CMPUT 312 Lab 1
Differential Drive Vehicle

Nikolai Philipenko
Yuyang Wang
Fall 2025

TABLE OF CONTENTS

1 Introduction

Mobile robots are playing an increasingly important role in modern society, with applications ranging from industrial automation to personal assistance. A widely used and cost-effective example is the differential drive robot, which moves using two independently controlled wheels mounted on either side of its axle. By varying the speed of the wheels, the robot can follow curved trajectories, while precise wheel control enables forward, reverse, and rotational movements.

This lab experiment involved a series of progressive tasks, beginning with drawing a simple straight line and culminating in the (theoretically) precise tracing of a geometric lemniscate. Accomplishing these complex movements required the development of custom implementations for both a robot state estimator and a velocity controller.

In order to smoothly develop in the project, we designed the project structure as follow:

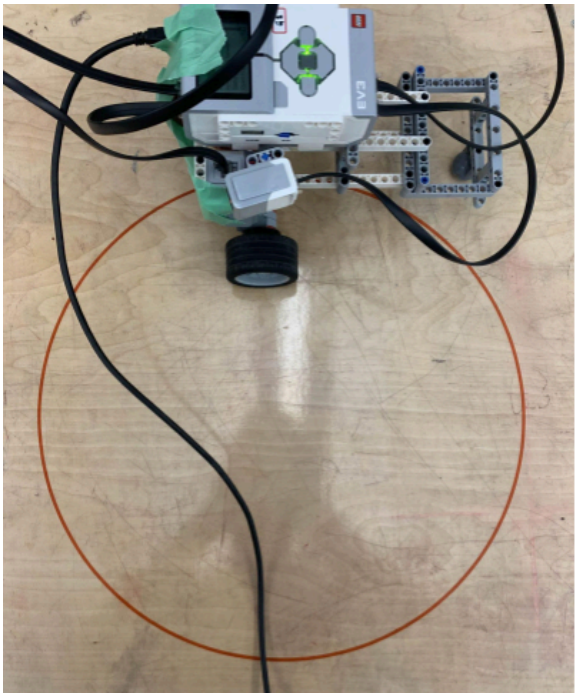
- **Low-level layer - robot_core:** This layer interacts directly with ev3dev2 APIs and provides essential control functions for the robot's motors and sensors needed to operate the robot.
- **Middle-level layer - kinematics:** This layer handles all kinematic and geometric calculations, such as a state estimator for dead reckoning the current pose, functions to generate shape waypoints, and a well-designed pure-pursuit velocity controller.
- **High-level layer - lab_tasks:** This layer is used to implement tasks directly from our lab assignments.

```
CMPUT312-LAB-01/
├── main.py                    # main entrance of the program
├── robot_core/                # store all low-level robot control functions
│   ├── __init__.py
│   ├── constants.py           # store constants like wheel radius, axle track
│   ├── driver.py              # encapsulate motor control functions
│   └── sensors.py             # encapsulate sensor reading functions
│
├── kinematics/                # store kinematics and geometry related functions
│   ├── __init__.py
│   ├── estimator.py           # dead reckoning calculations for robot position
│   ├── helper.py              # helper functions for our controller
│   ├── controller.py          # pure-pursuit controller for following shapes
│   ├── state.py               # managing the current state of the robot
│   └── geometry.py            # geometry of generating waypoints for shapes
│
└── lab_tasks/                 # store all lab task implementations
    ├── __init__.py
    ├── task2.py
    ├── task3.py
    ├── task4.py
    └── task5.py
```

This layered structure was to promote code modularity, reusability, and a clear separation of concerns, allowing us to focus on one project submodule at a time.

Error analysis was a central theme throughout this lab. As we progressed through the experiments, the gap between theoretical expectations and real-world performance became increasingly apparent. We measured errors using a Euclidean coordinate system and attempted to reduce them, only to discover that some stemmed from limitations in our algorithms, while others arose from the inherent ambiguities of the physical system. For instance, adjusting the main axle length improved the accuracy of our circular trajectories but degraded the precision of the lemniscate. This highlighted a recurring pattern: solving one problem often uncovered deeper, more subtle challenges.

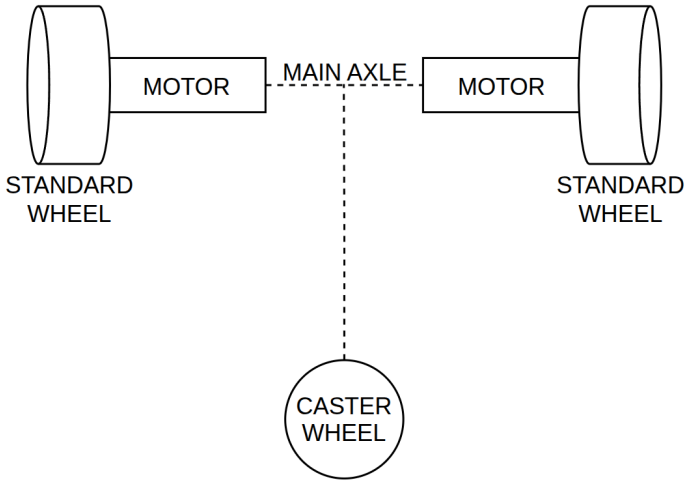
“As our circle of knowledge expands, so does the circumference of darkness surrounding it.”
- Albert Einstein



2 Building a Differential Drive Vehicle

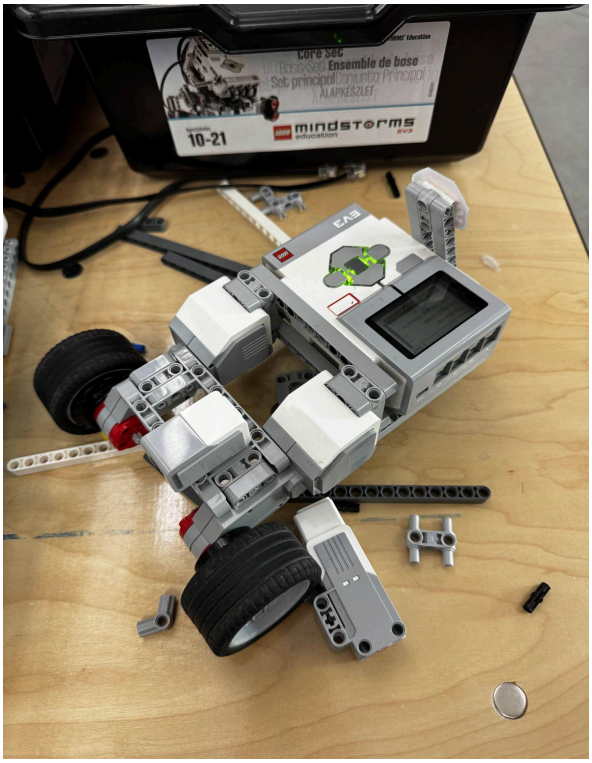
In order to build our differential drive vehicle, two Lego Mindstorms EV3 Large Servo Motors were used to provide torque to the two standard wheels mounted in parallel on the main axle. A free turning wheel, or caster wheel, was attached to the back of the vehicle to provide stability.

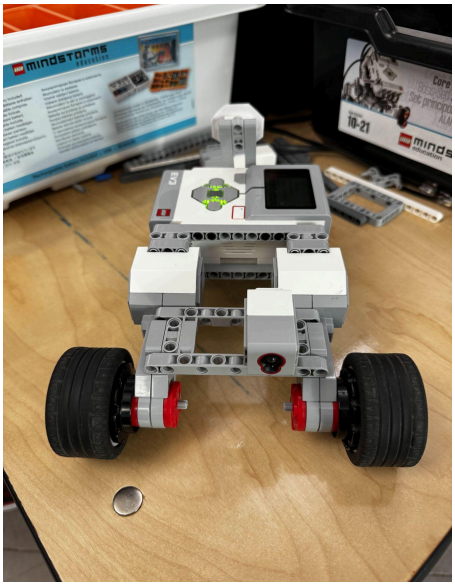
Figure 1: Differential drive motor and wheel assembly.



The first version of our differential drive robot (known as V1), included a large main axle length which enhanced the robot's stability and improved the weight distribution across both wheels. A balanced weight distribution across the wheels helps ensure that both wheels experience a similar amount of slip.

Figure 2: Differential drive robot build V1.





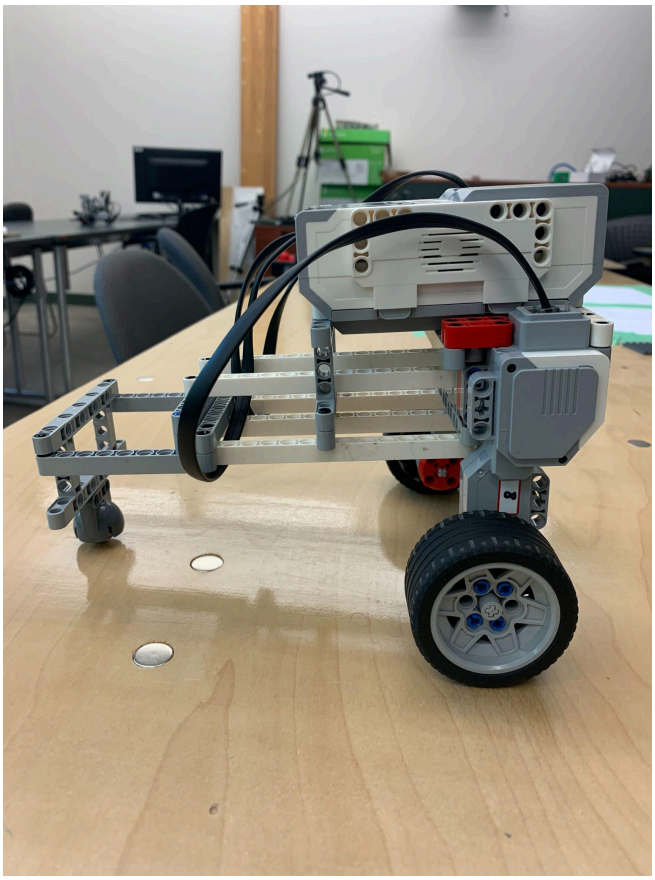
Flaws in the V1 design quickly showed during our extensive testing. The EV3 computer makes up the majority of the vehicle's weight. As seen in Figure 2, the computer was placed at the back of the robot. This caused two unforeseen consequences severely limiting the robot's performance.

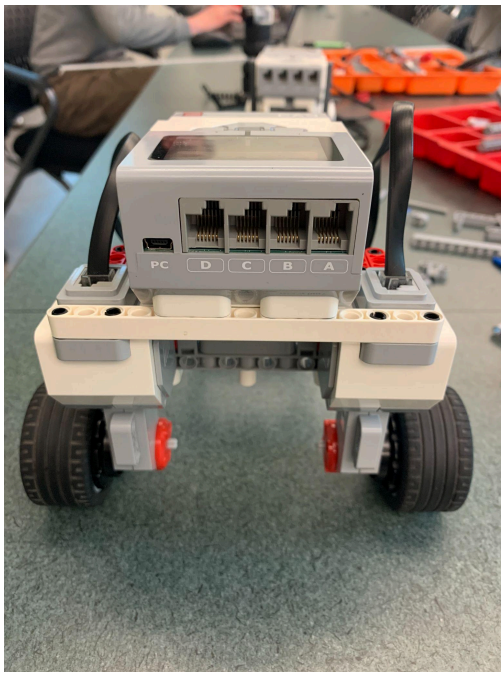
First, the computer at the back of the robot reduced the weight on the front wheels. As most maneuvers we performed used a closed-loop architecture with the feedback being the robot's state, wheel slippage caused severe drift between where the robot thought it was and where it actually was. As a result, path traversals became very inaccurate over a short period of time. We needed to increase the friction between the wheels and the ground to reduce this slippage.

Secondly, when any moderate force was applied to the castor wheel, it became very difficult to rotate. The internal bearing exhibited a high level of friction, and with the computer resting directly on top of it, the wheel effectively behaved like a rigid support.

The solution to the weight distribution problem was straightforward: reposition the computer higher and further forward to increase the normal force on the drive wheels and reduce the load on the caster wheel. With this adjustment, we created the second version of our differential drive robot.

Figure 3: Differential drive robot build V2.





3 Linear and Angular Error Data Collection and Analysis

Quantifying the robot's errors is essential, as they directly affect the accuracy of all subsequent tasks.

3.1 Methodology

We employed two distinct methods to measure the error in our robot's motion:

1. **Physical Measurements**
   This method directly reflected the robot's trajectory in the real world. We attached a marker to the robot to trace its path, then measured the linear displacement using a metric ruler and the final heading using a protractor, all with respect to a fixed coordinate frame. Although the width of the marker trace introduced some measurement uncertainty, this method provided the most direct approximation of real-world error.

2. **Dead Reckoning Estimator**
   The dead reckoning estimator relied on the onboard motor encoders. By estimating the instantaneous wheel speeds and integrating them over time, we obtained the robot's internal belief of its state within a predefined Euclidean coordinate system.

During preliminary testing, we also attempted to incorporate a gyroscope. However, significant drift made its readings unreliable, and we ultimately excluded it from our experiments.

3.2 Straight Line Error

We conducted three tests to evaluate the robot's straight-line driving performance. In each trial, the robot was programmed to travel a fixed distance of 250 mm, chosen to remain within the limits of our ruler. For successive tests, the linear speed was incrementally increased in order to analyze how error varied with the power applied to the robot's motors.

Table 1: Error in straight line. Distance = 250 mm

Run	Speed (mm/s)	State Estimator Distance (mm)	State Estimator Distance Error (mm)	Ruler Measurement (mm)	Ruler Measurement Error (mm)
1	30	247.8929	-2.4278	256.0	+6.0
2	60	249.4686	-0.5384	254.0	+4.0
3	90	247.0728	-2.9350	256.0	+6.0

3.2.1 Analysis

A clear discrepancy was observed between the two measurement methods. The dead reckoning estimator suggested that the robot slightly undershot the target, whereas physical measurements with the ruler showed consistent overshooting. Several factors may explain this difference:

- The robot's internal wheel encoders are innaccurate.
- Residual inertia after the motor impulse causes additional movement.
- Small inaccuracies in measuring the wheel diameter propagate into the calculations.

Variations across test runs were primarily influenced by linear speed, with a distinctly non-linear relationship between speed and error. The lowest error occurred at the intermediate speed of 60 mm/s, while the highest error was recorded at 90 mm/s. This trend is expected, as higher speeds increase inertia and the likelihood of wheel slippage. However, the unexpectedly high error at the lowest speed may be attributed to motor inconsistencies: at very slow rotations, the wheels exhibited slight jerkiness, which likely introduced additional friction and reduced accuracy in the encoder readings.

3.3 Angular Error

We programmed the robot to perform an in-place 360° rotation while collecting data at three different angular speeds. To ensure the trajectory was visible, the marker was repositioned so that the robot traced a circle rather than a single point.

Table 2: Error in angle. Rotation = 360 degrees.

Run	Angular Speed (deg/s)	State Estimator Angle (deg)	State Estimator Angle Error (deg)	Protractor Measurement (deg)	Protractor Measurement Error (deg)
1	20	357	-3.00	384	+24
2	40	357.59	-2.41	397	+37
3	60	357.21	-2.79	391	+31

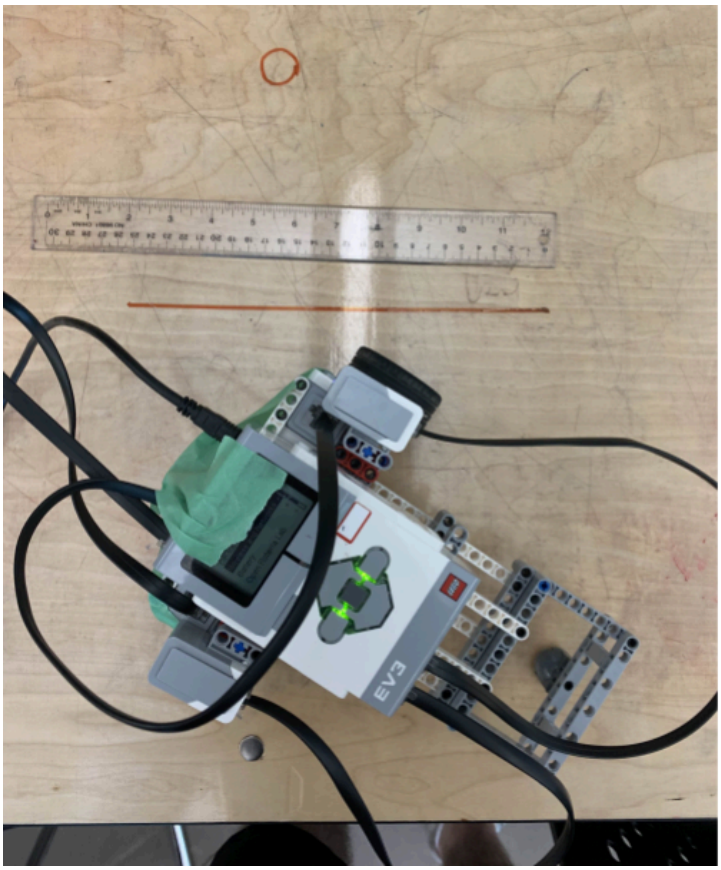
3.3.1 Analysis

There is a notable difference between the dead reckoning estimator and the protractor measurements. The dead reckoning estimator consistently reports that the vehicle slightly undershoots the target angle, whereas the protractor measurements indicate that the robot actually overshoots the target. For example, in Run 1 at 20 deg/s, the state estimator reports an angle of 357°, under by 3°, while the protractor shows 384°, overshooting by 24°. This discrepancy can be attributed to several factors:

- The robot's internal kinematic model, based on wheel encoder readings, does not perfectly capture the physical behavior.
- Inertia effects cause the robot to continue rotating briefly after motor commands stop.
- Small inaccuracies in measuring the axle length amplify errors in angle estimation.

The variation in error across runs is mainly influenced by the angular speed. Interestingly, the error does not increase linearly with speed. The smallest protractor measurement error (+24°) occurs at the lowest speed (20 deg/s, Run 1), while the largest error (+37°) appears at the middle speed (40 deg/s, Run 2). This trend is nearly the opposite of what was observed in the linear distance error analysis, indicating a tradeoff between linear and angular accuracy. A likely cause of the angular error is the inner wheel slipping during rotation, which accumulates over time and contributes significantly to the overshoot observed in the protractor measurements.

Figure 4: Measurement of linear and angular errors.



4 Shape Movements

There are a lot of ways to achieve this task:

- A top-down approach is to give the robot a series of targets and let it find the path.
- A bottom-up approach is to tell the robot the ICC and turning rate constantly.
- A hard-code approach is to use several simple shapes to approximate a complex shape.

We chose the top-down approach and identified three equally important software components required to achieve this task.

The first was **state estimation**, which allowed the robot to determine its position in the world frame at all times. This implementation is detailed in *Section 5: Dead Reckoning Position Controller*.

The second requirement was **path generation**, which involved calculating a trajectory that describes the desired motion. In our case, the path was represented as a sequence of *[x,y]* coordinates in the world frame that outlined the shape of interest.

The third and final component was a **path-following controller**. For this, we implemented the pure pursuit controller algorithm, which computes an angular velocity command that directs the robot from its current position to a point on the path located a specified look-ahead distance away.

4.1 Path Generation

Path Generation is to translate an abstract concept of shape into a concrete, machine-readable format. For our path-following controller, this format is a discrete, ordered sequence of waypoints, where each waypoint is an $[x,\ y]$ coordinate in the Euclidean coordinate system we defined. We used a generator that can pass in any parametric equations for complex curves and a direct vertex definition for polygons.
For continuous curves generation, the process follows three steps:

1. Define the Domain: We first specify the valid range of the parameter θ for the shape.
2. Discretize the Domain: The continuous domain is sampled to create a finite number of evenly spaced θ values. The number of samples determines the smoothness of the final path.
3. Generate Waypoints: The parametric functions $x(\theta)$ and $y(\theta)$ are evaluated at each discrete θ value to compute the final list of $[x,\ y]$ waypoints.

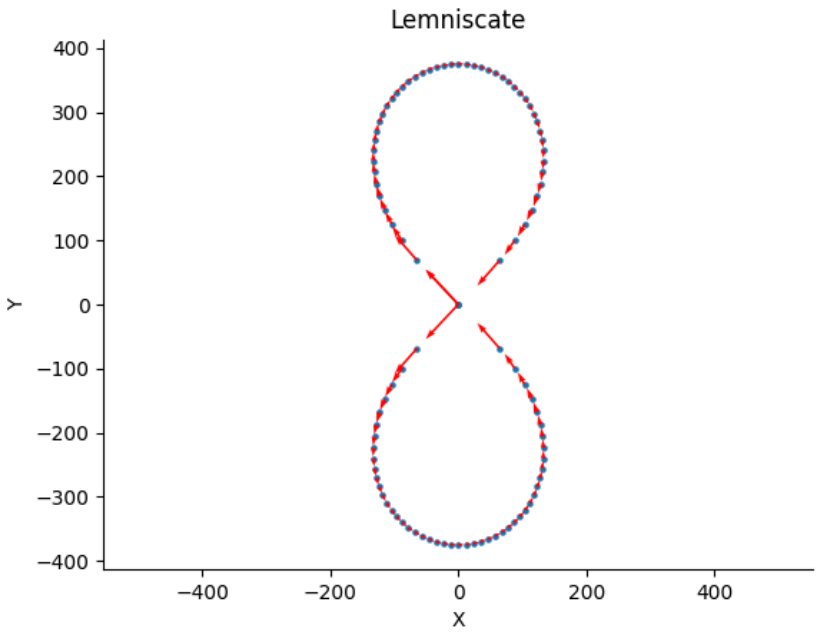
4.1.1 Lemniscate

The shape of lemniscate is defined as the polar equation $r^2 = a^2 cos(2\theta)$. We then calculated the parametric equations:

$$x(\theta) = asin(\theta)\sqrt{cos(2\theta)}$$
$$y(\theta) = acos(\theta)\sqrt{cos(2\theta)}$$

We then calculated the domain of it as $[-\pi/4, \pi/4] \cup [5\pi/4, 3\pi/4]$. We reversed the second segment to generate a continuous curve.

Figure 5: Trace of Lemniscate



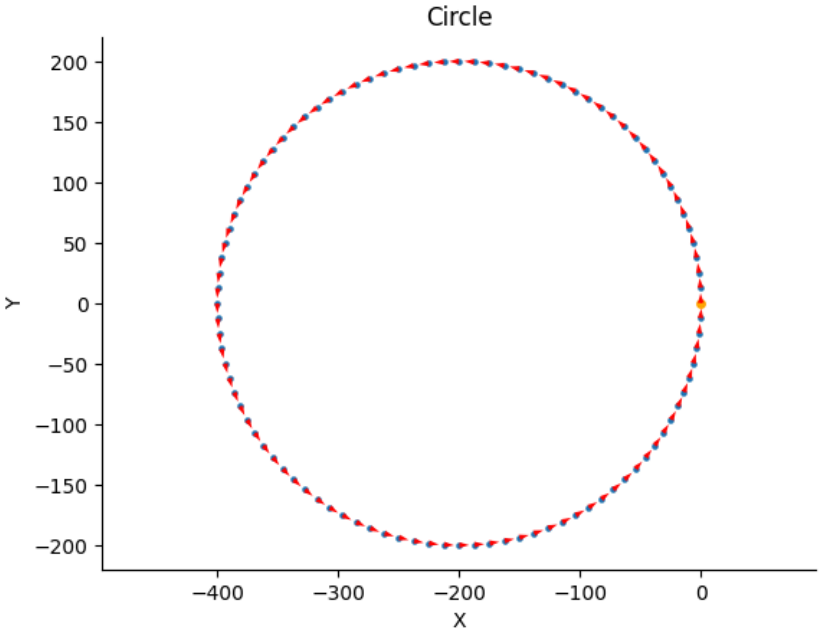
4.1.2 Circle

The circle is defined as the following parametric equations:

$$x(\theta) = Rcos(\theta) - R$$
$$y(\theta) = Rsin(\theta)$$

for θ in the domain $[0,\ 2\pi]$

Figure 6: Trace of Circle



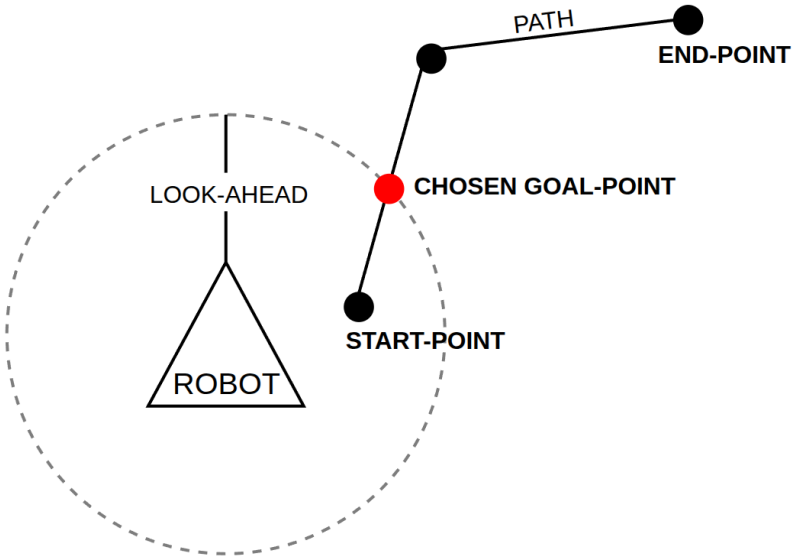
4.2 Path-Following Controller

Pure pursuit is a widely used path-tracking algorithm for differential drive robots. This particular algorithm was chosen for implementation due to its simplicity, geometric intuition, and adaptability to different path trajectories.

Given the current robot position and a precomputed path, the pure pursuit controller determines the necessary angular velocity for a robot to stay on the path. Linear velocity is treated as constant, therefore a simple p-controller was added to slow the robot down as it approached the final point on the path (the endpoint).

A configurable **look-ahead** distance is a key parameter in determining how the angular velocity is calculated. Pure pursuit will calculate a goal-point on the path that is a certain look-ahead distance away from the robot. This goal point is used in the angular velocity calculation.

Figure 7: Pure pursuit algorithm goal-point selection example.



A new goal-point is chosen at a high frequency update rate. Therefore, the robot will never reach the goal-point unless it is at the end of the path. This results in the robot following the path.

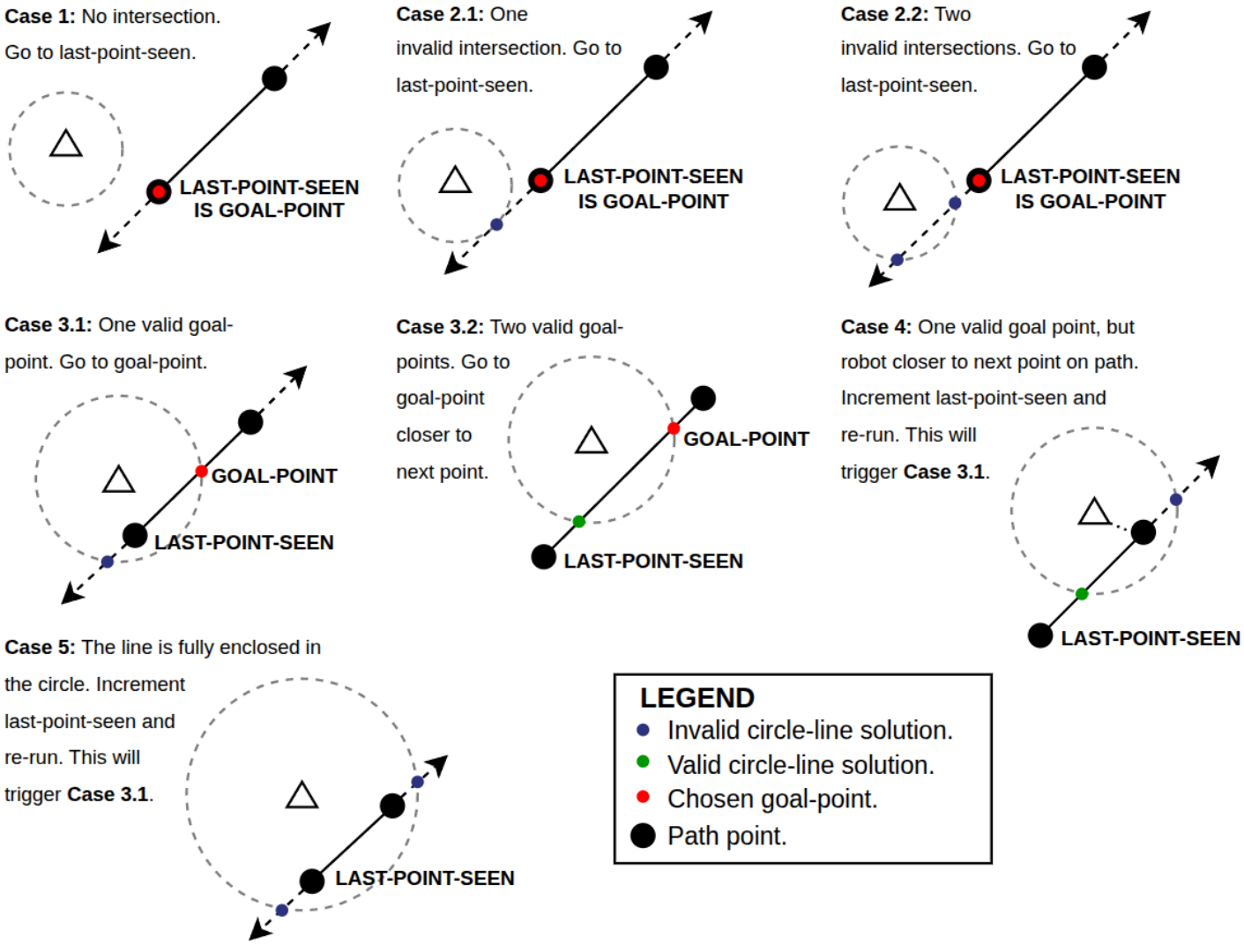
The look-ahead distance greatly affects the actual path traced. A small look-ahead distance provides tight path following but can result in robot overshoots and oscillations about the path. A large look-ahead distance provides a very smooth path following but can result in large curvatures near corners. Therefore the look-ahead distance influences the responsiveness and smoothness of the trajectory tracking.

4.2.1 Algorithm

The pure pursuit algorithm involves three major steps. First, the goal-point must be chosen. Second, given the goal-point and the robot's position and heading, the linear and angular velocities must be calculated. Finally, the linear and angular velocities must be converted to wheel velocities and sent to the robot's motors.

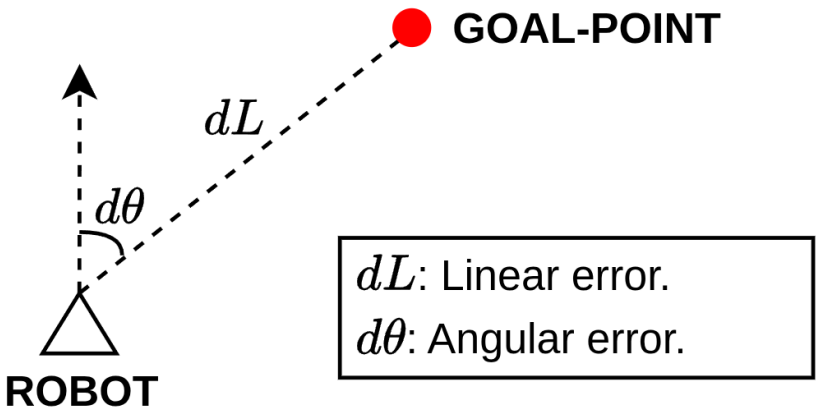
In order to choose a goal-point, pure pursuit finds the points of intersection between the circle centered at the robot with radius look-ahead distance and the infinite line drawn between the last-point-seen on the path and the next point on the path. The last-point-seen is tracked and incremented as the robot continues along the path. This results in several edge cases when finding the correct goal-point.

Figure 8: Circle-line intersection edge cases for pure pursuit algorithm.



Once the goal-point has been chosen, the linear and angular errors are calculated. The linear error is simply the distance between the robot and the goal-point. The angular error is the angle difference between the robot's current heading and the heading required to face the goal-point. Two P-controllers convert these errors into velocities by applying a configurable gain. These velocities are limited to a max linear and angular speed determined by configurable pure pursuit controller parameters.

Figure 9: Linear and angular errors.



Once the linear velocity V and angular velocity ω have been calculated, we use the standard differential drive kinematics to convert these velocities into the left and right wheel velocities, denoted V_L and V_R respectively:

$$V_L = V - \omega \times \frac{L}{2}$$

$$V_R = V + \omega \times \frac{L}{2}$$

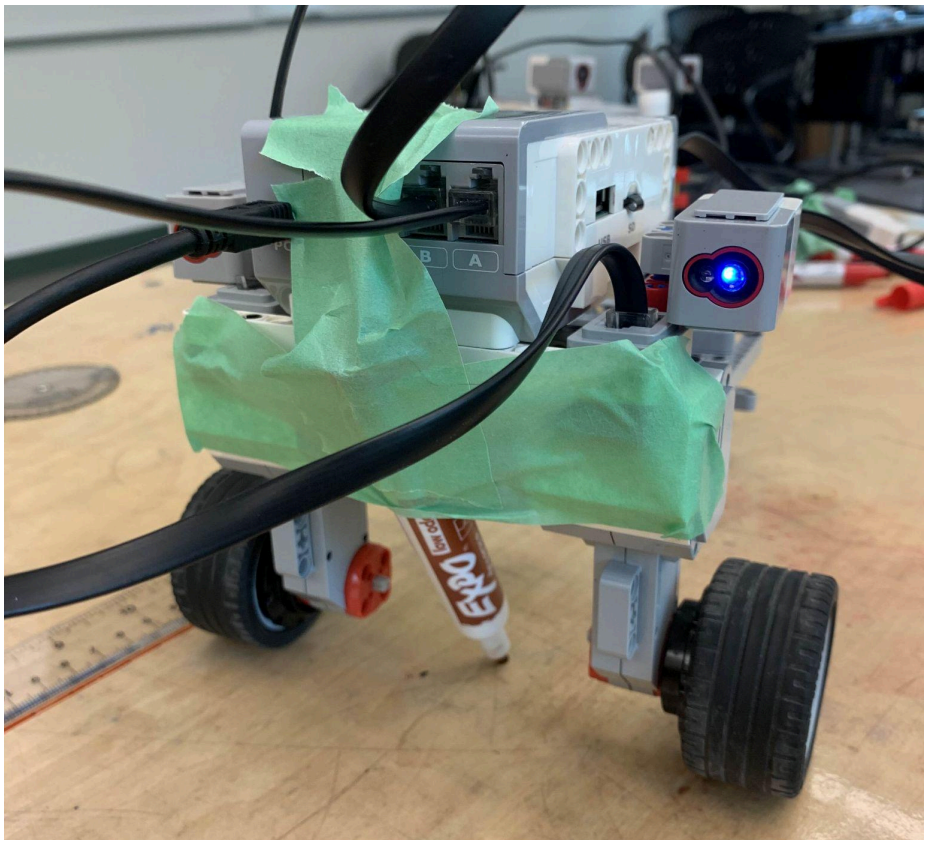
where L is the length of the main axle between the two wheels.

4.3 Testing and Results

For testing, three shapes were chosen for the robot to follow: a circle, a rectangle, and a lemniscate. The pure pursuit controller was used to follow the circle and lemniscate paths due to its excellence ability to trace out smooth curves. For the rectangle, a hard-coded approach was used because of the pure pursuit controller's inability to make sharp turns.

To capture the actual path traced by the robot, a dry-erase marker was mounted at the front of the robot along its main axle. This placement ensured that the recorded trajectory was not affected by the drifting or sliding behavior observed at the rear of the vehicle.

Figure 10: Dry-erase marker mounted on the robot.



4.3.1 Circle Path

Figure 11: Circle path performed by pure pursuit controller.

The circle path was performed three times immediately in succession. Figure 6 shows the path the robot took as it traversed a circle three times. The subscript s denotes the start of that run and the subscript e marks the end of that run.

$$1_s - start\ of\ run\ 1$$
$$2_s - start\ of\ run\ 2$$
$$3_s - start\ of\ run\ 3$$
$$3_e - end\ of\ run\ 3$$

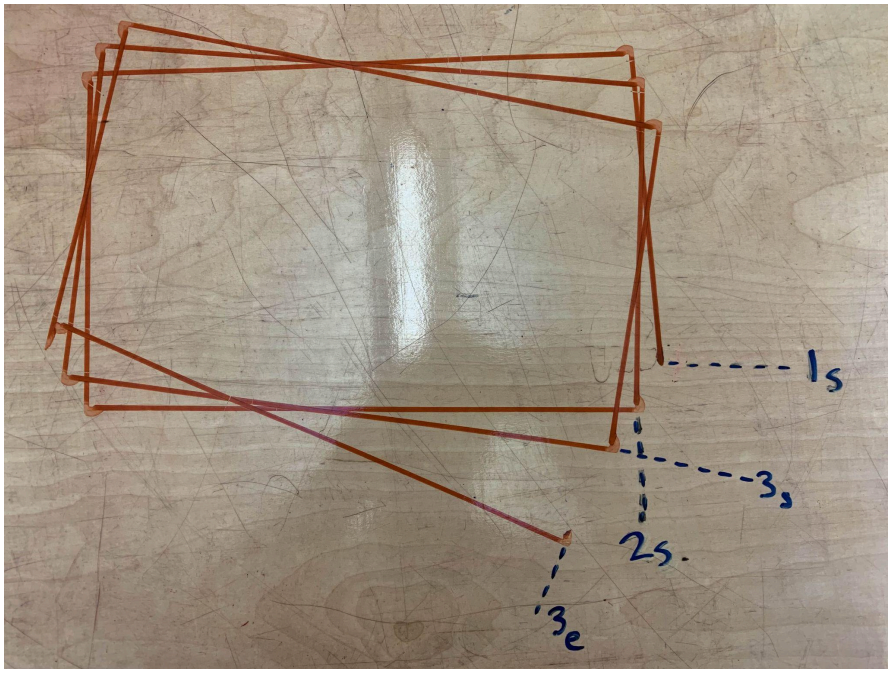
Note that 2_s and 3_s are also the end points for run 1 and run 2 respectively. The arc length between the start and end points for each run were measured. These errors are detailed in Table 3.

Table 3: Arc length errors of each circle path run.

Run Number N	**Arc Length Error** $(N_e - N_s)$ **(cm)**
1	1.6 cm
2	0.1 cm
3	3.4 cm

4.3.2 Rectangle Path

Figure 12: Rectangular path performed by hard coding.



Since the pure pursuit controller performs poorly when given paths with sharp corners, a dead reckoning or hard coding approach was taken for the rectangle path. The robot was commanded in open-loop control to drive in fixed-length straight lines split up by 90° turns.

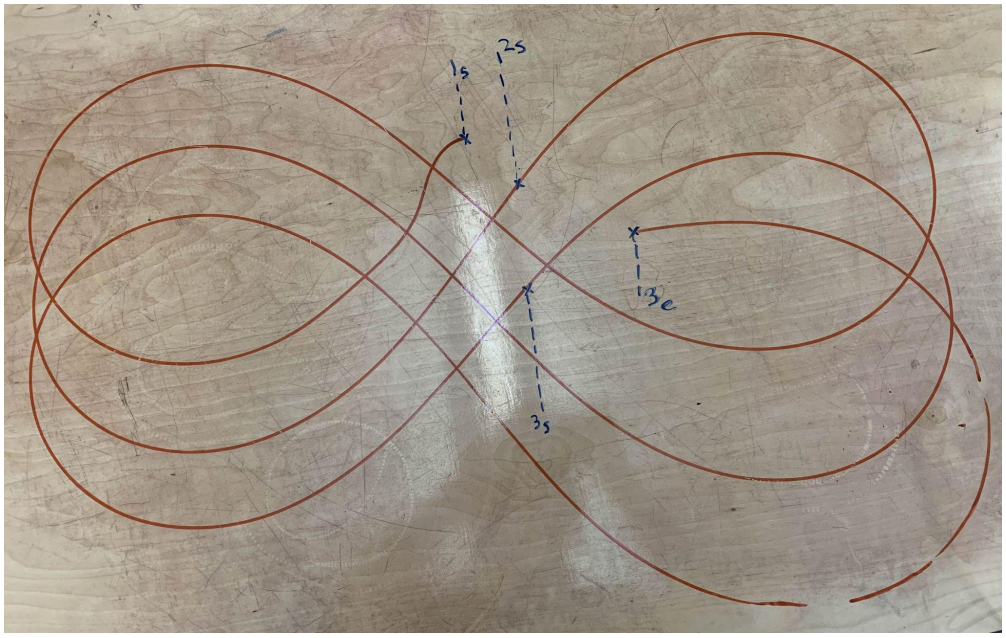
The rectangular path was performed three times immediately in succession. The distance between the start and end points of each rectangle was measured, along with the angle between the first and last sides of that rectangle. Ideally, the distance should be zero and the angle should be 90°. The measured errors are summarized in Table 4.

Table 4: Distance and angle errors of each rectangle path run.

Run	Distance Error between Start and End Points	Measured Angle between First and Last Sides, θ (deg)	Angle Error $\lvert 90^\circ - \theta \rvert$
1	2.4 cm	85°	5°
2	2.4 cm	83.5°	6.5°
3	5.1 cm	76°	14°

4.3.3 Lemniscate Path

Figure 13: Lemniscate path performed by pure pursuit controller.



By far the most interesting and complex shape, the lemniscate path was traced by the robot three times immediately in succession. The distance between the start and end points of each lemniscate was measured. These errors are detailed in Table 2.

Table 5: Distance errors of each lemniscate path run.

Run	Distance Error between Start and End Point (cm)
1	6.2 cm
2	8.8 cm
3	9.7 cm

4.3.4 Error Analysis

Since the circle and lemniscate paths were executed with the pure pursuit controller in a closed-loop architecture, accurate state estimation was essential for mapping the correct trajectory. Drift in the robot's estimated position significantly affected the accuracy of the actual path, as reflected in Tables 3 and 5, where error increased with each consecutive run.

For the rectangle path, a hard-coded open-loop approach was used. This method relied on the accuracy of the Lego Mindstorms motor API to achieve the commanded number of wheel rotations. Discrepancies between the commanded setpoints and the actual motor movements resulted in the increasing distance and angle errors reported in Table 4.

5 Dead Reckoning Position Controller

5.1 Implementation

The dead reckoning position controller estimated the robot's pose by polling the current wheel velocities and applying standard differential drive kinematics to compute the linear and angular velocities about the center of the main axle. These velocities were then integrated over time to update the robot's position and orientation.

The state estimator was critical for executing the complex movement tasks described in *Section 4: Shape Movements*, since the closed-loop path controller required continuous state feedback. To minimize integration error, the estimator operated at a high update rate.

First, the left and right wheel velocities are obtained. The Lego Mindstorms Motor API has a *speed* variable that can be polled, however it is in degrees per second. Therefore, the linear velocity of each wheel can be calculated from the following equations:

$$V_L = R * d\theta_L * \frac{\pi}{180}$$

$$V_R = R * d\theta_R * \frac{\pi}{180}$$

where V_L and V_R are the left and right linear wheel velocities, R is the radius of the wheel, and $d\theta_L$ and $d\theta_R$ are the left and right angular wheel velocities in degrees per second.

Once the wheel velocities are found, the following differential drive robot kinematic equations are used to obtain the robot's linear and angular velocities about the main axle midpoint:

$$V = (V_R + V_L) / 2$$

$$\omega = (V_R - V_L) / L$$

where V is the robot's linear velocity, ω is the robot's angular velocity, and L is the length of the main axle between the two wheels.

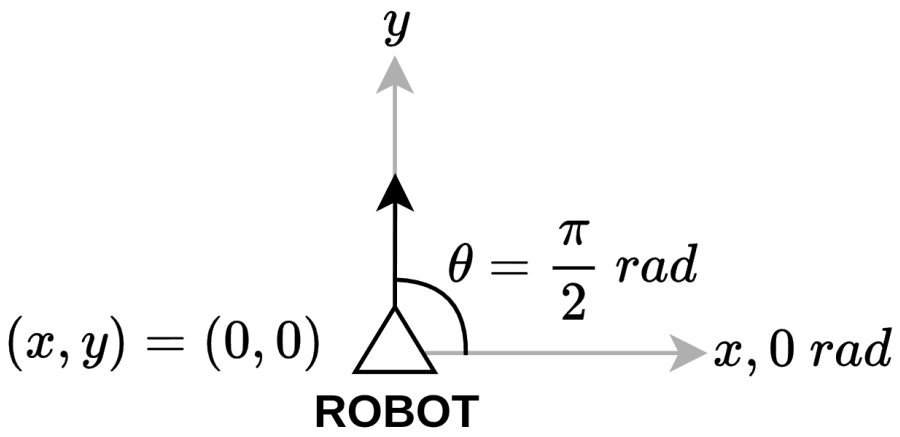
With the linear and angular velocity calculated, integrations can be performed to find the robot's current position and orientation in the world frame. Upon startup, we assume the robot is in the $[\, x = 0,\ y = 0,\ yaw = \frac{\pi}{2}\ rad\,]$ pose. The following equations are used to convert the linear velocity into its x, y components:

$$V_x = V cos(\theta)$$

$$V_y = V sin(\theta)$$

where V_x and V_y are the x, y components of the robot's linear velocity vector and θ is the robot's current yaw in the world frame.

Figure 14: Robot starting pose in world frame.



The time difference between every estimator state update was measured and used during the trapezoidal integrations. The trapezoidal rule was chosen as the software integration strategy due to its relatively high accuracy and low computational cost:

$$\theta_n = \theta_{n-1} + (\omega_n + \omega_{n-1}) / 2 \ * \ dt$$
$$x_n = x_{n-1} + (V_{x_n} + V_{x_{n-1}}) / 2 \ * \ dt$$
$$y_n = y_{n-1} + (V_{y_n} + V_{y_{n-1}}) / 2 \ * \ dt$$

where x_n, y_n, θ_n make up the robot's current pose and dt is the time difference between the n and $n - 1$ measurements.

5.2 Testing and Results

In order to test the accuracy of the dead reckoning position controller, the robot was aligned with a (0,0) coordinate reference frame. Then, three commands were sent to the motors in open-loop control. The commands defined the power to each wheel and the duration of the command.

Table 6: Commands sent in open-loop control to motors for state estimation testing.

Command Number	Left Wheel Power (%)	Right Wheel Power (%)	Duration (s)
1	20	10	2
2	10	10	1
3	-10	30	1

In each test, the three commands (1, 2, and 3) were executed sequentially. The test was repeated three times and after each the robot's state was recorded from the estimator and its pose was manually measured relative to the defined (0,0) coordinate reference frame.

Figure 15: Dead reckoning controller testing setup with (0,0) coordinate reference frame.

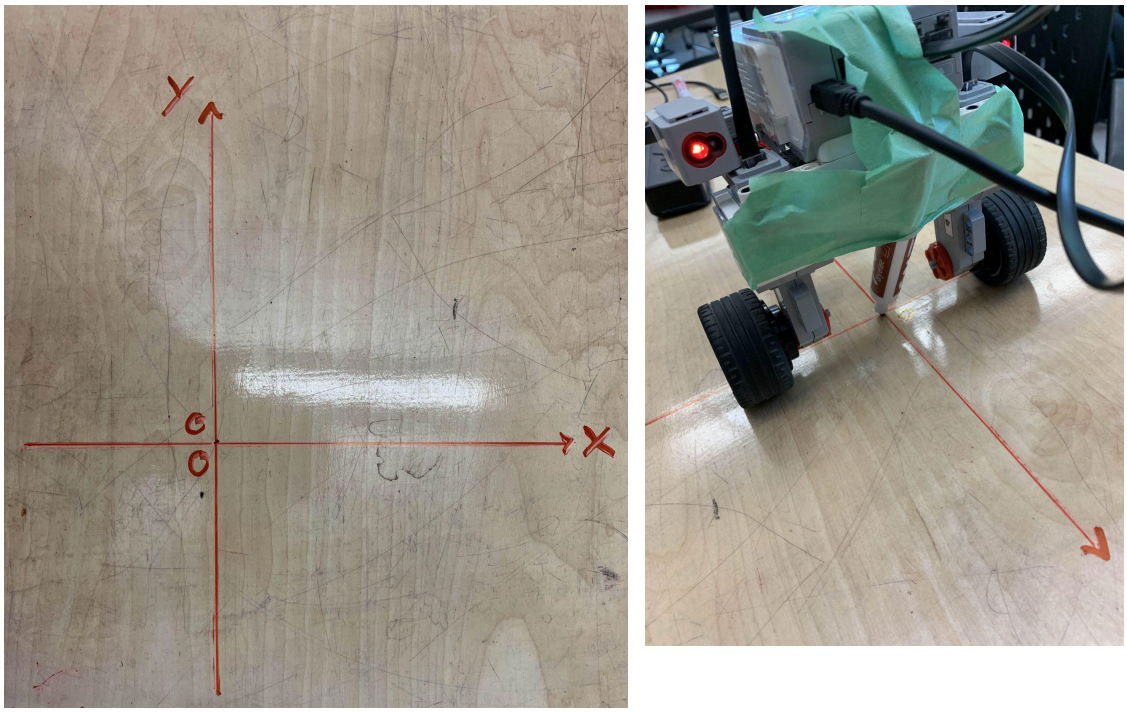
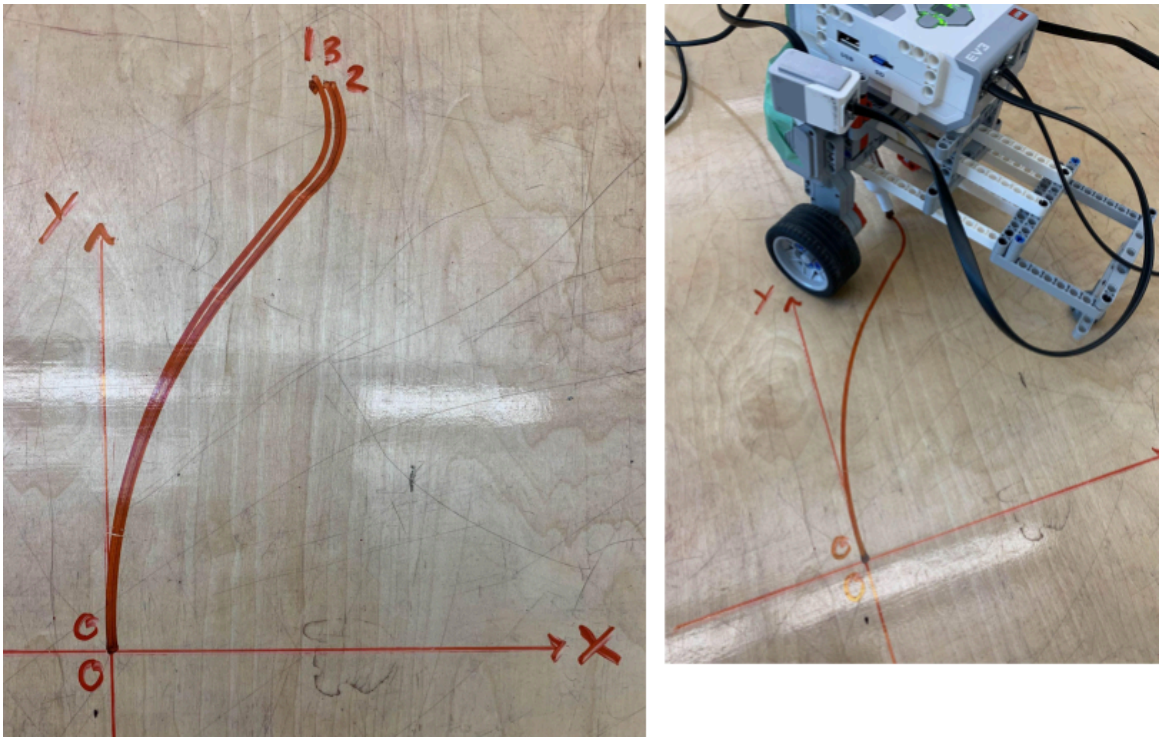


Figure 16: State estimator testing results.



From the results, the error was calculated between the robot's estimated position from the dead reckoning controller and its actual position from the (0,0) reference frame. The real final pose of the robot was measured manually using a ruler and protractor.

Table 7: Position and orientation errors between estimated and real robot state.

Test Number	Estimator Position (x,y) (cm)	Estimator Orientation (deg)	Real Position (x,y) (cm)	Real Orientation (deg)	Position Error (dx,dy) (cm)	Orientation Error (deg)
1	(8.12, 22.52)	131.29	(8.60, 22.90)	114.00	(0.48, 0.38)	17.29
2	(8.23, 22.40)	127.36	(9.40, 22.70)	115.00	(1.17, 0.30)	12.36
3	(8.19, 22.43)	128.08	(9.10, 22.70)	115.00	(0.91, 0.27)	13.08

5.2.1 Error Analysis

The accuracy of dead reckoning in a differential drive robot depends heavily on both geometric parameters and sensor fidelity. Two principal sources of error were identified in this experiment: uncertainties in model parameters and cumulative integration drift.

First, the kinematic model requires precise knowledge of the wheel diameter and main axle track length. These values were measured manually with a ruler; however, due to the robot's geometry, the exact measurement points were ambiguous. As a result, multiple reasonable interpretations of these parameters were possible. Even small variations in these values propagate directly into the kinematic equations, producing noticeable deviations in the estimated pose.

Second, dead reckoning inherently relies on the integration of wheel velocity measurements over time. Any noise in the encoder signals is accumulated during integration, resulting in a gradual divergence between the estimated and actual state. This drift is systematic rather than random, meaning that it consistently biases the estimated trajectory away from the ground truth.

The data in Table 7 illustrates these effects clearly. Across all three trials, the position error remained relatively small, typically less than 1.2 cm in the x-direction and less than 0.4 cm in the y-direction, indicating that translational estimates were reasonably accurate. However, the orientation error was consistently much larger, ranging from 12° to 17°. This systematic angular bias suggests either encoder miscalibration or inaccuracies in the measured main axle track length. While the position estimates were repeatable across trials, the consistent orientation offset indicates that the estimator was precise but biased.

6 Braitenberg Vehicle

Braitenberg Vehicles are simple autonomous machines whose straightforward control systems can generate surprisingly complex behaviors. Unlike previous tasks that relied on pre-calculated paths or fixed motor power, this task requires the vehicle to sense its environment and respond dynamically.

6.1 Sensor Calibration and Normalization

Raw sensor values are highly sensitive to ambient lighting conditions and may vary even under the same illumination. To build a robust system that adapts to different environments, we first calibrate the color sensors before operating the vehicle. During calibration, each sensor records 30 samples while a light source is placed directly in front of it, establishing the intensity range for that environment. The sensor readings are then normalized to a standardized range of [0, 1], allowing the vehicle to respond to relative light intensity as a percentage of the maximum value measured during calibration.

6.2 Motion Control

The vehicle is controlled by two spontaneously updated components: linear velocity and angular velocity. Linear velocity is proportional to the average light intensity the two sensors returned and angular velocity is proportional to the difference between the light intensity of the left and right sensors. To avoid a sharp turn that causes vehicle rollover, we added a smooth function for the angular velocity as following:

$$\omega_t = \alpha \cdot \omega_t + (1 - \alpha) \cdot \omega_{t-1}$$

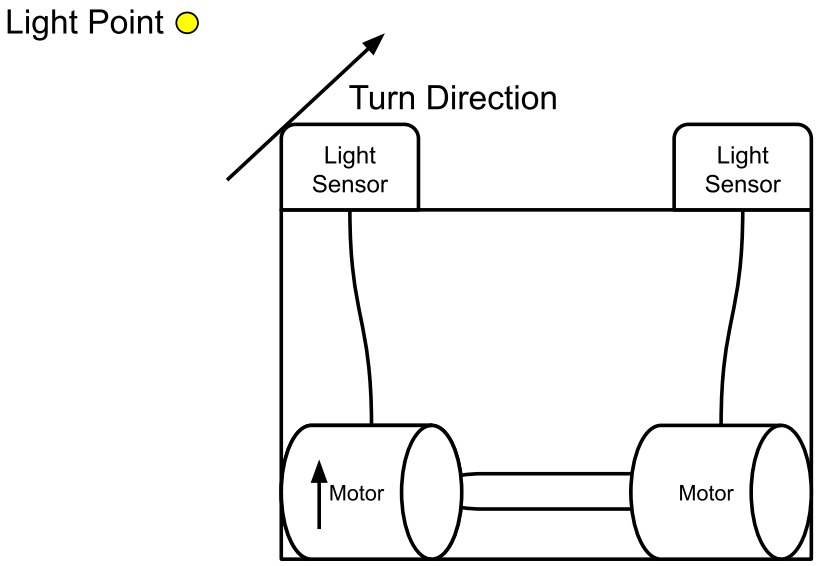
where ω is the angular speed and α is the smoothing factor.

6.3 Cowardice

Cowardice is a behavior in which the robot avoids the light source. This behavior is implemented using same-side excitatory connections: light detected by the left sensor increases the command to turn right, while light detected by the right sensor increases the command to turn left. The robot's angular velocity is then computed proportionally to the difference between the left and right sensor readings.

$$\omega \propto norm_{left} - norm_{right}$$

Figure 17: Cowardice Behavior

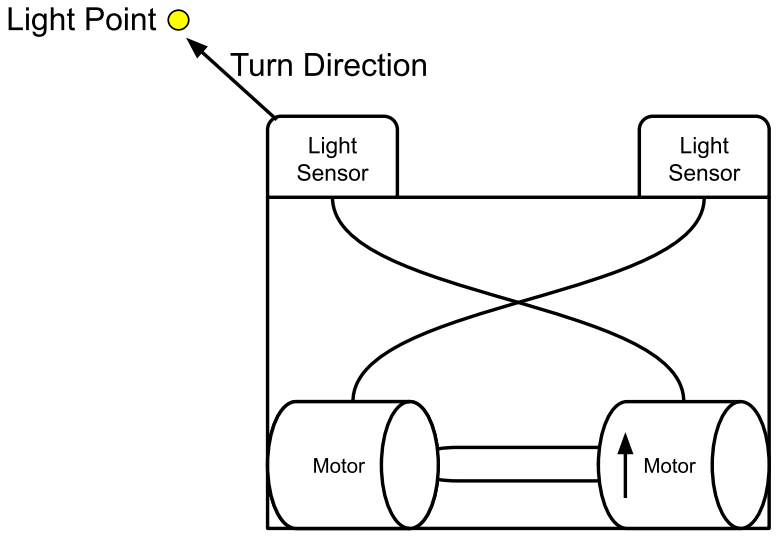


6.4 Aggression

Aggression is a behavior that drives the robot toward a light source. It is implemented using contralateral (opposite-side) excitatory connections: light detected by the left sensor commands a left turn, while light detected by the right sensor commands a right turn. Compared to Cowardice, the angular velocity calculation is inverted, causing the robot to steer toward rather than away from the light.

$$\omega \propto norm_{right} - norm_{left}$$

Figure 18: Aggression Behavior



7 References

1. Circle-Line Intersection Equations:
   https://mathworld.wolfram.com/Circle-LineIntersection.html,
2. MATLAB Pure Pursuit Controller:
   https://www.mathworks.com/help/nav/ug/pure-pursuit-controller.html,
3. Basic Pure Pursuit Implementation:
   https://wiki.purduesigbots.com/software/control-algorithms/basic-pure-pursuit,