Team 10 Section 2 ROB 550 BotLab Report

Anurekha Ravikumar, Billy Mazotti, Hongjiao Qiang
{anurekha, bmazotti, hqiang}@umich.edu

***Abstract*—This report presents the development and implementation of a robotic forklift using wheel odometry, LiDAR sensor measurements, and a custom lifting mechanism for automated forklift tasks. Monte-Carlo Localization, a particle-filter based localization algorithm, is used for 2D simultaneous localization and mapping (SLAM), while A* path planning is used for navigating the generated 2D occupancy grid. For detecting the pose of and stacking AprilTag fiducial-marked crates, the robot uses computer vision and a 4-bar linkage forklift design respectively. The robot demonstrated the ability to carry out complex exploration and stacking tasks, including semi-autonomous mapping of an unknown maze along with fully-autonomously stacking one crate.**

I. Introduction

THIS report outlines the development of a mbot-classic ground driving forklift robot that uses motion control, simultaneous localization and mapping (SLAM) and path planning to compete in a variety of competitions. Several key components are required for the system to work properly, including wheel speed motion control for differential drive motion control, integration of odometry and LiDAR sensor data into action and sensor models for Monte-Carlo localization, A* Path Planning for exploring unknown areas, and mechanism design, manufacturing, and integration for forklift capabilities. Our paper describes these techniques and features developed and integrated to implement the obstacle avoiding, crate lifting, and terrain exploring robot. The performance is validated on tasks requiring dead-reckoning path following, 2D occupancy map generation, unknown map exploration, and stacking crates.

II. Methodology

A. Motion and Odometry

*1) **Wheel Speed Calibration**:* We used the calibration program provided, which measures the robot's motion, to determine the polarity of the sensors and motors and perform a wheel speed calibration. By launching `mbot_calibrate_classic.uf2`, we automatically find the terrain-specific PWM to wheel rotationla velocity relationship necessary for wheel velocity control.

*2) **Odometry**:* : For ensuring functional and accurate odometry, we verify encoder performance using `mbot_encoder_test.uf2` and edit `mbot_params.h` to include the manually measured wheel baseline radius. In particular, accurate wheel baseline radius is necessary when incorporating the kinematics of the robot through PID control of the wheel speed. While incorporating IMU through gyrodometry was considered, we found sufficient performance with solely estimating odoemtry based on the wheel encoders.

*3) **Wheel Speed PID Controller**:* The final controller in the robot's firmware is designed to drive the motors based on two different control modes: Open Loop (OL), as seen in Fig. 11, and Proportional-Integral-Derivative (PID) control, as seen in Fig. 12, which is part of a closed-loop system.

Open Loop Control (MODE_MOTOR_VEL_OL): In this mode, the controller sends pre-calculated PWM signals directly to the left and right motors without using sensor feedback to adjust commands. This is based on a function correlating desired velocity with PWM, assuming predictable motor response. However, it doesn't account for variations between commanded and actual wheel speeds, which can result in motion inaccuracies.

PID Control (MODE_MOTOR_VEL_PID): In this mode, feedback control via PID controllers enhances speed precision. Linear/angular velocities undergo low-pass filtering (rc_filter_march with lowpass_linear/angular) for noise reduction. Angular error, defined as the difference between filtered commanded angular velocity and actual sensor-measured velocity, is processed by the angular velocity PID controller (rc_filter_march with pid_angular). Wheel velocities are derived from commanded linear/angular velocities using kinematic equations, considering robot dimensions (DIFF_BASE_RADIUS, DIFF_WHEEL_RADIUS). Left/right wheel errors, the disparity between desired and actual velocities (measured by encoders), are managed by individual wheel PID controllers to adjust velocity commands. Final PWM commands for each wheel combine calibrated PWM feedforward terms with PID controller outputs (ff_left + pid_output_left and ff_right + pid_output_right).

Feed-forward (FF) Control: The baseline control directly computes motor commands from robot kinematics in an open-loop system. It uses calibration data to convert desired wheel speeds into PWM motor commands.

Feedback (FB) Control with PID Controllers: The feedback control uses PID controllers to modify motor commands, correcting discrepancies between desired and actual wheel speeds. The proportional term tackles current error, the integral term deals with accumulated past errors, and the derivative term anticipates future errors using the current rate of change. Separate PID controllers are implemented for each wheel in the code.

Low-pass Filters: Velocity estimates from encoders are low-pass filtered to minimize noise, particularly at low speeds. These filtered velocities are then utilized by PID controllers for smoother motor command adjustments.

The final controller leverages feedback from wheel encoders and other sensors like IMUs to correct motion errors, ensuring more reliable and accurate control over the robot's trajectory.

*4) **Motion Controller**:* The motion control algorithm employed for navigating between waypoints integrates several distinct maneuver controllers. Central to the algorithm are three primary controllers: `StraightManeuverController`, `TurnManeuverController`, and `SmartManeuverController`. Each controller is designed for specific navigational tasks: the `StraightManeuverController` for linear motion towards a target, the `TurnManeuverController` for rotational alignment with the target orientation, and the `SmartManeuverController` which amalgamates linear and rotational movements for efficient path following. The core mechanism underlying these controllers is the Proportional-Integral-Derivative (PID) control system, utilized for computing the velocity commands based on the positional and orientational errors. The PID parameters for forward and turning movements (`fwd_pid` and `turn_pid`) are calibrated to optimize the responsiveness of the Mbot.

In practice, the `MotionController` class efficiently coordinates the Mbot's navigation, seamlessly transitioning between controllers (`INITIAL_TURN`, `DRIVE`, `FINAL_TURN`) based on the robot's status and upcoming waypoints. This process is enhanced by a real-time feedback loop within the main program, where the `updateCommand` function continuously adjusts motor commands in response to odometry data, ensuring the Mbot's actions align with current navigational needs. Safety protocols are embedded to regulate the maximum forward and angular velocities, maintaining the Mbot's operational integrity.

B. Simultaneous Localization and Mapping (SLAM)

Our SLAM framework combines an occupancy grid for initial mapping and Monte Carlo Localization for enhanced accuracy within a predefined map. This integration forms a sophisticated SLAM system, effectively addressing navigation and mapping challenges in specific environments.

*1) **Mapping**:* The mapping component plays a pivotal role, which primarily utilizes an Occupancy Grid—a data structure representing the environment—is dynamically updated based on sensor inputs.

Our mapping process initiates with the `Mapping` class, where key parameters like `maxLaserDistance`, `hitOdds`, and `missOdds` are set for laser scan interpretation. The central function, `updateMap`, updates the Occupancy Grid map using a new lidar scan (`scan`) and the current pose (`pose`). This involves two primary actions: scoring the endpoint of each laser ray in `scoreEndpoint`, where we increase occupancy probability log odds for cells with detected obstacles, and implementing `scoreRay` using Bresenham's algorithm to decrease log odds for cells indicating free space. Methods `increaseCellOdds` and `decreaseCellOdds` dynamically adjust cell occupancy odds based on lidar data, ensuring an up-to-date environmental representation.

The objective of a moving laser scan is to accurately capture and interpret laser scan data while the robot is in motion. This process involves adjusting the laser rays based on the scanner's movement (between `beginPose` and `endPose`) during the scan period. By interpolating the position and orientation of each ray using the `interpolate_pose_by_time` function, it accounts for the robot's dynamic nature, ensuring that the scan accurately reflects the environment as the robot navigates through it.

*2) **Monte Carlo Localization**:* Monte Carlo Localization (MCL) is a crucial particle filtering algorithm for estimating a robot's position. It combines an action model, predicting the robot's pose from its movements, and a sensor model, evaluating pose likelihood from sensor data. MCL's process involves drawing and normalizing particle samples to represent probabilities and calculating their weighted mean to pinpoint the robot's most probable position.

*a) **MCL - Action Model**:* The Action Model in our framework plays a vital role in predicting the robot's new pose using its movements. It uses the updateAction function to compute changes in position and orientation from odometry data and checks against minimum thresholds for significant movement. When such movement is detected, it calculates standard deviations for these movements, which the applyAction method uses to sample new poses from normal distributions. This process

includes generating new values for rotation and translation, updating the current pose to predict the robot's next position. Overall, the model combines translational and rotational dynamics with probabilistic modeling to accurately estimate the robot's new pose.

Below are the key equations and parameters used in the action model.

Modeling Action Equations:

- $(x_{t-1}, y_{t-1}, \theta_{t-1})$ – previous pose
- (x_t, y_t, θ_t) – current pose
- $(\Delta x, \Delta y, \Delta\theta)$ – change in pose

The change in distance squared is given by:

$$\Delta s^2 = \Delta x^2 + \Delta y^2 \tag{1}$$

The angle α is computed as:

$$\alpha = \text{atan2}(\Delta y, \Delta x) - \theta_{t-1} \tag{2}$$

The model action as a rotation, translation, and rotation is:

$$\mathbf{u} = \begin{bmatrix} \alpha \\ \Delta s \\ \Delta\theta - \alpha \end{bmatrix} \tag{3}$$

Given action: $\mathbf{u} = [\alpha \quad \Delta s \quad \Delta\theta - \alpha]$

The action model $P(\mathbf{x}_t | \mathbf{u}_{t-1}, \mathbf{x}_{t-1})$ is defined as:

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

Where:

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

TABLE I: Uncertainty Parameters in Action Model

Parameter	Description	Value
$k1$	Noise in rotational movement	0.0005f
$k2$	Noise in translational movement	0.0005f
min_dist	Minimum significant distance	0.000
min_theta	Minimum significant angle	0.000

The values for $k1$ and $k2$ were chosen to represent the noise present in rotational and translational movements of the robot, respectively. These values are small, indicating a relatively low level of noise in the robot's movement, and are typically determined through empirical testing and calibration. The thresholds min_dist and min_theta are set to zero, implying that any movement, no matter how small, is considered significant. This setting allows the model to be more responsive to minor positional changes, crucial in high-precision environments.

*b) **MCL - Sensor Model**:* Our sensor model is based on the implementation of the Likelihood Field Model. To overcome the lack-of-smoothness and computational limitations of the Beam Model, instead of calculating the entire ray casting, the Likelihood Field model only checks the end point. It first finds the end point of rays, find the nearest cell with breadth first search, and calculate the distance between the two points. The distance will then be taken into the likelihood function prob$(dist, \sigma_{hit})$, which is given by a Gaussian distribution $\frac{1}{\sqrt{2\pi}(\sigma_{hit})} exp(-\frac{dist^2}{2(\sigma_{hit})^2})$. In our implementation, we define the parameters $z_{hit} = 0.6$, $z_{random} = 0.005$, $z_{max} = 1000$ for calculating the `q` value, which is returned as the weight of each particle in proposal.

*c) **Particle Filter**:* The particle filter is where we combine both action and sensor models to apply action and update pose for the robot. For initialization, we set the pose of each particle to a specified 2D pose. The particles are assigned with the same pose, and their weights are initialized based on the inverse of the total number of particles. The `updateFilter()` Function is repeatedly called to use the action model to estimate if the robot has moved, apply the action, perform resampling, and update particle weights accordingly. For resampling, we implemented the Low Variance Resampling algorithm to avoid amplifying the variance.

*3) **Combined Implementation**:* Combining mapping and localization models, we have our SLAM system shown as Figure 1. The initial laser scan will be taken into the mapping algorithm and the original map will be produced. The original map will then be taken into the localization algorithm. The action model will update and apply the actions with the robot, and the sensor model will compute the estimated posterior pose. Then the updated pose information with the new laser scans will be taken into the mapping algorithm again to update the map. This forms an iteration and will continuously and simultaneously realize mapping and localization.

C. Planning and Exploration

*1) **Planning**:* The path planning algorithm we implemented is A*, with an obstacle distance grid incorporated to avoid being close to obstacles.

*a) **Obstacle Distance Grid**:* We used the method of obstacle distance grid to generate the distance to the nearest obstacle for each cell in a given occupancy map. The algorithm uses a priority-queue-based approach to efficiently compute the distance to obstacle and expand for cells. The generated obstacle distance grid were used to give extra penalty to cells that are closed to obstacles.

*b) **A* Algorithm Implementation**:* For planning a path from one pose in the environment to another, we

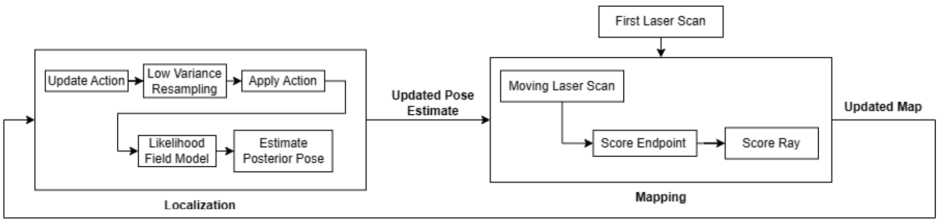

Fig. 1: SLAM System Diagram

implemented the A* algorithm, which employs a priority queue to explore nodes in order of increasing cost.

The core function `search_for_path()` repeatedly pops the node with the lowest `f_cost` from the open queue. The `f_cost` is the sum of the cost to reach the current node (`g_cost`) and the heuristic estimate from the current node to the goal (`h_cost`). In our implementation, the `h_cost` is calculated in the 8-way style, considering both cardinal and diagonal distances. Because of A* algorithm always finds the shortest past towards the goal, we had this issue that optimal paths tended to follow obstacles. The risk of travelling close to obstacles may negatively affect the accuracy of mapping and localization. Therefore, when calculating the `g_cost`, we incorporated the obstacle distance grid so that we penalize cells that are close to the walls. We added the value $1/obstacle\_distance(x, y)$ to the `g_cost` of the cells. Therefore, closer a cell is to the obstacle, larger the penalty it will get.

The popped node will be set to the `current_Node`. The algorithm checks if the `current_Node` is the goal. If it is, the path is considered found. If the `current_Node` is not the goal, the function expands the node by generating its children using the `expand_Node()` function, which explores adjacent grid cells in 8 directions and returns a vector of possible child nodes.

For each child node, the algorithm checks whether it has not been visited before, whether it is within the grid bounds, and whether it is traversable based on obstacle distances. If these conditions are met, the function calculates the `g_cost` and `h_cost` for the child node and adds it to both the open list and visited list. If the child node has been visited but it was with a higher `g_cost`, the function updates its cost in the visited list and adds it back to the open list.

After a path is found, a list of nodes will be extracted by the `extract_node_path()` function. It takes the goal node as the original `current_Node`, and keeps visiting the parent of the `current_Node` until the start node is reached. Then the list of nodes will be transformed to a list of poses, which is the valid input that the robot can take for execution. Consisting of `x`, `y`, and `theta`, the poses are generated such that`x` and `y` are calculated by transferring the corresponding `x` and `y` of the node from grid to global position, and `theta` is calculated by `{atan2}({dx,dy})`, where `dx` and `dy` are the displacement between two adjacent nodes.

*2) **Exploration**:*

*a) **Frontier Searching Logic**:* Our definition of a frontier is a cell with 0 log-odds, and with a neighbour that has log-odds smaller than 0. According to this definition, we created a function called `is_frontier_cell()` to validate if a cell can be considered as a frontier.

To find frontiers, we use a connected components search in the occupancy grid. Each connected components consists only of cells where `is_frontier_cell()` returns true. We scan the grid until an unvisited frontier cell is encountered, then we grow that frontier until all connected cells are found. We then continue scanning through the grid. .

*b) **Exploration State Machine**:* In the exploration state machine, there are states including `INITIALIZING`, `EXPLORING_MAP`, `RETURNING_HOME`, `COMPLETED_EXPLORATION`, `FAILED_EXPLORATION`.

Immediately after the first bit of data has arrived the lcm channel, the state machine will transtition from the `INITIALIZING` state to the `EXPLORING_MAP` state. In the `EXPLORING_MAP` state, all the frontiers in the map will be found, the one with the nearest centroid will be selected, and a path towards it will be planned. The planned path will then be sent to the `CONTROLLER_PATH_CHANNEL` and a sleep time for its execution will be given. The sleep time is calculated by multiplying a time unit of 0.5 second with the number of nodes in the generated path.

After the current path is executed, if all the frontiers found have been explored, the state machine will transition to `RETURNING_HOME` state, where the robot will follow a path generated by setting the home pose as the goal; if there is still at least an unexplored frontier, it will remain in the `EXPLORING_MAP` state; if there is no valid path to follow and the robot is not at the home pose, it will transition to the `FAILED_EXPLORATION` state.

D. *Gripper Design*

During the design concept process, we prioritized simplicity of motor integration. For this reason, we chose the Tiankongrc MG90s Micro Servor and a 4-bar linkage mechanism design (see Fig. 2. While the original design only employed one motor on one side of the mechanism, an additional motor was added to provide additional structural support and torque.

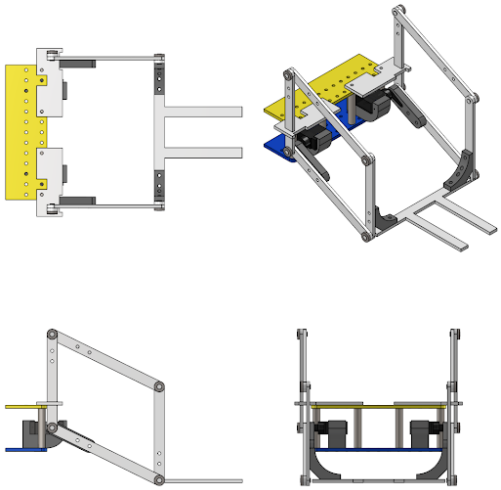

Fig. 2: 4-view of Forklift Gripper Design

All components were provided by the course or custom manufactured using a CO2 Laser Cutter for rapid prototyping of simple parts and FDM 3D printer effective fabrication of complicated parts. The vertical beam holders, 4-bar linkage beams, and fork component were made out of laser cut acrylic. We chose 3.175mm thick acrylic over the course's original supply of thicker acrylic in order to use the course's OLLO 9mm long Rivet's as detachable pivots for our 4-bar linkage. The adapters holding the servo motor to the robot, the servo lever arm to the bottom arm of the 4-bar linkage, and the outer vertical column of the 4-bar linkage to the fork component were made out of 3D printed PLA. Each 3D printed part used the course provided threaded inserts and M2.5 x 6mm screws for fastening together all components of the mechanism. For more details on the design, manufacturing, and assembly of our forklift, please see design_files.zip.

The columns (vertical) and arms of the 4-bar linage, along with the fork beams and

The process for stacking a crate came down the the following steps:

1) align the robot in front of the crate of interest with a distance of approximately 20cm between box and robot's front camera
2) lower the forklift (Fig. 3) and drive forward to insert the forklift's fork into the pallet of the crate
3) raise the forklift (Fig. 3) s.t. the bottom of the lifted crate reaches clears the heights of unstacked crates
4) drive to the second crate of interest and lower the forklift until the bottom of the crate is in contact with the top of the second crate and the fork beams are no longer in contact with the pallet
5) reverse the robot until a safe distance has been reached for rotating towards the next waypoint

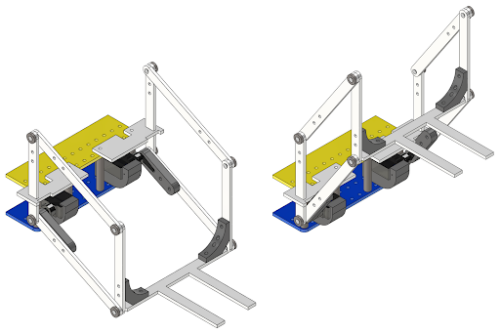

Fig. 3: Forklift in the down (left) and up (right) positions.

While the mechanical integration of the servo motors was simple, our mechanical design required calibration and additional motor software development. Due to right motor being flipped compared to the left motor, the normalized angle commands sent to the right motor needed to be flipped both during motor initialization and new forklift commands. Additionally, since normalized angle commands were used rather than radians or degrees, set angles for the aforementioned lowered, raised, and stacking steps needed to be manually tested and calibrated for.

III. Results

A. *Motion and Odometry*

To test the accuracy of motion controller, we commanded the robot to drive a 1 meter by 1 meter square 4 times. Using tape, we created an outline of the square on the floor to serve as a ground truth visual of where the robot should ideally drive. During the first loop, we recorded the linear and rotational velocities as seen in Fig. 4. From this plot we observed that both the maximum linear and rotational velocities limits of 0.2 m/s and $\pi/4$.

For all of the 4 loops, we also plotted the dead-reckoning odometry position estimates as seen in Fig. 5. The odometry estimates do not perfectly match the observed performance, especially for the 3rd and 4th loops when the accumulated error drift was greatest. However, we found even this level of estimates to be sufficient for competing in events 1 and 3 discussed in III-E.

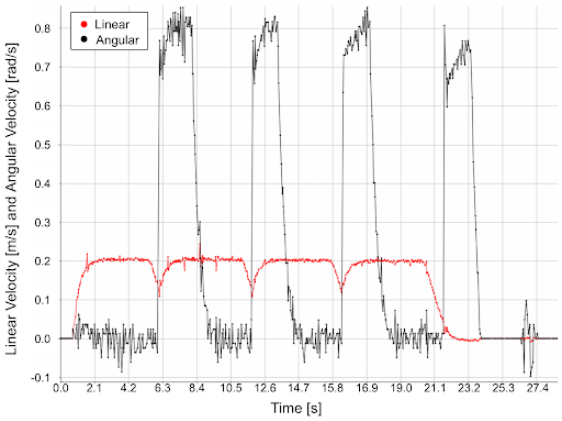

Fig. 4: Robot Velocities for Single Square Loop

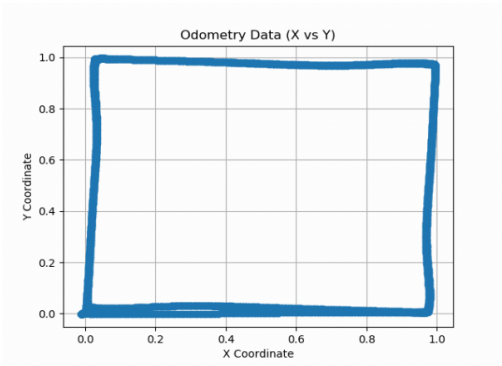

Fig. 5: Dead reckoning pose estimates while the robot drives a 1m x 1m square 4 times

TABLE II shows the result from multiple calibration trials of the robot's motors on a concrete floor, which revealed a pattern in the drive system's performance. The variation in calibration, as indicated by the variances, is minimal for the slopes but more significant for the intercepts. This suggests that while the robot can consistently translate PWM signals into changes in speed, the starting speed or base speed without any PWM signal is subject to more fluctuation.

The sources of variation could include mechanical inconsistencies, such as slight differences in motor construction or wheel diameter, variability in the encoder readings due to environmental factors such as sensor noise, differences in friction or traction between the robot's wheels and the concrete surface, which may be affected by surface texture or debris and potential thermal effects where the motor or electronic components may perform differently at varying temperatures.

TABLE III outlines the parameters for the controller, including the gains for the PID controllers and the parameters for the low-pass filters used for linear and angular velocities. These parameters are vital for the tuning of the controller

TABLE II: Motor calibration with variance for concrete

Parameter	Mean	Variance
Positive Slope Left	0.050927	4.87e-08
Positive Slope Right	0.062507	4.47e-08
Positive Intercept Left	0.064994	6.61e-06
Positive Intercept Right	0.061415	1.15e-06
Negative Slope Left	0.058345	1.69e-07
Negative Slope Right	0.056040	2.74e-07
Negative Intercept Left	-0.043988	1.30e-05
Negative Intercept Right	-0.078355	2.52e-05

TABLE III: Controller Parameters

Parameter	Value
SAMPLE_RATE	100 Hz (derived)
dt (Delta Time)	0.01 seconds
tc (Time Constant)	0.1 seconds
Low-pass Filter - Linear	
lowpass_linear	First-order
Low-pass Filter - Angular	
lowpass_angular	First-order
PID Controller - Angular	
(kp, ki, kd)	(0.08, 0.0, 0.0)
Tf_angular	0.01 seconds
PID Controller - Left Wheel	
(kp, ki, kd)	(0.08, 0.0, 0.001)
PID Controller - Right Wheel	
(kp, ki, kd)	(0.07, 0.0, 0.001)

For evaluating our controller's performance, we adopted an iterative tuning process, starting with the proportional gain (Kp) and incrementally increasing it to the point of system oscillation, before reducing to stabilize the response. The integral (Ki) and derivative (Kd) gains were then introduced to minimize steady-state error and overshoot, respectively. Additionally, we implemented feed-forward control to enhance the predictability of responses and low-pass filters to mitigate sensor noise, leading to smoother operation. The culmination of these features resulted in a robust and responsive control system, with improvements in adherence to desired trajectories and agility in maneuvering.

B. Simultaneous Localization and Mapping (SLAM)

Fig 6 shows the example of mapping result of running the log file $mbot\_example\_logs/drive\_maze.log$.

TABLE IV shows the time it takes to update 100, 500 and 1000 particles. From our estimation, the maximum number of particles our filter can support running at 10 Hz is around 750 particles.

Fig. 7 shows the example result of running our particle filter on $mbot\_example\_logs/drive\_sqaure.log$. From

TABLE IV: Update Times for Various Particle Quantities

Number of Particles	100	500	1000
Update time (ms)	11.7	72.3	142.9

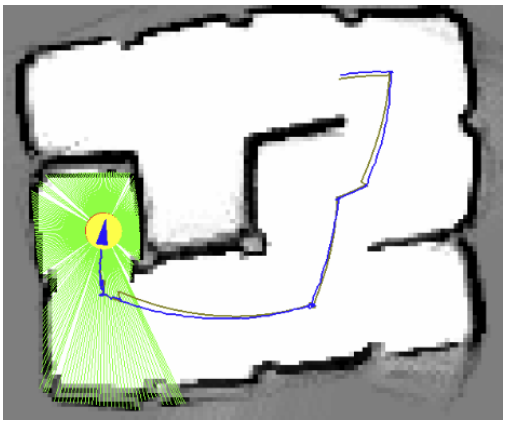

Fig. 6: Mapping Implementation From the Log File mbot_example_logs/drive_maze.log

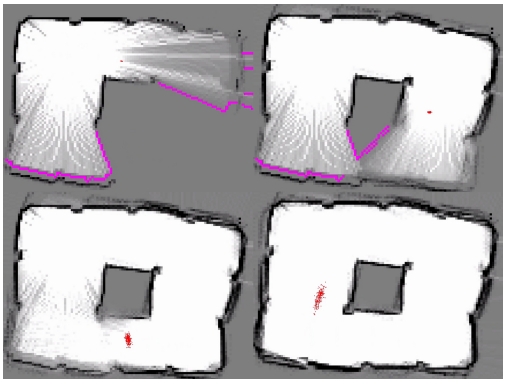

Fig. 7: Plot of 300 Particles Running drive_square.log

upper left to lower right, the four subplots are the particles' plot at 4 chronological intervals along the path the robot takes.

Fig. 8 is the absolute trajectory error (ATE) aligned plot comparing the estimated poses from our SLAM system with the ground truth poses in drive_maze_full_rays.log. The pose estimated by our SLAM system is more accurate in the first half of the path, but deviates slightly more in the second half.

TABLE V shows the statistics of the error between SLAM pose and the ground truth. From theses numbers, our SLAM system demonstrates overall good accuracy in pose estimation.

C. *Planning and Exploration*

TABLE VI shows the statistics on our path planning execution times for each of the example cases in

TABLE V: SLAM Pose Error in units of meters

RMS	Mean	Median	Std Dev	Min	Max
0.146	0.124	0.103	0.076	0.013	0.376

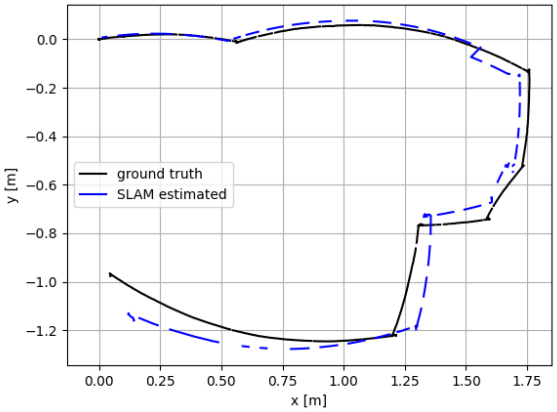

Fig. 8: SLAM Pose vs. Ground Truth ATE plot

`astar_test`. In most of the cases, our algorithm is optimal and fast enough. However, for one of the cases in the narrow constriction grid test, where the goal and start node are separated by the wall with a narrow gap, the computing time is outstandingly long. We used obstacle distance grid to penalize cells that are close to the wall and we used `minDistanceToObstacle` to avoid still selecting cells at the gap as the pathway. By giving the restrictions above, the algorithm was able to correctly not find a path, whereas the computing time is still long. The possible reason is that it still had to traverse every cell on one side of the wall, seeking for a promising path around the wall, before it makees the final decision.

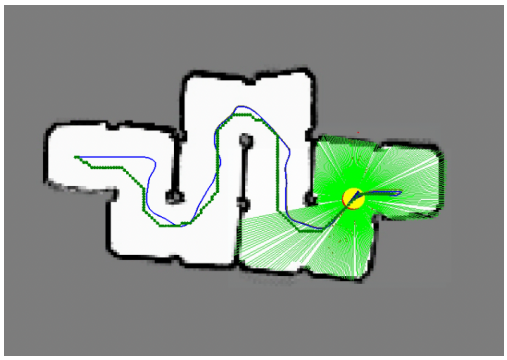

Fig. 9: Path Planned and Executed By Mbot

Fig. 9 shows an example of our MBot executing a planned path. The planned path is connected by the green dots. In this example image, the robot just finished exploring the map, so the origin was where the robot stopped at when it finished exploration, and the goal was set as the home position.

TABLE VI: A* Test Timing Statistics (μs)

Test	empty grid		filled grid		narrow constriction grid		wide constriction grid		convex grid		maze grid	
Outcome	**Success**	**Fail**	**Success**	**Fail**	**Success**	**Fail**	**Success**	**Fail**	**Success**	**Fail**	**Success**	**Fail**
Min	5899	67	N/A	26	6255	33	5844	246	71	34	4032	N/A
Mean	33274	68	N/A	2844	9582.5	1.43361e+08	14618.3	246	130	122	18941.2	N/A
Max	87716	69	N/A	14107	12910	4.30084e+08	25736	246	189	210	57109	N/A
Median	87716	0	N/A	28	0	50	25736	0	0	0	57109	N/A
Std dev	38496.5	1	N/A	5631.5	3327.5	2.02743e+08	8288.2	0	59	88	22097.5	N/A

D. Gripper Design

We found our gripper design capable of lifting and stacking a single crate. Due to loose tolerances in laser cutting the component keeping the columns of the 4-bar vertical, ensuring sufficient repeatable heights required a last minute duct taping alteration as seen by the gray duct tape wrapped around the white acrylic in Fig. 10.

E. Competition Tasks

For event 1: Speed Run, our team's robot successfully mapped the convex arena in the initial run, taking over 38 seconds. Leveraging this map, the subsequent speed run was completed in 28 seconds, faster than the mapping phase. The robot maintained a pose error within 5 cm, demonstrating both speed and precision in navigating the course.

For event 2, we were able to repeatedly stack 1 crate on top of another crate using relative odometry given the assumptions: the lifted block was originally placed forklift-side facing the robot and in the field of view of the robot's front camera and the second crate was placed approximately +20cm from the first crate in robot's forward direction of the initialized odometry frame.

For event 3, we successfully completed the task using exploration and autonomously returned to the home position in the maze.

IV. Discussion

A. Odometry Accuracy

As discussed in section III-A, we found that relying solely on wheel odometry for odometry frame dead reckoning pose estimates resulted in significant drift. Early attempts for using gyrodometry resulted in worse performance due to unfiltered gyroscope noise. Future investigation into incorporating the IMU's gyroscope data could result in better overall odometry performance which would be valuable for tasks when localization is only dependent on odometry such as in Competition 1 or when the robot needs to traverse long paths resulting in large accumulated drift such as in Competition 3. Additionally, tasks that involved higher amounts of rotation were found to have higher drift than transnational-heavy tasks, suggesting further highlighting the potential benefits of implementing effective gyrodometry.

B. Gripper Design

The chosen gripper design was able to effectively lift and stack crates. To go from a lowered to raised pose as see in Fig. 3, our design requires a single and swift rotation of the bottom link of the 4-bar linkage which can be much quicker than a conventional rack-and-pinion mechanism. However, due to the torque applied by the mechanism to the robot when trying to lift a crate at max speed, we needed to lower the speed of lift in order to prevent catastrophic tipping over. Solutions that we did not explore but may counteract this issue include adding counter weight and decreasing the length of the 4-bar linkage arms thereby decreasing the moment arm.

V. Conclusion

Our paper explored the development of a 2-wheeled robotic ground vehicle. Manual and automated calibration of the robot's geometry ensured precise odometry sufficient for driving slow long routs and fast short routes. Modeling the robot's action and sensor models through Monte Carlo Localization served as an effective 2D SLAM solution for exploring an unknown maze. Addition of a 4-bar linkage forklift mechanism enabled repeatable crate lifting and stacking. Future improvements include incorporation of gyrodometry and generalized autonomous crate stacking functionality.

In conclusion, the project serves as a proof of concept for the technologies in necessary for autonomous forklifts. The robot's versatility makes it valuable for broader automation applications, paving the way for ongoing research and development in the field.

References

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

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

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," vol. 2, 02 1999, pp. 1322 – 1328 vol.2.

[5] S.-G. Cui, H. Wang, and L. Yang, "A simulation study of a-star algorithm for robot path planning," pp. 506–509, 01 2012.

VI. Appendix

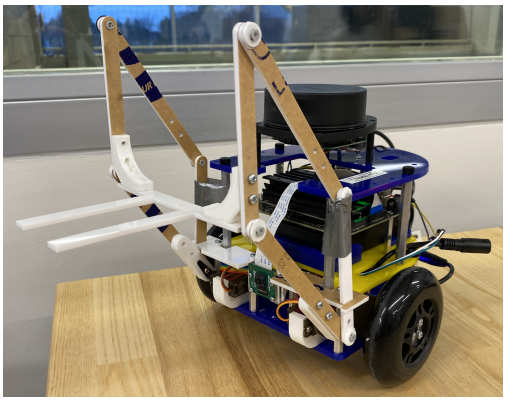

Fig. 10: Group 10 Mbot with Forklift Mechanism

Algorithm 1 Monte Carlo Localization

procedure InitializeParticles(N)
  for $i \leftarrow 1$ to N **do**
    Sample a particle x_i from the initial belief $bel(x_0)$
    Set the weight w_i to $1/N$
procedure MotionUpdate($u, \{x_i, w_i\}_{i=1}^N$)
  for $i \leftarrow 1$ to N **do**
    Sample a new pose x_i' by applying motion model $p(x_i'|u, x_i)$
    Update weight w_i using sensor model $p(z|x_i')$
  Normalize weights: $w_i \propto w_i / \sum_{i=1}^N w_i$
procedure Resampling($\{x_i, w_i\}_{i=1}^N$)
  Initialize an empty set X_{new}
  for $i \leftarrow 1$ to N **do**
    Sample k particles with replacement from $\{x_i\}_{i=1}^N$ based on weights $\{w_i\}_{i=1}^N$
    Add the sampled particles to X_{new}
  Set $\{x_i, w_i\}_{i=1}^N$ to the particles and weights in X_{new}

Algorithm 2 Likelihood Field Range Finder Model

Input: z_t x_t, m
Output q

$q = 1$
for all k **do**
  if $z_k \neq z_{max}$ **then**
    $x_{z_k^t} = x + x_{k,sens}\cos(\theta) - y_{k,sens}\sin(\theta) + z_k^t \cos(\theta + \theta_{k,sens})$
    $y_{z_k^t} = y + y_{k,sens}\cos(\theta) + x_{k,sens}\sin(\theta) + z_k^t \sin(\theta + \theta_{k,sens})$
    $dist = \min_{(x',y')}\{\sqrt{(x_{z_k^t} - x')^2 + (y_{z_k^t} - y')^2} \mid \langle x', y' \rangle \text{ occupied in } m\}$
    $q = q \cdot \left(z_{hit} \cdot \text{prob}(dist, \sigma_{hit}) + \frac{z_{random}}{z_{max}}\right)$
return q

Algorithm 3 Low Variance Resampling

Input: X_t W_t
Output $\bar{X}_t$

$\bar{X}_t = \emptyset$
r = rand$(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
for $m = 1$ to M **do**
  $U = r + (m-1) \cdot M^{-1}$
  while $U > c$ **do**
    $i = i + 1$
    $c = c + w_t^{[i]}$
  add $x_t^{[1]} to \bar{X}_t$
return $\bar{X}_t$

Algorithm 4 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 the *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*
return $\emptyset$

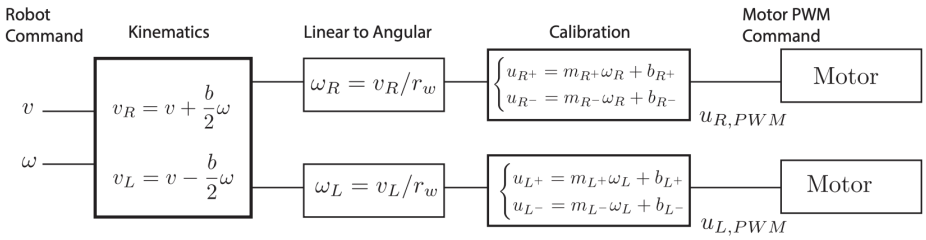

Fig. 11: Open Loop Wheel Speed Controller Design

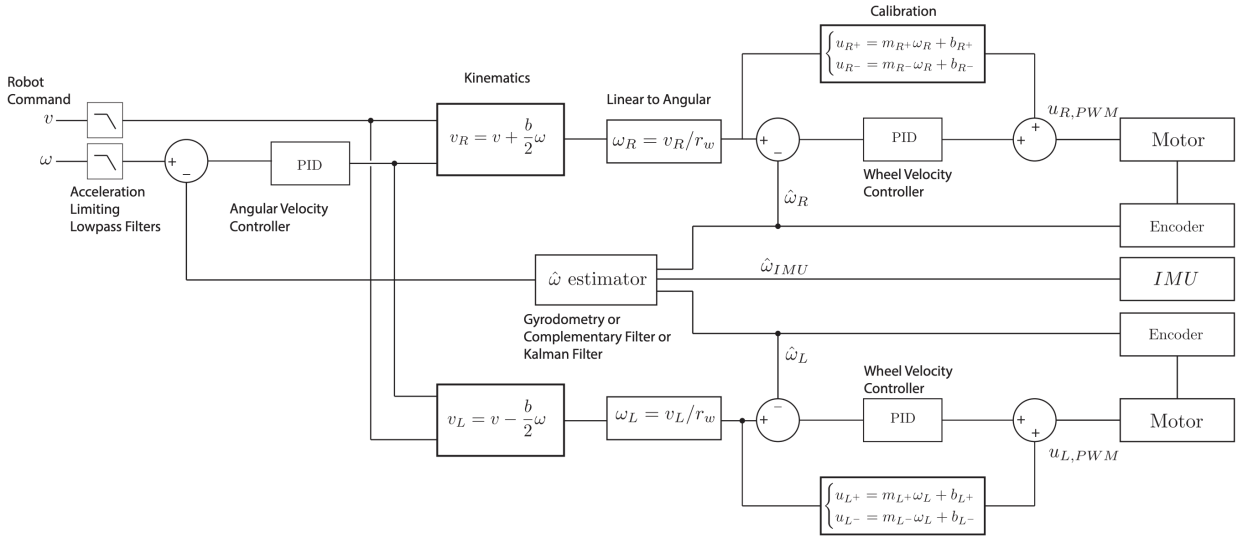

Fig. 12: Closed Loop Wheel Speed Controller Design