Technical Design Paper for Auracle

Amy Kibara, Emmanuel Mututa, Jacque Kyalo, Njuguna Wanjeri

I. Introduction

Robotics Dojo is a project-based robotics training program under the AFRICA-ai-JAPAN Project at the Jomo Kenyatta University of Agriculture and Technology with its primary objective being the encouragement of robotics research in Kenya. The program creates a robotics challenge each year in which students form teams and compete. The challenges are always evolving from previous years in an effort to build robots that can be beneficial in the real world environments such as industries.

This year's challenge was the design and development of an autonomous mobile robot that can collect and offload loads while navigating uneven terrain in a changing environment with dynamic obstacles, and that uses a camera to detect and classify waste for waste management. The robot's performance is judged on how accurately and quickly it can navigate, detect and classify objects, and collect and offload loads.

II. Design Criteria

The robot had to be able to do the following:

- Map its environment and save the map.
- Navigate its environment autonomously using the saved map.
- Carry a load on board and drop it at a designated location.
- Classify various classes of waste using a camera.

These requirements were met by mechanical, electrical and software subsystems working together toward a common goal: the robot frame, the offloading mechanism, the drivetrain and propulsion, the wheels and mounts, the control scheme, the environment detection devices, the power supply, the mapping, localization and navigation system, and the image detection system. Each is described in the sections that follow.

III. Development Process

The robot was developed in the following steps:

- A CAD (Computer-Aided Design) model of the robot frame and of the loading and offloading system was created, keeping the robot within the allowed limit of 300 × 300 × 300 mm.
- Acrylic and PLA were chosen as the fabrication materials. Parts were laser cut or 3D printed and then assembled (Section V).
- A Bill of Materials (BOM) was prepared for purchasing the electrical components and consumables.
- The electrical circuit was designed and soldered onto a perforated board with header pins for the connections, and the electromechanical components were connected to it.
- The mobile platform was tested to confirm that it followed instructions from a written algorithm.
- The sensors were added to the circuit, and the Raspberry Pi, connected to the lidar, was connected to the Arduino Nano over a serial link.
- Image detection and classification was implemented with the Raspberry Pi camera, and its output was used to coordinate the loading and offloading process.
- Mapping and navigation were then developed, tested and improved.

IV. Mechanical Design

A. Robot Frame

The frame is the physical structure that supports all the components, such as the sensors, actuators, electronics and other mechanical elements. It was made from acrylic, which is simple to fabricate on a laser cutter and strong enough to give the robot a rigid core that can sustain collisions without damage and reduce the transmission of vibration that could affect the accuracy of sensor data (Fig. 1).

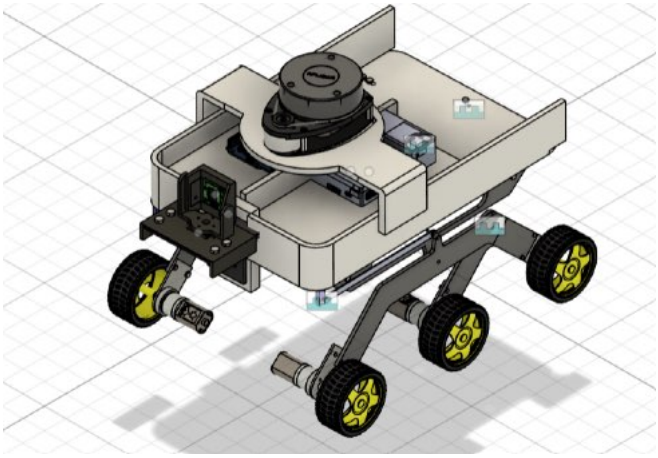


Fig. 1: CAD model of the Auracle robot, showing the frame, lidar, front camera and rocker-bogie wheels

B. *Offloading Mechanism*

The offloading mechanism (Fig. 2) uses a rack and pinion (Fig. 3) to convert the rotary motion of a stepper motor into linear motion. The load carriage is where the load is placed. Once the robot has reached the offloading station, the stepper motor is activated and moves the rack through the pinion, pushing the load carriage forward.

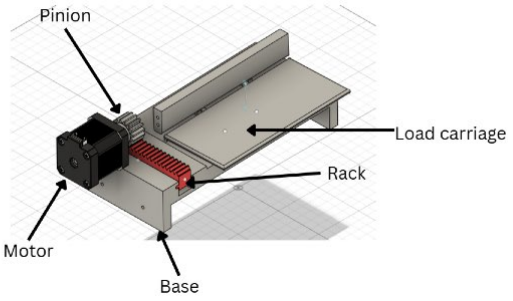


Fig. 2: Offloading mechanism: stepper motor, pinion, rack, load carriage and base

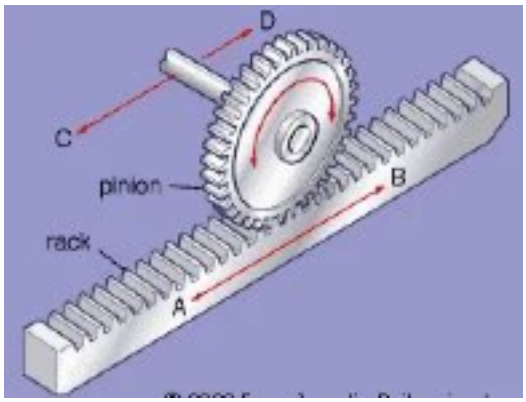


Fig. 3: Principle of the rack and pinion

The load carriage is made of a vertical and a horizontal plate connected by a revolute joint (Fig. 2). Once the carriage has been pushed far enough that the horizontal plate is no longer supported by the base, it rotates downwards and the load slides off.

C. *Drivetrain and Propulsion*

This system had to move the robot accurately and reliably on uneven terrain. It consists of six wheels, four of them driven by motors with rotary encoders that provide odometry data. The motors are powered by the lithium polymer (LiPo) battery through H-bridge motor driver modules. Four motors were chosen so that the energy consumption would not deplete the battery quickly. A rocker-bogie suspension keeps all the wheels in contact with the ground, especially on uneven terrain (Fig. 1).

The rocker-bogie is made up of the chassis, the rockers and the bogies. Each rocker is attached to the chassis by a revolute joint, and each bogie is attached to its rocker in the same way (Fig. 4). The left and right rockers are connected by a differential, which averages out the roll caused by uneven terrain.

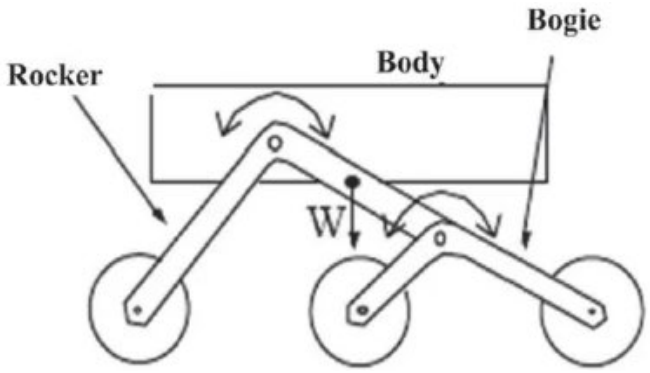


Fig. 4: Rocker-bogie suspension: body, rocker and bogie

D. *Wheels and Mounts*

The wheels are 85 mm in diameter and have spikes for a firm grip on the ground. Their thickness of 36 mm gives a larger ground contact area and improves the stability of the robot. Four motors with a maximum speed of 200 revolutions per minute (RPM) rotate the wheels, which are attached to the motors with hexagonal couplings. The wheel-motor assemblies are mounted on the chassis with mounting brackets.

E. *Motor Selection*

Speed. The robot has to cover its path in the arena in 3 minutes. The arena is 6.7 m long (L) and 4.9 m wide (w), and the path crosses the width twice and the length three and a half times (Fig. 5). The distance is therefore:

$$d = 2w + 3.5L = 2 \times 4.9 + 3.5 \times 6.7 = 33.25\ m$$

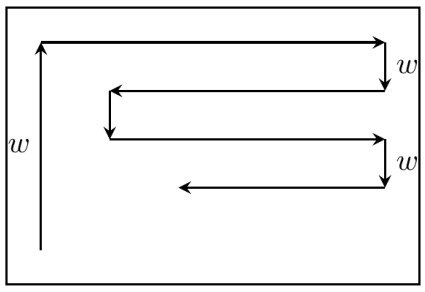


Fig. 5: Path of the robot in the arena, used to estimate the distance

For a time t of 3 minutes (180 s), the required speed is:

$$v = d / t = 33.25 / 180 = 0.185\ m/s = 11.1\ m/min$$

The wheel diameter is 65 mm, so its circumference is $C = \pi d = \pi \times 0.065 = 0.204$ m. The wheel must therefore turn at 11.1 / 0.204 = 54.4 rpm. With a safety factor of 1.5, this becomes 54.4 × 1.5 = 81.6 rpm, which is within the 130 rpm rating of the motor.

Forces. The robot mass is taken as 8 kg, the steepest slope as 7° and the surface as gravel, with a rolling resistance coefficient $C_{rr} = 0.05$. The force along the slope is:

$$F_s = mg \sin\theta = 8 \times 9.81 \times \sin 7° = 9.564\ N$$

The rolling resistance is:

$$F_r = C_{rr}\, mg \cos\theta = 0.05 \times 8 \times 9.81 \times \cos 7° = 3.895\ N$$

The robot must also accelerate. Taking 1 s to reach the cruise speed of 0.185 m/s gives an acceleration a = 0.185 m/s², and the force needed is:

$$F_a = ma = 8 \times 0.185 = 1.480\ N$$

Wheel inertia. The wheels also have to be spun up. Each wheel is modelled as a solid cylinder of diameter 65 mm (r = 32.5 mm) and thickness 36 mm, with a density of 1250 kg/m^3. Its mass is $m_w = \rho\pi r^2 t = 0.149$ kg, and its second moment of inertia about its axis is:

$$I = \tfrac{1}{2}\, m_w\, r^2 = \tfrac{1}{2} \times 0.149 \times 0.0325^2 = 7.89 \times 10^{-5}\ kg{\cdot}m^2$$

The angular acceleration is $\alpha = a / r = 0.185 / 0.0325 = 5.69$ rad/s², so the torque needed to spin up one wheel is $I\alpha = 4.5 \times 10^{-4}$ N·m. For the six wheels it is 2.7×10^{-3} N·m, which is equal to a force at the ground of:

$$F_I = 6I\alpha / r = 0.083\ N$$

Torque. The total force is F = 9.564 + 3.895 + 1.480 + 0.083 = 15.022 N. It is shared by the four driven wheels, so each carries F / 4 = 3.756 N, and the torque per motor is:

$$T = (F / 4)\, r = 3.756 \times 0.0325 = 0.122\ N{\cdot}m = 1.24\ kg{\cdot}cm$$

The 25GA-370 motor is rated 1.6 kg·cm at 130 rpm, so it meets the torque requirement with a margin of about 29%. The wheel inertia adds only 0.083 N to the 15.022 N, because the acceleration is small; the slope and the rolling resistance set the torque.

F. Motor Encoder Design

Problem. The drive motor is a 25GA-370 DC motor. It is rated 1.6 kg, 12 V and 130 RPM, with a 4 mm shaft. This motor has no rear shaft, so it has no encoder. An encoder had to be designed for it. Fig. 6 shows a motor that carries its encoder on the rear shaft.

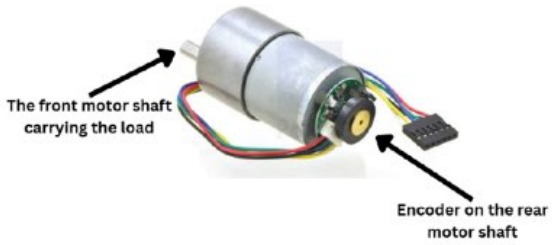


Fig. 6: Motor with an encoder on the rear shaft

Encoder. An encoder is a sensor that measures the rotation of a shaft, from which the distance travelled and the speed of a wheel are calculated. It does this by producing pulses, and its resolution is expressed in pulses per revolution (PPR).

Classification. Encoders are classified in two ways. By sensing mechanism, they are magnetic, optical, mechanical, capacitive or inductive. By output type, they are absolute or incremental. An absolute encoder outputs a unique position value for every location. It knows its exact position as soon as it is powered. An incremental encoder outputs the change in position as pulses on A and B quadrature channels. It does not know its absolute position at power-up. It only counts movement from where it starts. This design covers magnetic and optical incremental encoders.

Design constraints. The encoder had to meet four constraints. Fig. 7 shows the coupler, bracket and wheel rim that set them.

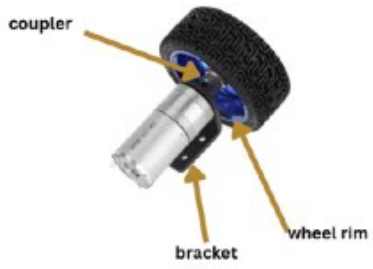


Fig. 7: Coupler, bracket and wheel rim, which set the encoder design constraints

- It must mount on the motor shaft, on the right-hand side of the bracket.
- Its length below the shaft must not exceed the wheel radius. This keeps the ground clearance.
- Its width must be less than the distance from the coupler to the rim. This avoids contact with the rim.
- Its cost must be below Ksh 500.

Resolution. Resolution is the smallest quantity that a measuring instrument can detect. Let p be the pulses per revolution and d the wheel diameter. The wheel circumference is πd. The resolution r is then:

$$r = \pi d / p$$

The circumference is fixed by the wheel. Resolution can therefore only be improved by increasing the PPR.

Magnetic encoder arrangements. Two arrangements are used for magnetic encoders.

Radial arrangement. The sensor sits at the rim of a magnetic disk. The disk has alternating north and south poles around its edge, as shown in Fig. 8.

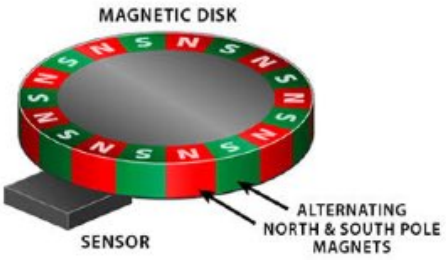


Fig. 8: Radial magnetic encoder arrangement

Axial arrangement. A ring magnet sits in a mounting. Two Hall sensors, Bx and By, sit on a circuit board that faces the magnet, as shown in Fig. 9.

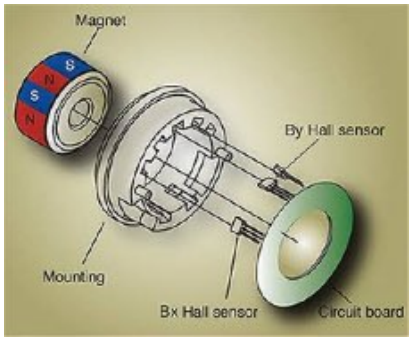


Fig. 9: Axial magnetic encoder arrangement

The axial arrangement was chosen. Small magnets were placed on the flat face of a cylindrical encoder disk, and the Hall-effect sensors were mounted on a holder facing it. A gear train using a 12-tooth and a 48-tooth gear increases the angular velocity (RPM) of the encoder disk, which raises the number of pulses per wheel revolution and so improves the resolution (Fig. 10). The 3D-printed assembly is shown in Fig. 11.

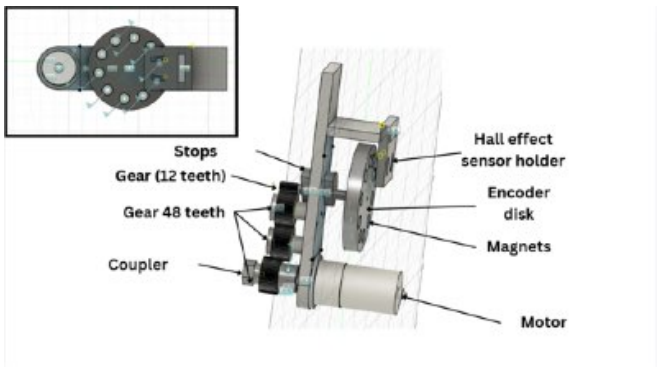


Fig. 10: CAD of the encoder assembly: coupler, gears, stops, magnets, encoder disk, Hall-effect sensor holder and motor

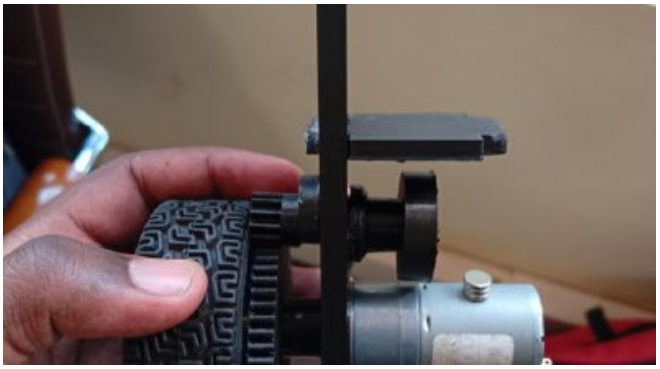


Fig. 11: 3D-printed encoder assembly mounted on the motor

G. Design Challenges

Maintaining the ground clearance and the clearance between the encoder and the wheel rim forced the team to iterate the design and its fabrication repeatedly, at the expense of resolution, moving from 24 pulses per revolution to 8 pulses per revolution. When the 3D printer was damaged, further print iterations became difficult, so the team sought an alternative.

The alternative was to seek for other motors. These motors, however, had their own faults and required a lot of calibration. The motors borrowed were 4 JGB37-520 12V 110RPM Encoder Motor, 12V, 0.45A/1A, 10kg.cm which fit well within our limits. However, 2 of the motors had their encoders damaged, the remaining two were placed in the rear and an offset applied to estimate the odometry of the front wheels as described further in section VII (B) on Calibration.

V. Fabrication

Two fabrication processes were used to produce the mechanical parts of Auracle: laser cutting of acrylic for the frame, and 3D printing of PLA, PETG and ABS for the encoder parts and some other structural components. Both processes work directly from the CAD model, and each design change could be cut or printed again quickly. Table I shows which process was used for each part. For ROS, the geometry of the robot is described in a URDF (Unified Robot Description Format) file; a step-by-step guide to writing one is given in [6].

Table I: Fabrication Process by Part

Part	Process	Material
Robot frame	Laser cutting	Acrylic
Encoder gears, disk, sensor holder and stops	3D printing	PLA
Other structural components	3D printing	PLA
Chassis assembly	Screws and hot glue	n/a

A. Laser Cutting

Laser cutting was chosen for the frame because it is precise, repeatable and non-contact, so the acrylic is not deformed by cutting forces, and because a redesigned part can be cut again within minutes. The machine is a CO_2 laser cutter (Fig. 12). A high voltage excites a mixture of nitrogen,

carbon dioxide and helium gases in a sealed tube. The excited CO_2 molecules emit photons, which are amplified between a fully reflective and a partially reflective mirror, and the beam that leaves through the partially reflective mirror is focused by a convex zinc selenide lens in the laser head. A compressor blows a ring of air around the head, which pushes molten material and fumes away from the cut, and an extractor removes the fumes from the machine.

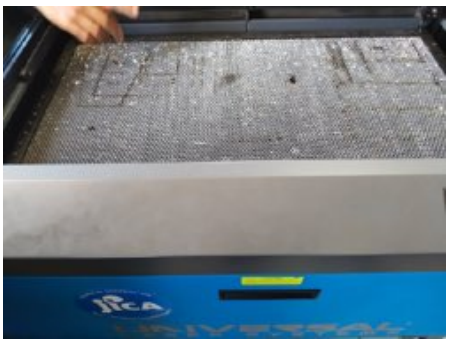


Fig. 12: Bed of the CO_2 laser cutter

Job preparation. A cutting job is prepared as follows:

- The 2D drawing is exported from the CAD software to PDF on a white page and opened in CorelDraw, where the dimensions and units are set.
- Hairline outlines are selected, which reduces wasted material.
- The colour of each line sets the operation that the laser performs (Table II).
- In the Universal Control Panel software, the laser head is positioned with the navigation panel and the cutting boundaries are previewed by moving the head. The material and its thickness are then selected, the settings are applied, and the job is started with the green play button.

Table II: Line Colours Used to Set Up a Laser Job

Colour	Operation
Red	Cutting
Green	Cut first, before the red outline
Blue	Engraving of lines
Black	Engraving of text and images

Operation and safety. The machine is started in a fixed order. The gas valve is first opened briefly to release any moisture in the gas barrel and then closed tightly. The extractor, the compressor and finally the laser are then switched on. Shutdown reverses this order, and the compressor valve is opened to depressurize it and closed again, which prevents moisture from forming and corroding the gas barrel. Materials such as PVC are never cut because they release toxic fumes.



Fig. 13: Sample piece cut from acrylic on the laser cutter

B. 3D Printing

The encoder assembly (gears, encoder disk, Hall-effect sensor holder and stops) and some other structural components were 3D printed in PLA (Fig. 10 and Fig. 11). 3D printing suits these parts because they have three-dimensional shapes, such as gear teeth and sensor holders, that cannot be made from flat sheet, and because a design change only requires the part to be printed again. The encoder had to be redesigned several times to meet the ground-clearance and rim-clearance constraints (Section IV-G). When the 3D printer was damaged, further reprints became difficult and the team had to change approach.

C. Assembly

The laser-cut frame and the 3D-printed parts were assembled with screws, with hot glue used where necessary. The wheels are attached to the motors with hexagonal couplings, and the wheel-motor assemblies are fixed to the chassis with mounting brackets (Section IV-D).

VI. Electrical Design

A. Control Scheme

The control scheme is divided between the mobile platform and the navigation system. An Arduino Nano microcontroller runs the mobile platform because it has enough input and output pins. It sends commands to the motors, reads the encoder and IMU data, and passes the data to the navigation system over a serial connection.

The navigation system needed a processor capable of hosting the Robot Operating System (ROS 2), and the Raspberry Pi 5 was chosen for its sufficient processing speed and relatively low power requirements. It receives data from the Arduino Nano and the lidar, processes them, and sends the appropriate motor commands back to the Arduino Nano.

B. Environment Detection Devices

The sensors used for the robot setup were quadrature encoders, MPU-6050, RPLIDAR A1 M8 and the Pi Camera Module 3.

The encoders were used to track wheel rotation. This information would assist the robot to calculate its estimated position and trajectory.

The MPU-6050 was used to measure the robot's orientation and movement by integrating a gyroscope and an accelerometer. To mitigate the odometry drift the encoder

data and inertial measurement unit (IMU) data were combined using sensor fusion.

The lidar was used to detect objects within the environment of the robot using an infrared (IR) laser scan. This data was used for simultaneous localization and mapping (SLAM) of the robot as well as during navigation to allow the robot to detect the introduction of new obstacles into its environment.

The camera captures images of the waste for classification (Section X-B).

C. Power Supply

A 3S (11.1 V) lithium polymer (LiPo) battery of 2.2 Ah powers the mobile platform and the navigation system. It was chosen for its high energy density and relatively low weight. The power requirement of each electrical device was calculated to determine the battery capacity (Section VI-E). Switches disconnect the battery from the electronics when the robot is not in use, which reduces the risk of accidents caused by short circuits.

D. Electrical Circuit and Component Selection

The circuit and the power flow of the robot are shown in Fig. 14.

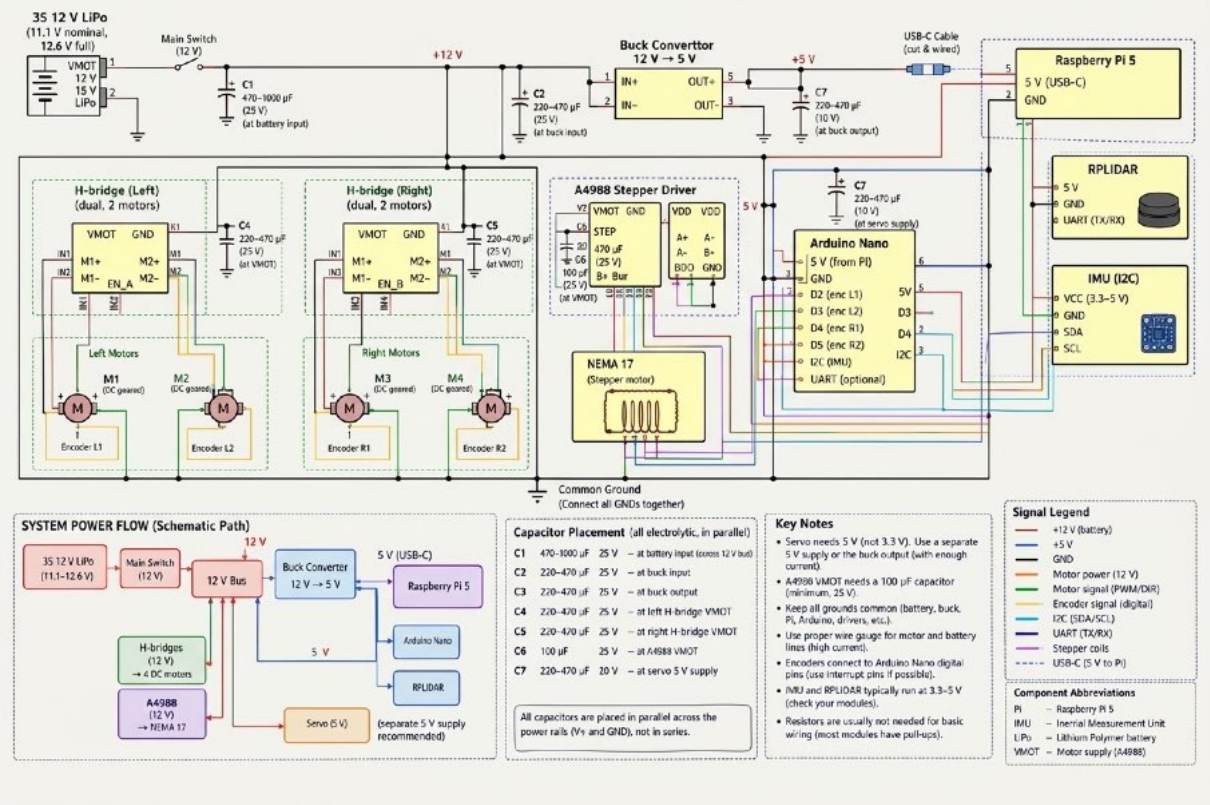


Fig. 14: Electrical circuit and system power flow

1. Power Distribution and Decoupling Topology

- Source & Bus Architecture: The system is powered by a 3S Lithium-Polymer (LiPo) battery (11.1 V nominal, 12.6 V fully charged) routed through a master power switch to establish a primary $+\ 12V$ power bus.
- Voltage Regulation: A step-down DC-DC buck converter regulates the primary $12V$ rail down to a clean $+\ 5V$ logic rail to power microcontrollers and peripheral sensors.
- Transient Suppression: A robust network of parallel electrolytic and ceramic decoupling capacitors (C_1 through C_7, ranging from $100pF$ to $1000\mu F$) is distributed across the power rails and driver VMOT inputs to suppress high-frequency switching noise, inductive back-EMF spikes, and voltage sag during high-current motor transients.

2. Actuation and Drive Subsystems

- Differential/Skid-Steer Propulsion: Four brushed DC geared motors (M_1 through M_4) are driven via dual

H-bridge motor driver modules configured for bidirectional velocity control.

- Odometry Feedback: Integrated quadrature encoders (enc L_1–L_2, R_1–R_2) provide high-resolution pulse trains for closed-loop speed control and dead-reckoning odometry.
- Auxiliary Actuation: An A4988 stepper motor driver interfaces with a NEMA 17 bipolar stepper motor to provide precise positional control for secondary mechanisms.

3. Sensor Interfaces

- An Inertial Measurement Unit (IMU) communicates via an I^2C serial bus (SDA/SCL lines) for orientation and motion tracking.
- An RPLIDAR connects via UART serial communication for environmental scanning and simultaneous localization and mapping (SLAM).

E. Power Budget and Battery Runtime

Battery Energy Capacity

The mobile platform uses a 3S Lithium Polymer (LiPo) battery pack featuring a nominal voltage $V = 11.1\ \text{V}$ and a total charge capacity

$$C = 2200\ \text{mAh}\ (2.2\ \text{Ah}).$$

The total stored energy capacity E is evaluated as:

$$E = V \times C,\ E = 11.1\ \text{V} \times 2.2\ \text{Ah} = 24.42\ \text{Wh}$$

Power Consumption: Actuation Subsystem

The propulsion drivetrain comprises four $12\ \text{V}$ DC geared motors. Operating at a nominal continuous current draw of $I_m = 0.5\ \text{A}$ per motor, the continuous electrical power per actuator P_m is given by:

$$P_m = V_m \times I_m = 12\ \text{V} \times 0.5\ \text{A} = 6.0\ \text{W}$$

For the 4-motor differential arrangement, total propulsion power consumption P_{motors} is:

$$P_{\text{motors}} = 4 \times P_m = 24.0\ \text{W}$$

Power Consumption: Embedded Processing and Perception Hardware

The power draw for the remaining on-board computing and sensing elements is summarized in Table III below:

Table III: Power Consumption Breakdown of Electronics

Component	Operating Voltage (V)	Current Draw (A)	Power (W)
Raspberry Pi 5	5.0	1.60	8.00
Arduino Nano	5.0	0.04	0.20
RPLiDAR A1 M8	5.0	0.50	2.50
AI Camera Module	5.0	0.20	1.00
MPU-6050 IMU	3.3	0.015	0.05

The total electrical power demand P_{Total} is computed as the sum across all functional modules:

$$P_{Total} = P_{motors} + P_{Pi} + P_{Nano} + P_{LiDAR} + P_{Camera} + P_{IMU}$$

$$P_{Total} = 24.0\,W + 8.00\,W + 0.20\,W + 2.50\,W + 1.00\,W + 0.05\,W = 35.75\,W$$

Continuous Operating Runtime

The expected continuous runtime t before depletion is evaluated by:

$$t = \frac{E}{P_{Total}} = \frac{24.42\,Wh}{35.75\,W} = 0.683\,hours\ (\approx 41.0\,minutes)$$

Dynamic Discharge and C-Rating Capability

To ensure stable supply voltage during dynamic transients (motor starting currents, torque spikes on uneven terrain), the continuous discharge rating of the battery was verified using its C-rating ($C_{\text{rating}} = 35$):

$$I_{\max} = C_{\text{rating}} \times \text{Capacity} = 35 \times 2.2\ \text{Ah} = 77.0\ \text{A}$$

Since $I_{\max} = 77.0\ \text{A}$ significantly exceeds the nominal full-load system current ($\approx 3.22\ \text{A}$), the selected battery provides a safe operational margin against voltage drops and thermal stress under transient overload conditions.

VII. Firmware and Control

The computation is split in two. The Arduino Nano runs the real-time layer that controls the wheels, and the Raspberry

Pi 5 runs ROS 2, which handles mapping, navigation and the mission. A PC is used only to display what the robot is doing and to start missions.

A. Firmware Setup

The firmware on the Nano runs a control loop 40 times a second. It talks to the Pi through plain text lines over a USB serial link at 500,000 baud. The Pi sends the wanted left and right wheel speeds, and the Nano replies with the encoder counts and IMU readings at the same rate. If the Nano hears nothing from the Pi for half a second, it stops all the motors.

Each side of the robot shares one pair of direction pins, while every wheel has its own speed signal. All four speed signals are set to the same frequency, about 976 Hz, so the wheels respond alike. When a motor starts, its power is raised in small steps instead of all at once, which avoids the current spike described earlier, and the motors are briefly braked before they reverse. The two rear wheels run in a closed loop.

The firmware first works out the power that a given speed should need, using a straight line fitted to measured data, and a proportional-integral controller then corrects the remaining error using the encoder speed. A short burst of extra power helps each wheel overcome static friction when it starts from rest. The controller also detects faults, such as a wheel turning the wrong way, a wheel at full power that cannot reach its target, and a wheel that does not move at all, which is what an unplugged encoder looks like.

The front wheels have no encoders, so each one copies the power of the rear wheel on its side and adds a fixed offset to cover its extra friction. On the Pi, a ros2_control hardware plugin reads the data from the Nano, turns the encoder counts into wheel positions and sends back the wheel speeds. A differential drive controller then turns the robot's forward and turning speed commands into left and right wheel speeds, and publishes the wheel odometry.

B. Calibration and Tests

Each part of the firmware was calibrated with a test, most of them first with the robot propped up so that the wheels turned freely, and then again on the floor.

Encoder resolution: The number of encoder pulses per wheel turn was found by turning each rear wheel by hand for ten full turns and counting the pulses. This gave 985 pulses per turn on the left and 990 on the right. The first values used were wrong by different amounts on each side, which made the robot curve while its odometry showed a straight line. On the right side, one encoder channel sometimes lost pulses, so that wheel is counted using its other channel.

Front wheel offsets: To find the offsets, each wheel was raised off the ground, its power was slowly increased until it started turning, and then decreased until it stopped. The rear wheels started at about 56 PWM and stopped at about 30. The right front wheel was similar, but the original left front motor needed about 96 to start, which showed that it was damaged, and it needed to be replaced. The offset of each front wheel was then set from these results and tuned on the floor. The robot was driven 0.5 m forward and back, and spun through 360 degrees to the left and to the right, and the offsets were adjusted until it neither veered nor stalled. In the last tuning run the robot's heading changed by just over two degrees at most over 0.5 m, and the spins measured about 360 degrees in both directions.

Speed controller: The power needed for a given speed was measured by running the rear wheels at several fixed speeds in the air and fitting a straight line to the steady results. The two rear motors turned out nearly identical, reaching about 1420 pulses per second at a power of 250, which is about 0.34 m/s with no load. The controller gains were then chosen from two automatic sweeps of 18 step tests each, ranked by accuracy in both directions. In the confirming run the steady error stayed within 1.3% of the target and the ripple within 2.7%.

IMU: A pin test of the Nano showed that three of its six analog pins were faulty, including the hardware I2C clock pin. The IMU was therefore moved to a software I2C bus on two other pins, after which it answered correctly. The chip identified itself as a compatible clone of the MPU-6050. At rest it read 1.06 g for gravity, with a gyroscope drift of 0.004 degrees per second and noise of 0.07 degrees per second. Turning the robot by hand through 360 degrees gave 354.6 degrees to the left and 352.6 degrees to the right, so the signs were correct and the scale was close. The readings stayed valid while both rear motors were running.

Wheel geometry and slip: The lidar was used as a reference to calibrate the wheel geometry. Comparing the turning measured by the wheels with the turning measured by the lidar gave an effective track width of about 0.47 m, and the wheel radius was set to 37.5 mm. A slip test on a grass strip showed that the wheels reported about 0.9 m for a run in which the robot really moved about 0.45 m, so roughly half of the wheel travel was slip. This is the reason laser odometry was added to the mapping system.

VIII. Mapping with Teleoperation

Before the robot can navigate, it needs a map. The map is made by driving the robot by hand while SLAM builds it from the lidar scans.

A. Network and Timing

The network architecture connects a host PC and a Raspberry Pi via ROS 2, utilizing Zenoh middleware for reliable data exchange and Chrony for precise time alignment. To establish communication, the Pi continuously runs a dedicated Zenoh router as a background service. Rather than relying on static configuration files, the setup uses a dynamic initialization script that locates the Pi on the local network and configures the host PC to connect directly to the Pi's router over TCP.

This scripted approach bypasses standard multicast discovery limitations and allows the network configuration to be rapidly deployed across multiple host laptops without permanently modifying their environment variables.

Because the Raspberry Pi has no battery-backed real-time clock fitted, ensuring perfectly synchronized system time is critical for accurate data timestamping across the network. The same script designates the host PC as the master time

server, snapping the PC clock into line first. It then remotely commands the Pi to point its time service to the PC's IP address and forces an immediate time step. This snaps the Pi's clock into exact alignment with the host, ensuring the system loops remain synchronized before any distributed operations begin.

B. Teleoperation

The robot is driven by hand with a keyboard or a game controller. The keyboard runs in a terminal on the Pi, so only the key presses cross the Wi-Fi. The game controller is read on a Windows computer and sent to the Pi as network packets. Both are turned into smooth speed commands with limits on acceleration. The robot moves only while a key is held, and it stops if the signal goes quiet. A command selector gives teleoperation the highest priority, above navigation, so a person can always take over the robot. Speeds are limited to about 0.25 m/s.

C. Mapping

Mapping [1] uses the laser scan from the lidar to build a representation of the robot's environment. It is carried out with SLAM Toolbox, using a parameter file that sets, among other things, the scan range, the map resolution and the maximum loop-closure distance, so that a clear map is generated for the robot to refer to later. An Extended Kalman Filter (EKF) [2]–[4] fuses the IMU data with the encoder data to give an accurate and reliable estimate of the robot's position and velocity. Laser odometry computed from the lidar scans is also added to the filter because the wheels slip on uneven ground. As a result, if the robot skids, the skid is not registered as motion in RViz2. The lidar scan is filtered before use so that the robot's own wheels and frame do not appear as obstacles.

The mapping parameters were tuned for a small arena with narrow corridors: a 3 cm map resolution, a laser range of 0.2 to 6 m, and strict loop closure so that corridors that look alike are not wrongly matched. SLAM runs on the Pi, next to the lidar, so only the finished map is sent over the Wi-Fi to the PC. The map is built by driving slowly and steadily and finishing at the start point, so that the loop closes. The map is visualized in RViz2 and later, if satisfied with its clarity, saved and serialized for use in localization. A small editing tool can remove false walls from a saved map, for example near the start point, without changing the original.

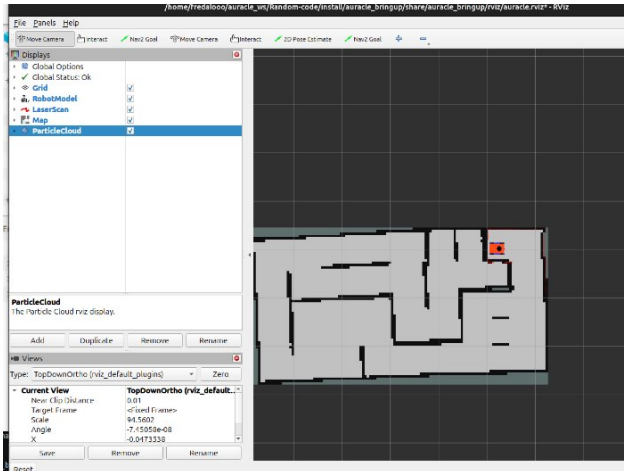


Fig. 15: Arena in Gazebo simulation (left) and its saved map loaded in RViz2 (right)

D. Mapping Runs

Four runs show how the map changed as the odometry changed. The first run was made on game day, with SLAM and RViz both running on the PC. It was the first map made with the new 75 mm wheels. Map drift caused a closed box to be drawn at the start, so Nav2 found no path (Fig. 16).

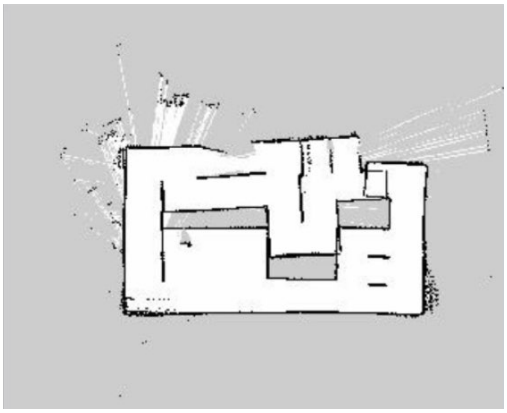


Fig. 16: First run, with SLAM and RViz on the PC. A closed box was drawn at the start

In the second run, SLAM ran on the Pi and RViz on the PC, reading /auracle/map, /auracle/scan, /auracle/slam_toolbox/graph_visualization and TF. This run was changed to compensate for the time offset between the Pi and the PC. The right encoder was counted on channel B only, after wire dropouts in the first attempt. The walls were clean and straight (Fig. 17). Navigation was not tried on this map.

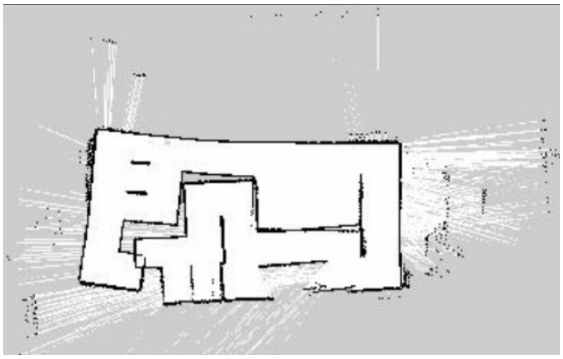


Fig. 17: Second run, with SLAM on the Pi and the right encoder counted on channel B only

The third run used the same setup as the second, with lidar odometry (the rf2o package) added, since the encoder connection is not reliable enough for the motors. It did not work well: the map had doubled walls, a bowed wall and ghost lines (Fig. 18).

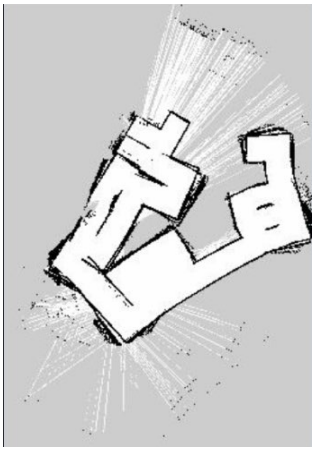


Fig. 18: Third run, with lidar odometry added. The map has doubled walls, a bowed wall and ghost lines

IX. Navigation on the Saved Map

Localization. To localize, the saved map is loaded as a parameter when the localization launch file starts. The map then appears in RViz2, and if the estimated pose does not match the robot's actual location, an initial pose estimate is given. Localization uses AMCL, which compares the lidar scan with the map and keeps only the position guesses that match.

Navigation. After localization, the robot is ready to navigate autonomously through a set of waypoints or to a single goal. The Nav2 stack is used with parameters tuned away from the defaults. Its global and local costmaps assign costs to the space around obstacles, so that the planner can find an obstacle-free path of any length.

A notable implementation is the adoption of the Regulated Pure Pursuit controller, which is much lighter on the Pi's processor than the Model Predictive Path Integral (MPPI) controller used in earlier simulation runs, and which can be given a minimum speed to suit the motor controller. A collision monitor is placed between Nav2 and the motors to stop the robot, using the live laser scan, when an obstacle is too close ahead.

The path is planned with the NavFn planner and followed at about 0.2 m/s, slowing down near obstacles and turning on the spot once when the robot is facing well away from the path. Both costmaps treat the robot as a 34 by 31 cm rectangle and add a safety margin around obstacles, and the local one is rebuilt from the live lidar scan. The final heading at a goal is ignored, because the corridors are too narrow to turn in.

Commands from navigation go through the same command selector as the teleoperation commands, but at a lower priority. In an early test on a saved map, the robot reported reaching 15 goals over 17.7 m with no stalls and two wall contacts, and the safety margins and speeds were tuned further after that.

Latest run: In the latest run, the lidar odometry, the wheels and the IMU were fused. There was still some drift, but navigation worked well on this map, accepting three short goals up to the loading station and one long goal to the offloading station (Fig. 19). The working theory is that the map need not be geometrically exact, as long as the robot sees the same thing while mapping and navigating. This is to be tested on every map.

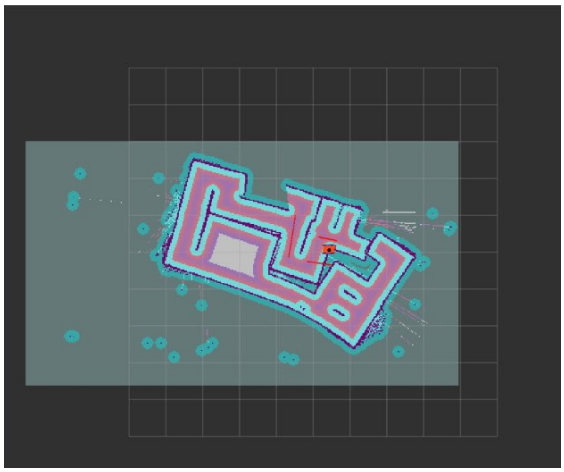


Fig. 19: Latest run, the map Nav2 drove well, with lidar odometry, wheels and IMU fused

Collision monitor: The collision monitor sits between Nav2 and the wheels. It reads the live filtered lidar scan and stops the robot when something enters a short zone ahead or behind. It is independent of the map, so it reacts to obstacles that were not there during mapping. Joystick control bypasses it, so the robot can always be driven out. It is an optional launch parameter while it is tuned. In the trials so far, the first trial handled dynamic obstacles well, while two later runs made contact with walls and objects; the zone size and the corridor clearance are under review. Navigation with moving obstacles on the field is yet to be tested, to see whether the robot re-plans around them and stops in time.

X. Behaviour Trees and Computer Vision

A. Camera Setup and Troubleshooting

The camera is a Raspberry Pi Camera Module V2 (Sony IMX219) on a Raspberry Pi running Ubuntu 24.04.5 LTS and ROS 2 Jazzy, using libcamera with the Raspberry Pi PiSP pipeline. Its images are to be streamed over Zenoh to an Ubuntu PC for viewing.

On the robot's Raspberry Pi (auracle) the camera initially did not work: the rpicam-hello command failed with an undefined libcamera symbol error. Inspecting the loaded libraries with ldd showed a version mismatch. The camera application was loading a manually installed libcamera 0.7.2 from /usr/local, while the Ubuntu package provides version 0.7.1. The system libraries were deliberately left unchanged, because replacing them could affect other software such as ROS 2 and the Hailo components. Instead, a second Raspberry Pi with a working camera (smartai) was used as a reference. Its libcamera source commit was identified and the same commit was built on auracle in an isolated directory, with three environment variables pointing to it. With this build, the cam utility detected the IMX219 sensor and captured a test frame of 800 × 600 pixels. The variables apply only to the terminal session in which they are set, and the packaged rpicam-hello remains incompatible with the new build. Table IV summarizes the problems and how each was handled.

Table IV: Camera Problems and Resolutions

Problem	Investigation	Resolution
`rpicam-hello` undefined symbol error	Checked library paths with `ldd`	Found an incompatible application and library pair
Multiple libcamera versions	Inspected `/usr/local` and `/usr/lib`	Avoided replacing system libraries
Unsure of the correct libcamera version	Examined the working Pi, smartai	Reproduced its exact source commit
Detection failed through `rpicam-hello`	Built libcamera 0.7.2 separately	Used the matching `cam` application
Camera hardware needed checking	Ran `cam -l`	IMX219 detected
Image capture needed checking	Ran `cam --capture=1`	Captured a 1.4 MB PPM image

The next step is to integrate the working camera with ROS 2 Jazzy. The planned data flow is IMX219 camera, libcamera 0.7.2, ROS 2 camera node, /camera/image_raw, Zenoh transport, Ubuntu PC, RViz2. The first task is to confirm that the camera node publishes images locally on auracle, and remote streaming is configured after that.

B. Image Classification

The camera captures images of the waste, which is displayed on a screen, and a machine learning model classifies it. An AI HAT on the Raspberry Pi runs the model to reduce the computational load on the Pi. The model is a YOLO-based detector in ONNX format that publishes the class and confidence of each object it finds. The possible categories are "organic", "plastic" and "metallic", and detections below 50% confidence are ignored.

The model was tested live with the camera (Fig. 20). Detections are published on ROS 2 at about 10–15 FPS. Each detection message carries the class and the score; in the example in Fig. 21, the class is "plastic" with a score of 0.62, for the plastic bottle in the matching camera frame (Fig. 22).

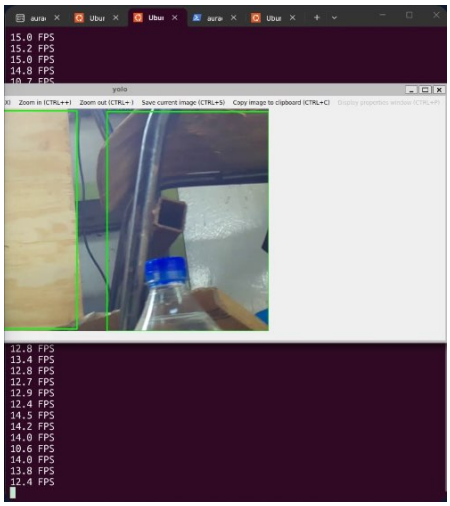


Fig. 20: Live camera test. Green boxes mark detections in the live camera window

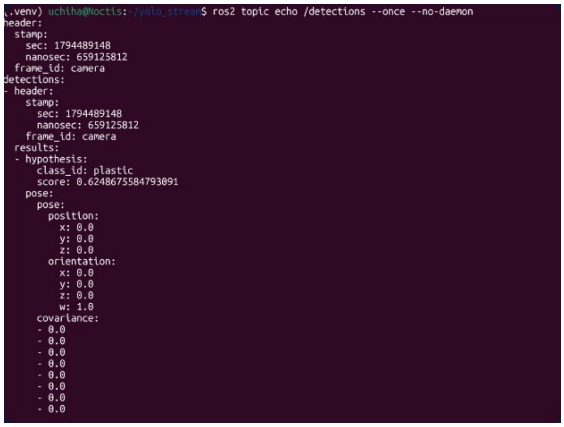


Fig. 21: Detection message on /detections: class "plastic", score 0.62

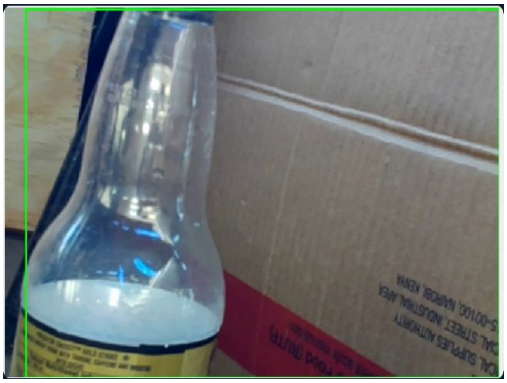


Fig. 22: The matching camera frame: a plastic bottle with its bounding box

The accuracy of the YOLO26n model on the Hailo-8L was tested by comparing the ONNX reference with the float and INT8 compiled graphs (HAR). The test used 30 labelled images with 52 objects of 3 classes, and a hit needed a confidence of 0.25 and an IoU of 0.50 (Table V). In the table, TP is the number of true positives, FN the number of missed objects and FP the number of false positives. The float HAR matched the ONNX recall. The unoptimised INT8 graph traded recall for precision. The model should therefore be optimised with a calibration set before the compiled model is relied on.

Table V: Hailo-8L Accuracy Test of YOLO26n

Variant	TP	FN	FP	Recall	Precision
ONNX	37	15	21	0.71	0.64
HAR native	37	15	15	0.71	0.71
HAR INT8	29	23	4	0.56	0.88

C. Mission Behaviour Tree

The behaviour tree package is a small mission runner that sits on top of Nav2. It is built with the BehaviorTree.CPP library. It sends goals to Nav2 and awaits a success response from it. A behaviour tree is a good fit for this job because a mission is a list of steps with rules about what to do when a step fails, and a tree makes those rules easy to read and change. The runner waits for Nav2 to be ready and tells it the robot's starting pose. It then detects an object using the computer vision model, which decides the region to serve, drives to the loading area of that region, pauses for three seconds to stand in for loading, and drives to the matching offload area. The loading and offloading areas are marked once on the saved map by clicking on their centres in RViz2, and the runner stores them as coordinates in the map frame. These coordinates only belong to that map, so they are marked again for a new one. If one trip fails, the runner does not stop. It goes back to waiting for the next detected object.

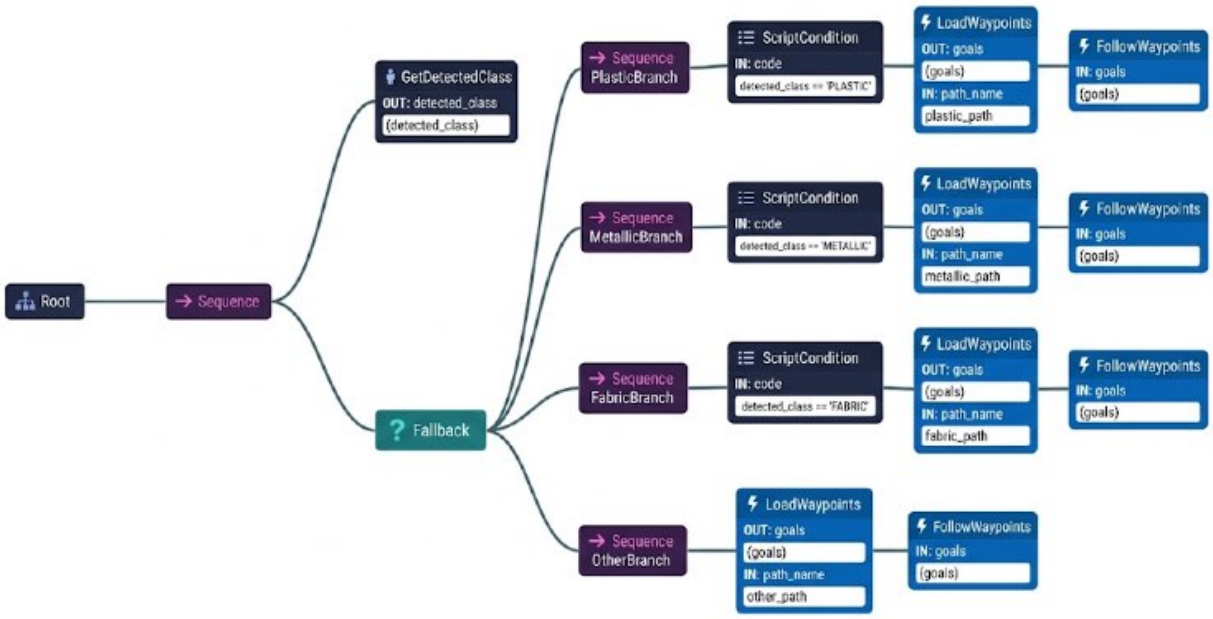


Fig. 23: Mission behaviour tree

The mission behaviour tree was run over predetermined routes 1 to 3. Each route pairs one loading region with one offloading region, marked once on the saved map. The robot drives to loading region N, pauses to load, then drives to offloading region N. Obstacle and collision monitoring was erratic, and the controller and recovery limits are being tuned. The test code launches the tree with a region as a parameter. The robot passes through that region, then docks on the next

point to stop (Fig. 24). The coordinates are to be compared with those measured physically on the game field, or with a Nav2 pose estimate set before the tree starts moving.

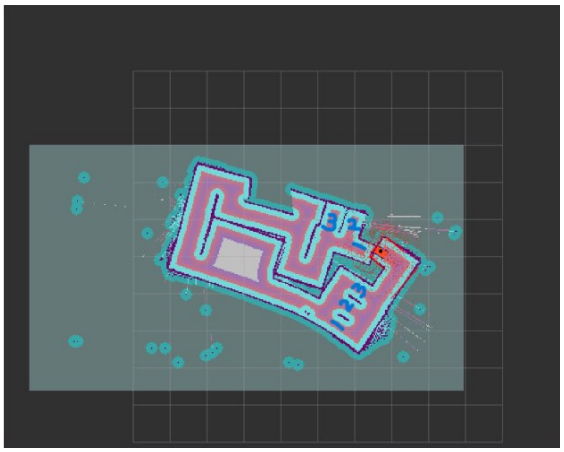


Fig. 24: Behaviour tree test on the saved map, with numbered regions

Control Flow & Execution Logic

Root Sequence (Sequence)

Serves as the primary orchestrator, ticking its children sequentially from top to bottom.

First invokes GetDetectedClass to acquire the target classification; only upon successful retrieval does execution proceed to the routing Fallback selector.

Perception Acquisition (GetDetectedClass)

Role: Leaf Action node interfacing with the perception subsystem.

Blackboard Output: Writes the string identifier of the classified object to the {detected_class} blackboard key.

Material Routing Selector (Fallback)

Evaluates four prioritized branch sequences (PlasticBranch, MetallicBranch, FabricBranch, and OtherBranch) until one succeeds:

Table VI: Behaviour Tree Branch Logic

Branch Sequence	Guard Condition (ScriptCondition)	Path Identifier (path_name)	Execution Pipeline
PlasticBranch	detected_class == 'PLASTIC'	plastic_path	Evaluates guard → loads plastic bin trajectory → navigates waypoints
MetallicBranch	detected_class == 'METALLIC'	metallic_path	Evaluates guard → loads metallic bin trajectory → navigates waypoints

FabricBranch	detected_class == 'FABRIC'	fabric_path	Evaluates guard → loads fabric bin trajectory → navigates waypoints
OtherBranch	*None (Default Catch-All)*	other_path	Directly loads fallback trajectory → navigates waypoints

Blackboard Data Flow & Reusable Action Primitives

Rather than hardcoding separate navigation behaviors for each material class, the tree achieves modularity through parameterized action nodes and dynamic blackboard remapping:

- ScriptCondition: Evaluates inline boolean expressions against {detected_class}. If a condition evaluates to false, the parent Sequence immediately halts and returns FAILURE, prompting the Fallback node to evaluate the next branch.
- LoadWaypoints: Accepts a static string input (path_name) corresponding to a pre-defined trajectory profile and populates the {goals} blackboard entry with the resulting pose array.
- FollowWaypoints: Consumes the {goals} pose array from the blackboard and dispatches the trajectory to the navigation controller, returning RUNNING during transit and SUCCESS upon reaching the target drop-off zone.
- Fail-Safe Default Handling (OtherBranch): Because OtherBranch omits a ScriptCondition guard, it acts as a guaranteed fallback whenever {detected_class} contains an unrecognized, noisy, or uncategorized label—ensuring the robot always clears the sorting queue safely.

XI. Conclusion

Auracle is a four-motor rover on a rocker-bogie suspension that can map an arena, localize on the saved map, navigate through it, and use a camera to decide where to take a load. Laser-cut acrylic and 3D-printed PLA parts allowed quick design iterations, although the loss of the 3D printer slowed the encoder development. The mechanical, electrical and computational parts were developed together, and each was calibrated with a simple test before the next part was built on it. Calibrating the encoders, the front wheel offsets and the IMU removed the curving and stalling seen at first, and fusing laser odometry with the wheel and IMU data gave usable maps even where the wheels slip. Some problems remain. The right rear encoder still drops pulses, three analog pins on the Nano are damaged, and the Pi's power supply was not fully solved, so replacing the Nano and making the Pi's power path more reliable are the next hardware steps. On the software side, the next step is to run the whole mission, from detection to offloading, on the robot as one continuous test.

The work that remains is to connect the detected class to the behaviour tree, so that a detection selects the loading and offloading route; to tune detection for speed against reliability, by optimising the model with a calibration set, re-scoring recall and precision and then setting the confidence threshold; to assemble the offloading mechanism, mount it on the robot and trigger it at the offload step of the tree; and to test navigation with moving obstacles on the field.

References

[1] E. Renard, ROS 2 from Scratch: Get Started with ROS 2 and Create Robotics Applications with Python and C++. Birmingham, U.K.: Packt Publishing, 2024, pp. 343–344.

[2] Ribeiro, M. I. I. (2004). Kalman and extended Kalman filters: Concept, derivation and properties. Institute for Systems and Robotics, 43(46), 3736-3741.

[3] Automatic Addison, Extended Kalman Filter with Python, 2025, Online https://automaticaddison.com/extended-kalman-filter-ekf-wth-python-code-example/.

[4] Tom Moore, Robot Localization Wiki, Online Article, 2024, https://docs.ros.org/en/melodic/api/robot_localization/html/index.html

[5] Raguraman, P., Meghana, A., Navya, Y., Karishma, S. K., & Iswarya, S. (2021). Color detection of RGB images using Python and OpenCv. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 7(1), 109-112.

[6] F. Wanjeri, "URDF for Dummies," Robotics Dojo, Sep. 14, 2026. [Online]. Available: https://roboticsdojo.substack.com/p/urdf-for-dummies