Technical Design Paper for the Robotics Dojo Competition 2026

Team Limit Breakers

Oct 26, 2026

Team Members:
Brumley Ogari – Team Leader
Junior Iregi
Ndirangu James
Gideon Mosomi
Victor Muriithi

I. Introduction

Robotics Dojo is a project-based robotics training programme under the AFRICA-ai-JAPAN Project at the Jomo Kenyatta University of Agriculture and Technology (JKUAT). Its main objective is to encourage robotics research in Kenya. Each year the programme sets a robotics challenge in which student teams design, build and compete with their own robots, and the challenges evolve so that the resulting robots stay relevant to real-world settings such as industry.

The global waste crisis (driven by massive volumes of municipal solid waste and fast-growing electronic waste) overwhelms traditional manual sorting and recycling capacity, with global e-waste climbing past 62 million tonnes annually while a fraction is recycled. Robotics, paired with artificial intelligence (AI) and computer vision, provides scalable, high-speed solutions across waste management streams. This year's challenge to beat is built to develop this solution. The robot is an autonomous mobile device that navigates a game field, identifies and collects waste items and offloads them at a drop zone. It must use a camera and an onboard-trained AI model to classify waste into three classes: plastic, metal and organic. Performance is judged on how accurately and how fast the robot performs navigation, waste classification, collection and offloading. This builds on previous years, whose challenges centred on autonomous navigation and last year, potato-leaf disease classification.

This paper describes the design and development of the robot built by Team Limit Breakers. It covers the mechanical, electrical and software systems, the mathematical models used to size and control them and the results of tests in simulation and on the real robot.

II. Design Strategy

The robot had to do the following:

- Map the game-field and save the map.
- Navigate autonomously on the saved map, including climbing the ramp to the raised zone.
- Classify waste as plastic, metal or organic using a camera and a trained model.
- Carry items on board and drop them at a designated location.

These requirements are met by mechanical, electrical and software systems working toward one goal. The robot is split between two computers: a low-level controller that drives the motors and reads the sensors, and a navigation and AI stack that decides where to go and what it sees. The subsystems are:

Subsystem	Main components	Role
Robot frame	Laser-cut acrylic layers	Rigid structure for all parts
Offloading mechanism	TT gear motor driving a rack and pinion sliding door (L293D driver)	Releases the load at the drop zone
Drivetrain	4×12 V 70 rpm geared DC motors with encoders, $2 \times$ L298N drivers	Four-wheel-drive propulsion on rough ground and ramp
Power supply	$4 \times$ 18650 cells, buck converter	Motor and logic power
Control scheme	Arduino Uno (ROS-Arduino-Bridge firmware), Raspberry Pi 4, laptop	Low-level control on the Uno, ROS 2 on the Pi and laptop
Environment detection	Quadrature encoders, MPU-6050 IMU, Slamtec RPLIDAR, camera	Odometry, orientation, obstacles, waste images
Mapping, localisation, navigation	EKF, slam_toolbox, AMCL, Nav2	Map, pose estimate, path planning and control
Image detection	Trained 3-class model on the laptop	Classifies plastic, metal, organic

III. Vehicle Design

The vehicle design is described in three parts: the mobile platform, the navigation system and the computer vision system. Symbols are defined where first used; values come from the CAD model and from measurements on the robot, and any value that is assumed is stated as such.

1. Mobile Platform

The mobile platform consists of the frame, the four-wheel-drive drivetrain, the power system, the low-level controller and the offloading mechanism.

Robot frame

The frame is built from laser-cut acrylic layers, which are light, dimensionally stable, easy to fabricate and transparent, so wiring faults can be seen inside the body. The robot measures about 0.30 m long by 0.30 m wide by 0.27 m high lidar included and has a mass of 3.3 kg. The heaviest parts (battery pack and motor drivers) sit on the lowest layer to keep the centre of gravity low. This matters because the gamefield includes a ramp. With track width t, centre-of-gravity height h_{cg}, and l_r the horizontal distance from the centre of gravity to the rear axle, the robot tips sideways or backwards when the slope angle reaches:

$$\theta_{\mathrm{lat}} = \arctan\left(\frac{t}{2\,h_{\mathrm{cg}}}\right) \tag{1}$$

$$\theta_{\mathrm{long}} = \arctan\left(\frac{l_r}{h_{\mathrm{cg}}}\right) \tag{2}$$

The design requires the ramp angle to stay well below both values, and below the traction limit $\tan\theta \le \mu_s$, where μ_s is the static friction coefficient between wheel and surface. For our robot $t = 0.26\,\mathrm{m}$ (wheel centre to wheel centre), $l_r = 0.10\,\mathrm{m}$, and the centre of gravity height is estimated from the CAD layout at $h_{\mathrm{cg}} = 0.10\,\mathrm{m}$ (assumed, to be checked by a tilt-table test). This gives $\theta_{\mathrm{lat}} = 52^\circ$ and $\theta_{\mathrm{long}} = 45^\circ$, both well above the 30° ramp, and the traction condition needs $\mu_s \ge \tan 30^\circ = 0.58$, which the spiked rubber tyres are expected to meet. The ramp is therefore climbed slowly and straight on.

Figure 1 shows the tagged 3D model of the robot and Figure 2 shows the assembled robot.

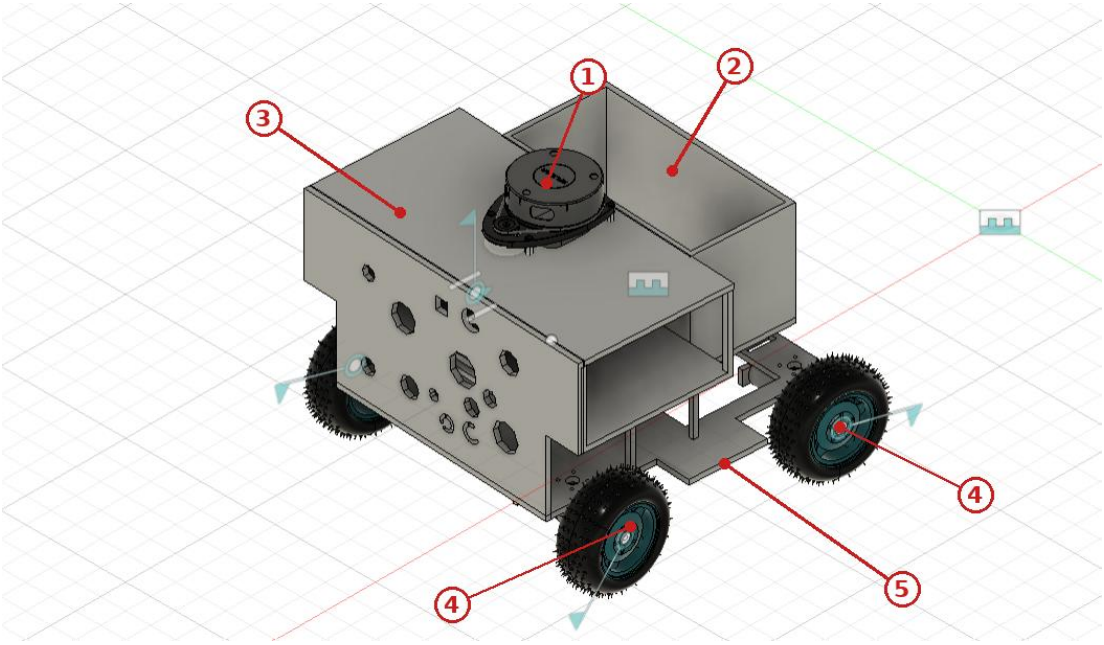


Figure 1: *Tagged 3D robot model. 1: RPLIDAR. 2: offloading bin. 3: acrylic body holding the electronics. 4: spiked wheels (four driven wheels). 5: lower chassis plate.*



Figure 2: *The assembled robot, built from laser-cut acrylic layers on four spiked wheels.*

Drive kinematics

The robot is a four-wheel skid-steer vehicle: the two wheels on each side are driven together and the robot turns by running the left and right sides at different speeds. Because the wheels slide sideways during a turn, the effective track width is larger than the geometric one. We write it as χb, where b is the measured wheel-to-wheel distance and $\chi \geq 1$ is found experimentally from a rotation test. With left and right side speeds v_L and v_R:

$$v = \frac{v_R + v_L}{2} \tag{3}$$

$$\omega = \frac{v_R - v_L}{\chi\, b} \tag{4}$$

$$v_L = v - \frac{\omega\, \chi\, b}{2} \tag{5}$$

$$v_R = v + \frac{\omega\, \chi\, b}{2} \tag{6}$$

The inverse relations are used by the ROS 2 differential-drive controller to convert velocity commands (v, ω) from Nav2 into wheel targets. Each motor has a quadrature encoder. If a wheel of radius r produces N_{rev} encoder ticks per revolution (encoder pulses per motor revolution $\times$ decoding factor $\times$ gearbox ratio), the distance per tick and the wheel speed measured over a control period Δt are:

$$\Delta s = \frac{2\pi r}{N_{\text{rev}}} \tag{7}$$

$$v_{L,R} = \frac{\Delta s\, n_{L,R}}{\Delta t} \tag{8}$$

where n_L and n_R are the ticks counted in Δt. Our controller runs at 30 Hz, so $\Delta t = 1/30\,\text{s}$. Odometry is taken from the front-left and front-right encoders. Our front motors give $N_{\text{rev}} = 2290$ ticks per wheel revolution and the wheel diameter is 98 mm ($r = 0.049\,\text{m}$), so

$$\Delta s = \frac{2\pi \times 0.049}{2290} = 0.134\,\text{mm per tick.}$$

At the 70 rpm no-load speed the top speed is

$$v_{\max} = 2\pi r \cdot \frac{70}{60} = 0.36\,\text{m/s},$$

which is the speed limit used in navigation.

Motor sizing

The drivetrain uses four 12 V, 70 rpm geared DC motors with encoders. To check that they are sufficient, we size them for the worst case, climbing the ramp. Let m be the robot mass, r the wheel radius, $N = 4$ the number of driven wheels, θ the ramp angle, C_{rr} the rolling resistance coefficient, a the target acceleration and η the drivetrain efficiency. The traction force and the torque each motor must supply are:

$$F_{\text{req}} = m\,(a + g\sin\theta + C_{rr}\, g\cos\theta) \tag{9}$$

$$T_{\text{motor}} = \frac{F_{\text{req}}\, r}{N\,\eta} \tag{10}$$

For a target speed v the required wheel speed is:

$$n_{\text{req}} = \frac{60\,v}{2\pi r}\ \text{rpm} \tag{11}$$

A brushed DC motor has an approximately linear torque-speed curve between its no-load speed ω_0 (70 rpm at 12 V) and its stall torque T_s. The motor can do the job only if the required operating point lies under that line, and the current drawn follows from the torque constant k_t and the no-load current I_0:

$$\omega_{\text{req}} \le \omega_0 \left(1 - \frac{T_{\text{motor}}}{T_s}\right) \tag{12}$$

$$I = \frac{T_{\text{motor}}}{k_t} + I_0 \tag{13}$$

Finally, the wheels must not slip on the ramp, which requires

$$F_{\text{req}} \le \mu_s\, m\, g \cos\theta. \tag{14}$$

Our values are $m = 3.3\,\text{kg}$, $r = 0.049\,\text{m}$ (98 mm wheels), $N = 4$ and $\theta_{\text{ramp}} = 30^\circ$, with an assumed $C_{rr} = 0.03$, $a = 0.1\,\text{m/s}^2$ and $\eta = 0.8$. This gives

$$F_{\text{req}} = 3.3\,(0.1 + 9.81\sin 30^\circ + 0.03 \times 9.81\cos 30^\circ) = 17.4\,\text{N},$$
$$T_{\text{motor}} = \frac{17.4 \times 0.049}{4 \times 0.8} = 0.27\,\text{N·m}\ (2.7\,\text{kg·cm})\ \text{per motor.}$$

Crossing the ramp at $v = 0.25\,\text{m/s}$ needs

$$n_{\text{req}} = \frac{60 \times 0.25}{2\pi \times 0.049} = 49\,\text{rpm},$$

which is below the 70 rpm no-load speed. For the operating point to lie under the torque-speed line, the motor stall torque must satisfy

$$T_s \ge \frac{T_{\text{motor}}}{1 - n_{\text{req}}/70} = \frac{0.27}{0.30} = 0.89\,\text{N·m},$$

so the motors must have a stall torque of at least about 0.9 N·m (9 kg·cm); the high-ratio 70 rpm gearbox was chosen to supply torque of this order at low speed, and the supplier's stall-torque figure should be checked against this limit. The wheels do not slip on the ramp if

$$\mu_s \ge \frac{F_{\text{req}}}{m\,g\cos\theta} = 0.62.$$

Four driven wheels share the load, so each motor works well below its stall torque on the ramp, and the wheels keep traction on loose or uneven surfaces.

Turning in place

Skid-steer robots are much harder to turn on the spot than to drive straight, because all four contact patches must slide sideways. For a robot of length L and width b with weight shared equally by the four wheels, each wheel sits at distance d from the robot centre, and the friction moment the motors must overcome is:

$$d = \tfrac{1}{2}\sqrt{L^2 + b^2} \tag{15}$$

$$M_{\text{turn}} = \mu_k\, m\, g\, d \tag{16}$$

The motors create this moment through opposing traction forces F_w at the four wheels, which sit $b/2$ either side of the centreline. Balancing moments gives the traction per wheel and the corresponding motor torque:

$$4\,F_w\,\frac{b}{2} = M_{\text{turn}} \;\Rightarrow\; F_w = \frac{\mu_k\, m\, g\, d}{2\,b} \tag{17}$$

$$T_{\text{spin}} = \frac{F_w\, r}{\eta} \tag{18}$$

This is an approximation (it ignores the load shifting between wheels), but it shows why turning in place demands far more torque than straight driving. For our robot ($L = b = 0.30\,\text{m}$, $m = 3.3\,\text{kg}$ and an assumed $\mu_k = 0.6$) $d = 0.21\,\text{m}$, $M_{\text{turn}} = 4.1\,\text{N·m}$ and each motor needs $T_{\text{spin}} = 0.42\,\text{N·m}$ (4.3 kg·cm), about 1.6 times the torque needed to climb the 30° ramp. Spinning in place is therefore the worst case for the drivetrain. It matches what we measured: on the floor, in-place rotation was reliable only at a duty cycle near the maximum (PWM 255), while intermediate duty cycles (about 150 to 210) gave erratic results with a wheel sometimes failing to turn. We therefore use a dedicated spin command that bypasses the PID loop and applies a fixed duty cycle for in-place rotation (SPIN_PWM = 200 out of 255, about 78% duty).

Power system

The motors are powered by a pack of four 18650 lithium cells connected in series ($n_s = 4$, $n_p = 1$), which gives 14.8 V nominal and 16.8 V fully charged, and the logic side (Raspberry Pi 4 and lidar) by a separate power bank over USB-C. For n_s cells in series and n_p strings in parallel, the pack voltage, capacity and stored energy are:

$$V_{\text{bat}} = n_s\, V_{\text{cell}} \tag{19}$$

$$C_{\text{bat}} = n_p\, C_{\text{cell}} \tag{20}$$

$$E = V_{\text{bat}}\, C_{\text{bat}} \tag{21}$$

The supply was sized from the sum of the loads, and the expected runtime follows from the average power, the usable fraction of the pack (depth of discharge, DoD) and the efficiency of the converters η_d:

$$P_{\text{peak}} = \sum_i V_i\, I_i \tag{22}$$

$$t_{\text{run}} = \frac{\eta_d\, \text{DoD}\; E}{P_{\text{avg}}} \tag{23}$$

The L298N drivers are bipolar H-bridges and drop roughly 2 V across the bridge (see datasheet), so the voltage reaching a motor at duty cycle u (0 to 255) is lower than the battery voltage, and the drop is dissipated as heat in the driver:

$$V_m = \frac{u}{255}\, V_{\text{bat}} - V_{\text{drop}} \tag{24}$$

$$P_{\text{loss}} = V_{\text{drop}}\, I_m \tag{25}$$

With our 14.8 V pack the motors see about $14.8 - 2 = 12.8\,\text{V}$ at full duty, which matches their 12 V rating, and only about $0.78 \times 14.8 - 2 = 9.6\,\text{V}$ at a duty of 200/255. This is one reason the motors reach their full speed only near the top of the duty-cycle range. A buck converter supplies 5 V to the Arduino Uno and the sensors. Switches isolate the battery from the electronics when the robot is not in use.

Low-level controller

An Arduino Uno running the ROSArduinoBridge firmware handles the real-time tasks: it drives the two L298N motor drivers and the offloader driver (L293D), reads the four encoders and the MPU-6050 IMU over I2C, and talks to the Raspberry Pi over USB serial at 57600 baud. The speed loop runs at 30 Hz and the motors stop automatically if no command arrives for 1000 ms. Serial throughput is limited to about baud / 10 bytes per second (8N1 framing), so each 33 ms cycle can carry at most roughly 190 bytes:

$$B_{\max} = \frac{\text{baud}}{10} = \frac{57600}{10} = 5760\,\text{bytes/s} \tag{26}$$

$$\text{bytes per cycle} = \frac{B_{\max}}{f_{\text{loop}}} = \frac{5760}{30} \approx 192\ \text{bytes} \tag{27}$$

Figure 3 shows the base plate that carries the Arduino Uno, the two L298N motor drivers and the MPU-6050 IMU.

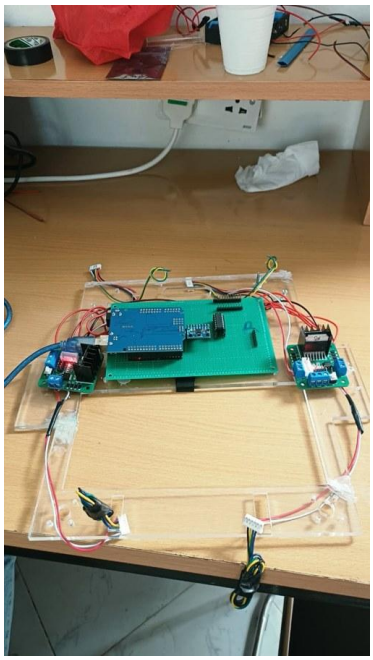


Figure 3: *Base plate with the Arduino Uno on its perforated board, the MPU-6050 IMU and the two L298N motor drivers.*

Closed-loop speed control

Each wheel speed is controlled with a discrete PID loop. The target is expressed in encoder ticks per control frame, and the measured value n_k is the number of ticks counted in the last frame. At the 70 rpm no-load speed the largest value n can take is $n_{\max} = 2290 \times (70/60)/30 = 89$ ticks per frame:

$$n_{\max} = N_{\text{rev}} \cdot \frac{70}{60} \cdot \frac{1}{30}\ \text{ticks per frame} \tag{28}$$

With target n_k^*, the error, its accumulated sum and the output PWM command are:

$$e_k = n_k^* - n_k \tag{29}$$

$$I_k = I_{k-1} + e_k \tag{30}$$

$$u_k = \frac{K_p\, e_k + K_i\, I_k + K_d\,(e_k - e_{k-1})}{K_o} \tag{31}$$

The gains are $K_p = 60$, $K_d = 12$, $K_i = 0$ and $K_o = 50$ (the final tuned values), so the effective proportional gain is $K_p/K_o = 1.2$ PWM counts per tick of error. A limit on how fast u_k may change gives a smooth acceleration ramp and a smooth stop, which protects the gearboxes and reduces wheel slip. Modelling the motor as a first-order system with gain K_m (ticks per frame per PWM count) and time constant τ_m, proportional control alone leaves a steady-state error:

$$e_{ss} = \frac{n^*}{1 + K_m K_p / K_o} \tag{32}$$

The integral term removes this error but can cause overshoot and instability if it is large, so it is kept small or zero, and the derivative term damps overshoot. The gains were tuned by test runs on the bench (wheels in the air) and then on the floor, as reported in Section IV.

Offloading mechanism

The load is released through a sliding door driven by a rack and pinion. A TT gear motor, through its own L293D driver, turns a pinion of pitch radius r_p. The pinion meshes with a rack fixed to the door, so the rotation of the motor becomes a straight movement of the door, and a single actuator both opens and closes it. Because the door slides in its guides, the motor only has to overcome friction and the weight of the load it carries. For a door of mass m_d carrying a load m_L, a guide friction coefficient μ_d, a gear efficiency η_g, a door stroke s and a pinion speed ω_p, the door motion, the motor torque and the opening time are:

$$x = r_p\, \theta_p \tag{33}$$

$$v = r_p\, \omega_p \tag{34}$$

$$t_{\text{open}} = \frac{s}{r_p\, \omega_p} \tag{35}$$

$$F_{\text{door}} = \mu_d\,(m_d + m_L)\, g \tag{36}$$

$$\tau_{\text{off}} = \frac{F_{\text{door}}\, r_p}{\eta_g} \tag{37}$$

$$T_{\text{rated}} \geq \text{SF} \cdot \tau_{\text{off}}, \qquad \text{SF} \geq 1.5 \tag{38}$$

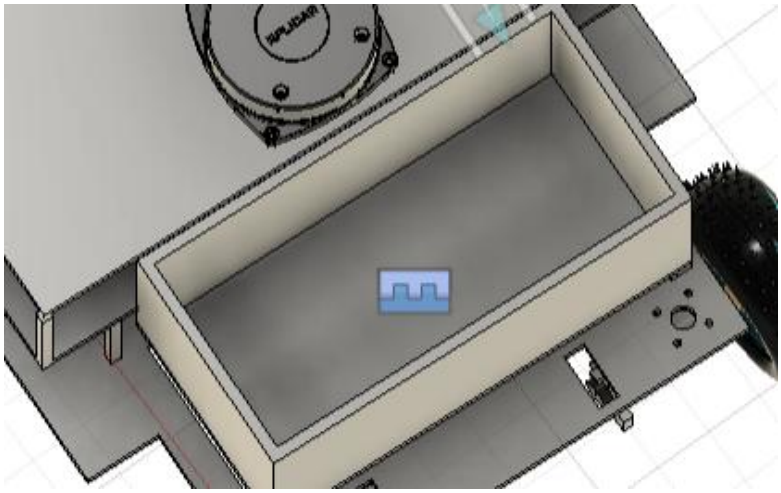


Figure 4: *The offloading bin in the CAD model, with the sliding door driven by a rack and pinion.*

The off-loader is separate from the drivetrain, so its power draw is added to the peak-power sum above only for the short time it operates.

Challenges in hardware implementation

1. **Uneven wheel start-up at intermediate duty cycles.** During floor tests with opposite-sign commands, at mid-range duty (about 150 to 210) one front wheel sometimes did not turn while the others did, and all four wheels spun only near full duty. A motor starts only when the torque it produces exceeds the static friction torque T_f of its gearbox and wheel. With armature resistance R_a this sets a minimum duty:

$$\frac{k_t}{R_a}\left(\frac{u_{\min}}{255}V_{\text{bat}} - V_{\text{drop}}\right) \geq T_f \tag{39}$$

   Because T_f differs from motor to motor, the minimum duty differs per wheel. The fix was to command in-place turns at a fixed, higher duty cycle and to let the PID loop work above the dead zone for straight driving.

2. **Serial timeouts and control-loop overruns.** During testing the base controller sometimes stopped responding, with overruns in the controller loop and serial timeouts between the Pi and the Arduino. In the worst cases the controller entered an uncontrolled loop, executing commands continuously, and recovered only when the whole robot was powered off and on again. We did not fully isolate the cause. The behaviour is typical of a supply dip or electrical noise from the motor drivers disturbing the microcontroller or its serial link, so the 1 s command watchdog in the firmware limits the effect, and the motor supply wiring and USB connection are the first things we check when it appears.

2. Navigation System

Navigation runs on ROS 2 Jazzy with Ubuntu 24.04. The nodes close to the hardware (motor control, sensor drivers and camera) run on the Raspberry Pi 4, while mapping, localisation, path planning and AI inference run on a laptop on the same network. The two communicate through ROS 2 topics. The robot gets its pose from fused odometry, builds the map with slam_toolbox, localises on the saved map with AMCL and drives with the Nav2 stack. This split keeps the Pi free of heavy computation.

Wheel odometry

Between two control cycles the left and right sides travel Δs_L and Δs_R, computed from the tick counts and the distance per tick Δs defined earlier. The robot pose (x, y, θ) is then updated by dead reckoning:

$$\Delta s = \frac{\Delta s_R + \Delta s_L}{2} \tag{40}$$

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{\chi\, b} \tag{41}$$

$$x_k = x_{k-1} + \Delta s \cos\left(\theta_{k-1} + \tfrac{\Delta\theta}{2}\right) \tag{42}$$

$$y_k = y_{k-1} + \Delta s \sin\left(\theta_{k-1} + \tfrac{\Delta\theta}{2}\right) \tag{43}$$

$$\theta_k = \theta_{k-1} + \Delta\theta \tag{44}$$

This is accurate only while the wheels do not slip. A skid-steer robot slips sideways every time it turns, so the heading error grows with each turn and the position error grows with the distance travelled, and neither is corrected by wheel encoders alone. The effective width factor χ absorbs part of this error but not the random part. The odometry therefore needs a second, independent source of heading.

Inertial measurement

The MPU-6050 provides a gyroscope and an accelerometer. The gyroscope measures the yaw rate ω_z directly, and does not depend on wheel slip. After subtracting a bias b_g, which is estimated as the average gyroscope reading while the robot is stationary at start-up, the heading is obtained by integration:

$$\theta_k = \theta_{k-1} + (\omega_{z,k} - b_g)\,\Delta t \tag{45}$$

Integration alone drifts slowly because of residual bias and noise, so neither sensor is trusted on its own: wheel odometry is good over short distances in a straight line, and the gyroscope is good for rotation. They are combined by a filter.

Sensor fusion with the Extended Kalman Filter

The robot_localization package runs an Extended Kalman Filter (EKF) that fuses the wheel odometry and the IMU into a single odometry estimate. The state is the planar pose and velocity, $\mathbf{x} = [x,\, y,\, \theta,\, v,\, \omega]^{\mathsf{T}}$. In the prediction step the state is projected forward with the motion model f and the covariance P grows by the process noise Q. In the update step each measurement z, with noise covariance R, corrects the prediction by the Kalman gain K:

$$\hat{\mathbf{x}}_{k|k-1} = f(\hat{\mathbf{x}}_{k-1},\, u_k) \tag{46}$$

$$P_{k|k-1} = F_k\, P_{k-1}\, F_k^{\mathsf{T}} + Q \tag{47}$$

$$K_k = P_{k|k-1}\, H^{\mathsf{T}} \left(H\, P_{k|k-1}\, H^{\mathsf{T}} + R\right)^{-1} \tag{48}$$

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k|k-1} + K_k \left(z_k - h\big(\hat{\mathbf{x}}_{k|k-1}\big)\right) \tag{49}$$

$$P_k = (I - K_k H)\, P_{k|k-1} \tag{50}$$

The motion model for the planar robot and its Jacobian $F = \partial f / \partial \mathbf{x}$ are:

$$\hat{\mathbf{x}} = \begin{bmatrix} x \\ y \\ \theta \\ v \\ \omega \end{bmatrix}, \qquad f(\hat{\mathbf{x}}) = \begin{bmatrix} x + v\cos\theta\,\Delta t \\ y + v\sin\theta\,\Delta t \\ \theta + \omega\,\Delta t \\ v \\ \omega \end{bmatrix} \tag{51}$$

The encoder velocities (forward speed v and yaw rate ω from the differential-drive controller) and the gyroscope yaw rate are the measurements. Because the measurement matrix H simply selects these quantities, the filter weights them by their noise covariances: the gyroscope gets more weight for yaw rate and the encoders more weight for forward speed. If the measurements are briefly lost, the prediction step carries the estimate forward using the motion model. The filter publishes the filtered odometry and the `odom` to `base_link` transform that the rest of the stack uses.

Mapping (SLAM)

The Slamtec RPLIDAR returns, for each beam at angle α, a range r. With the robot at pose (x, y, θ), the point where the beam hits an obstacle in the map frame is:

$$\mathbf{p}_{\text{hit}} = \begin{bmatrix} x + r\cos(\theta + \alpha) \\ y + r\sin(\theta + \alpha) \end{bmatrix} \tag{52}$$

The lidar has a fixed angular step $\Delta\alpha$, so neighbouring beams are $s = r\,\Delta\alpha$ apart at range r. For the map to have no gaps, s must not exceed the map resolution ρ, which limits the range over which the lidar can build a solid wall:

$$s = r\,\Delta\alpha \le \rho \;\Rightarrow\; r \le \frac{\rho}{\Delta\alpha} \tag{53}$$

The slam_toolbox package builds the map in two stages. First, scan matching finds the pose ξ that best aligns the new scan with the existing map M:

$$\xi^{*} = \arg\max_{\xi} \sum_{i} M\big(T_{\xi}\, h_i\big) \tag{54}$$

where h_i are the scan end points and T_{ξ} is the transform that places them in the map frame. Second, when the robot returns to a known place (loop closure), a pose graph is optimised so that the accumulated drift is spread over the whole trajectory:

$$\min_{\xi_1,\ldots,\xi_n} \sum_{(i,j)} e_{ij}^{\mathsf{T}}\, \Omega_{ij}\, e_{ij} \tag{55}$$

Here e_{ij} is the difference between the measured and the predicted relative pose of nodes i and j, and Ω_{ij} is the confidence in that measurement. The map itself is an occupancy grid. Each cell stores the log-odds of being occupied, which is updated additively with every scan, so repeated hits make a wall more certain and a few noisy returns are averaged out:

$$l_t(m_i) = l_{t-1}(m_i) + \log\frac{p(m_i \mid z_t)}{1 - p(m_i \mid z_t)} - l_0 \tag{56}$$

$$p(m_i) = 1 - \frac{1}{1 + e^{\,l_t(m_i)}} \tag{57}$$

The quality of the map depends strongly on the odometry that feeds the scan matcher, which is why the fused odometry above is used. We also tuned the slam_toolbox parameters (resolution, maximum laser range, minimum travel distance before an update, loop-closure thresholds) and we drive slowly and avoid sharp in-place turns during mapping. The finished map is saved and serialised for localisation.

Localisation (AMCL)

During a run, the saved map is loaded and the robot localises on it with the adaptive Monte Carlo localisation (AMCL) particle filter. The pose belief is a set of N weighted particles. After each motion the particles are moved with the odometry plus noise, each particle i is weighted by how well the current scan z fits the map from that pose, and the weights are normalised:

$$w_i \propto \prod_{j=1}^{K} p(z_j \mid x_i, m) \tag{58}$$

$$\sum_i w_i = 1 \tag{59}$$

The particles are resampled when too few of them carry most of the weight, which is measured by the effective particle count, and AMCL adapts N to the uncertainty:

$$N_{\text{eff}} = \frac{1}{\sum_i w_i^2} \tag{60}$$

The initial pose is given in RViz with the 2D Pose Estimate tool, and the filter corrects the odom to map transform from there.

Path planning and control (Nav2)

Nav2 combines a global planner, which finds a route across the whole map, and a local controller, which follows it while avoiding new obstacles. Both work on costmaps built from the map and the live lidar scan. A grid search such as A* expands the cell with the lowest estimated total cost, where g is the cost so far and h the estimated remaining cost to the goal:

$$f(n) = g(n) + h(n) \tag{61}$$

Each costmap cell has a lethal cost where there is an obstacle. Around obstacles the cost decays with distance d, so the robot prefers paths that stay away from walls. With the inscribed radius r_{ins} of the robot, the inflation radius r_{inf} and a decay factor k, the cost is:

$$C(d) = \begin{cases} 254, & d \le r_{\text{ins}} \\ 252\, e^{-k\,(d - r_{\text{ins}})}, & r_{\text{ins}} < d \le r_{\text{inf}} \\ 0, & d > r_{\text{inf}} \end{cases} \tag{62}$$

The robot is rectangular, so its circumscribed radius is the same distance d computed earlier, about 0.21 m for our 0.30 m by 0.30 m footprint. A corridor or gate of free width w_c can be passed only if:

$$w_c \ge 2\,(r_{\text{ins}} + \varepsilon) \tag{63}$$

where ε is a small safety margin. The gamefield has narrow passages, so we tightened the inflation radius and the footprint in both costmaps until the planner would route through them, while keeping r_{inf} large enough

that the robot does not scrape walls. The local controller samples or optimises velocity commands (v, ω) within the limits of the robot, scores them against the path, the goal and the obstacle costs, and sends the best command to the base controller. Velocity limits are set from the drivetrain analysis above, with a lower speed and acceleration limit near the ramp. Waypoints are sent with the Nav2 waypoint follower, so the robot visits a list of goals in order across the gamefield.

Operating on the ramp

A 2D lidar scans one horizontal plane. When the robot pitches by an angle φ on the ramp, the scan plane tilts and may hit the ramp surface itself, which would appear as a wall. If the lidar is mounted at height h_l above the ground, the beam meets the ground at a distance:

$$r_g = \frac{h_l}{\tan \varphi} \tag{64}$$

To avoid false obstacles, r_g must be larger than the range used to mark obstacles in the local costmap, or the obstacle layer must filter returns closer than r_g. With $h_l \approx 0.25\,\text{m}$ (estimated from the CAD, as the lidar sits on top of the 0.27 m high robot) and a pitch equal to the 30° ramp angle,

$$r_g = \frac{0.25}{\tan 30^\circ} = 0.43\,\text{m},$$

so a return closer than about 0.4 m during the climb may be the ramp itself and not an obstacle. The IMU pitch is used to confirm that the robot is on the ramp.

3. Computer Vision System

The camera lets the robot recognise waste items and classify each one as plastic, metal or organic. The task rules require teams to train their own model. Our Raspberry Pi 4 has no AI accelerator, so the camera runs on the Pi and the heavy inference runs on the laptop, which acts as the AI processor. The pipeline has four steps: image capture on the Pi, transfer of frames to the laptop, inference with the trained model, and publication of the result as a ROS 2 topic that the behaviour logic reads.

Dataset and training

The model was trained on Google Colab using public waste datasets with images and annotations, relabelled into our three. Images are resized to the network input size and normalised with the dataset mean μ and standard deviation σ:

$$x' = \frac{x - \mu}{\sigma} \tag{65}$$

The classes are unlikely to have equal numbers of images. If class c has N_c images out of N in total and there are C classes, a class weight in the loss prevents the model from favouring the largest class. Data augmentation (flips, brightness and contrast changes, small rotations and crops) is used so that the model copes with the lighting of the gamefield, which differs from the lighting in online photos:

$$w_c = \frac{N}{C N_c} \tag{66}$$

For a classifier, the network outputs a score z_c for each class, which is turned into a probability by the softmax function. Training minimises the weighted cross-entropy between the probabilities and the true label y:

$$p_c = \frac{e^{z_c}}{\sum_{k=1}^{C} e^{z_k}} \tag{67}$$

$$\mathscr{L} = -\sum_{c=1}^{C} w_c \, y_c \log p_c \tag{68}$$

If the model also locates the item in the image (object detection), the overlap between a predicted box A and the true box B is measured by the intersection over union (IoU), and a detection counts as correct when IoU is above a threshold, usually 0.5:

$$\mathrm{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{69}$$

Evaluation metrics

Model quality is judged on a validation set kept apart from the training images. With true positives TP, false positives FP and false negatives FN for each class, the metrics are:

$$\mathrm{Precision} = \frac{TP}{TP + FP} \tag{70}$$

$$\mathrm{Recall} = \frac{TP}{TP + FN} \tag{71}$$

$$F_1 = \frac{2\,\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}} \tag{72}$$

For detection, the average precision (AP) of a class is the area under its precision-recall curve, and mAP is the mean over the C classes:

$$\mathrm{AP}_c = \int_0^1 p_c(r)\, dr \tag{73}$$

$$\mathrm{mAP} = \frac{1}{C} \sum_{c=1}^{C} \mathrm{AP}_c \tag{74}$$

Results on the validation set and on photos taken in the game-field are reported in Section IV.

Camera geometry

The camera is modelled as a pinhole. A point at (X, Y, Z) in the camera frame appears at pixel (u, v), where f_x and f_y are the focal lengths in pixels and (c_x, c_y) is the image centre:

$$u = f_x \frac{X}{Z} + c_x \tag{75}$$

$$v = f_y \frac{Y}{Z} + c_y \tag{76}$$

If an item has a known real height H and appears h pixels tall, its distance is estimated by inverting this relation, and the bearing to the item follows from its horizontal pixel position. The robot uses these two values to align with the item or the drop zone:

$$Z = \frac{f_y H}{h} \tag{77}$$

$$\beta = \arctan\left(\frac{u - c_x}{f_x}\right) \tag{78}$$

The focal lengths and the centre are found by calibrating the camera with a chessboard pattern.

Image transport and latency

Sending raw frames over the network needs a lot of bandwidth. For W by H pixels, 3 colour bytes per pixel and f frames per second:

$$B = 8\,W\,H \cdot 3 \cdot f \ \text{ bit/s} \tag{79}$$

As an illustration, 640 by 480 pixels at 15 frames per second would need about 110 Mbit/s raw, which is too much for a shared Wi-Fi link, so the frames are compressed (JPEG) before transfer. The delay between an item appearing in front of the camera and the result being published is the sum of four parts, and must stay short enough for the robot to act while the item is still in view:

$$T_{\text{total}} = T_{\text{capture}} + T_{\text{transfer}} + T_{\text{inference}} + T_{\text{publish}} \tag{80}$$

Decision logic

A single frame can be wrong, so the robot acts on a class only when the model reports it with confidence above a threshold τ in k consecutive frames. If the per-frame error rate is p_e and frames are treated as independent, the probability of a false trigger falls rapidly with k:

$$P_{\text{false}} \approx p_e^{\,k} \tag{81}$$

For example, with $p_e = 0.1$ and $k = 3$, the false-trigger probability is about 0.001. The cost is a delay of k frames, so k is chosen as small as the accuracy allows.

IV. Experimental Results

The robot was tested and assessed in four areas: drivetrain and speed control, quality of the map, autonomous navigation, and waste classification.

Drivetrain and speed control

Each test used the same routine: reset the encoders, send a command, wait for the 2 s run to finish, and read back the left and right encoder counts. The first rounds were run with the wheels in the air to isolate the PID loop, and then repeated on the floor with the robot's full weight on the wheels.

Test	Command	Left ticks	Right ticks	Comment
Floor, round 1	[0 200 200]	1902	1912	Both sides agree within 1%
Floor, round 2	[0 255 −255]	−2007	−1879	Reverse, sides differ by about 6%
Floor, round 3	[0 200 −60]	1653	−34	Turn command (opposite signs); right side barely moved
Floor, round 4	[0 −150 150]	−93	51	Turn command (opposite signs); almost no movement

Straight-line driving was well matched between the two sides. A mismatch in counts over a run means the robot veers, and the heading error it causes follows from the odometry equations:

$$\delta\theta = \frac{\Delta s\,(n_R - n_L)}{\chi\, b} \tag{82}$$

The in-place turns under PID control were unreliable, which agrees with the turning model in Section III: the torque needed to slide four wheels sideways is high compared to straight driving. We tested raw duty cycles to find where in-place turning becomes reliable.

Raw command (left, right)	Left ticks	Right ticks	Observation
150, −150	800	−151	Front right wheel did not turn
180, −180	1525	−204	Rotated, then stopped
210, −210	3037	38	Right wheel not moving
255, −255	3967	−4926	All four wheels spinning well
−255, 255	−4006	4570	All four wheels spinning well

Only the full-duty commands turned all four wheels reliably, so in-place rotation is commanded at a fixed high duty cycle. At full duty the robot turned a full rotation without struggling, but too aggressively for mapping, so the spin duty was then lowered to a final SPIN_PWM of 200, which gives a gentler in-place turn that is suitable for mapping. The effective width factor was set to $\chi = 1.5$, a typical value for four-wheel skid-steer robots [7], and is refined by comparing the heading from the encoders with the gyroscope heading in a 360 degree rotation test.

Quality of the map

We compared maps built from wheel odometry alone and from the fused (encoder plus IMU) odometry, in simulation and on the real robot, with two simple tests: turning in place by 90 degrees, and driving a complete loop. Our first real-robot map of a room, made before the fusion and tuning were complete, was very noisy. The corridor map made later was clean apart from speckle along a railing, whose thin bars return the lidar beams inconsistently. The gamefield simulation map is shown in Figure 5.

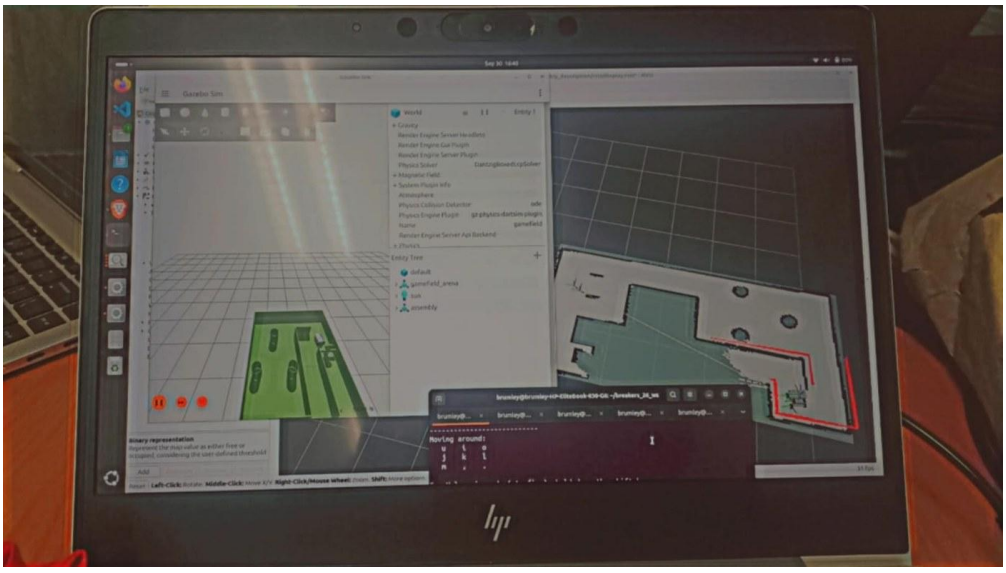


Figure 5: *Gamefield simulation, with the Gazebo world on the left and the map being built by slam_toolbox in RViz on the right.*

Autonomous navigation

On a saved map of a narrow corridor, used as a stand-in for the gamefield, the robot reached a goal set in RViz in 13 s with no recovery behaviours, after the costmaps were tightened. Before the costmaps were tightened, the robot did not move towards goals set in the same corridor.

Waste classification

Overall mAP at IoU 0.5 is measured on a held-out validation set, and on photos taken in the gamefield lighting.

V. Acknowledgement

The development of our robot depended greatly on the tutorials, presentations and support provided by the Robotics Dojo interns and mentors, and on the parts made available through the Japan International Cooperation Agency (JICA) and JKUAT. The team also thanks the open-source ROS 2, slam_toolbox, Nav2 and robot_localization communities, whose software this work builds on.

VI. 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.
2. M. I. Ribeiro, "Kalman and extended Kalman filters: Concept, derivation and properties," Institute for Systems and Robotics, 2004.
3. T. Moore and D. Stouch, "A generalized extended Kalman filter implementation for the Robot Operating System," in *Proc. 13th Int. Conf. on Intelligent Autonomous Systems (IAS-13)*, 2014.
4. S. Macenski and I. Jambrecic, "SLAM Toolbox: SLAM for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021.
5. S. Macenski, F. Martin, R. White and J. Gines Clavero, "The Marathon 2: A navigation system," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2020.
6. S. Thrun, W. Burgard and D. Fox, *Probabilistic Robotics*. Cambridge, MA, USA: MIT Press, 2005.
7. A. Mandow, J. L. Martinez, J. Morales, J. L. Blanco, A. Garcia-Cerezo and J. Gonzalez, "Experimental kinematics for wheeled skid-steer mobile robots," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2007.
8. G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.