Project report and progress: Mobile Robotics

Autonomous ball catcher

Shivansh Pratap Singh (2024702024), Siddharth Jain (2025701046)

Summary

This report presents a structured overview of the initial development phase of the Autonomous Ball-Catching Mobile Robot project. It documents the establishment of a stable and reliable mobile base, detailing the hardware integration of the power, sensing, and actuation subsystems, along with the corresponding control architecture. The report further outlines the implementation of the baseline PID controller for the vehicle's motion regulation and describes the calibration procedures used to align the dual-camera system within a common world frame. Following calibration, a preliminary triangulation pipeline was developed to enable consistent 3D localisation of the robot and target objects. The report concludes with the current status of the trajectory-estimation module, which is under active development and will form the foundation for predictive interception in the subsequent stages of the project.

Hardware Architecture

The robot's locomotion system is built upon a four-wheel differential drive chassis. The architecture is divided into two layers: the *High-Level Planner* (Raspberry Pi) and the *Low-Level Controller* (Arduino Mega).

Component Integration

- **Actuation:** Four DC motors (Micro DC12V High Torsion DC Geared Motors) equipped with quadrature encoders (600 PPR) provide propulsion and feedback.
- **Power Electronics:** An LM298 (L298N) Dual H-Bridge Motor Driver manages high-current power delivery to the motors.
- **Controller:** An Arduino Mega serves as the dedicated real-time controller. It reads encoder interrupts and computes PID outputs to maintain target RPMs.
- **Planner:** A Raspberry Pi 4B acts as the brain, sending directional commands (Forward, Backward, Left, Right, Stop) via USB Serial communication (WiFi enabled).

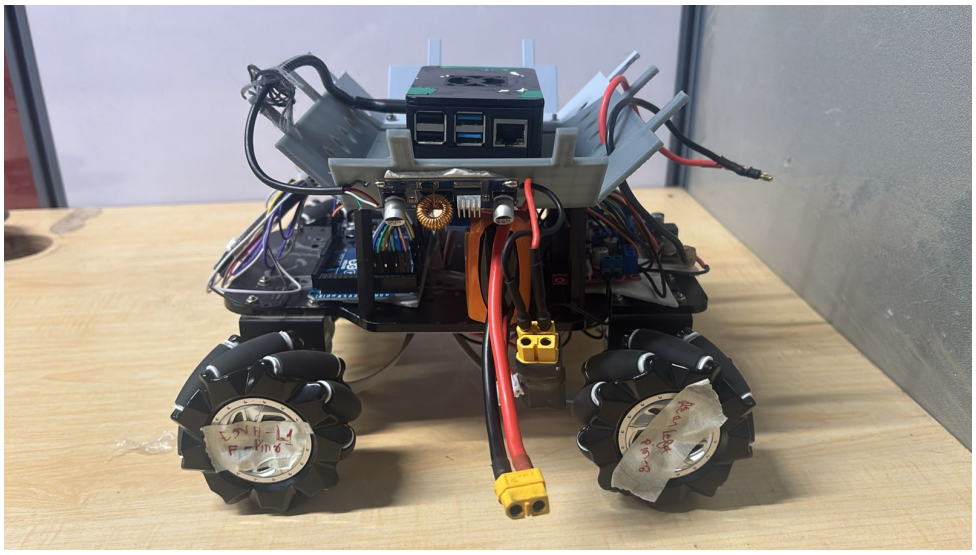


Figure 1: The assembled 4WD mobile base showing the component stacking and wiring layout.

Software Implementation

Low-Level Control (Arduino)

The firmware (`four_wheel_drive.ino`) implements a Proportional-Integral-Derivative (PID) controller to ensure consistent motor speeds despite friction or battery voltage variance.

PID Configuration: To synchronize the four independent wheels, we utilized the `PID_v1.h` library. Through empirical tuning, the following gains provided the most stable response without oscillation:

$$K_p = 1.5, \quad K_i = 0.5, \quad K_d = 0.1$$

High-Level Navigation (Python)

The navigation script (`navigator1.py`) running on the Raspberry Pi handles the robot's trajectory logic. It establishes a Serial connection at 9600 baud to the Arduino.

```
# Heading Correction Logic
angle_to_goal = math.atan2(error_y, error_x)
heading_error = angle_to_goal - current_angle

# Normalize error to [-pi, pi]
while heading_error > math.pi: heading_error -= 2 * math.pi
while heading_error < -math.pi: heading_error += 2 * math.pi

if abs(heading_error) > HEADING_TOLERANCE:
    # Send turn command ('a' or 'd')
else:
    # Send forward command ('w')
```

Listing 1: Navigation Logic Snippet

Currently, the script uses a simulated position generator to validate the logic flow. In upcoming weeks, this will be replaced with real-time data from the overhead camera system.

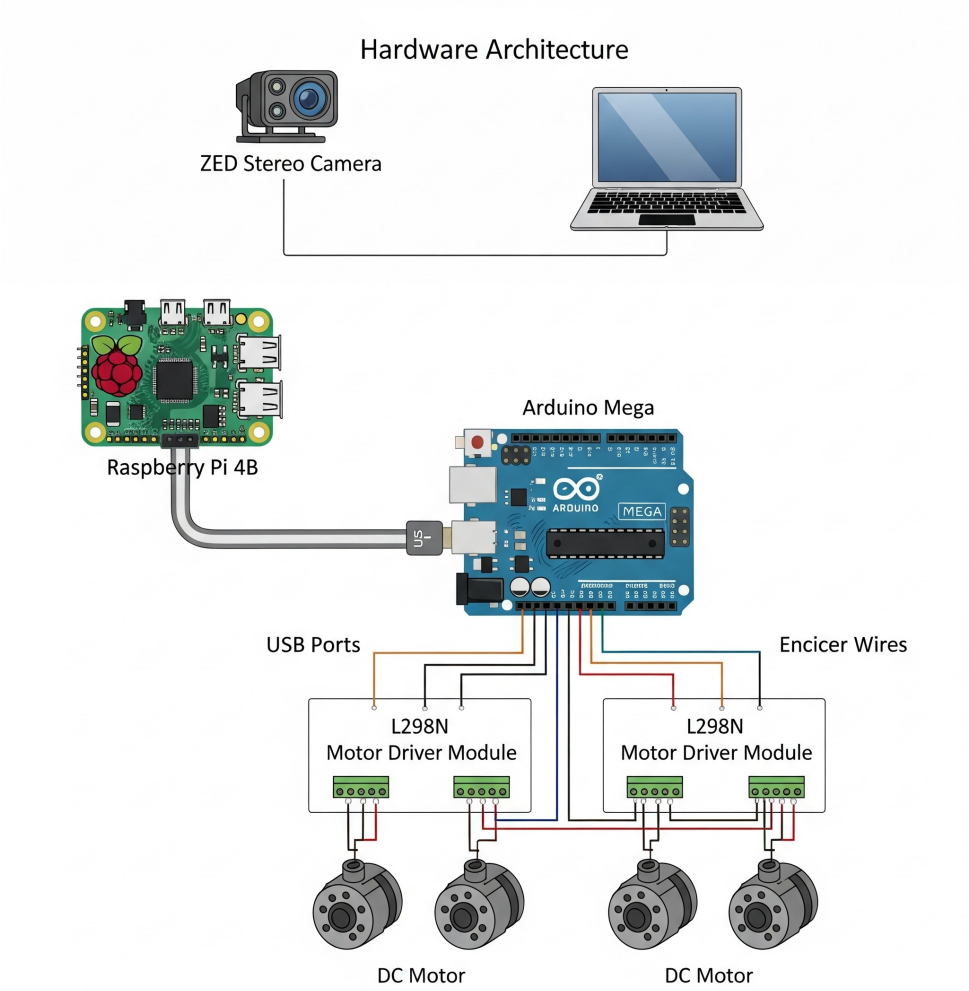


Figure 2: Project Pipeline

Hardware Testing and Validation

Bench Testing

Initial tests were conducted with the robot raised to verify motor directionality and encoder signal integrity. We confirmed that all four encoders (Interrupts 0, 1, 4, 5) correctly register pulses and that the H-bridge logic aligns with the kinematic model.

Performance Metrics

Ground tests were conducted to evaluate the limits of the drive train.

Metric	**Observed Value**
Max Linear Velocity	$\approx 1.1\,\mathrm{m/s}$
Command Latency (Serial)	Negligible ($< 10\,\mathrm{ms}$)

Table 1: Week 1 Performance Benchmarks

The maximum speed of 1.1 m/s is sufficient for intercepting a paper ball thrown at moderate speeds. The PID controller successfully maintained straight-line motion, correcting for the inherent mechanical differences between the four motors.

Stereo Vision & Calibration

To enable the robot to catch a flying object, accurate 3D localization is critical. We utilized the stereo capabilities of the ZED 2i camera, treating the left and right lenses as independent observers to compute a shared 3D world frame.

Theoretical Formulation

The vision system is modeled using the standard Pinhole Camera Model. A 3D point $\mathbf{X} = [X, Y, Z, 1]^T$ in the world frame is mapped to a 2D pixel $\mathbf{x} = [u, v, 1]^T$ via the projection matrix P:

$$s\mathbf{x} = \mathbf{K}[\mathbf{R}|\mathbf{t}]\mathbf{X} \tag{1}$$

Where:

- $\mathbf{K}$ is the Intrinsic Matrix containing focal lengths (f_x, f_y) and principal points (c_x, c_y).
- $\mathbf{R}$ and $\mathbf{t}$ are the Extrinsic Rotation and Translation parameters relating the camera frame to the world frame.
- s is a scale factor.

Calibration Procedure

We developed a custom Python script using the OpenCV library to calibrate the cameras. The process involved two stages:

1. **Intrinsic Calibration:** We captured a series of checkerboard images individually for the left and right cameras. Using `cv2.calibrateCamera`, we minimized the reprojection error to solve for the camera matrices $\mathbf{K}_{left}$ and $\mathbf{K}_{right}$ and their respective distortion coefficients.
2. **Stereo Extrinsic Calibration:** Using synchronized image pairs, we computed the rigid transformation $(\mathbf{R}, \mathbf{T})$ between the left and right cameras using `cv2.stereoCalibrate`. This step is crucial for triangulation as it defines the baseline geometry.

Triangulation via Direct Linear Transform (DLT)

Once the projection matrices $P_1 = \mathbf{K}_{left}[\mathbf{I}|\mathbf{0}]$ and $P_2 = \mathbf{K}_{right}[\mathbf{R}|\mathbf{T}]$ were established, we implemented the DLT algorithm to reconstruct 3D points from matched 2D pixel observations.

For a single 3D point $\mathbf{X}$ observed as $\mathbf{x}_1$ in view 1 and $\mathbf{x}_2$ in view 2, the cross product constraint $\mathbf{x}_i \times (P_i\mathbf{X}) = \mathbf{0}$ yields a linear system of the form $\mathbf{AX} = \mathbf{0}$.

We implemented this solver using Singular Value Decomposition (SVD), where the solution $\mathbf{X}$ corresponds to the right singular vector associated with the smallest singular value of $\mathbf{A}$.

```python
def DLT(P1, P2, point1, point2):
    # Construct the A matrix for AX = 0
    A = [point1[1]*P1[2,:] - P1[1,:],
         P1[0,:] - point1[0]*P1[2,:],
         point2[1]*P2[2,:] - P2[1,:],
         P2[0,:] - point2[0]*P2[2,:]]

    A = np.array(A).reshape((4,4))

    # Solve using SVD
    B = A.transpose() @ A
    U, s, Vh = linalg.svd(B, full_matrices=False)

```

```
    # Normalize homogeneous coordinate
    return Vh[3,0:3] / Vh[3,3]
```

Listing 2: Implementation of DLT Triangulation

Mobile Base Control

PID Speed Control

The Arduino firmware implements a PID controller to synchronize the four independent wheels. Through empirical tuning, the gains were set to $K_p = 1.5, K_i = 0.5, K_d = 0.1$. This ensures the robot maintains a straight heading despite uneven friction or battery voltage sag.

Navigation Logic

The high-level planner on the Raspberry Pi calculates the heading error θ_{err} relative to the target. The control law applies a simple proportional correction:

$$\text{Action} = \begin{cases} \text{Turn Right} & \text{if } \theta_{err} > \text{Tolerance} \\ \text{Turn Left} & \text{if } \theta_{err} < -\text{Tolerance} \\ \text{Move Forward} & \text{otherwise} \end{cases}$$

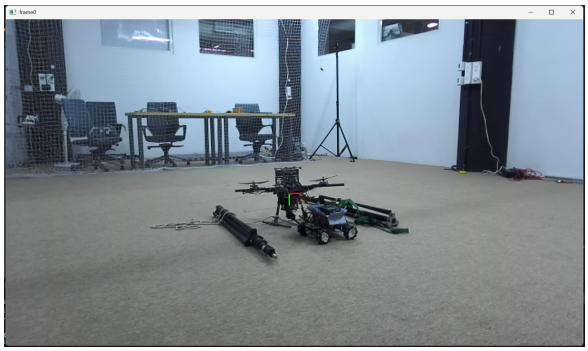
Performance Metrics

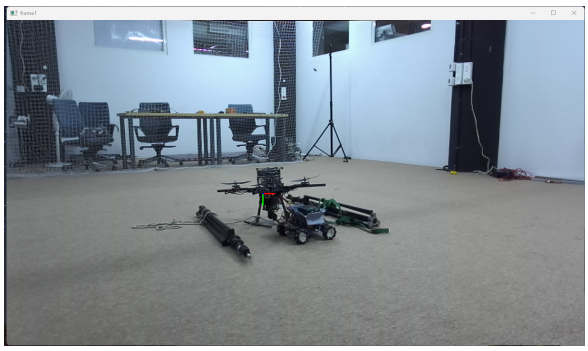
Metric	Result
Left camera calibration RMSE Error	0.18
Right camera calibration RMSE Error	0.19
Stereo camera calibration RMSE Error	0.31

Table 2: System Performance Summary

Result

We have successfully integrated the electromechanical platform with a calibrated stereo vision system. The robot can now move precisely, and the vision system can resolve 3D points in a shared world frame.





```
------------------------------------------------------------
STEP 2: Computing intrinsic parameters
------------------------------------------------------------
Calibrating left camera (camera0)...
rmse: 0.18444135558757915
camera matrix:
 [[1.03108372e+03 0.00000000e+00 6.28678778e+02]
 [0.00000000e+00 1.02907985e+03 3.84798584e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
distortion coeffs: [[-6.68917859e-02  7.37949511e-01  1.61220334e-03 -1.83042505e-03
  -1.82276527e+00]]

Calibrating right camera (camera1)...
rmse: 0.19268193668506486
camera matrix:
 [[1.00839658e+03 0.00000000e+00 6.39900247e+02]
 [0.00000000e+00 1.00631926e+03 3.77434640e+02]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
distortion coeffs: [[ 7.82720423e-02 -2.27853620e+00  6.74155370e-04 -1.53616216e-03
   1.81295997e+01]]

STEP 3: Capturing synchronized stereo pairs
------------------------------------------------------------
STEP 4: Computing stereo calibration (extrinsic parameters)
------------------------------------------------------------
rmse:  0.31278791024409797
```

Figure 3: Calibration Result