ME 37500 Final Project: Robot Control

Prepared by:

Ethan Ross (ross315@purdue.edu)
50% – Designed initial FSM for line following capabilities, designed final controller for distance-based tracking, lap counter algorithm, wrote sections I, II-A, III-A, and IV.

Gilbert Chang (chang940@purdue.edu)
50% – Tuned wheel velocity PID controllers, developed continuous offset algorithm and curvature based deadband scheduling. Wrote sections II-B, II-C, II-D.1, II-E, and III-B.

School of Mechanical Engineering,
Purdue University
585 Purdue Mall
West Lafayette, IN 47907

Date: 2025.05.01

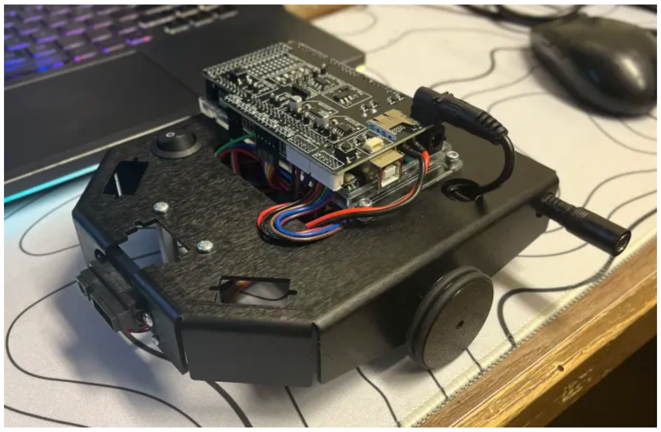

Fig. 1. AndyMark Skitter Robot

***Abstract*— This report details the system identification process performed on components of the AndyMark Skitter Robot, alongside the design and implementation of motor velocity controllers and finite state machines (FSMs) for line-following and distance-based tracking tasks.**

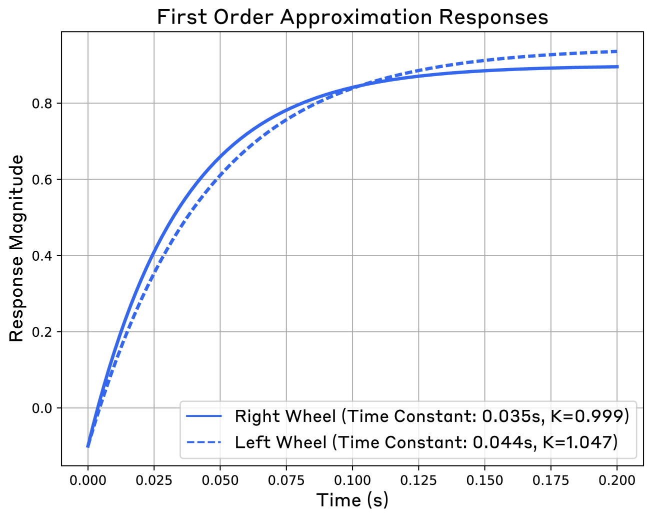

Fig. 2. Visualization of the predicted responses by the left wheel (solid line) and right wheel (dashed line) on the robot in response to a unit step input.

I. Introduction

The objective of the project is to program a robot to follow a line on a track autonomously and complete two full laps. After that, the robot will switch to parking mode and park eight inches from a wall.

Shown in Figure 1 is the AndyMark Skitter robot. The chassis of the two-wheeled robot is constructed from 0.09-inch hard-anodized aluminum sheet metal. It's powered by a 12V battery and controlled via an Arduino Mega 2560 with an expansion board that interfaces with the robot peripherals. [1] The robot has two 12V DC gearmotors with Hall effect encoders, each providing three counts per revolution at the motor shaft and 182 per revolution of the output shaft, enabling accurate feedback. Each motor has six pins, the first and second corresponding to the negative and positive driving voltages of the motor, the third and fourth dedicated to powering the Hall effect encoder, and the others being the sensor outputs. [1, 2] Shown in Table I are the technical specifications of the two sensors available on the Skitter robot.

Subsystem	Specifications
Gearmotors	12V DC, 340 RPM, 300 g-cm, 14:1
Line Sensor	3 reflectance sensors at 0.375” pitch
Distance Sensor	Operating range of 0-80 cm

Table I. Subsystem Specifications of the AndyMark Skitter Robot [1–4]

II. Subsystem Characterization and Full System Model

A. Sensor Functionality Verification

Before conducting system identification, the functionalities of each sensor was verified utilizing Simulink's Arduino compatibility add-ons. [5, 6] The block diagrams shown in Figure 3 were constructed with the assistance of the ME375 Robot Library built for Simulink.

First, the input voltage to the gearmotors was normalized, and an input that varied between -1 and 1 was sent to the gearmotors to verify their functionality. It was noted that the gearmotors did not move until the input magnitude exceeded 0.23 due to static friction. The value was noted for friction compensation in controller design.

Second, the shaft encoders were tested with a function block that converted the count difference between two samples of the encoder sensor readings into a velocity reading. The gearmotors were then ran at their maximum velocity, and it was verified that one rotation of the gearmotor output shaft corresponded to 182 pulses from the encoder.

Third, the line follower sensor was tested with the corresponding function block from the provided library. Shown in diagram three of Figure 3, there are three ports corresponding to each reflectance sensor in the array, and a fourth port that maps the reading of all three sensors to a continuous position offset from a black line. It was verified that each reflectance sensor sent a one when the color black was directly under it, and that it sent a zero in response to white. The binary threshold of the function block that determined when the signal would be determined high enough to be considered one was adjusted to be 300.

Finally, the distance sensor was tested using the Simulink implementation of the rightmost block diagram in Figure 3. The provided function block gives the raw output in centimeters, and a calibration function that converts the voltage reading into a distance reading in centimeters. A test object was held at immediate and far distances to determine that the output voltage saturated at the extreme distances. Refer to section II-C for the determination of the calibration function for the distance sensor.

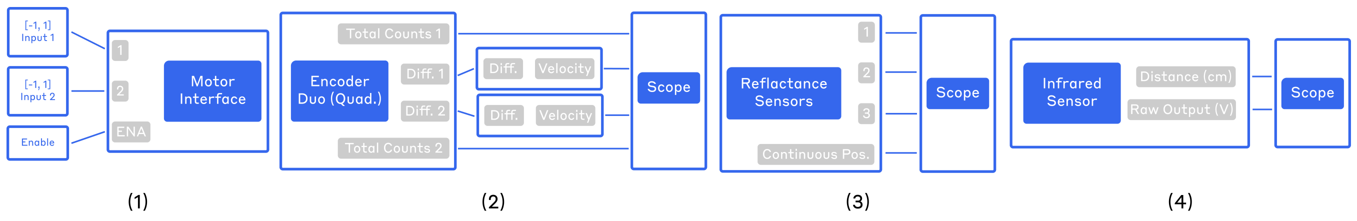

Fig. 3. (1) Motor testing block with two inputs ranging from -1 to 1, (2) Encoder testing block with two encoders' outputs being used to calculate total encoder counts and rotational velocity, (3) Line following sensor testing block, with 3 reflectance sensors being converted into binary outputs, their values being used for the calculation of total offset from the line, (4) Infrared sensor testing block, with the raw output in volts being used to calculate the distance in centimeters.

B. Drive Motors

To create a plant model for the motors, they were treated as first-order models for computational simplicity. Using MATLAB's data logging capabilities, the velocities of the motors in response to a unit step test were recorded. Nonlinear least-squares regression was performed with a Python script using the Matplotlib and NumPy libraries [7, 8] to fit a curve of the form 1 to the data.

$$y = K(1 - e^{\frac{-t}{\tau}}) \tag{1}$$

K is the static gain, t is the time, and τ is the time constant of the system. First order transfer functions were logged in the form:

$$P(s) = \frac{K}{\tau s + 1} \tag{2}$$

Shown in Figure 2 is the predicted velocity responses of each wheel based on the first-order approximation, as well as their corresponding static gain and time constant values. We also note that an input of magnitude greater than 0.23 is required to overcome static friction.

C. Distance Sensor

Shown in Figure 4, the distance sensor has extreme nonlinearities in behavior, with three distinct states: (1) quadratic growth, (2) saturation, (3) logarithmic fall-off. We can see that in the first, the voltage-distance relationship resembles the left side of a negative quadratic curve. Utilizing the Pandas and SciPy libraries [9, 10], the data was truncated at 5.1 centimeters, and a quadratic curve was fit to the data, the resulting formula being Equation 3.

$$v = -0.0778d^2 + 0.8786v + 0.6663 \tag{3}$$

For the next, it can be seen that the sensor seems to "saturate" at a value around 3 V, so a linear model: equation 4, is fit to the data between 5.1 and 7.0 centimeters.

$$v = 0.0061d + 3.0739 \tag{4}$$

Finally, for the majority of the data, from 7.0 to 80.0 centimeters, the inverse logarithmic curve: equation 5 is fit to the remaining data.

$$v = \frac{9.1021}{\ln(d) - 0.1902} - 1.8459 \tag{5}$$

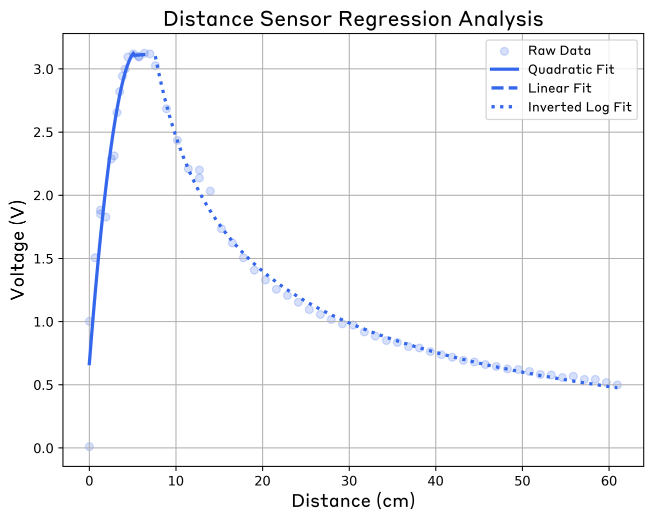

Fig. 4. Visualization of the raw data collected during the distance sensor calibration process, along with the truncated results of regression conducted on the three intervals of raw data.

sL	sC	sR	Interval	Midpoint
0	0	0	$[-0.414, -0.406] \cup [0.406, 0.414]$	± 0.410
0	0	1	$[-0.781, -0.414] \cup [-0.039, -0.031]$	-0.598
0	1	0	$[-0.406, -0.336] \cup [0.336, 0.406]$	± 0.371
0	1	1	$[-0.336, -0.039]$	-0.188
1	0	1	$[-0.031, 0.031]$	0.000
1	0	0	$[0.031, 0.039] \cup [0.414, 0.781]$	0.598
1	1	0	$[0.039, 0.336]$	0.188
1	1	1	$\varnothing$ (not attainable)	$\varnothing$

Table II. A mapping of sensor outputs to intervals of offset from the center of the line, with the midpoint of each interval being provided. All offset measurements are in inches. sL, sC, and sR are the binary inputs from the left, center, and right reflectance sensors on the array respectively.

D. Line Follower Sensor

1) Reflectance Sensor Array: The competition line consists of two 23.5/64" thick lines, separated by a 5/64" gap. Since the line follower sensor is a reflectance sensor array, a mathematical model mapping the collective outputs of the line follower sensor to intervals in offset from the center of the line is needed for robust controller design. Shown in Table II are 8 of the possible sensor array readings, along with the corresponding offset intervals from the center of the stripe and the midpoint of those intervals.

While the mapping shown in Table II is definite and nott susceptible to noise, due to the relative large size of each interval, the midpoints are coarse and prone to delay. We have a singular measurement: δ_{meas}, quantized to five values and drift-free.

In MATLAB, a function named ”estimateDelta” is created, which accepts three binary inputs: sL (left sensor), sC (center sensor), sR (right sensor). The function initializes a persistent lookup table (LUT) that maps each sensor combination to the midpoint reading value seen in Table II. The LUT intentionally maps impossible readings, such as all-black or all-white, to the last read offset, ensuring that the robot can rediscover the competition line.

2) Odometry-based Drift Estimation: We have access to four different sources of information: the two readings from each encoder on either motor (n_L^k, n_R^k), and the command signal sent to either motor ($v_L^{\text{cmd}}, v_R^{\text{cmd}}$).

Note that in the following equations, k denotes how many sample intervals have passed. In the case of the robot, one sample interval is 0.002 seconds, so k would equal to the time elapsed divided by that interval length.

Utilizing these sources, we can create a rudimentary odometry to estimate the drift, giving the LUT's estimation of robot offset more accuracy. Considering the width of the wheelbase in inches as B, we find the robot's angle relative to the line in terms of the two sets of variables:

$$d\theta_k = \frac{2\pi r}{BN}\left((n_R^k - n_R^{k-1}) - (n_L^k - n_L^{k-1})\right) \tag{6}$$

Because the command signal has a magnitude of $[0, 1]$ we can map the command signal to a linear velocity of $v_n^{\text{cmd}} \cdot v_{\text{max}}$ where v_{max} is the maximum linear velocity of the robot.

$$d\theta_{\text{PWM}} = \frac{v_{\text{max}}}{B}\left(v_R^{\text{cmd}} - v_L^{\text{cmd}}\right) \tag{7}$$

To derive the angle over time, we integrate over each timestep T_S.

$$\theta_k = \theta_{k-1} + T_s \cdot d\theta_k \tag{8}$$

We take a weighted average of the two values to fuse the two θ estimations together:

$$\theta_{\text{true}} = \theta_{\text{command signal}} + K_\theta(\theta_{\text{encoder}} - \theta_{\text{command signal}}) \tag{9}$$

In Equation 9, $K_\theta \in [0, 1]$ and denotes how much the measurement relies on the angle estimation based on encoder measurements.

Stepping back, we utilize θ_{encoder} and the explicit midpoint method in the Runge-Kutta family of integration methods to derive our estimated drift. dS_k denotes the linear distance traveled by the robot during sample interval k.

$$y_k = y_{k-1} + dS_k \sin\left(\theta_{\text{encoder},k-1} + \frac{1}{2}d\theta_k\right) \tag{10}$$

We employ our blended measurement, θ_{true}, to transform y_k and account for any possible error accumulated in the integration process with encoder measurements. x_k is derived in a similar fashion to Equation 10, but with $\cos$ utilized instead of $\sin$.

$$y_{\text{final},k} = x_k \sin(\theta_{\text{true}}) + y_k \cos(\theta_{\text{true}}) \tag{11}$$

Finally, we blend y_{final} with δ_{meas} to obtain an odometry-based estimation of the drift from the line using a weighted average in Equation 12. $K_{\text{sensitivity}} \in [0, 1]$ denotes how much the final measurement relies on the estimation from the reflectance sensor array.

$$\boxed{\delta_{\text{final}} = y_{\text{final}} + K_{\text{sensitivity}}(\delta_{\text{meas}} - y_{\text{final}})} \tag{12}$$

E. Full System Model

1) Wheel Rotational Velocity Transfer Function: Referring to section II-B, we recall our K and τ values from Figure 2 to construct transfer functions for the left and right wheels respectively.

$$P_{\text{left}}(s) = \frac{1.047}{0.044s + 1}, \quad P_{\text{right}}(s) = \frac{0.999}{0.035s + 1} \tag{13}$$

In Equation 13, the input for each transfer function is the duty cycle of the PWM input sent to the motors, and the output is the rotational velocity in encoder counts per second.

2) Input Nonlinearities and Actuator Saturation: In section II-A, the stiction value for each motor was identified to be 0.23, so the duty cycles of PWM inputs to each motor should have their magnitudes bounded to the tight interval of $[0.23, 1]$.

3) Robot Kinematics: We are able to predict the rotational velocity and linear velocity of the robot utilizing Equations 14 and 15. v_1 and v_2 are defined to be the linear velocity of the robot at its left and right wheels respectively. They may be calculated by multiplying their respective rotational velocities ω_n by the wheel diameter D.

$$v_A = \frac{v_1 + v_2}{2} \tag{14}$$

$$\dot{\theta} = \frac{v_2 - v_1}{2h} \tag{15}$$

With the rotational and linear velocities of the robot obtained, we utilize basic trigonometry to obtain the rate of change of the x and y coordinates of the center of the robot.

$$\frac{dx_A}{dt} = v_A \cos\theta \tag{16}$$

$$\frac{dy_A}{dt} = v_A \sin\theta \tag{17}$$

Utilizing Euler integration in code, we can obtain an estimation of $x_A(t)$, $y_A(t)$ and $\theta_A(t)$ during robot execution.

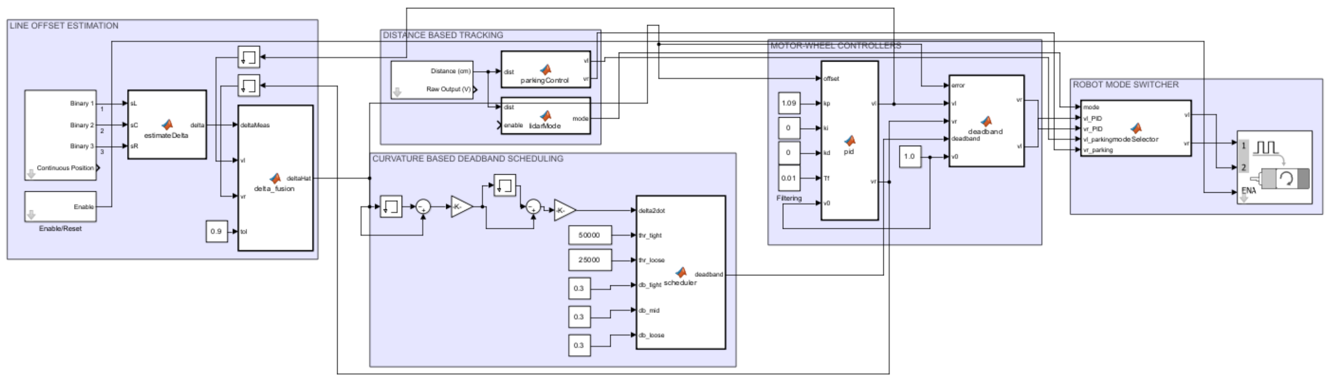

Fig. 5. Finalized controller architecture designed inside MATLAB Simulink. Note that the PID block on the right contains both line following and wheel velocity PID controllers.

III. Proposed Controller Structure and Design

A. Wheel Rotational Velocity Controllers

For controlling the rotational velocity of each wheel on the robot a PID controller architecture was considered. The general form of the controller is as follows:

$$G_{PID}(s) = \frac{K_D s^2 + K_P s + K_I}{s} \tag{18}$$

In Equations (18), K_D, K_P, and K_I represent the derivative, proportional, and integral gain values, respectively. Two methods were explored to determine the appropriate gains.

1) MATLAB Root Locus Design: In MATLAB, the Root Locus method can be used interactively to design a PID controller that meets the desired percent overshoot (PO) and 2% settling time ($T_{2\%}$). This is typically done using the `rltool` interface.

After entering the open-loop plant transfer function $P(s)$, the desired PO and $T_{2\%}$ were input. These are internally converted into desired dominant pole locations based on the standard second-order system characteristic equation:

$$s^2 + 2\zeta\omega_n s + \omega_n^2 = 0 \tag{19}$$

where ζ is the damping ratio and ω_n is the natural frequency of the system.

MATLAB then uses these pole locations to determine the required phase margin (PM) and gain margin (GM) that the closed-loop system must meet. These margins are used to place constraints on the open-loop frequency response, ensuring stability and performance.

A PID controller of the form

$$C(s) = K_p + \frac{K_i}{s} + K_d s \tag{20}$$

is created by MATLAB such that the root locus of $C(s)P(s)$ passes through the desired closed-loop poles. This provides a controller that satisfies the given specifications.

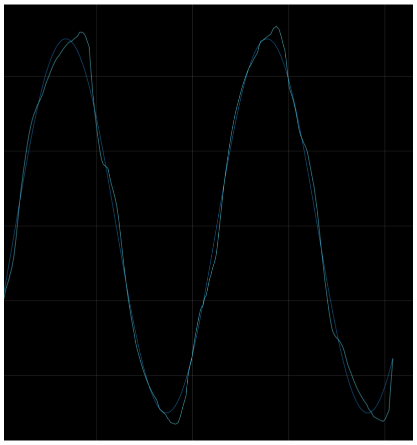

Fig. 6. An example of Ziegler-Nichols tuned PID controller performance. Shown in blue is a sinusoidal command signal, and in cyan is the motor's corresponding velocity.

2) Ziegler-Nichols Method: The Ziegler–Nichols method is an empirical approach to tuning controller gains based on system response. The procedure is to first set the integral and derivative gains to zero: $K_i = 0$, $K_d = 0$. Then gradually increase the proportional gain K_p until the system exhibits sustained oscillations. Finally, record the gain at this point as the *ultimate gain* K_u, and the oscillation period as the *ultimate period* T_u.

For a PI controller, the recommended gains are:

$$K_p = 0.45K_u, \quad K_i = \frac{1.2K_u}{T_u} \tag{21}$$

For a PID controller, the classic Ziegler–Nichols tuning rules are:

$$K_p = 0.6K_u, \quad K_i = \frac{1.2K_u}{T_u}, \quad K_d = 0.075K_uT_u \tag{22}$$

Figure 6 shows the performance of a PID controller tuned using this method in the context of our system. In our initial tuning attempt, we observed an ultimate gain of $K_u = 0.7$ and an ultimate period of $T_u = 0.07$. These values produced strong disturbance rejection and had a decent overshoot. We will likely continue to tune these gain values up until competition day.

System	Controller	Input/Output Structure
Motor-Wheels	PID	PWM Duty Cycle $\rightarrow$ Wheel Velocity
Robot (Line Tracking)	PID w/ db. sched.	Line Offset $\rightarrow$ PWM Duty Cycles
Robot (Distance Tracking)	PID w/ FSM	Distance Reading $\rightarrow$ PWM Duty Cycles

Table III. A comprehensive table attributing tasks or systems to controller architectures and input-output pairs.

B. PID Controller for Line Following Capabilities

In addition to utilizing a PID controller to ensure that the rotational velocity control signals are met swiftly and accurately by the system, the higher level goal of following the two-striped competition line still remains.

1) Raw PID Controller: We set the error used in the PID controller to the blended offset measurement from section II-D.1. Relying on the relatively high accuracy of the measurement, we construct the general form of a PID controller as described in III-A.

There are two important insights regarding the PID controller's implementation in the line-following use-case. The first is the prevention of integral wind-up. If the integral term skyrockets during performance, the robot could be left circling.

Therefore, the integral term is bounded in magnitude in comparison to some nominal value.

$$\min(\text{Integral Output},\ \text{Nominal Output}) \tag{23}$$

During performance, the offset measurement may fluctuate wildly, as oscillations are inherent to the PID controller's performance. Moreover, the discrete nature of the offset from using a LUT in computing the robot's offset can cause the value to jump around. Therefore, the derivative term $D(s)$ is filtered as shown in Equation 24:

$$D(s) = K_d \cdot \frac{N}{1 + \frac{N}{s}} \tag{24}$$

2) Deadband Scheduling: During the robot's performance, it may converge on the center of the line vibrate around it. In this case, neither of the motors are spinning at maximum velocity. To optimize for speed on the straight parts of the track, this problem significantly increases lap times.

A deadband is consequently introduced, a region on the line where PID control signals will be ignored, and the maximum PWM duty cycle possible is sent to the motors. The performance on the straight portions of the track is assured, however, on the curved portions of the track, the static deadband will result in a choppy motion, as the robot traverses the curved portion in linear segments.

To solve this, deadband scheduling is introduced, where the second derivative of the offset reading is taken, and tighter deadbands are assigned to the PID controller depending on the curvature of the path.

C. Finite State Machine for Distance Based Tracking

When the robot makes two complete laps around the track, it's required to stop exactly 8 inches from a barrier erected in its path on the track. To tackle this problem, two major components of a possible controller are needed.

1) Lap Counter: The first component is a lap counter, for which the triggering of a "distance following" state will be used for. The trigger mark present on the track is the only case where all three sensors in the reflectance array will read 1. Therefore, to identify when the robot has completed two laps, we will count how many times the line sensor reads all black. The first time will be when the robot starts its run and crosses the line, the second will be after the robot concludes its first lap, and the third will be after the robot finishes its second lap.

A naive algorithm counting the number of instances of all black will log instances where all three sensors read black for a brief moment as a line crossing. To combat this issue, we introduce a threshold term, where if the sensors read all black long enough, then the algorithm will count it as a valid crossing.

2) Distance Sensor Based Parking: Converting 8 inches from the barrier to centimeters, we get 20.32 centimeters. As shown in Figure 4, 20.32 centimeters is well into the logarithmic regime, so the quadratic and linear region can be ignored if the robot does not overshoot more than 13 centimeters. For the purposes of a FSM for parking, Equation 5 can be used.

In a FSM for distance based parking, there are 3 states: (1) far away, (2) close to setpoint, (3) too close. To activate the FSM, the lap counter must be equal to three.

The first state assumes that the distance sensor reading d is above 40 centimeters. In this state, the robot will continue following the line at maximum speed. If the reading is below 40 centimeters and above 10 centimeters, we move to the second state. If the reading is below 10 centimeters, we're too close to the barrier, and move to the third state.

In the second state, the robot begins using a PID controller to dynamically adjust the nominal velocity in the line following PID controller in response to error from the distance setpoint. In essence, the error in this new PID controller is the difference between d and 20.32 centimeters. The output of the PID controller is the desired velocity of the robot, meaning that if the robot slightly overshoots its distance setpoint, it's able to "back up" and return itself to equilibrium.

Finally, the third state is for if the robot is too close to the barrier. If the reading drops below 10 centimeters, the robot reverses itself at maximum velocity.

Upon successful execution of the PID loop in keeping the robot at exactly 20.32 centimeters of the barrier, the task is completed and the robot can exit its program.

D. Controller Justification

We are convinced that the controller structure will work because it combines a proven PID control approach with tuning and filtering. PID controllers are widely used in industry because they are simple and effective when it comes to correcting steady errors and smoothing out sudden changes. Also, our design makes sure that the wheel speed, line tracking, and parking are managed by its own control loop, with a state machine ensuring smooth transitions. Lastly, our controller structure avoids overshooting and jitter, making it reliable and optimized for the competition.

IV. Potential Issues with Controller Implementation

Several challenges may arise when implementing the proposed control scheme on the robot:

- **Sensor Noise and Inaccuracy:** The line-following and distance sensors might give inconsistent readings because of sharp changes in the lookup table. This could make the controller behave less predictably. *To help with this, we'll use simple filters like low-pass filters, and we'll carefully adjust the controller gains so it doesn't overreact to noisy signals.*
- **Motor Dead Zones and Saturation:** The motors won't move at low PWM signals due to stiction, which can make slow or precise movements difficult. *To fix this, we will use deadband compensation and make sure the PWM signals stay within the range where the motors actually respond.*
- **Integral Windup and Derivative Lag:** The integral part of the controller can build up too much over time, and the derivative part might react slowly if we filter it too much. *We will use anti-windup methods and pick a filter setting that balances fast response with low noise under control.*
- **Overfitting to Track Conditions:** Tuning based on specific track conditions in Potter may cause the robot to not perform well on competition day. *We will validate the system under varying lighting and surface conditions to improve robustness.*

By planning ahead for these issues and building in ways to handle them, we hope to keep the robot running smoothly and reliably during operation.

References

[1] AndyMark Inc. *Skitter Classroom Robot*. https://www.andymark.com/products/skitter-classroom-robot. Accessed: 2025-04-27. 2025.

[2] AndyMark Inc. *340 RPM 12V 14:1 Ratio Gearmotor with 2 Channel Encoder*. https://www.andymark.com/products/340-rpm-12v-14-1-ratio-gearmotor-with-2-channel-encoder. Accessed: 2025-04-27. 2025.

[3] Pololu Corporation. *QTR-3RC Reflectance Sensor Array*. https://www.pololu.com/product/2457. Accessed: 2025-04-27. 2025.

[4] Cytron Technologies. *Analog Distance Sensor (10-80cm)*. https://www.cytron.io/p-analog-distance-sensor-10-80cm. Accessed: [April 27th, 2025]. 2025.

[5] MathWorks. *MATLAB Support Package for Arduino Hardware*. https://www.mathworks.com/matlabcentral/fileexchange/47522-matlab-support-package-for-arduino-hardware. Accessed: 2025-04-27. 2024.

[6] MathWorks. *Simulink Support Package for Arduino Hardware*. https://www.mathworks.com/matlabcentral/fileexchange/40312-simulink-support-package-for-arduino-hardware. Accessed: 2025-04-27. 2024.

[7] The Matplotlib Development Team. *Matplotlib API Reference*. Accessed: 2025-04-27. 2025.

[8] NumPy Developers. *NumPy User Guide*. Accessed: 2025-04-27. 2024.

[9] The pandas development team. *pandas Documentation*. https://pandas.pydata.org/docs/. Accessed: 2025-04-27. 2025.

[10] The SciPy community. *SciPy Documentation*. https://docs.scipy.org/doc/scipy/. Accessed: 2025-04-27. 2025.