. The gripper was used along with the camera to find, pick and place warehouse boxes.

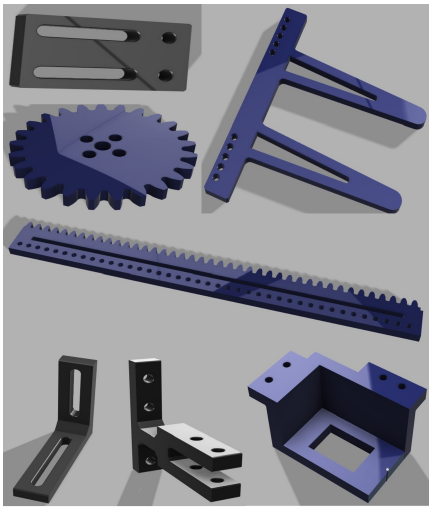

Fig. 20: Rendered CAD designs

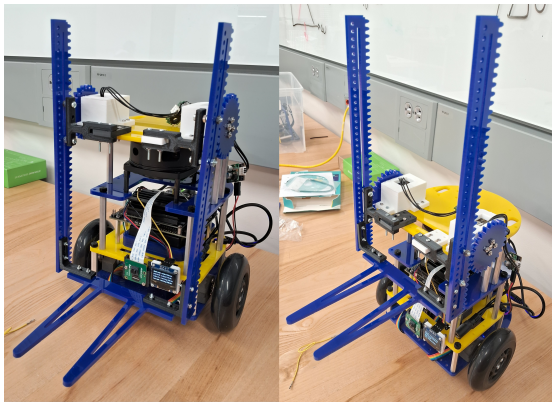

Fig. 21: Integrated Gripper Design

REFERENCES

[1] RPLidar Specification Manual.
[2] Bosch IMU Specification Manual.
[3] R. D. Fonso, "Navigation and motors control of a differential drive mobile robot," *International Conference on Control, Automation and Diagnosis*, 2023.
[4] "Gyrodometry: a new method for combining data from gyros and odometry in mobile robots," 1996.
[5] Implementation of the Pure Pursuit Path Tracking Algorithm.
[6] "High resolution maps from wide angle sonar," 1985.