Autonomous Systems Lab

Path Control

Abstract

This document discusses the theoretical background and practical implementation of path control for a two-wheel mobile robot. We cover wheel odometry, sensor fusion (wheel odometry + IMU), path-tracking algorithms (pure pursuit, Stanley), trapezoidal velocity profiling, and PID control. Laboratory simulations and exercises are included to illustrate how these concepts translate to real-world scenarios.

1. Introduction

Path control is fundamental to autonomous mobile robotics, ensuring that a robot follows a predefined trajectory while accounting for noise and uncertainties in sensor measurements. This paper provides both the theoretical underpinnings (odometry, sensor fusion, motion profiles) and practical implementation details (PID, code structures, and lab simulations) for a small Zumo-like robot.

1.1 Roadmap

- **Section 2** covers wheel odometry basics, including the derivations for estimating position and orientation.
- **Section 3** introduces sensor fusion between IMU and odometry for smooth and more stable estimate
- **Section 4** introduces path-control algorithms, comparing pure pursuit, Stanley, and point-to-point (P2P) strategies.
- **Section 5** explores trapezoidal velocity profiles and how they can be combined with path control for smooth, jerk-free motion.
- **Section 6** discusses PID control and how it is integrated into motor speed regulation and heading control.
- **Section 7** outlines lab simulation instructions and recommended parameter-tuning steps.
- **Section 8** presents in-lab tasks for testing these ideas on a physical platform.

2. Wheel Odometry and Kinematic Model

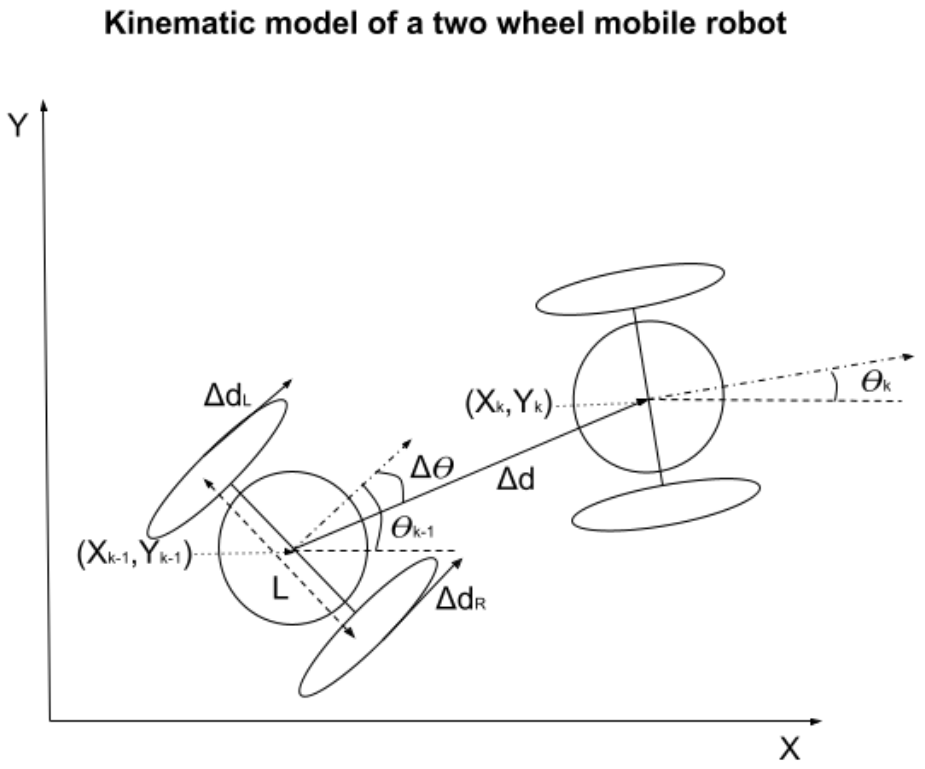
2.1 Basic Wheel Odometry

Wheel odometry estimates robot motion using rotation measurements from wheel encoders.

- **Key Components**
  - **Wheel Encoders:** Measure wheel rotation in discrete ticks per revolution.
  - **Robot Geometry:** Wheel diameter and the baseline (distance between the two wheels) are critical for calculating forward motion and turns.
- **Advantages**
  - Straightforward, purely on-board sensing (no external infrastructure).
  - Relatively easy to implement with off-the-shelf encoder hardware.
- **Limitations**
  - Susceptible to slipping and skidding (assumes stable contact with the ground).
  - Errors accumulate over time without external correction.

2.2 Kinematic Model

A common two-wheel mobile robot model (with differential drive) can be derived from left- and right-wheel travel distances:



- $\Delta d = \frac{\Delta d_R + \Delta d_L}{2}$
- $\Delta\Theta = \frac{\Delta d_R - \Delta d_L}{L}$
- $\Delta d_{(L,R)} = \boldsymbol{\pi} D_{(L,R)} \frac{EncoderPulses}{EncoderPulsesPerRevolution}$
- $X_k = X_{k-1} + \Delta d \cdot cos(\Theta_{k-1} + \Delta\Theta/2)$
- $Y_k = Y_{k-1} + \Delta d \cdot sin(\Theta_{k-1} + \Delta\Theta/2)$
- $\Theta_k = \Theta_{k-1} + \Delta\Theta$

where:

- Δd_R and Δd_L are distances traveled by the left and right wheels.
- L is the wheelbase (distance between left and right wheels).
- X_k, Y_k and Θ_k are the new position and orientation at timestep kkk.

3. Sensor Fusion: Wheel Odometry + IMU

3.1 IMU Components

An Inertial Measurement Unit (IMU) typically includes:

- **Accelerometer** (measures linear acceleration),
- **Gyroscope** (measures angular velocity),
- **Magnetometer** (measures orientation relative to Earth's magnetic field).

3.2 Complementary Filter for Fusion

Because gyros drift over time and accelerometers introduce noise, combining IMU and wheel-encoder data yields a more stable estimate. A complementary filter is a simple yet effective approach:

Gyro Orientation Change:

$$\Delta\theta_{gyro} = \omega \cdot dt$$

Wheel Odometry Orientation Change:

$$\Delta\theta_{odo} = \frac{\Delta d_R - \Delta d_L}{L}$$

Complementary filter:

$$\Theta_k = \alpha \cdot (\Theta_{k-1} + \Delta\theta_{gyro}) + (1 - \alpha) \cdot (\Theta_{k-1} + \Delta\theta_{odo})$$

Where:

- d: is the distance travelled
- L: is the distance between wheels
- ω: is angular velocity from the gyro,
- α: is a tuning parameter (0 to 1).

Note: More sophisticated methods (e.g., Extended Kalman Filter) can also be employed, especially if the environment is prone to noise, drift, or large disturbances.

4. Path Control

Accurate path control ensures the robot follows a desired path, often represented by discrete waypoints or a continuous curve. Two widely known approaches are **Pure Pursuit** and **Stanley**.

4.1 Pure Pursuit

1. **Core Idea**: The robot looks ahead to a “target point” on the path and steers to intercept it.
2. **Key Parameter**: **Lookahead Distance**, which determines how far ahead the robot aims along the path.
3. **Advantages**:
    - Simple to implement, easy to tune.
    - Effective on relatively smooth paths.
4. **Disadvantages**:
    - Might overshoot tighter curves.
    - Performance depends heavily on the lookahead parameter.

4.2 Stanley Algorithm

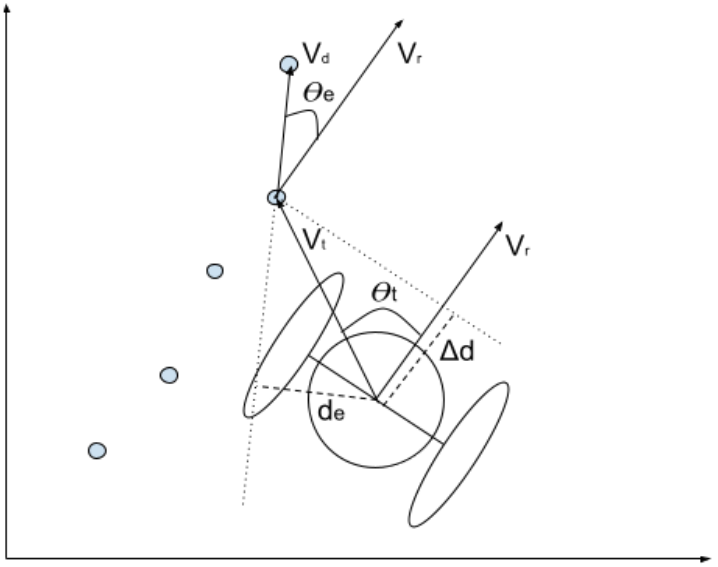
1. **Core Idea**: Uses cross-track error and heading error relative to the trajectory to compute a steering command.
2. **Advantages**:
    - Often better at handling sharper turns and “stickier” path tracking.
3. **Disadvantages**:
    - More complex, slightly higher computation.
    - Requires careful tuning of proportional gains for cross-track error and heading alignment.

4.3 Point-to-Point (P2P) Control

For simpler tasks, you can steer from your current position to a single desired point by calculating:

1. The bearing difference Θ_t between the robot’s heading and the vector from the robot to the target.
2. A forward command proportional to distance to the target.
3. A turning command proportional to Θ_t

Path control of a two wheel mobile robot



Cross product:

- $\overline{a} \times \overline{b} = ||a|| \cdot ||b|| \cdot sin(\Theta)\, \widehat{n}$

Dot product

- $\overline{a} \cdot \overline{b} = ||a|| \cdot ||b|| \cdot cos(\Theta)$

Robot direction vector

- $V_r = [cos(\Theta)\ ,\ sin(\Theta)]$

Point to Point (P2P) Control can be express by

- $\Theta_t = arccos(\frac{V_t \cdot V_r}{||V_t|| \cdot ||V_r||}) \cdot sign(V_r \times V_t)$

Trajectory Control can be expressed by

- $\Theta_e = arccos(\frac{V_d \cdot V_r}{||V_d|| \cdot ||V_r||}) \cdot sign(V_r \times V_d)$
- $d_e = \frac{V_t \times V_d}{||V_d||}$
- $\Delta d = \frac{V_t \cdot V_r}{||V_r||}$

5. Trapezoidal Velocity Profiles

5.1 Rationale

A trapezoidal velocity profile transitions smoothly through acceleration, constant velocity, and deceleration phases. This improves motion accuracy and reduces wear on motors/gears.

Key Phases

1. **Acceleration**: Increase velocity from 0 to V_{max} at a_{max}
2. **Constant Velocity**: Maintain V_{max}
3. **Deceleration**: Reduce velocity back to 0 by $-a_{max}$

5.2 Governing Equations

Using basic kinematics:

- $V_f^2 = V_0^2 + 2a\Delta s$

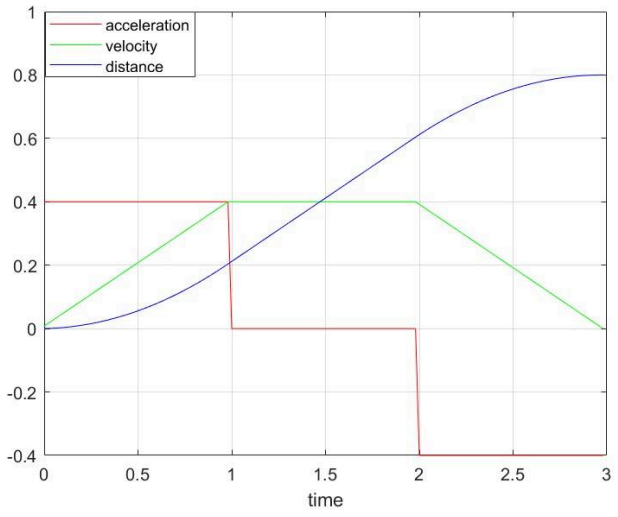
and for distance-based scheduling:

- $V_k = \sqrt{2a \cdot s}$

subject to constraints:

- $V_k \leq V_{max}$, $\frac{\Delta V_k}{\Delta t} \leq a_{max}$

An example of a desired motion profile, with max acceleration and speed limit.



6. PID Control

A **Proportional-Integral-Derivative (PID)** controller reduces error between a setpoint (desired speed or heading) and the current measured value.

$$u(t) = K_p \cdot e(t) + K_i \cdot \int e(t)dt + K_d \cdot \frac{de(t)}{dt}$$

where:

- $e(t)$ is the error,
- K_p, K_i, K_d are gains for proportional, integral, and derivative terms.

6.1 Example: PID with FIR Derivative Filter

```
float kp = 1.0;
float ki = 0.1;
float kd = 0.2;
float Ts = 0.01;
float integral = 0.0;
float prev_error = 0.0;
float prev_derivative = 0.0;
float derivative = 0.0;
float alpha = 0.1;
float filtered_derivative = 0.0;

float pid_control(float setpoint, float process_variable) {

    float error = setpoint - process_variable;


    integral += error * Ts;


    derivative = (error - prev_error) / Ts;
    filtered_derivative = alpha * derivative + (1 - alpha) * prev_derivative;


    float control_signal = kp * error + ki * integral + kd * filtered_derivative;


    prev_error = error;
    prev_derivative = filtered_derivative;

    return control_signal;
}
```

Tip: In real applications, consider clamping the integral term to avoid “windup” and manage noise in derivative calculations.

7. Pre-Lab Activity

Overview & Objectives

Before proceeding with the hardware-based lab session, you will explore four MATLAB simulations that model a two-wheel mobile robot's motion and control. The goals of this pre-lab are:

1. **Familiarize yourself with the simulation environment** (MATLAB/Simulink blocks, scripts, and scopes).
2. **Observe how changes to key control parameters** (e.g., PID gains, trapezoidal velocity limits, path-control gains) affect robot behavior.
3. **Compare different path control strategies** (point-to-point vs. path-following) to understand their strengths and limitations.
4. **Gain insight into how simulation responses differ from real-world performance**, laying the groundwork for your in-lab experiments.

1. Simulation Roadmap

The simulations listed below are located in the project GitHub folder. Ensure the file `InitZumo.m` is present in the same folder so all parameters load correctly.

Simulation	Purpose	Key Observations/Outcomes
Simplified_OpenLoop	Demonstrates robot motion in an open-loop manner (no PID).	Notice the limitations of open-loop control (no correction).
ZumoSimulatiom_P2P	Implements Point-to-Point (P2P) with PID motor control.	See how PID influences reaching a single target (overshoot, settling time).
ZumoSimulatiom_P2P_Multi Point	Extends P2P to multiple sequential waypoints.	Observe transitions between waypoints and overall path error.
ZumoSimulatiom_Path_Control	Full path-following (continuous path) with multiple points.	Compare performance to P2P. Tune path-control gains.

2. Key Parameters to Explore

In each simulation, you will see various parameters that influence the robot's behavior:

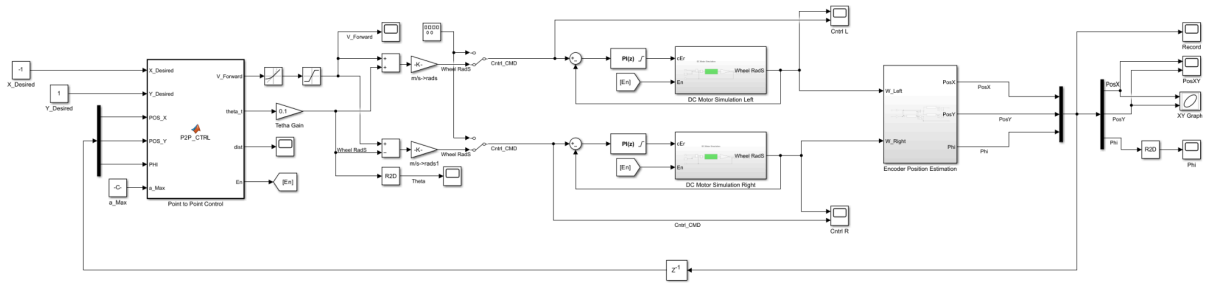
- **Motor PID Gains**: K_p, K_i, for the motors' speed control.
- **Trapezoidal Profile Settings**: Acceleration a_{max}, maximum velocity V_{max}.
- **Path Control Gains**: For heading (Θ gain) and cross-track error

Note: *Changing these parameters can increase or decrease overshoot, settling time, and steady-state error.*

3. Step-by-Step Instructions

1. **Setup**
    - Open MATLAB/Simulink.
    - Ensure `InitZumo.m` is in your current working folder.
2. **Simplified_OpenLoop**
    - Open `Simplified_OpenLoop.slx` and run the simulation.
    - Observe the robot's trajectory (x-y position over time) and note how the absence of feedback control affects accuracy.
3. **ZumoSimulation_P2P**

ZumoPi Simulation - P2P

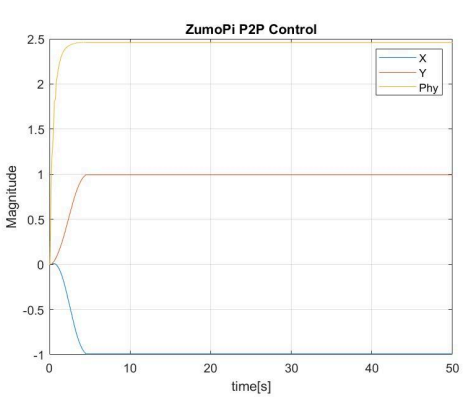


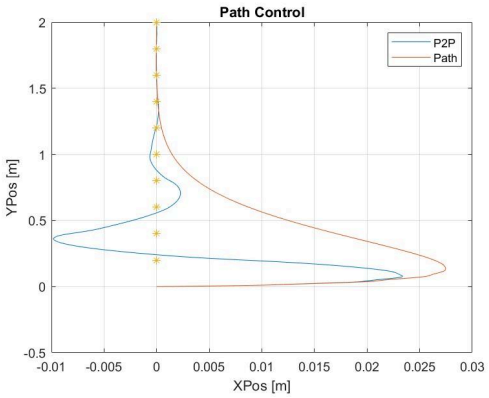
    - Open `ZumoSimulatiom_P2P.slx`
    - Set the desired x,y coordinates to [-1,1] and run the simulation.
    - Inspect how the robot moves to a single target (x,y).
    - Adjust the **motor PID gains** for example, increase K_p , K_i or add more θ gain value, and observe the changes in the trajectory.
    - **Record** your observations in a short table (parameter changed → resulting behavior).
4. **ZumoSimulatiom_P2P_MultiPoint**
    - Open `ZumoSimulatiom_P2P_MultiPoint.slx`, set a sequence of multiple waypoints (e.g., `[0 0.2; 0 0.4; 0 0.6; 0 0.8; 0 1]`).
    - Run the simulation and examine the scope outputs (e.g., robot x-y position vs. time).
    - **Try** extending the path to 2 meters (e.g., `[0 0.5; 0 1.0; 0 1.5; 0 2.0]`).
    - Observe how the robot transitions from one waypoint to the next and whether the trapezoidal velocity profile or PID gains cause any instability between points.
5. **ZumoSimulatiom_Path_Control**
    - Open `ZumoSimulatiom_Path_Control.slx`, specifying multiple points as in the previous step.
    - Pay special attention to the **heading error** (θ gain) and **cross-track error** (de gain).

- Gradually increase or decrease these gains to see how the robot “tracks” a continuous path.
- **Plot** the final path the robot follows and compare it to the desired path. Look for consistent offsets, oscillations, or overshoot around tight turns.

6. **Data Plotting and Analysis**
   - Each simulation has built-in scopes that record the robot’s position, and orientation.
   - Use `plot(...)` commands in MATLAB (e.g., `plot(Record_P2P.time, Record_P2P.signals.values)`) to visualize and compare different runs.
7. **Concluding tasks**
   - Replace the desired coordinates with a variable **DesiredCoordinates** in the simulations and run the following script `plotScript.m`

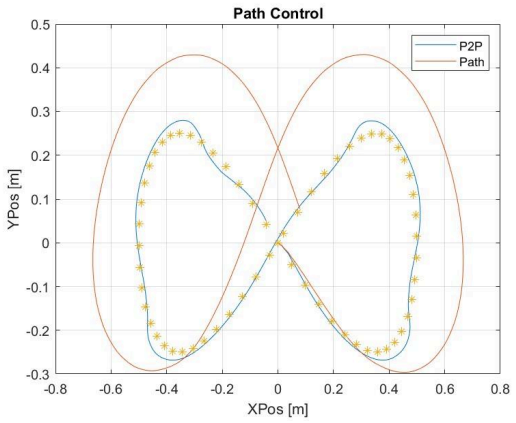
   - Adjust the algorithm's gains for optimal results without inducing jitter in the robot's angle. Plot the response and explain how each gain affects the system's behaviour.

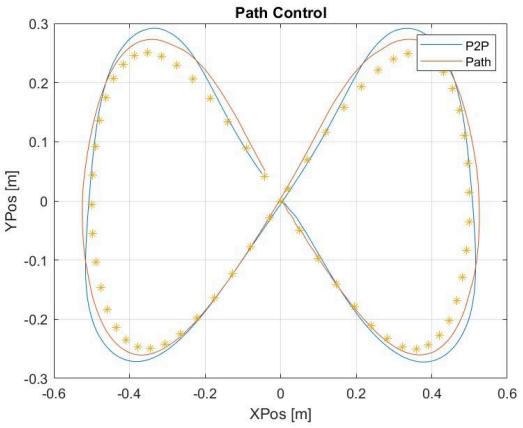




   - Define a path of your choosing (up to 1x1 metres) either by using discrete points or a continuous path from which 50-100 discrete points are derived. Run the simulation using the coordinates you have defined. Adjust the algorithm`s gains / parameters as needed for optimal performance and plot the response.

Example:





5. Deliverables

By the end of the pre-lab, you should have:

1. **A short summary** (bullet points or a paragraph) of your observations for each simulation (OpenLoop, P2P, MultiPoint, PathControl).
2. **Plots or screenshots** illustrating at least one gain-tuning experiment.
3. **Written answers** (brief notes) to the guiding questions, focusing on how parameter variations changed the robot's simulated performance and why.

Conclusion

Completing this Pre-Lab Activity will deepen your understanding of basic control concepts—odometry, PID, trapezoidal velocity profiles, and path-tracking algorithms—before working hands-on with the physical robot.

8. In-Lab Activity

Overview & Objectives

This in-lab activity will allow you to:

- **Deploy and test your path-control algorithms** on an actual two-wheel mobile robot
- **Compare real-world performance** to the pre-lab simulation results, identifying similarities and differences due to factors like wheel slip, encoder noise, and imperfections in sensor measurements.
- **Refine your control gains** (motor PI, path-control parameters) to account for real dynamics.
- **Log and analyze** data collected from the robot to evaluate performance metrics such as cross-track error, oscillation. Compare the actual path to the desired path.

1. Running Simple Path Control

Note that the actual platform uses different units than the simulation. (The simulation uses meters, m/s. while the code on the platform implemented as millimetre, mm/s)

- Open Example **PathControl.ino** under WS_Zumo/Arduino/Examples/PathControl/
- Copy the project to your student folder.
- Make sure the path coordinates are se to:

```
// Define desired path (coordinates in mm).
float desiredPath[][2] = {
  {100, 0},
  {200, 100},
  {300, 100}
};
```

- Under **PathController.cpp** you will find the various path control settings. Make sure to start with the default values.

```
// Initialize control parameters.
dt_time  = 0.01f;           // 10 ms update interval
WHEELS_DISTANCE = 98.0f;    // mm between wheels
a_max = 200.0f;             // maximum acceleration (mm/s^2)
v_max = 100.0f;             // maximum forward velocity (mm/s)
Kp = 1.0f;                  // Velocity control Proportional gain
Ki = 0.1f;                  // Velocity control Integral gain
Kp_theta = 100.0f;          // Path control theta heading error gain
Kp_de = 1.0f;               // Path Control cross-track error gain
```

- Run the code. Power on the motors and press A button to start motion.
- Observe how well it reaches the marked target points:

2. Real time observer application

- Open Example **PathExample.py** under WS_Zumo/Python/Examples/
- Copy the project to your student folder.
- The example records the odometry messages to **recorded_messages.txt** In addition it will plot the X,Y position in 2D. In a separate plot it will present several motion parameters over time.
- Run the application open chrome or explorer navigate to your car ip at port 8500 (eg: http://192.168.0.120:8500/)
- View the position in real time during the motion.
- Plot the path from the **recorded_messages.txt** using matlab or python.
- Mark the path coordinates on the plot.

3. Tuning the path control gains, PI Velocity controller.

- Open **PathControl.ino** under your student folder.
- Review the Path control gains.
- Adjust the Path control gains. Θ_e, d_e (**Kp_theta, Kp_de**) Change only one parameter at a time.
- Test what happens when one of the parameters is set to 0.
- Record several tuning experiments and create a plot in matlab or python comparing the results.

4. Trapezoidal Profile Experiments

- Increase **v_max** or **a_max** in small increments to see if the robot can maintain stable control at higher speeds with higher acceleration.
- Observe if higher speeds lead to larger final errors, especially in turning maneuvers.

5. Full path control

- Implement the path from the prelab. Remember to scale the parameters accordingly.
- Record the results and compare them to the desired path.

6. Optional: Introduce Sensor Fusion

- Using the gyroscope from the IMU try implementing a complementary filter for better heading estimation.
- Compare the heading stability with and without gyro integration.

7. Deliverables

By the end of the pre-lab, you should have:

Experiment Log / Tuning Table

- Document each set of gains (e.g., motor PI gains, path-control gains, trapezoidal profile parameters) that you test on the real robot.
- Note the resulting behavior: overshoot, final position error, time to settle, or cross-track error.

Plots / Trajectory Visualizations

- At least one plot showing the **actual path** vs. the **desired path** on the real robot.
- Overlay the **simulated** vs. **real-world** trajectories in the same coordinate system to highlight discrepancies.

Comparison & Analysis

- A short **written comparison** between simulation results (from the Pre-Lab) and observed real-robot performance.
- Identify key reasons for any deviations (e.g., wheel slip, sensor noise, unit conversion issues).

Final Controller Settings

- Provide your **final, tuned gains** (e.g., Kp,Ki, path-control θ gain, cross-track error gain) that yielded acceptable performance.
- Include brief justifications for these parameter choices (why each gain was increased/decreased).

Optional Sensor Fusion Results

- If you implemented gyro-odometry fusion, show **before vs. after** performance metrics or plots.
- Note any reduction in heading drift or improvement in path accuracy.

Short Reflection or Lab Report

- Summarize the most important lessons learned about how real-world factors (slip, noise, friction) influence path control.
- Outline potential future improvements (e.g., better sensor calibration, advanced filters).

Conclusion

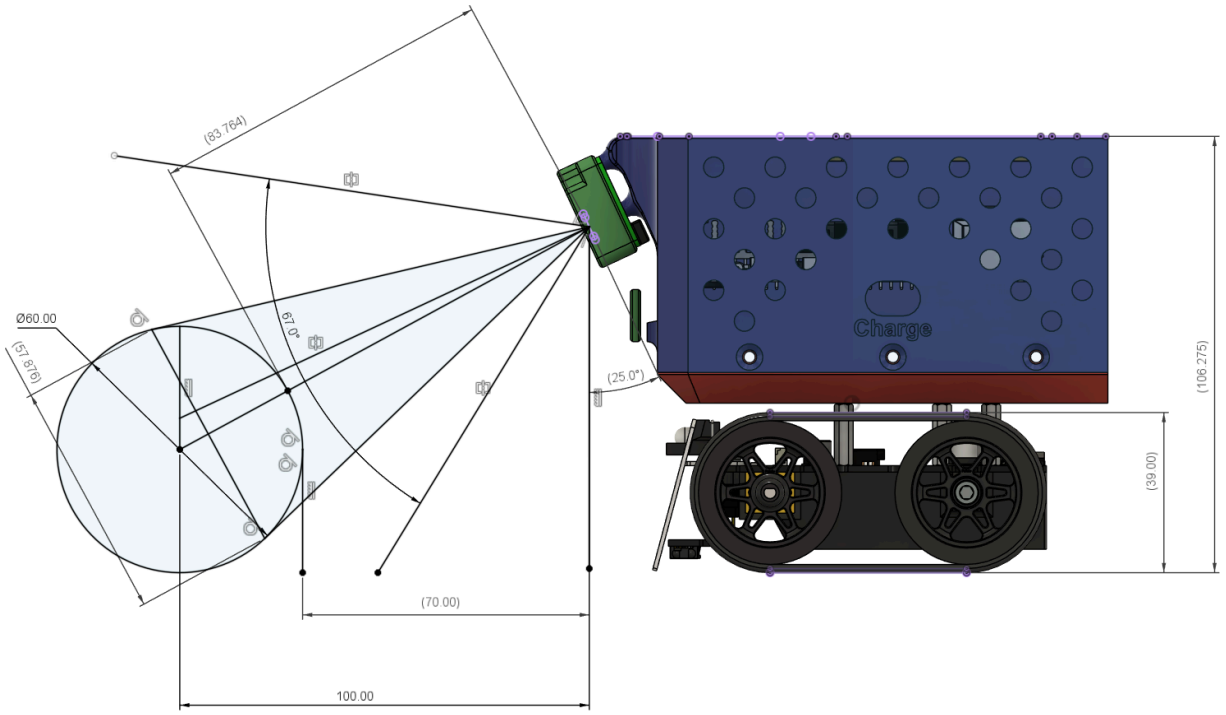
By completing this In-Lab Activity, you will gain practical experience implementing path control on an actual mobile robot. Comparing simulation to reality will illuminate the nuances of real sensor data, motor dynamics, and environmental factors. Through systematic tuning and data analysis, you'll refine your understanding of core control principles and better appreciate the complexities of robotics in a non-ideal world.

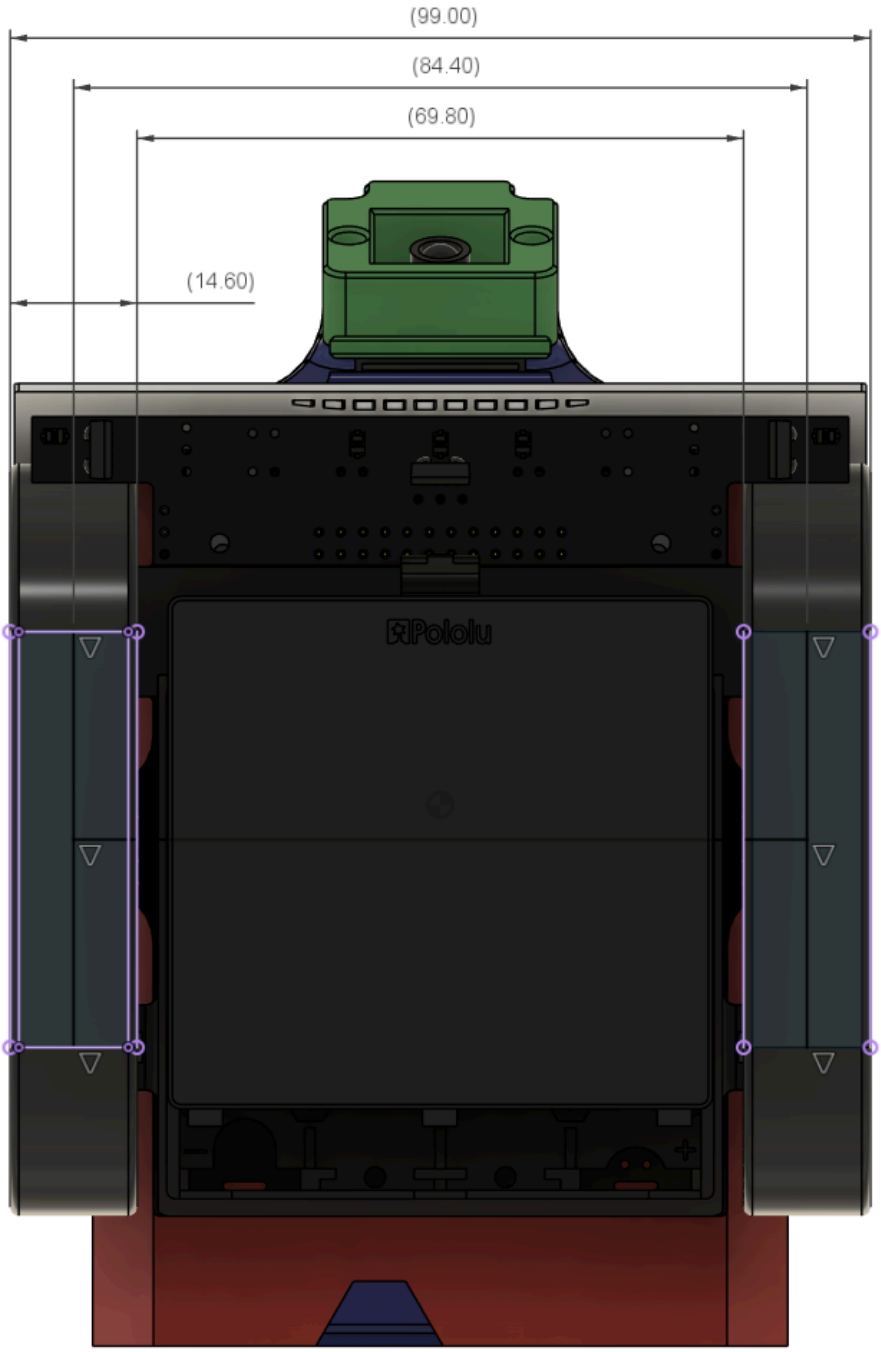
Appendix A:

Matlab script to convert matlab points to C++ style

```
%% convert Car coordinates to c style
CarVec = [0 , 0 ; 1 , 1 ; 2 , 2]; % desired path in matlab style
output_str ='float desired_path[][2] = { ';
output_str = [output_str, '{', num2str(CarVec(1, 1)), ' , ', num2str(CarVec(1, 2)), '}'];
for i = 2:size(CarVec, 1)
% Append each row to the output as a C++ style array element
output_str = [output_str, ' , {', num2str(CarVec(i, 1)), ',', num2str(CarVec(i, 2)), '}'];
end
output_str = output_str + " };";
output_str
```

ZumoPi platform dimensions




(99.00)
(84.40)
(69.80)
(14.60)
Pololu