ROB 550 BotLab Team 11 Report

Jayjun Lee, Tyler Smithline, Xiangbo Gao

***Abstract*—This paper presents a comprehensive account of our work in the BotLab project, emphasizing the development and integration of various robotic systems on the MBot. The project encompasses a broad range of robotics concepts, including cascade control and PID, planar kinematics, motion models with uncertainty, perception, and reasoning. The project was divided into several parts, each focusing on a specific aspect of robotics: robot motion control and odometry, vision, SLAM, and implementing planning and exploration. These sections entailed designing a feedback controller for motor speed, a gripping mechanism, implementing 2D mapping with LIDAR, building a SLAM system, and exploring path planning and autonomous navigation. This paper summarizes the methodologies, challenges, and outcomes from each part of the project. The culmination of these efforts is demonstrated in a series of lab checkpoints and a competition showcasing the MBot's capabilities in real-world scenarios where our robot won the 2nd place.**

I. Introduction

The field of robotics continues to advance rapidly, with applications in diverse areas from industrial automation to exploration in hazardous environments. This paper documents our journey through the BotLab project, where we engaged in the assembly, programming, and testing of the MBot – a differential drive mobile robot.

Our project was structured in multiple parts. The initial phase involved assembling the MBot and designing a gripping mechanism, laying the groundwork for advanced operations. Subsequent parts of the project delved into motion and odometry. We developed controllers for precise robot movement and designed algorithms for accurate positioning and orientation. In the vision section, we worked with camera and apriltag detection for obstacle identification. We also designed a gripping mechanism that can lift objects smoothly. The simultaneous localization and mapping (SLAM) component was pivotal, combining sensor inputs and algorithms to create dynamic maps of the unknown environment and to localize the robot in that map. Our work also delved into the reasoning aspect, where we implemented algorithms for localization, mapping, path planning, and autonomous navigation. Using Monte Carlo localization and A* search algorithms, we developed a system that could navigate complex environments while adapting to uncertainties. This paper presents a comprehensive overview of the project, detailing the methodologies, challenges encountered, and solutions implemented.

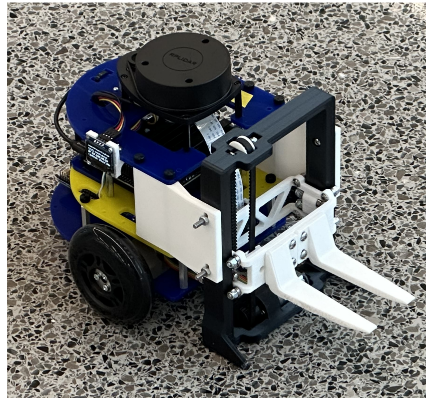


Fig. 1: Our MBot with the 3D printed lifting elevator.

II. Methodology

A. Motion and Odometry

1) Wheel Calibration: In order for the robot to achieve smooth driving motions, it is important to understand the relationship between the pulse width modulation (PWM) signal applied to the DC motors and their output speed under normal driving conditions. This is accomplished by a calibration process that sends PWM signals with known duty cycles to the drive motors and measuring their resulting speed from encoder.

Since the wheels need to overcome static friction, the PWM signal needs to ramp up very quickly until the wheel begins spinning. Then the relation between duty cycle and wheel velocity becomes roughly linear. Because of this, a least squares fit is used to estimate the best linear approximation of this relation.

2) Odometry: The robot's odometry system uses the wheel encoders to estimate the pose of the robot. By using the number of wheel revolutions counted by the encoders combined with the known parameters of the robot, a pose estimate is computed. These are used in a kinematic model of the robot illustrated by Fig. 2. To keep track of the robot's current pose in the world frame, all changes in robot pose are measured relative to the starting pose of the robot. As a result, the distance the robot has traveled in x and y can be calculated based on the robot's heading in the world frame.

This information can be calculated using the known quantities of how much each wheel has rotated since the last measurement, which is known directly from the encoder measurements. These values are known as

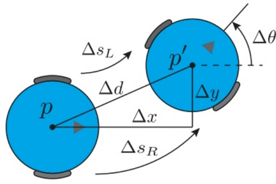


Fig. 2: A diagram of a differential drive model.

Δs_R and Δs_L respectively. Using these, the change in the robot's heading and total distance traveled can be calculated as:

$$\Delta\theta = \frac{\Delta s_r - \Delta s_L}{b}$$
$$\Delta d = \frac{\Delta s_r + \Delta s_L}{2} \tag{1}$$

Next, the change in x and y can be calculated using the change in heading, distance traveled, and initial heading of the robot in the world frame:

$$\Delta x = \Delta d\, cos(\theta + \Delta\theta/2)$$
$$\Delta y = \Delta d\, sin(\theta + \Delta\theta/2) \tag{2}$$

These equations produce an odometry model that can keep track of the robot's pose but is notably susceptible to accumulation in error the further the robot has traveled. In order to help mitigate non-systematic error, gyrodometry was implemented to incorporate IMU data into the robot's pose estimate [1].

Gyrodometry is a simple method of fusing encoder and gyroscope data into a single heading estimate. If the measured change in heading since the previous time step from the gyroscope exceeds that of the encoders by a specified threshold, then the gyroscope measurement is used. If not, the encoder odometry measurement is used. In other words, if the wheels slip significantly or if the robot is subject to a rotation from a force other than the wheels, the heading will be updated from the gyroscope. This helps reduce error in the pose estimate when unexpected behavior not accounted for in the odometry model is experienced.

3) Wheel Speed PID: If the PWM signal sent to the motors is toggled on and off in order to achieve a desired moving speed, the robot will experience sudden acceleration and deceleration that may lead to consequences such as wheels slipping or robot tipping. Additionally, this method of open-loop control does not guarantee the robot will move at the intended velocity set by the motion controller. In order to help mitigate these problems, a closed-loop PID controller is used.

The control system uses a separate PID loop on the left and right wheels in addition to a PID loop on the linear body velocity (v_x) and rotational body velocity (w_z). For each of these loops, only proportional gain is used. From testing, the wheels had negligible overshoot so there was no need for derivative gain and the calibrated PWM function minimized accumulated error, so integral gain was not needed either. Therefore, the following gain table was used:

TABLE I: PID gains for differential drive.

PID Loop	k_P	k_I	k_D	T_f
Left	0.09	0	0	2.1 ∗ `loop_period`
Right	0.09	0	0	2.1 ∗ `loop_period`
V_x	0.09	0	0	2.1 ∗ `loop_period`
W_z	0.09	0	0	2.1 ∗ `loop_period`

In addition to the PID loop, the left and right wheels have a low-pass filter to reduce the impact of high-frequency noise and sudden changes in the control input. This filter has a cutoff frequency of 0.25 Hz such that only sustained control inputs are kept.

One final feature of the wheel controller is the use of a PWM to velocity calibration function obtained from the calibration routine described in the wheel calibration section. The output from the PID loop is a desired velocity, which is sent to the calibration function which returns the appropriate PWM duty cycle level to achieve the desired velocity. The calibration function accounts for static, viscous, and coulomb friction in order to achieve the correct velocity.

4) Motion Controller: The motion controller is necessary to determine how to convert a series of waypoints into linear and rotational velocity commands for the robot. From testing, it was evident that different control methods achieve better results depending on the task at hand. For instance, the Rotate-Translate-Rotate (RTR) controller excelled at closely following lines for the drive square and Event 1 mapping tasks. For map exploration, however, RTR was much slower than using a smart controller. Due to its simplicity, however, the RTR controller was chosen for the vast majority of tasks.

The RTR controller has three steps: Rotate towards the desired goal, drive towards that goal, and then rotate to the final target orientation. Each of these steps uses proportional control to determine the appropriate turning and linear velocity based on the error between the current pose and target pose. The control equations can be understood in relation to Fig. 3.

These errors are calculated as follows:

$$\Delta x = x_g - x$$
$$\Delta y = y_g - y$$
$$d = \sqrt{\Delta x^2 + \Delta y^2} \tag{3}$$
$$\alpha = atan2(\Delta y, \Delta x) - \theta$$
$$\beta = \theta_g - \theta$$

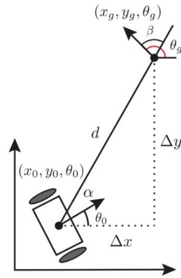


Fig. 3: Motion Controller Diagram

Then, proportional control is used to determine the appropriate linear and turning velocity for the robot, w_{sp} and v_{sp}. For the first rotation, the velocity is: $w_{sp} = K_w \alpha$ where K_w is the proportional gain. For the straight segment, the rotational velocity is $w_{sp} = K_w \alpha$ and the linear velocity is $v_{sp} = K_v d$. This ensures that the robot starts its motion at a higher velocity but slows as it approaches its target location, while also steering to remains pointed towards its goal in case it drifts off course due to non-systematic error. The final rotation has velocity $w_{sp} = K_w \beta$, which simply aligns the robot's heading with its desired heading.

B. Vision

This section describes the method for performing camera calibration and how the blocks' locations are detected, as well as how their rotation within the camera's view frustum is determined.

1) Camera Calibration and Apriltag Detection: In our calibration procedure, a checkerboard with specified dimensions is employed. During calibration roughly 50 frames of the checkerboard image are captured, including various angles and distances. For each image, the function `cv2.findChessboardCorners()` is used to detect the corners of the checkerboard. After frame capture, `cv2.calibrateCamera()` is used to calculate the intrinsic matrix of the camera. For the extrinsic matrix, the mapping between the image coordinate and the world coordinate is calculated by detecting Apriltags of a given size. The results are shown in section III-B1.

C. Gripper Design

The gripper design was developed with a few key design constraints in mind. It was designed to deflect as little as possible under applied loads from lifting, and needed to achieve roughly 125 mm of travel in order to stack cubes. Additionally, the mechanism needed to maintain a smooth vertical linear motion and keep the forks level when lifting. Finally, the lift was designed to not block the camera when the lift was in its lowest position and also was also intended to not block the LIDAR at any point in its range of motion. Since the mechanism design was started before the SLAM and vision detection components were implemented, these final design constraints turned out to be of little importance, but were still reflected in the final design.

A 3D printed carriage and elevator system was used because it allows for smooth linear motion from only one motor, while also being lightweight and having the design versatility that comes from 3D printing. The shape of the carriage and elevator were carefully chosen to optimize the travel and rigidity of the elevator. Finally, a belt drive was used because it allows for the motor to be placed at the bottom of the lift, reducing the center of gravity of the robot. Additionally, a belt drive can produce linear motion without any part of the mechanism extending beyond the top of the elevator like a rack and pinion does, which simplifies the SLAM programming since there is no need to tell the LIDAR system to ignore the rays hitting the lift during specific states.

The final design is given in Fig. 11 in the Appendix.

D. Simultaneous Localization and Mapping (SLAM)

SLAM is a fundamental problem in robotics that enables an autonomous robot to build a map of an unknown environment while simultaneously tracking its own pose within that map. In this project, we map the environment using the LIDAR scans and estimate the robot's pose via Monte Carlo localization. In this section, we present our SLAM system in detail.

1) Mapping: For mapping, our system focuses on updating an 2D occupancy grid map using a 2D LIDAR. The occupancy of each cell is determined by estimated log-probability of the cell being occupied and is updated by incorporating the information from each ray of the LiDAR scan. The LiDAR scan is converted to the global frame and each ray that is shorter than `maxLaserDistance` is decomposed into cells that the laser hit and the cells that it passed through where the log odds are increased by `hitOdds` and decreased by `missOdds`. We employ the Bresenham's line algorithm for tracing the cells that the laser passed through. To consider the robot's movement while scanning, each ray is modified to `movingLaserScan` and these are used to build the map.

2) Monte Carlo Localization:

a) Action Model: The action model aims to predict the robot's pose after taking its action. We model the action model based on how the motion controller uses the RTR method. Each rotation and translation is bound to have some uncertainty in the real world that for each prior particle, the standard deviation of Gaussian noises parameterized by `r1Std`, `trStd`, `r2Std` are injected along with the odometry information as its mean. This

results in a set of particles or a belief of where the robot could be after its motion.

Algorithm 1 RTR Action Model for SLAM

Initialize: $k_1 \leftarrow 0.005$, $k_2 \leftarrow 0.025$
function UPDATEACT(odom)
  $\Delta X \leftarrow$ odom.x − prev.x
  $\Delta Y \leftarrow$ odom.y − prev.y
  $\Delta\theta \leftarrow$ odom.θ − prev.θ
  tr $\leftarrow \sqrt{\Delta X^2 + \Delta Y^2}$
  r1 $\leftarrow$ angleDiff(atan2($\Delta Y, \Delta X$), prev.θ)
  if $|\text{r1}| > \frac{\pi}{2}$ **then**
    r1 $\leftarrow$ angleDiff(π, r1)
    dir $\leftarrow -1.0$
  r2 $\leftarrow$ angleDiff($\Delta\theta$, r1)
  r1Std $\leftarrow \sqrt{k_1 \cdot |\text{r1}|}$
  trStd $\leftarrow \sqrt{k_2 \cdot |\text{tr}|}$
  r2Std $\leftarrow \sqrt{k_1 \cdot |\text{r2}|}$
  tr $\leftarrow$ tr $\cdot$ dir
  prev $\leftarrow$ odom
  return moved
function APPLYACT(samp)
  if moved **then**
    sR1 $\leftarrow$ Normal(r1, r1Std)
    sTr $\leftarrow$ Normal(tr, trStd)
    sR2 $\leftarrow$ Normal(r2, r2Std)
    newS.x $\leftarrow$ newS.x + sTr $\cdot \cos$(samp.θ + sR1)
    newS.y $\leftarrow$ newS.y + sTr $\cdot \sin$(samp.θ + sR1)
    newS.$\theta \leftarrow$ wrapPi(samp.θ + sR1 + sR2)
  return newS

b) Sensor Model: The sensor model scores the likelihood of each particle estimates based on the LIDAR scans after applying the action model by using a simplified likelihood field model. Each particle is scored by iterating through the rays of the LIDAR scan to compute an overall sum of scores.

c) Particle Filter: The pose of the robot is modeled as a particle distribution. The particle filtering algorithm starts by creating a prior distribution by resampling from the previous posterior and then a proposal distribution is computed by applying the action model on the prior particles. Subsequently, based on the LiDAR scans, the likelihood scores are assigned for each particle based on the sensor model. These scores are normalized across particles to get particle weights and its top 10% particles, sorted by weights. These result in a new posterior distribution to resample from for the next timestep.

3) Combined Implementation: The sequential components above are combined to form the full SLAM system as shown in Fig. 4. For initial localization on a given map, a robot's unknown position is estimated by dispersing particles across a map. As the robot moves and collects sensor data, particles are weighted by how closely their predicted sensor readings align with actual data. Through resampling, particles with higher weights are favored, leading to a convergence on the robot's most likely position.

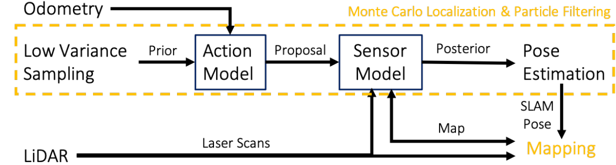


Fig. 4: A block diagram of the SLAM system and interaction between subsystems.

E. Planning and Exploration

1) Path Planning: The objective of the path planning algorithm is to find the optimal path from the robot's current position to its goal position while avoiding obstacles in a computationally efficient manner. The algorithm implemented is an A* algorithm, which combines elements from Djikstra's algorithm and a greedy best-first algorithm to find the shortest path to the destination without the caveats of either standalone algorithm. Djikstra's algorithm traverses cells by expanding from the start node evenly in all directions, which can take a long time to find an optimal path. It is complemented well by a greedy algorithm, which uses a guiding heuristic to determine the order of cells to traverse. For this algorithm, the heuristic used was the Manhattan distance between the current and the goal cell. While this finds a path to the goal very quickly for grids with no obstacles, it struggles in complex mazes and does not explore all cells, often missing out on more efficient paths that Djikstra's algorithm would have found.

These two path planning algorithms are combined in A*, which calculates a g-cost based on Djikstra's algorithm and an h-cost based on the greedy algorithm heuristic. More specifically, the g-cost is the length of the path in cells from the start cell to the current one. This is computed by defining each cell with a parent and adding one to the parent's g-cost count each time a new cell is explored. Additionally, this parent can be reassigned to maintain the shortest path to the start. Finally, the distance to the nearest obstacle is added to the g-cost to prioritize exploring cells further from obstacles. The g- and h-costs are summed to compute an f-cost. The A* algorithm maintains two priority queues: the open queue keeps track of nodes to traverse, and the closed queue keeps track of nodes that have already been traversed and should not be revisited. For each node in the open queue, the algorithm loops through its neighbors and calculates the f-cost for each. If the node is not already in the closed queue, it is added to the open queue. To add on,

if a node in the open queue is traversed and one of its neighbors has a lower g-cost than its parent node, its parent will now point to that neighbor. Since the priority queue is sorted by f-cost, when the loop repeats it will look at the node in the open queue with the lowest f-cost, thus bringing the traversal closer to the goal. Once the traversal reaches the goal, the path from the start cell to end cell is generated based on the path of parents associated with g-cost from the goal cell to the start, which will always be the optimal path because of how Djikstra's algorithm reassigns parent nodes.

2) Exploration: The objective of the exploration algorithm is to provide a set of paths such that the robot can map the entire space that it is contained in with SLAM. The algorithm has the primary objective of navigating to frontiers, which are regions of cells on the border between empty and un-mapped space. More specifically, empty cells are defined as having log odds less than zero and un-mapped cells have a log odds of 0. The `find_map_frontiers()` function finds these regions by finding singular frontier cells and growing these regions until all connected frontier cells are found. If they are above a specified size threshold, they are kept and if not they are ignored. Once the set of valid frontiers is found, their centroids are determined and a path is planned from the robots current position to the centroid of the nearest frontier using A*. Once the robot has travelled close enough to the nearest frontier for it to be mapped, it will no longer be considered a frontier and a path to the new closest frontier is generated.

III. Results

In the following sections, the results of using the previously described methodology is discussed.

A. Motion and Odometry

1) Wheel Calibration: The wheel calibration was evaluated by running many calibration tests and evaluating the variance across these tests. This data is given in Table II below:

TABLE II: Calibration results for left and right motors averaged over five trials.

Trial	L Avg.	R Avg.	L Variance	R Variance
+ Slope	0.0769	0.0671	5.65×10^{-5}	4.47×10^{-5}
+ Intercept	0.0593	0.0544	1.12×10^{-6}	5.56×10^{-6}
− Slope	0.0686	0.0752	4.18×10^{-5}	2.95×10^{-5}
− Intercept	−0.0760	−0.0446	7.52×10^{-5}	2.38×10^{-5}

2) Odometry: The odometry system was evaluated by comparing measurements of known distances traveled by the robot to the reported distance given by the encoders and odometry calculations. In Table III, the average errors in x, y, and z were 2.0%, 4.53%, and 7.1% respectively. These tests indicated that the odometry was more accurate for straight line travel than for turning and that the error accumulated the further the robot traveled.

TABLE III: A comparison of odometry measurements against ground-truth movements.

Measurement	Odom. Result	Actual Travel	Error
x_1	50.0 cm	48.5 cm	3.0 %
x_2	50.0 cm	49.0 cm	2.0 %
x_3	50.0 cm	49.5 cm	1.0 %
y_1	32.5 cm	30.0 cm	7.7 %
y_2	30.0 cm	31.0 cm	3.3 %
y_3	38.0 cm	39.0 cm	2.6 %
θ_1	−3.10 rad	−3.14 rad	7.7 %
θ_2	1.68 rad	1.57 rad	6.5 %
θ_3	0.002 rad	0.0 rad	- %

Implementing gyrodometry helped reduce this error accumulation, but had negligible impact on simple paths. As seen in Fig. 5, the odometry system performs quite well on a square path. The same test was ran using gyrodometry, and the results were nearly identical.

3) Wheel Speed PID: The wheel speed controller was successful in smoothly accelerating the robot to its desired velocity. For a desired angular velocity of 2π, or 6.28 rad/s, the wheels achieved roughly 6.20 rad/s. This is 1.3% error, which is well within the range needed for this project. Without feed-forward control from calibration, the wheels spun at 3.15 rad/s.

4) Motion Controller: The rotate-translate-rotate motion controller excelled on simple paths with straight lines and stationary turns, such as the drive square path shown in 5. Similarly, the RTR controller excelled at navigating around the cones in event 1 of the competition.

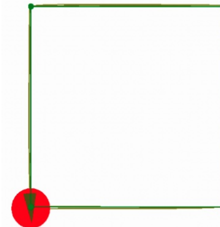


Fig. 5: A drive square path (dark green) followed by the robot 4x by dead-reckoning (lighter green).

Fig. 6 shows the linear and rotational velocity of the robot as it drives around the square in Fig. 5. From the plot, it is apparent that the robot alternates between linear translation and rotation motions.

B. Vision

1) Camera Calibration: After camera calibration, the intrinsic K and extrinsic matrix $[R|t]$ are found.

$$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1266.2 & 0.0 & 645.1 \\ 0.0 & 1268.8 & 370.9 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

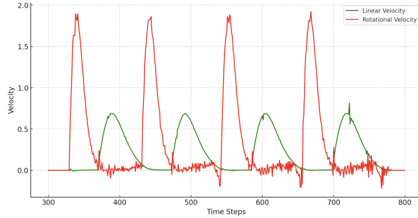


Fig. 6: A graph of linear (green) and rotational (red) velocities of the mbot while driving in a square.

TABLE IV: A comparison of AprilTag detection with odometry translation.

Time (t)	Translation (x, y, z)	Angle (deg)	Distance (m)	AprilTags Trans. (y + Dist.)
1	(0, 0, 0)	4.97	0.232	0.2326
2	(-0.165, -0.0075, 0.0611)	3.5	0.452	0.287
3	(-0.377, -0.0217, 0.0766)	1.42	0.625	0.248
4	(-0.600, -0.0493, 0.063)	3.1	0.787	0.187
5	(-0.709, 0.106, -0.452)	3.4	0.961	0.252

$$[R|t] = \begin{bmatrix} 0.773 & -0.329 & -0.543 & -0.210 \\ 0.542 & 0.787 & 0.294 & -0.058 \\ 0.331 & -0.521 & 0.787 & 0.631 \end{bmatrix}$$

Here, f_x and f_y are the focal lengths expressed in pixel units, s is the skew coefficient (often zero in modern cameras), and (x_0, y_0) represents the coordinates of the principal point in the image. R is a 3x3 rotation matrix and t is a 3x1 translation vector.

2) Apriltags Detection: For evaluating the performance of the apriltags detection, we compare the camera detection of the apriltag with your odometry by driving backwards and while keeping the apriltag in sight. The results are shown on Table IV. Based on five different measurements, the detected AprilTag locations have a low variance of 0.00105.

C. Gripper

The gripper mechanism functioned exactly as expected in the competition. It was quite stiff and was able to lift many blocks at once while holding the stack straight up and down. Additionally, the elevator had enough travel to stack blocks while not blocking LIDAR rays. Unfortunately, there was insufficient time to integrate AprilTag detection with the lifter so it was only controlled manually during the competition.

D. Simultaneous Localization and Mapping (SLAM)

The mapping accuracy of our SLAM system is qualitatively assessed by comparing against the real world map and a pre-recorded log file such as `drive_maze.log`.

Fig. 7: Mapping of the drive_maze.log file.

The localization accuracy of our SLAM system is estimated by comparing and computing the RMS error against given the `drive_maze_full_rays.log` file with ground-truth poses of the robot. The RMS error for the position is 0.123 m and the orientation is 0.282 rad. In Fig. 8, our particle filter is assessed by replaying the `drive_square.log` file and by recording our particles at regular intervals. As shown, the particles form a near-perfect square path with low spread of particles, which indicates that the action and the sensor models are performing well as a part of particle filtering.

Table V consists of the action model parameters used to compute the standard deviations of Gaussian noises to model uncertainties in the rotational (k_1) and translational (k_2) movements of the RTR controller. These parameters are used in Algorithm 1 and were chosen by several trials. As in Table VI, the maximum number of paticles that our particle filter can support at 10Hz is estimated to be 1077 particles by linear regression. Overall, our SLAM is fast, stable, and functional.

TABLE V: Uncertainty parameters of the action model.

Action Model Parameters	k_1	k_2
Value	0.005	0.025

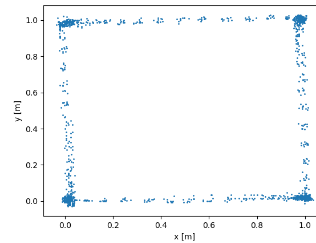


Fig. 8: 300 particles drawn at regular intervals along the drive_square.log path using the particle filter.

TABLE VI: Time taken for updating the particle filter when using 100, 500, and 1000 particles.

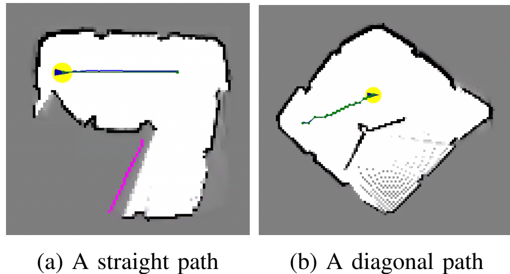
Number of Particles	100	500	1000
Update Time (ms)	12.7044	58.0963	88.6907

E. Planning and Exploration

The planning and exploration algorithms were validated with a set of provided test cases and through the robot's performance on Event 3 of the competition.

1) Path Planning: Overall, the A* path planner excelled in choosing an optimal path in a quick manner. While it required tuning to balance the g-cost, h-cost, and obstacle avoidance goals, it was ultimately successful in producing a path that allowed the robot to explore an entire maze without collisions and in a reasonable computational time. In Table VII, the statistics on the A* path planning execution on the test maps are provided.

As seen in Fig. 9a, 9b, and 10a, the robot followed the planned A* path very closely for simple and complex paths. It succeeded in planning more difficult paths, as in Fig. 10b, but it was difficult to check the accuracy of the robot's actual motion since the planned path constantly changed during the exploration of the map.



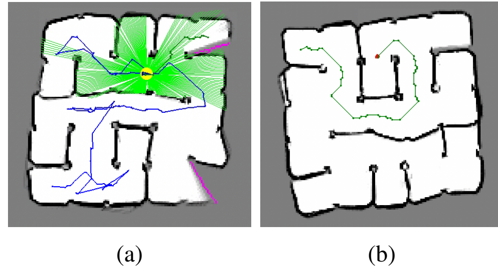
(a) A straight path (b) A diagonal path

Fig. 9: Comparison of A* (green) and actual path (blue).

2) Exploration: The exploration algorithm excelled in finding frontiers and using A* to plan a path to the centroids of the frontiers. As seen in Fig. 10a, the robot has explored nearly the entire map and is in the process of driving to another frontier during the time of the screenshot. From the image, it is evident that the robot drove just far enough to create a map of every corner of the maze before planning a path to the next frontier.

TABLE VII: Timing information for successful A* planning for various test maps in microseconds.

Map	Min	Mean	Max	Median	Std dev
empty	5947	9209.33	13649	13649	3252.67
filled	14	27	58	23	16.53
narrow	4139	356591	1061170	4139	498212
wide	4278	312658	929164	4531	435936
convex	139	15982	31825	16841	15843
maze	3817	30629.8	75558	18462	27022.7



(a) (b)

Fig. 10: (a) Exploration and (b) return paths.

While the exploration algorithm was successful in planning paths to frontiers, it struggled to get the robot to follow the planned path. The robot exhibited some undesirable behavior such as over-rotating past the desired direction of travel according to the A* path. Additionally, the robot still hit walls on occasion despite the A* algorithm being programmed to generate paths with a buffer between the robot and obstacles.

F. Competition Results

Our robot was validated through a competition with four events. In the first event, Speed Run, the robot successfully mapped the arena and returned to the starting pose with an error of under 20 cm. In the second event, Heavy Lift, we were unable to fully integrate elevator lift control with AprilTag navigation but were able to manually stack three sets of cubes using PyGame remote control. In the third event, Maze Explorer, we successfully explored and mapped the maze environment and returned to our starting pose with an error of under 5 cm. We completed the fourth event via teleoperation. Overall, our robot's performance earned 600 points, which tied for 2nd place overall in the AM leaderboard.

IV. Discussion

A. Motion and Odometry

The average variance for the slope of the linear portion is 4.31×10^{-5} and 2.64×10^{-5} for the intercepts, which is small. This indicates that the calibration routine is consistent, but some variation exists. There are many potential causes of this variation. One such source is that each calibration routine was run in a slightly different spot on the floor, and these slight changes in floor roughness and flatness will impact the variation. Other sources may include mechanical wear, power supply variation, and environmental conditions such as dust.

Our odometry performed well for straight lines, but struggled to accurately measure changes in the robot's heading and on longer paths. This was evident from comparing the SLAM pose to the odometry pose. As the robot got closer to its goal, the difference between these

poses increased and the SLAM pose better reflected the actual state of the robot. While gyrodometry helped to mitigate this error by accounting for slipping and other non-systematic error, it still failed to accurately localize the robot on longer paths through the maze.

The Wheel Speed PID controller was successful in getting the wheels to spin at the desired velocity. Additionally, the low-pass filter was successful in filtering out high-frequency noise in the input command. When using `mbot_test_drive.py`, the signal would constantly drop out, causing the robot to frequently start, stop, and face-plant before the low-pass filter was implemented. The PID parameters were tuned through trial and error in order to achieve desirable performance. The metrics were the rotational velocity of the wheel compared to its target velocity and acceleration. The parameters were deemed to be tuned when the robot had smooth acceleration and deceleration and achieved the target velocity.

While the RTR motion controller excelled on straight line paths, its drawbacks were evident in the map exploration portion of the competition. Since RTR rotates in place before it begins translating, it wastes substantial time aligning itself with its target direction. Since the exploration algorithm constantly updates the optimal path, the robot would frequently rotate back and forth but make very slow progress towards the goal cell. A different motion controller algorithm such as pure pursuit would have likely resulted in faster exploration times.

B. Vision

While visual servoing towards a warehouse block via AprilTag detection was functional, yet it mostlyonly worked mostly in a constrained and manually designed settings. To build a fully autonomous Mbot, further efforts would beare required in automatic block finding the blocks within an unknown map using SLAM and more sophisticated maneuvering to manipulate the blocks.

C. Gripper

The gripper worked exactly as intended but was designed to a tighter set of design constraints than was necessary. It would have been simpler to design if it would have been allowed to obstruct the camera and LIDAR systems. The strengths of the gripper was the smoothness of the elevator, stiffness of the forks, and high lifting capacity. It could have been improved by increasing the travel of the elevator to make it easier to stack blocks on top of each other.

D. Simultaneous Localization and Mapping (SLAM)

As shown by the mapped figures in the report, the SLAM system was functional most of the time. It was able to map the environment fast and simultaneously localize itself within that map via particle filtering based on the likelihood scores from the LiDAR scans. Occasionally, the robot jumped within the map with rotational symmetry of the map when it lost track of its own pose. However, this was resolved after reducing the range of neighborhood for the simplified likelihood field model.

E. Planning and Exploration

The A* path planning worked quite well with few caveats. It was computationally fast enough to be practical even on long paths, and successfully maintained a safe distance from obstacles while having a preference for cells further from obstacles. The main room for improvement would be better pruning of the path to remove way-points clustered near each other.

The exploration algorithm was successful on some occasions but not all. It's possible that it failed on occasion because the initial rotation in the RTR motion controller would cause enough movement for a new path to be generated, thus causing the robot to stay at its current location and inhibiting progress towards the goal. Additionally, SLAM would occasionally localize the robot incorrectly, causing the robot to collide with a wall during its exploration. If more time were available, the most valuable improvement would be changes to the motion controller and A* path pruning to prevent the robot from getting stuck at the start of its path.

V. Conclusion

To conclude, the project covered the topics of odometry, PID and motion control of a differential drive robot along with the occupancy grid-based SLAM system with Monte Carlo localization using the LiDAR. Furthermore, A* path planning and a state machine-based frontier exploration methods were covered to build a map of an unknown environment autonomously. Visual servoing and manipulation of the cubes is accomplished by a RGB camera and a 3D printed elevator design. These software and hardware stacks were successfully integrated resulting in a 2nd place finish in the competition.

References

[1] J. Borenstein and L. Feng, "Gyrodometry: a new method for combining data from gyros and odometry in mobile robots," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, 1996, pp. 423–428 vol.1.

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

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

VI. APPENDICES

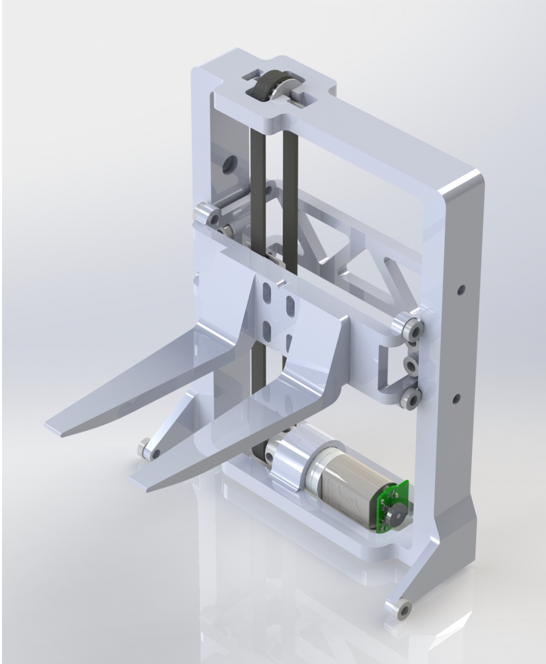


Fig. 11: The final elevator and gripper design for the lifting mechanism of the mbot.