Botlab: Motion and Odometry

Andrew Greer
University of Michigan
alogreer@umich.edu

Dhillon Patel
University of Michigan
dhipatel@umich.edu

John Pye
University of Michigan
jepye@umich.edu

***Abstract*—Before advanced sensing and path planning algorithms can be developed or implemented, a reliable low-level controller is needed. Towards this end, we developed an open-loop controller and a series of closed-loop PID controllers to determine which approach was best to control an MBot. Lack of feedback control rendered open-loop innaccurate, and so an advanced PID controller with frame velocity control, integral saturation, and feed-forward terms was chosen instead. To guide the robot, pose estimation approaches using odometry and a sensor fusion of gyroscopic and odometry data were tested for accuracy and reliability, and a method of dead reckoning using odometry data was additionally implemented. Due to the unreliability of the MBot Inertial Measurement Unit (IMU), with the component shutting off at random, standard odometry was chosen.**

I. Introduction

Part 1 of Botlab deals with control: the sensing needed (dead reconing and pose estimation), and the algorithms to command the motors. To build a sophisticated robot controller, we broke the process down into development phases that start from a simple open loop controller with no feedback control, progressing steadily to a feedback controller which utilizes encoder data to control the wheel speed and converge to a set objective velocity. Our goal is to build the robot controller that can enable precise control for the foundation of further Botlab development.

II. Wheel Speed Characterization and Calibration

The calibration procedures for both MBot #26 and MBot #105 were completed on the lab's rubber floor mat and yielded similar calibration functions. These calibration functions are characterized with y being our PWM% and x being wheel speed. The slope of the calibration line thus converts from desired velocity to the necessary PWM signal to obtain that velocity.

In order to increase accuracy, we calibrated the robot motors both moving clockwise, and counterclockwise (forward and reverse). Thus, four separate calibration function were generated: two for the right motor, and two for the left. The appropriate function is selected by the control script, depending on the desired direction of motion. The left motor forward and reverse, and right motor forward and reverse calibration functions for MBot #26, respectively, were:

$$y = 1.6411 + 0.1214 \quad (1)$$

$$y = 1.7047 - 0.0991 \quad (2)$$

$$y = 1.5508 + 0.09 \quad (3)$$

$$y = 1.55 - 0.0942 \quad (4)$$

These calibration functions differed slightly from those of MBot #105, which, in the same respective order of left forward, left reverse, right forward, and right reverse were:

$$y = 1.449x + 0.0922 \quad (5)$$

$$y = 1.5149x - 0.0924 \quad (6)$$

$$y = 1.5339 + 0.1102 \quad (7)$$

$$y = 1.5583 - 0.0975 \quad (8)$$

The variation in the left and right wheels for MBot #26 was relatively high, as shown by comparing the slopes between the wheels. The left wheel had slopes of 1.6411 and 1.7047 for the forward and reverse cases; however, the right wheel had slopes of 1.5508 and 1.55 for both cases with the largest difference between the left and right wheels coming from the reverse case and was 0.1547, showing an apparent difference in the PWM% and wheel speed. This variation was not as drastic for MBot #105, where the largest difference in slopes between the left and right wheels came from the forward case, yielding 0.0849, showing nearly half the difference as that of MBot #26.

We believe the source of this variation to be the quality of the robot motor mounting itself. MBot #105 was reassembled, with wheels spaced far from the chassis, and motors supported to ensure horizontal operation. This meant both wheels had adequate space, and were at an approximately right-angle orientation with respect to the ground, eliminating wheel-to-wheel variance. MBot #26, meanwhile, was deliberately left in an as-is state to quantify variation between a robot in good condition, and one in poor condition. With insufficient clearance, the wheels would occasionally and sporadically rub against the chassis, and were at a variable angle with respect to the ground. This caused variation both between the wheels, as well as between the robots. The results, as seen in the calibration functions, are significant and should be accounted for in future development.

III. Open Loop Control and PID Closed Loop Control

With calibration functions, we were able to begin implementing two controllers on the MBot, one being an open loop controller with feed-forward command only, and the second

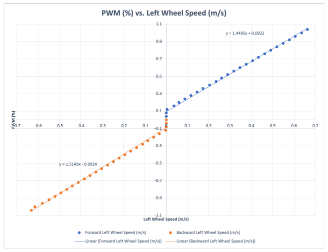

Fig. 1. MBot #105 Left Motor Calibration

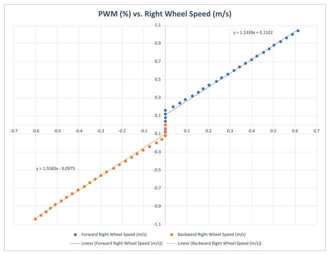

Fig. 2. MBot #105 Right Motor Calibration

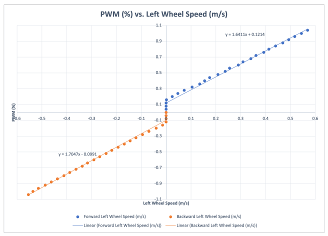

Fig. 3. MBot #26 Left Motor Calibration

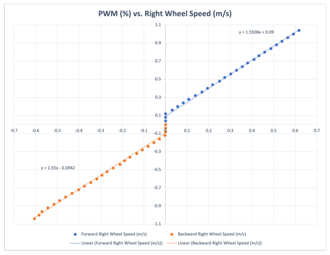

Fig. 4. MBot #26 Right Motor Calibration

being a simple wheel speed PID controller, with feedback from encoders.

A. *Open Loop Controller*

The open loop controller method receives no feedback on the current speed of the wheel, which means that it is an inaccurate method of controlling the wheel speed and is prone to either undershoot or overshoot. With this controller, the previously generated calibration functions are used to calculate the Pulse Width Modulation (PWM) signal per wheel for the robot to drive at some desired velocity.

As seen in Figure 6, this leads to a noticeable steady-state offset from our commanded set-point, in addition to random variation in control signal and thus velocity. This is not ideal from the point of view of control, for which we turn to a feedback controller able to correct error on-the-fly. In order to capture the full spectrum of the calibration effectiveness, the robot first drives forwards, then reverses.

B. *Simplified PID Controller*

To address open loop inaccuracies, PID (Proportional-Integral-Derivative) control incorporates a feedback term which allows for convergence over time. This feedback is provided by utilizing the encoder data to derive current wheel speed. Using error calculated from the difference between measured and commanded velocities, we tuned the PID for each wheel to quickly converge to a step velocity set-point.

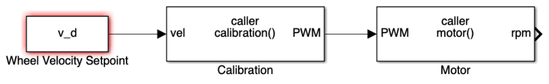

Fig. 5. Open Loop Controller Block Diagram

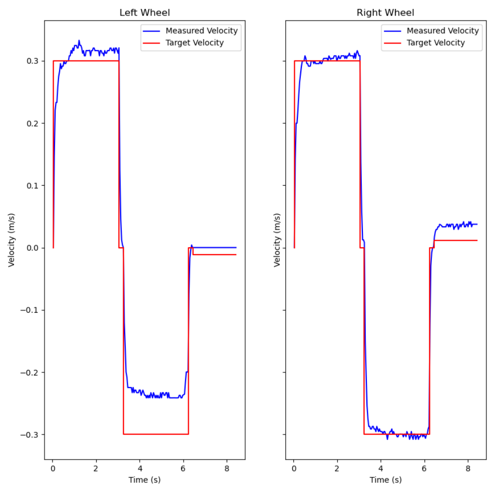

Fig. 6. MBot #2 Open Loop Control Step Response

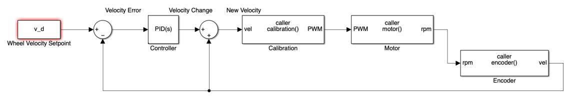

Fig. 7. PID Closed Loop Controller Block Diagram

The closed loop PID controller is, in fact, 3 separate controllers - Proportional, Integral, and Derivative, which seek to converge to a set-point. First, the error between the current state (provided by the wheel encoders) and the set-point is calculated. Based on the magnitude of this error, the Proportional controller exerts some effort to change the velocity in a way which reduces the error. Second, the Integral controller examines the integral of all accumulated error up until this point, and exerts an effort to change the velocity in such a way that the accumulated error decreases - this allows for convergence over time, as the integral term becomes more powerful as error accumulates. Finally, the Derivative controller examines the rate at which the error is changing - analogous to the speed itself of the robot - and attempts to drive this rate to 0. This results in smoother control, but risks undershooting the target as the Derivative controller prioritizes slowing down.

The force exerted by each component of the PID controller is dictated by gains, which we found through experimental trial and error. A higher K_p strengthens the P term, increasing overshoot and oscillation but also increasing convergence speed. A higher K_i strengthens the I term, increasing instability if too high, but otherwise allowing for eventual set-point convergence. A higher K_d strengthens the D term, increasing undershoot but potentially eliminating control oscillations in controller response. We tuned our controller to reduce overshoot, instability, and converge to the set-point over time, the results of which are listed below. We decided our robot was

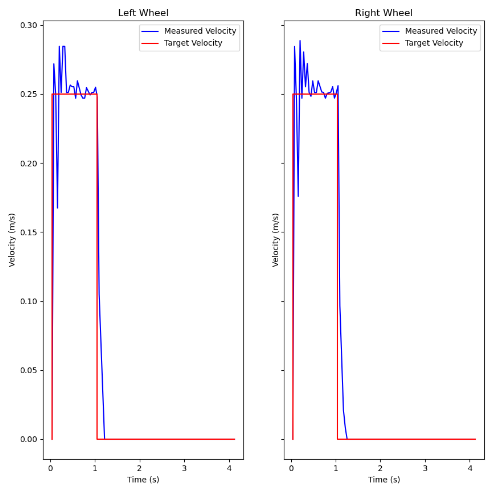

Fig. 8. MBot #105 PID Control Step Response

sufficiently tuned when we noticed steady-state offset had been eliminated, and the robot converged to the set-point quickly and without extreme oscillation.

Parameter	Left	Right
K_p	1.2	1.2
K_i	0.4	0.4
K_d	0.001	0.001
Filter (Hz)	20.0	20.0

C. Comparative Analysis Between Open-Loop and Closed-Loop PID Control

As shown in Figure 8, the tuned PID controller produces the desired result of fast convergence to our set-point velocity. Figure 6, meanwhile, demonstrates the comparative weakness of open-loop control, in which the controller was unable to converge to the set-point. Furthermore, the reliance of the open-loop controller upon calibration data alone precludes any possibility of terrain adaptation. While the MBot was calibrated on a rubber mat, the calibration curve would be different if driving on sand, or asphalt. The PID controller, on the other hand, would be able to still eventually converge to our set-point (albeit slower, due to the continued reliance on calibration to translate between velocity and control signal).

IV. ODOMETRY

Dead reckoning is vital to determining a robot's place in the world, without the use of external sensors, relying solely on itself. Dead reckoning involves measuring the rotation of the wheels of the robot over time (in our case, using an encoder) and, along with the physical characteristics of the wheel, determining the distance traveled. In order to enable dead reckoning, two methods can be used to calculate the robot's position and orientation.

odometry is a means to enable dead reckoning with wheel encoders only. The method calculates the position and orientation by distance traveled by the encoders and maps those changes to motion in both the x and theta directions. We validated our odometry model by measuring out 1m and manually moving the robot over the 1m and observing that the robot reported a displacement within 1% of 1 m. Likewise, we turned the robot $\pi/2$ radians, π radians, $3\pi/2$ radians, and finally 2π radians and observed each time a value within 10% of the expected radians (the increased margin to account for difficulties in manually rotating an object to a specific angle).

No specific correction was necessary to counteract steady-state errors in our odometry solution. However, in order to avoid angle runaway and ease analysis, we clamped recorded orientation to [-2π, 2π].

V. GYROSCOPE-ODOMETRY SENSOR FUSION

Gyroscope-odometry Sensor Fusion, or Gyrodometry for short, helps counteract the fundamental source of odometry inaccuracy: physical drift. By defaulting to the gyroscopic measurement during moments of large disagreement between the odometry and Gyroscopic methods, we both avoid the negative impact of drift (by defaulting to the Gyroscope in cases of disagreement), as well as noise (by using the odometry algorithms at all other times). Equations for Gyrodometry are listed below.

$$\phi = \frac{2\pi \cdot \Delta(encoder\ reading)}{(encoder\ resolution) \cdot (gear\ ratio)} \tag{9}$$

$$\Delta s_\omega = r_\omega \Delta\phi_\omega \tag{10}$$

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{wheel\ base} \tag{11}$$

$$\Delta d = \frac{\Delta s_R + \Delta s_L}{2} \tag{12}$$

$$\Delta x = \Delta d \cos(\theta + \frac{\Delta\theta}{2}) \tag{13}$$

$$\Delta y = \Delta d \sin(\theta + \frac{\Delta\theta}{2}) \tag{14}$$

$$p^{'} = \begin{bmatrix} x^{'} \\ y^{'} \\ \theta^{'} \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} \tag{15}$$

Algorithm 1 An algorithm with caption

```
0: $\Delta_{G-O} = \Delta\theta_{gyro} - \Delta\theta_{odo}$
0: if $|\Delta_{G-O}, i| > \Delta_{thres}$ then
0:     $\theta_i = \theta_{i-1} + \Delta\theta_{gyro,i} \cdot T$
0: else
0:     $\theta_i = \theta_{i-1} + \Delta\theta_{odo,i} \cdot T$
```

Parameter	Value
Wheel Radius	0.042
Wheel Base	0.15
Gear Ratio	78.0
Encoder Res	20.0
Threshold	0.005

We used a value of 0.005 radians as the threshold of difference between odometry and Gyrodometry, beyond which the controller would default to Gyroscopic heading. This parameter represents the difference between the instantaneous changes in heading recorded by odometry and the gyroscope. This parameter was defined through extensive trial and error testing and experimentation. We examined the difference between gyroscope-estimated heading and odometry-estimated heading during nominal operation (simple driving), as well as off-nominal operation (being picked up and spun around). We saw that, during nominal operation where odometry is appropriate, the difference between gyroscope estimation and odometry estimation was approximately 0.001 to 0.003 radians. Off-nominal operation, however, saw a difference on the order of 0.01 radians. The threshold of 0.005 radians was thus chosen to avoid premature activation of the gyroscope during nominal operation.

However, during normal operation, we were forced to disable Gyrodometry. This was due to unreliable IMUs on each team member's robot, within which lies the gyroscope sensor. The IMU would often not turn on with the rest of the robot, or would intermittently freeze. Thus, although accurate, the method was unreliable. Furthermore, the risk of Gyrodometry activation with an inactive IMU was great - due to the large difference between odometry (showing a moving robot) and the gyroscope (showing a stationary robot), Gyrodometry would by default trust the gyroscope, and thus believe the pose was fixed at the default value of 0 radians, as opposed to the actual heading.

VI. Robot Frame Velocity Controller

In order to smooth convergence and increase robot controllability, we next set out to augment our wheel speed controller with control of the robot's frame velocities - linear and angular - as well.

A. Simple Frame Velocity Controller

We first implemented a pure frame controller by additionally controlling based on Forward Velocity and Turning Velocity directly, as opposed to translating these velocities directly into wheel speeds and controlling those. The function of this controller is identical to that of the pure PID controller, with the exception of an additional control loop before the wheel speed control loop. This controller calculates the error between the measured robot angular and linear velocity derived from the encoder signals, and the target angular and linear velocity. The result of this controller is a command linear and angular velocity set-point, which is then fed into the wheel PID controllers. This controller is mainly effective in smooth convergence - in other words, avoiding overshoot and aggressive

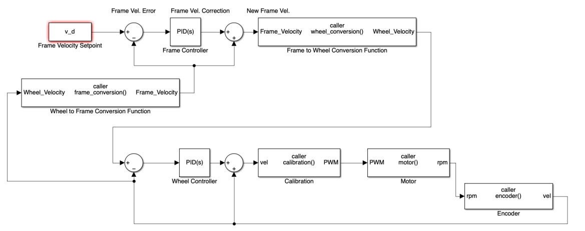

Fig. 9. PID Closed Loop Controller Block Diagram

oscillation by controlling not just the wheel speed, but the set-point provided to the wheels as well. However, this controller proved sensitive to low set-points, and overly damped - leaving significant overshoot which was not eliminated by Integral control.

In an effort to determine the best performance of the Simple Frame Velocity Controller, we tuned the controller as before - to minimize overshoot, oscillation, and error, and maximize convergence speed. Towards this end, we kept our wheel controller parameters the same as in the simple PID controller, and focused on tuning only the wheel velocity controller gains. The results of this tuning are listed below.

Parameter	Left	Right	Forward	Turn
K_p	1.8	1.8	0.9	0.3
K_i	0.4	0.4	0.05	0.0
K_d	0.001	0.001	0.0	0.0
Filter (Hz)	20.0	20.0	20.0	20.0

B. Advanced Frame Velocity Controller

To rectify the issues of the Simple Frame Velocity Controller, we augmented the control loop with feed-forward terms derived from our open loop controller, as well as integral term saturation. The feed-forward term sped convergence and helped to eliminate overshoot, as the robot essentially began with a control signal close to the correct control signal. The Integral saturation feature, meanwhile, eliminated the chief issue of Integral controllers in general: wind-up. Integral wind-up occurs when the controller maintains a steady-state error for too long. As the Integral term ordinarily has no maximum allowed control effort, it may destabilize the robot by over-correcting for past error. The saturation term prevents this, by implementing a minimum and maximum allowable control signal.

The performance of this controller at different step inputs is displayed below in Figures 11 through 15. Note that we graph the heading and linear displacement of the robot. This effectively captures the accumulated error of the Advanced Frame Velocity Controller. We see in each figure that the robot travels with linear speed (indicating quick set-point convergence and minimal overshoot or oscillation) and travels to the approximate state it should attain, given the velocity input (indicating minimal accumulated error). We deem the Advanced Frame Velocity Controller to be superior to all other

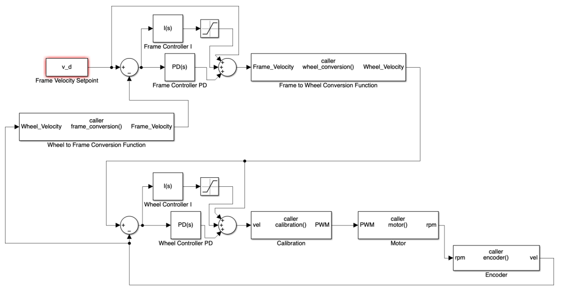

Fig. 10. PID Closed Loop Controller Block Diagram

tested, and will thus proceed with it as the primary controller used by the group.

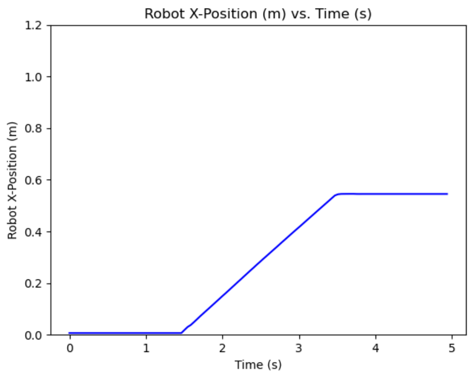

Fig. 11. 0.25 m/s for 2s Step Input

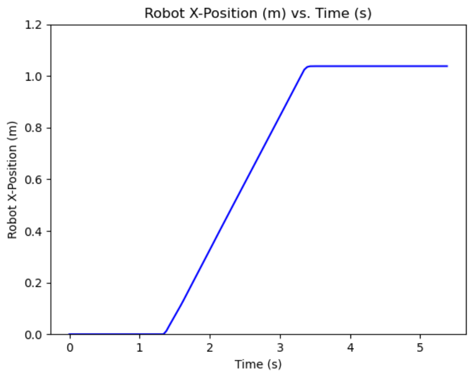

Fig. 12. 0.5 m/s for 2s Step Input

Based on these tests, we tuned the Advanced Frame Velocity Controller to minimize overshoot, error, and oscillation, while maximizing convergence speed. As before, we ceased tuning once we saw steady-state error eliminated in the long run, coupled with minimal oscillation and fast convergence. The results of this tuning are listed below.

Parameter	Left	Right	Forward	Turn
K_p	1.2	1.2	1.1	0.26
K_i	0.4	0.4	0.05	0.0
K_d	0.001	0.001	0.0	0.0
Filter (Hz)	20.0	20.0	20.0	20.0

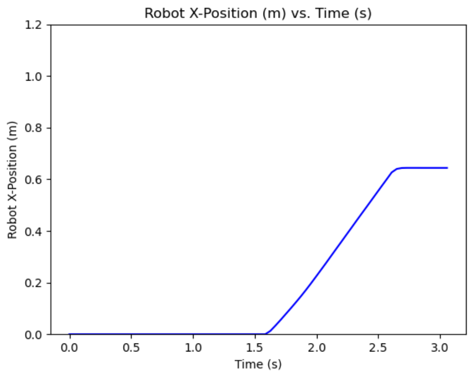

Fig. 13. 1 m/s for 1s Step Input

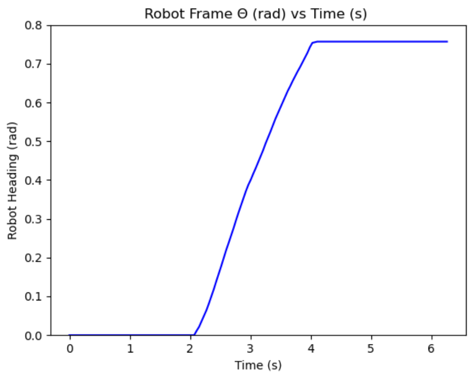

Fig. 14. $\pi/8$ rad/s for 2s Step Input

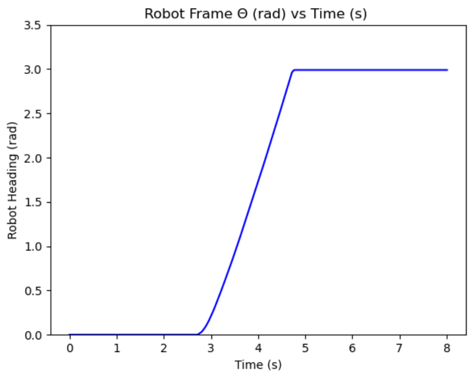

Fig. 15. $\pi/2$ rad/s for 2s Step Input

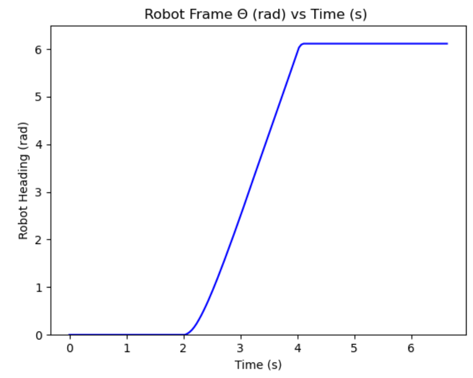

Fig. 16. π rad/s for 2s Step Input

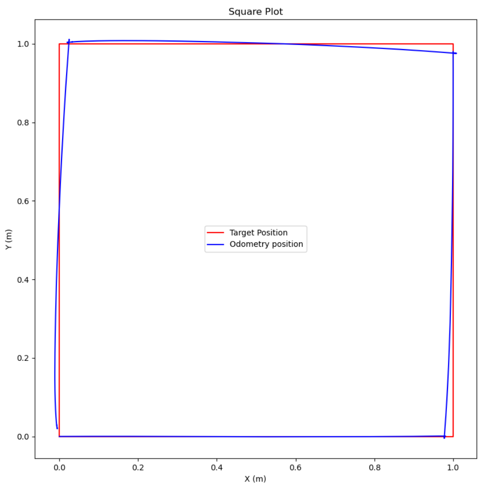

Fig. 17. MBot #105 Pose Estimation of 1m Square FIX

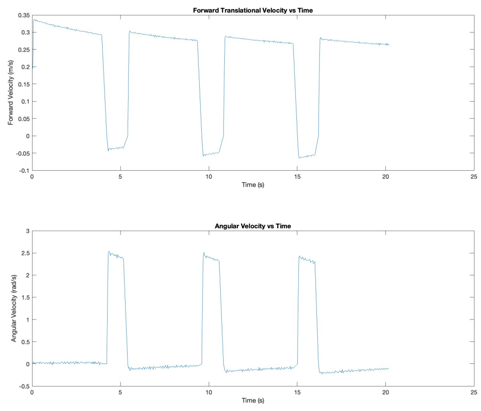

Fig. 18. MBot #105 Linear and Rotational Velocity FIX

VII. Dead Reckoning Square Driving

After implementing dead reckoning, we commanded the robot to drive a 1 m square one time. This allowed us to observe our robot executing a timed drive of a square, where we could compare the odometry position of the robot to the desired position, as shown in Figure 17. Along with this, we were able to make physical observations and comparisons to our initially untuned robot, which executed two timed squares: one based on its encoder readings and another based on approximately how many seconds it took to drive a meter and turn by $\pi/2$.

We observe that our robot with dead reckoning implemented performed significantly better than our robot without tuning, as it was better able to maintain a constant velocity while turning and driving. This reduced the drift over time, meaning that the actual path of the robot more closely resembled a square. We also saw that our robot performed very well on turns, barely producing near-perfect turns at $\pi/2$. We saw that our untuned robot veered heavily to the left and was not able to correct itself on straights or turns, so it was not consistent in its ending pose; whereas, we saw that the tuned robot was consistent in executing the square and having the same end pose (same as the start) almost every time.

However, Figure 18 still demonstrates the need for more controller tuning. As seen by the velocity changing on both the straight driving sections and the turning sections, we can surmise that our controller is overshooting its desired velocity.

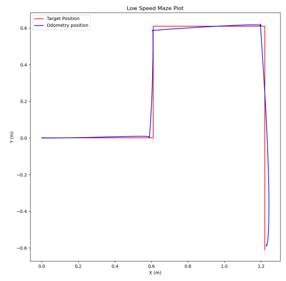

Fig. 19. MBot #105 Maze Drive at Slow Speed

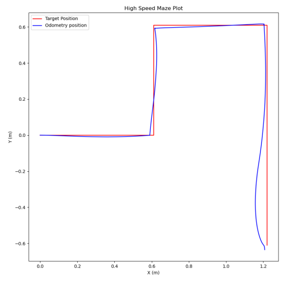

Fig. 20. MBot #105 Maze Drive at High Speed

VIII. Motion Controller Maze Driving

We recorded the MBot driving through the maze at a slow speed with translational velocity of 0.2m/s & an angular velocity of $\pi/4$ rad/s (Link to Slow Speed Maze Drive) and at a high speed with translational velocity of 0.8m/s & angular velocity of π rad/s (Link to High Speed Maze Drive). We turned the PID parameters in the motion_controller.cpp file, by iteratively running the robot through the maze. However, we saw that attempting to change the velocity limits saw a decrease in performance. As seen in Figures 19 and 20, the robot performed better at the slower speed setting. We observed that an increase from 0 for our K_i and K_d terms resulted in a large decrease in performance, thus our tuned PID parameters are as follows:

Parameter	Forward Velocity	Turn Velocity
K_p	1.2	3.0
K_i	0.0	0.0
K_d	0.0	0.0

IX. Conclusion

Through the design and testing of our controllers, the MBot was better at accomplishing the tasks using a fine-tuned PID controller with safeguards against Integrator wind-up, as well as feed-forward terms. The ability to respond to feedback was crucial to overall control of the robot in navigating the different scenarios, both when driving a 1m square once or driving a through a maze. During our testing, we encountered issues with IMU breakdown and as a result found relative success with the tasks using just odometry. Ultimately, odometry for position and a fine-tuned PID for motion resulted in an MBot that could readily move about accurately through its environment.

REFERENCES

[1] EECS 467 WN24 Slides, Google Drive Folder. Available online: https://drive.google.com/drive/folders/1k2LsNlQYenkD3fTVSE6PTB4Cy1tz1eGF?usp=drive_link (accessed on February 7, 2024).