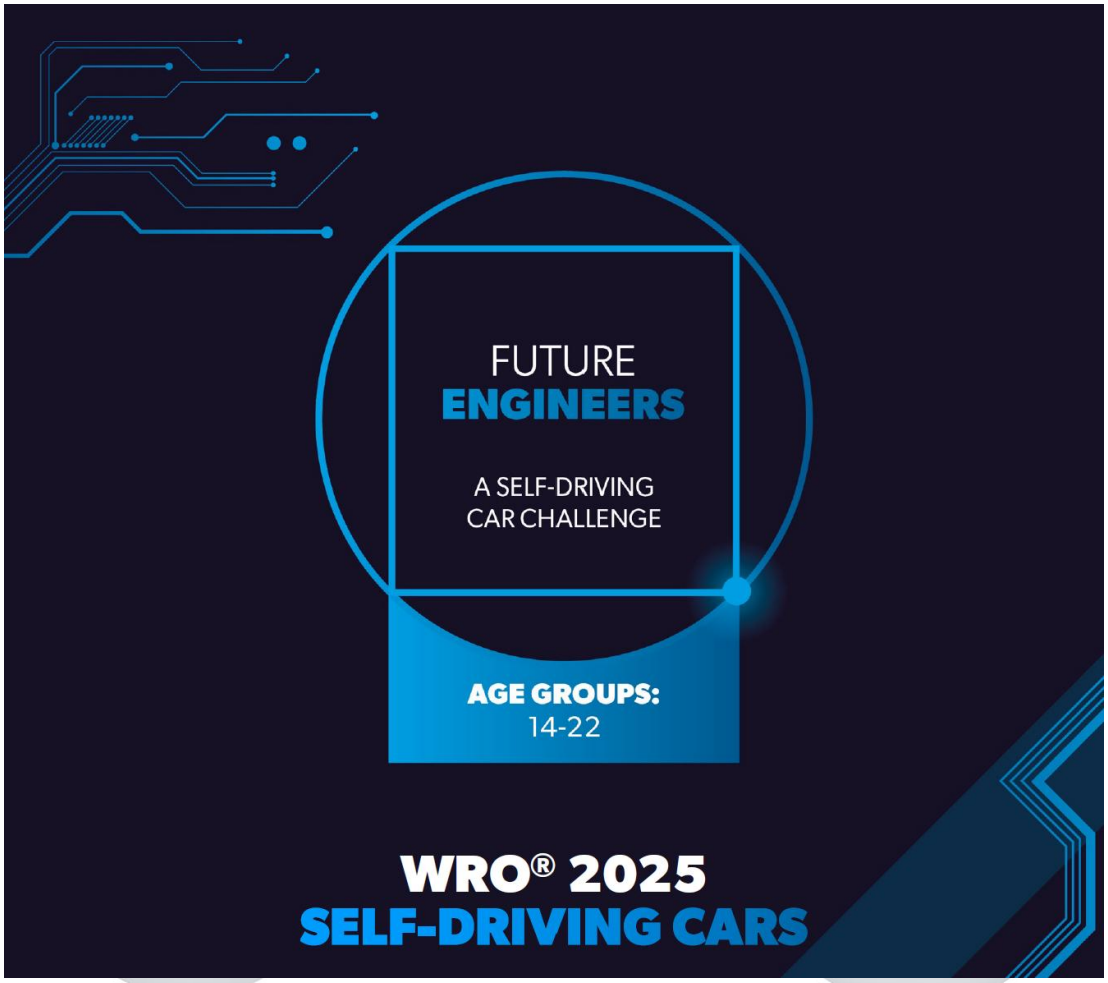
Viva La Vida — WRO FE Open Challenge



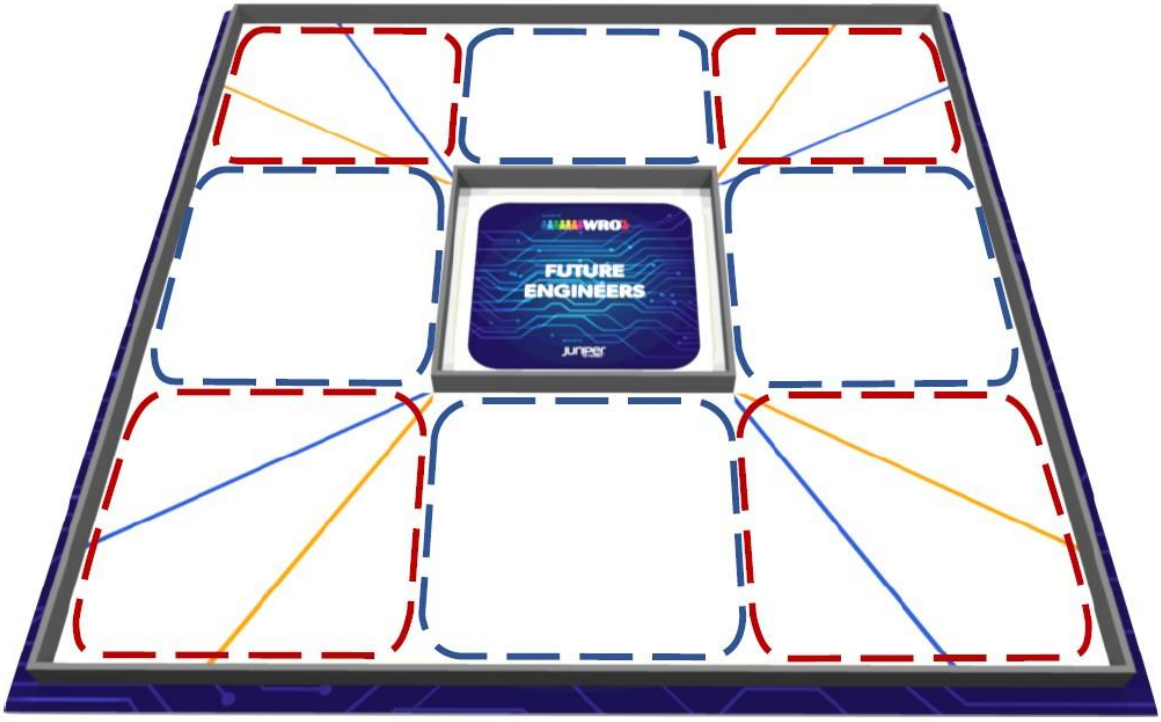
VIVA ΛA VIΔA

IN A ROBOTICS/ENGINEERING CONTEXT, IT IS A FUN, PLAYFUL SLOGAN MEANING “LONG LIVE THE SCREW!”, CELEBRATING THE SIMPLE BUT ESSENTIAL MECHANICAL COMPONENT!

Table of Contents

1 Overview & Scope

Mission: autonomously complete three laps inside a 3×3 m field containing a randomized 1×1 m inner island that creates corridors. Robot must maintain direction, avoid the walls, and stop safely. Competition rules: single power switch + Start button; no wireless during the run.



- The robot must autonomously navigate a dynamic track with varying corridor widths, ensuring it avoids walls and stays on course.
- Key challenges include detecting and responding to obstacles, managing tight turns, and adapting to different track conditions.
- The robot uses sensors to measure distances and avoid obstacles, with sensor fusion techniques to ensure accurate navigation in real-time.

2 The robot - Hardware & Wiring

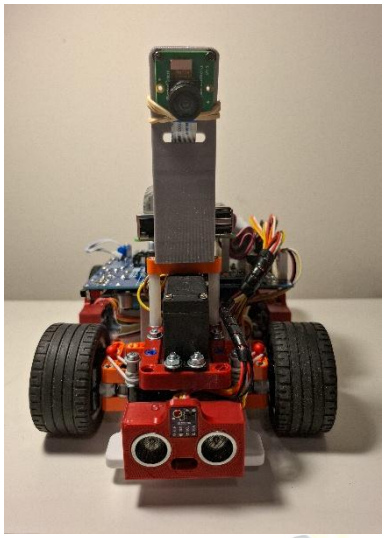
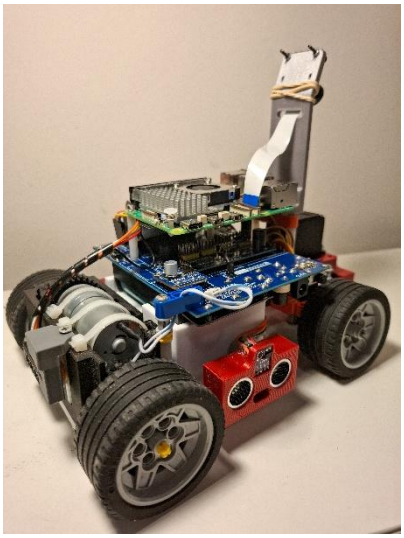
2.1 Bill of Materials

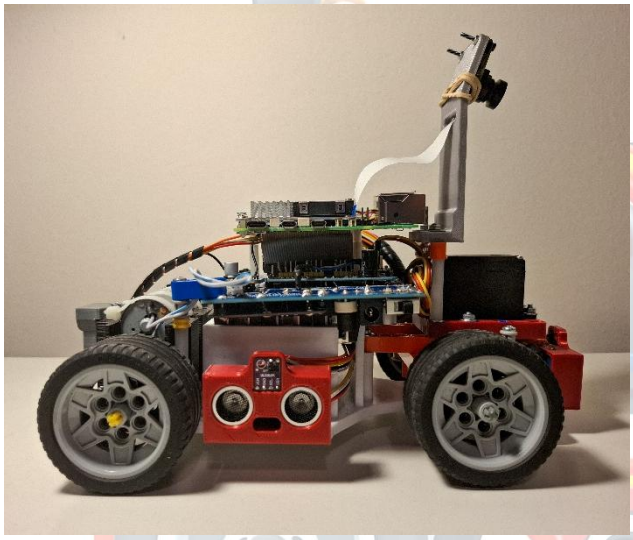
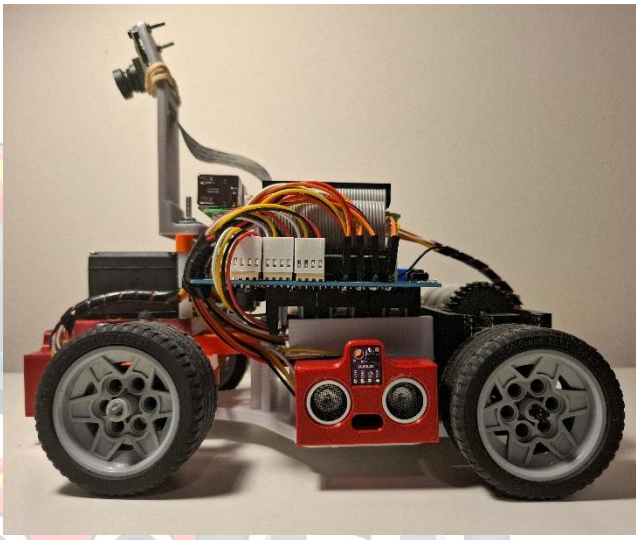
Component	Role	Interface/Notes
Raspberry Pi	Main controller	GPIO/I²C
PCA9685	Servo + motor PWM	50 Hz PWM; I²C
MPU6050	Gyro-Z for yaw	I²C
Ultrasonic x3	Front/Left/Right	Trig/Echo via gpiozero
VL53L0X (opt.)	ToF sensors	I²C + XSHUT addressing
Steering Servo	Single	1000–2000 µs pulse
Motor + Driver/ESC	Single	PWM ch1/ch2 (FWD/REV)
Start Button	Headless start	GPIO20
Status LEDs	Green / Red	GPIO19 / GPIO13
Battery/Power	5–6 V servo; 5 V logic	Common ground

2.2 Pin Map

Subsystem	Signal	Pin/Channel	Dir	Notes
Start Button	START_BTN	GPIO 20	IN	Pull-up; active LOW
LEDs	GREEN/RED	GPIO 19 / 13	OUT	Status/ready
US Front	TRIG / ECHO	22 / 23	OUT / IN	max_distance, queue_len
US Left	TRIG / ECHO	27 / 17	OUT / IN	max_distance, queue_len
US Right	TRIG / ECHO	5 / 6	OUT / IN	max_distance, queue_len
Servo	SERVO_CHANNEL	PCA ch 0	PWM	Steering
Motor	MOTOR_FWD / REV	PCA ch 1 / 2	PWM	Duty 0–100%
I²C	SCL / SDA	board.SCL / board.SDA	—	PCA/IMU/ToF
ToF XSHUT	L/R/F/B/LF/LR	D16 / D25 / D26 / D24 / D8 / D7	OUT	Addr 0x30/31/32/33/34/35

2.3 Robot photos

3 System Architecture

Code structure:

- imports & global configuration (speeds, thresholds, filtering, timing); hardware setup (GPIO/I²C, PCA9685, MPU6050, pin constants)
- sensor initialization (ToF with XSHUT addressing and ultrasonic fallbacks)
- data model & threading primitives (Events, Locks, deques for plotting/logging)
- RobotController class (low-level actuation, PWM/servo handling, distance filtering, lane-keeping corrections, turn-decision helpers)
- sensor_reader thread (continuous filtered measurements into a shared dict)
- robot_loop FSM (IDLE → CRUISE → TURN_INIT → TURNING → POST_TURN → STOPPED, with yaw integration, narrow-mode scaling, and safety guards)
- GUI subsystem (Tkinter plots, status, sliders, CSV export, preset save/load); start/stop handlers; utilities (CSV export, JSON overrides)
- The __main__ entry (GUI vs headless startup, thread lifecycles, cleanup)

The stack is organized into two real-time threads plus an optional GUI:

- sensor_reader: acquires ultrasonic/ToF ranges, applies median+EMA smoothing with spike rejection, and publishes under a Lock.
- robot_loop: integrates gyro Z to yaw, runs the finite state machine (FSM), computes steering/motor commands via PCA9685.
- GUI (optional): Tkinter with live plots, tuning sliders, CSV export, and start/stop controls.

Synchronization uses Events (readings_event, loop_event, sensor_tick) to gate progress without busy-waiting.

Outline:

- Software Workflow: The system operates with a multi-threaded architecture, where sensor readings are continuously acquired and processed, and decisions are made based on these readings. The robot transitions between different states (CRUISE, TURN_INIT, TURNING, STOPPED) based on sensor data and predefined thresholds.
- Threading: The robot utilizes two main threads: one for acquiring sensor data and another for executing the robot's finite state machine (FSM). This setup ensures real-time performance without busy-waiting, as synchronization is managed using events and locks.

3.1 Code Structure

3.1.1 Imports

3.1.1.1 External Libraries

Uses standard threading/time/collections plus: gpiozero.DistanceSensor for ultrasounds; Adafruit PCA9685 for PWM (servo & motor); adafruit_mpu6050 for IMU; adafruit_vl53l0x for ToF; Tkinter + matplotlib for GUI/plots; numpy for median/mean; RPi.GPIO for LEDs/button; digitalio + board for XSHUT and I²C lines. These define the runtime I/O model.

3.1.1.2 Internal Modules

The solution is a single Python program structured into clear sections (imports, configuration, hardware setup, RobotController class, sensor thread, FSM loop, GUI, utilities, main). No separate packages are required; this keeps deployment simple on the Pi.

3.1.2 Variables

3.1.2.1 Global Constants

Tunable thresholds & timings (e.g., SPEED_* per FSM state; SOFT_MARGIN/MAX_CORRECTION; STOP_THRESHOLD; FRONT_TURN_TRIGGER; TURN_DECISION_THRESHOLD; TURN_TIMEOUT; TURN_LOCKOUT; TARGET_TURN_ANGLE and tolerance; narrow-mode thresholds; filtering alphas/jumps; loop/sensor delays). These drive behavior and can be adjusted via GUI sliders or JSON overrides.

3.1.2.2 Robot-Specific Variables

Servo geometry (SERVO_CENTER/MIN/MAX and pulse widths), PCA channels for servo/motor, ultrasonic pins, ToF XSHUT pins, LEDs and Start button. Also stores immutable base values and environment scaling factors (SPEED_ENV_FACTOR, DIST_ENV_FACTOR) used by helper getters

(eff_*). JSON overrides are loaded by load_variables_from_json() early so all downstream logic sees updated values.

3.1.3 H/W Setup

3.1.3.1 Hardware Configuration

GPIO set to BCM mode; I²C bus initialized. PCA9685 set to 50 Hz (servo-friendly). IMU (MPU6050) instantiated and a 500-sample gyro bias is computed at startup (robot must be still). ToF sensors are optionally addressed via XSHUT (0x30..0x35) and started in continuous mode; fallback to ultrasonic is automatic if ToF init fails.

3.1.3.2 Pin Assignments

Front US (TRIG 22 / ECHO 23), Left (27 / 17), Right (5 / 6). Servo on PCA ch 0, motor driver on ch 1/2. Start button on GPIO20 (pull-up, active LOW). LEDs: GREEN 19, RED 13. I²C SCL/SDA.

3.1.4 Robot Classes

3.1.4.1 Sensor Handling

RobotController.filtered_distance() fuses raw ToF/US with a small rolling history (median), jump rejection (FILTER_JUMP[_TOF]) and EMA smoothing (FILTER_ALPHA[_TOF]). Each sensor type uses its own parameters. Distances are normalized to cm; sentinel 999 denotes invalid/unavailable.

3.1.4.2 Motor Control

rotate_motor() maps 0–100% duty to PCA9685 channels (FWD/REV). stop_motor() zeros both channels. set_servo() clamps to limits, applies optional slew limiting (SERVO_SLEW_DPS in deg/s) to limit command rate, converts angle to microsecond pulse and to PCA duty cycle. safe_straight_control() produces bounded lane-keeping corrections based on SOFT_MARGIN and MAX_CORRECTION.

3.1.4.3 State Management

RobotState enum defines IDLE, CRUISE, TURN_INIT, TURNING, POST_TURN, STOPPED. Events (readings_event, loop_event) gate progression; sensor_tick coordinates waits to keep loops responsive without busy-waiting. Narrow-mode derives from L+R with hysteresis to scale speeds/thresholds.

3.1.5 Robot Loop

3.1.5.1 Main Control Loop

robot_loop() runs continuously after readings_event is set. It integrates yaw from gyro Z (with a simple low-pass) and appends telemetry to deques for plotting. FSM actions drive servo/motor per state with state_speed_value() (affected by narrow-mode scaling). Lap/turn counters update, and POST_LAP behavior enforces stop at MAX_LAPS.

3.1.5.2 Decision Making

Turn preconditions: front < FRONT_TURN_TRIGGER and (t − last_turn) ≥ TURN_LOCKOUT. Direction requires XOR (open_left, open_right – only one Input is True), where open is None/999 or > TURN_DECISION_THRESHOLD. Optional LOCK_TURN_DIRECTION fixes the first chosen side. Turn termination uses |Δyaw − TARGET_TURN_ANGLE| ≤ tolerance, else timeout or max-angle as fail-safes. STOP_THRESHOLD forces immediate STOPPED with timed auto-retry.

3.1.5.3 Movement Logic

Before the first turn, servo stays centered to avoid premature biasing. After the first turn, gentle wall-following corrections are applied when a side distance falls below SOFT_MARGIN. POST_TURN holds straight briefly before resuming CRUISE.

3.1.6 GUI

3.1.6.1 Tkinter Setup

launch_gui() builds a 3-panel matplotlib view (front, sides, yaw) with live annotations, start/stop buttons, status indicators (including a circular traffic-light style), and labels for turns/laps. Clean shutdown stops ToF continuous mode and calls several cleanups.

3.1.6.2 Sliders & Data Visualization

Grouped sliders expose key parameters (speeds, margins, triggers, timeouts). Changes take effect live and can be saved/loaded as JSON presets. Export CSV captures time series, state, counters, and a snapshot of slider values for later analysis.

3.1.7 Main

3.1.7.1 Initialization

The __main__ guard prints mode and launches GUI or headless path. In headless mode, LEDs indicate readiness; a physical Start press arms readings_event, then loop_event. KeyboardInterrupt and window close handlers stop loops, halt sensors, and clean up GPIO.

3.1.7.2 Robot Execution

In GUI mode, sensor and loop threads are started on demand via the buttons. In headless mode, sensor_reader runs as a daemon and robot_loop runs in the foreground after Start. Both modes share exactly the same control logic, ensuring identical behavior during competition.

4 Decision-Making Architecture

The controller separates sensing, eligibility checks, and action commits. Each behavior is guarded by explicit thresholds, hysteresis, and lockouts so that small sensor fluctuations do not cause thrashing. All important decisions are explainable in logs and reproducible by fixed parameter sets (or slider presets).

4.1 Perception Pipeline

- Distances are read as meters (gpiozero) or mm (ToF) and normalized to cm.
- Median over N_READINGS suppresses outliers; EMA (FILTER_ALPHA) smooths; values jumping beyond FILTER_JUMP are rejected.
- For corridor reasoning we derive: (a) front distance, (b) left & right distances, and (c) left+right sum for narrow-mode.

Example calibration anchor: with 1.0 m corridor width and a 10 cm robot, left+right is expected near 90 cm; this informs NARROW_SUM_THRESHOLD setting.

4.2 Corridor Model & Narrow-Mode Scaling

We treat the track as four straight corridors and four corners. Narrow-mode activates when L+R < NARROW_SUM_THRESHOLD and deactivates above threshold+NARROW_HYSTERESIS. When

active, SPEED_ENV_FACTOR and DIST_ENV_FACTOR can scale state speeds and distance thresholds to keep the robot stable (e.g., drive slower and begin corrections earlier). Hysteresis prevents chatter when moving near the boundary of activation.

4.3 Turn Eligibility (Pre-conditions)

A turn is considered only when (i) the front distance falls below FRONT_TURN_TRIGGER and (ii) the last turn ended ≥ TURN_LOCKOUT seconds ago. This prevents premature or back-to-back turns on zig-zag geometries.

4.4 Direction Selection (Symmetry-Aware)

The side 'open' predicate is: distance is None/999 or distance > TURN_DECISION_THRESHOLD. We require XOR (open_left, open_right - only one Input is True) so that we only commit when exactly one side is clearly open; if both are open or both closed we keep waiting in TURN_INIT at a reduced speed. Optionally, LOCK_TURN_DIRECTION fixes the first chosen direction (LEFT/RIGHT) for the session for consistent lap direction.

4.5 Turn Commit & Steering Command

Upon selection, we: (a) record yaw_start and time, (b) set the servo to TURN_ANGLE_LEFT/RIGHT (hardware setpoints), and (c) switch to state TURNING with SPEED_TURN.

4.6 Termination Strategy (Multi-guard)

Primary: gyro-based angle tracking. We estimate Δyaw = yaw − yaw_start and stop when |Δyaw − TARGET_TURN_ANGLE| ≤ TURN_ANGLE_TOLERANCE. Fallback 1: timeout when (now − turn_start) > TURN_TIMEOUT to avoid deadlocks. Fallback 2: a hard cap when |Δyaw| ≥ MAX_TURN_ANGLE to contain extreme cases (e.g., slipping, missing side). After stopping we center the servo, optionally snap yaw to the exact target for continuity, and enter POST_TURN for a short straight stabilization.

Rationale: IMU bias accumulates slowly, but at corner time-scales the bias is quasi-constant; using relative yaw is robust to drift over a single corner.

4.7 Straight-Line Corrections

After the first turn (to establish lane context), wall-following applies only gentle corrections when min(left,right) < SOFT_MARGIN. The correction magnitude saturates at MAX_CORRECTION; if still near the wall, it can renew. This prevents over-steering on straights while maintaining a comfortable distance from the inner walls.

4.8 Obstacle Stop & Auto-Retry

If front < STOP_THRESHOLD at any time, motors stop and the state becomes STOPPED with a retry deadline now+OBSTACLE_WAIT_TIME. On expiry the controller rechecks front; if clear (≥ STOP_THRESHOLD) the robot resumes CRUISE.

4.9 Determinism, Hysteresis, and Lockouts

- Determinism: state transitions require explicit pre-conditions and never depend on transient, ambiguous readings.
- Hysteresis: both narrow-mode and angle tolerance avoid edge-case chatter.
- Lockouts: TURN_LOCKOUT guarantees spacing between turns; servo slew bounds steering rate.

5 Finite State Machine (FSM)

5.1 States

State	Enter	Actions	Exit
IDLE	Readings on, loop off	Motors off; servo centered	Loop on → CRUISE
CRUISE	Normal driving	Gentle wall-following after 1st turn	Stop → STOPPED; Turn pre-conditions → TURN_INIT
TURN_INIT	Front < trigger & lockout OK	Slow; wait XOR(only one open side)	Direction chosen → TURNING; Front re-safe → CRUISE
TURNING	Direction committed	Servo setpoint; track yaw; guards	Angle tolerance OR timeout OR max-angle → POST_TURN
POST_TURN	Turn finished	Short straight stabilization	Timer elapsed → CRUISE
STOPPED	Obstacle/user/laps	Motors off; optional auto-retry	Retry clear → CRUISE

5.2 Sate actions

State	On-Entry Actions	Transition Condition (guard)	Next State	Notes
START_IDLE	Motors off; center servo	—	**START_READINGS**	Initial state
START_READINGS	Start sensor thread	—	**CRUISE**	—
CRUISE	Ramp motor; center servo; apply **CRUISE corrections** (after 1st turn) with soft margin → ±MAX_CORRECTION	front < STOP_THRESHOLD	**STOPPED**	Safety stop
		front < FRONT_TURN_TRIGGER **AND** lockout_ok	**TURN_INIT**	Prepare to turn
		(any time) run **NARROW detector**	**CRUISE** (stay)	Detector scales speeds/thresholds; not a state change
TURN_INIT	Slow down; wait XOR(open-side)	direction_chosen	**TURNING**	If direction locking enabled, apply lock now
TURNING	Servo = target angle; monitor yaw	turn_completed	**POST_TURN**	Completion via yaw/angle
POST_TURN	Short straight segment	segment_done	**CRUISE**	Re-enter cruise loop
STOPPED	Wait window; auto-retry timer	retry_window_elapsed **AND** path_clear	**CRUISE**	If still blocked after retries, you may re-enter STOPPED or fail-safe

6 Parameters & Profiles

Category	Key Parameters (defaults)	Purpose
Speeds	SPEED_CRUISE/TURN_INIT/TURN/POST_TURN	State-specific duty
Driving	SOFT_MARGIN; CORR_EPS; CORRECTION_MULTIPLIER	Wall-following
Safety	STOP_THRESHOLD; TURN_TIMEOUT; TURN_LOCKOUT	Guards
Turns	FRONT_TURN_TRIGGER; TURN_DECISION_THRESHOLD	Eligibility & direction
Yaw	TARGET_TURN_ANGLE; TURN_ANGLE_TOLERANCE; MAX_TURN_ANGLE	Termination
Narrow	NARROW_SUM_THRESHOLD; NARROW_HYSTERESIS	Scaling & chatter-free toggling
Filters	FILTER_ALPHA; FILTER_JUMP; N_READINGS	Smoothing & spike reject
Timing	LOOP_DELAY; SENSOR_DELAY	Loop pacing

6.1 Preset Profiles

Profile	Cruise	Turn	Target°	Tol°	Front Trigger	Lockout
Safe	20	12	84	6	100 cm	2.0 s
Balanced (default)	25	15	86	5	90 cm	1.5 s
Aggressive	40	16	88	4	80 cm	1.2 s

6.2 Operations Runbook

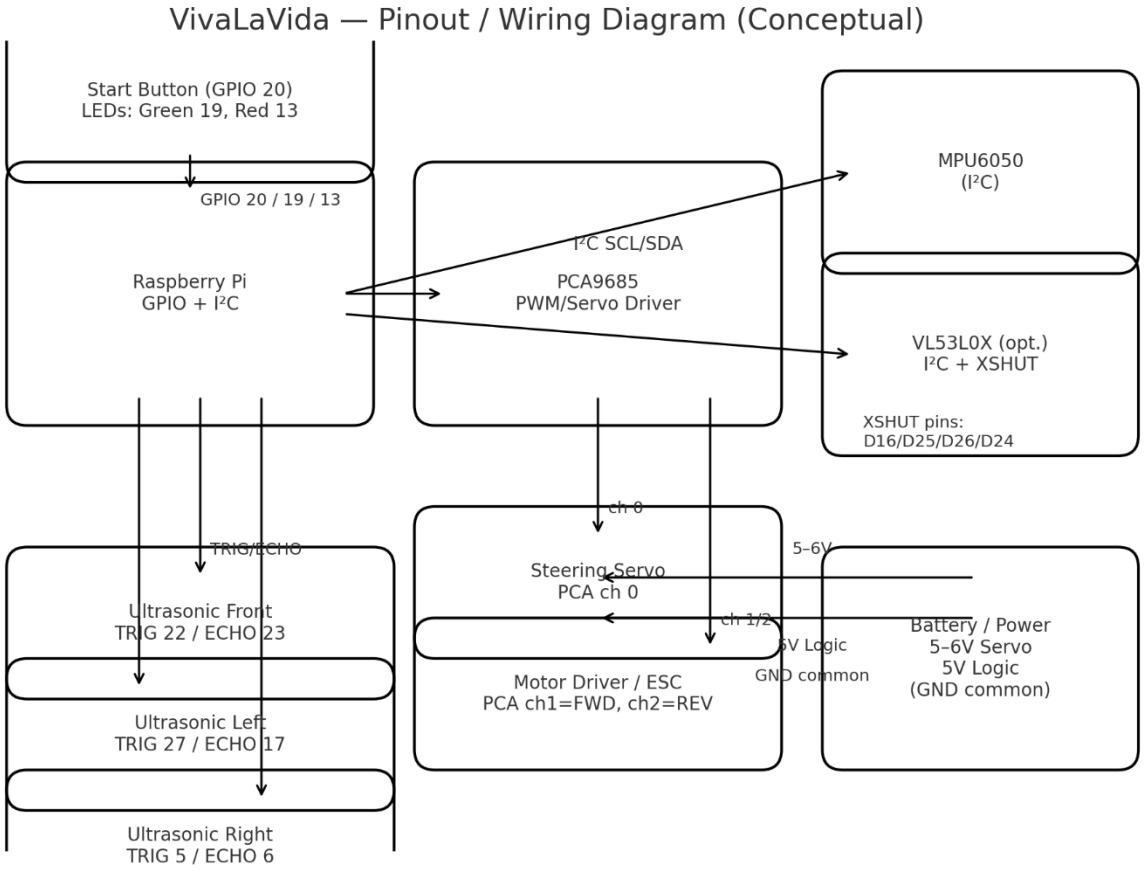
Power → READY LED or GUI → Start Readings → Start Loop (or headless Start). Observe plots; export CSV and save slider presets after runs.

6.3 Boot & Headless Start

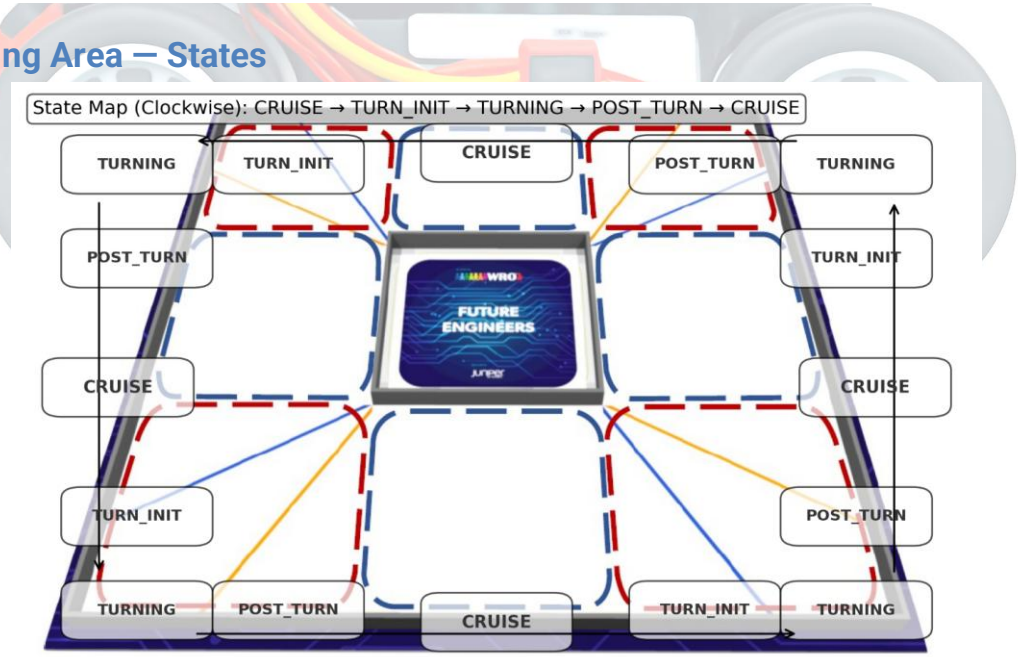
Boot sequence: RED LED on → sensors init → GREEN LED on blinking (ready). Headless: press Start (GPIO20 goes LOW) to arm readings_event and then loop_event.

7 Diagrams

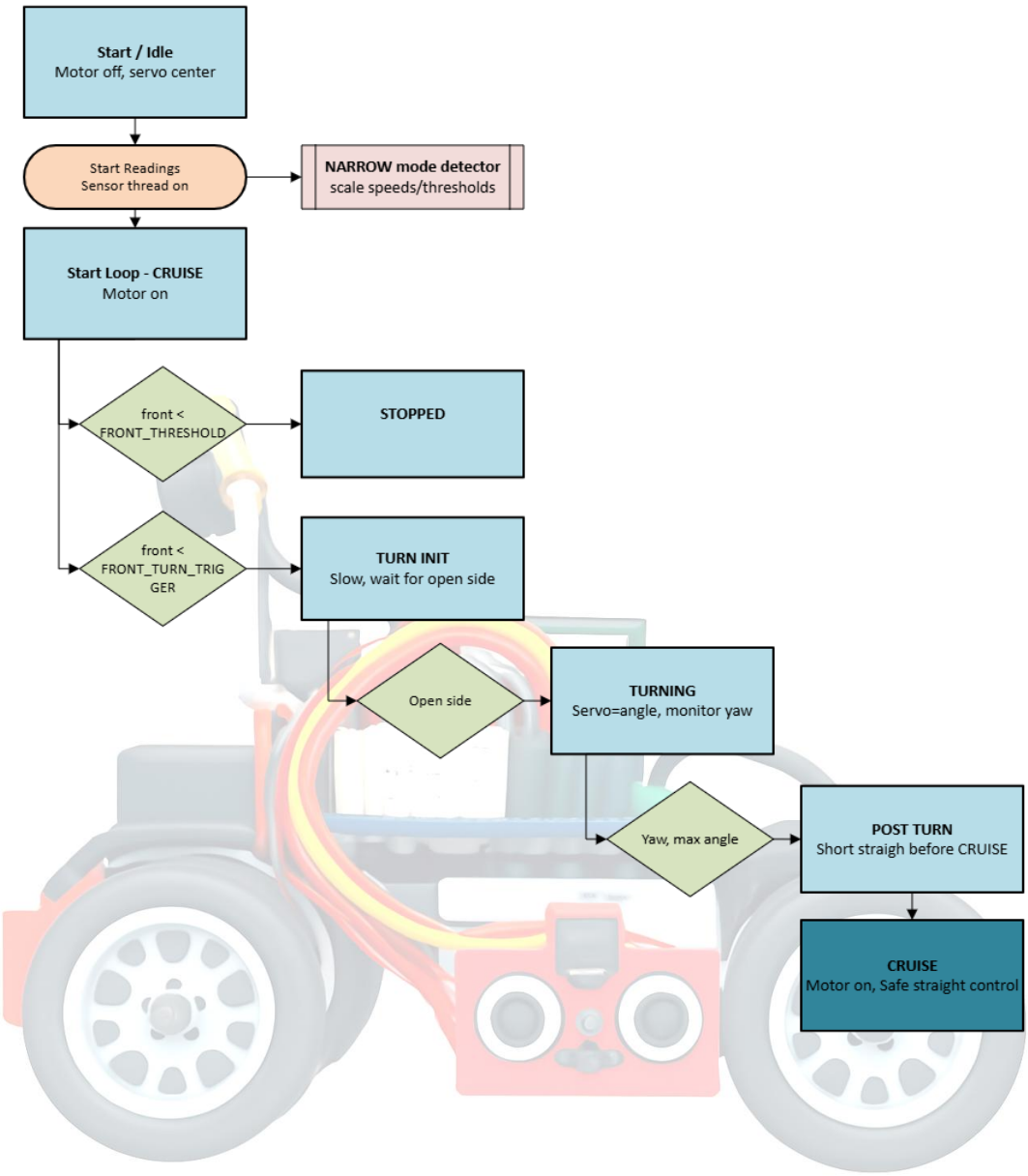
7.1 Pinout / Wiring (Conceptual)



7.2 Driving Area — States



7.3 Decision Flowchart (FSM + Safety + Narrow Mode)



8 Appendix A - Variables

Notes:

- Many thresholds are dynamically scaled when the robot enters a narrow corridor. Speeds are multiplied by NARROW_FACTOR_SPEED (via SPEED_ENV_FACTOR) and distance thresholds by NARROW_FACTOR_DIST (via DIST_ENV_FACTOR). The code uses helper functions eff_* to compute effective values.
- Units: distances in centimeters (cm), time in seconds (s), angles in degrees (°), speeds as 0–100% PWM.

8.1 Variables

Variable	Scope & Detailed Explanation	Accepted values & meanings
— Initialization & Operating Mode —		
USE_GUI	Selects runtime mode. If 1, starts the Tkinter GUI with live plots, sliders, and control buttons. If 0, runs headless and waits for the physical START button (GPIO).	0 = headless (competition), 1 = GUI (debug).
USE_TOF_SIDES	Chooses sensor type for side distances. 1 initializes VL53L0X ToF for left/right; 0 uses ultrasonic.	0 = ultrasonic, 1 = ToF (VL53L0X).
USE_TOF_FRONT	Chooses sensor type for the front distance sensor.	0 = ultrasonic, 1 = ToF (VL53L0X).
USE_TOF_CORNERS	Enable or disable corner sensors.	0 = disabled, 1 = enabled.
— Narrow-Corridor Behavior —		
NARROW_SUM_THRESHOLD	If (left + right) < threshold → narrow mode ON.	cm; typical 40–120.
NARROW_HYSTERESIS	Exit narrow mode only when (left + right) > threshold + hysteresis.	cm; typical 5–30.
NARROW_FACTOR_SPEED	Multiplier applied to state speeds in narrow mode (via SPEED_ENV_FACTOR).	Float ≥ 0.0; 1.0 = no change.
NARROW_FACTOR_DIST	Multiplier applied to distance thresholds in narrow mode (via DIST_ENV_FACTOR).	Float ≥ 0.0; 1.0 = no change.
— Speeds (0–100%) —		
SPEED_IDLE	Motor command in IDLE.	0–100; usually 0.
SPEED_STOPPED	Motor command in STOPPED.	0–100; usually 0.
SPEED_CRUISE	Baseline forward speed during CRUISE.	0–100.
SPEED_TURN_INIT	Speed while waiting for open side to turn.	0–100.
SPEED_TURN	Speed during TURNING.	0–100.

SPEED_POST_TURN	Speed after a turn during POST_TURN.	0–100.
— Driving & Safety —		
SOFT_MARGIN	Distance from wall when gentle steering corrections begin (safe_straight_control).	cm; typical 10–60.
MAX_CORRECTION	Max steering correction magnitude from safe_straight_control.	degrees; typical 3–15°.
CORR_EPS	cm: treat the side as 'steady' if within ±1.5 cm of trigger value.	Float; typical 1.5.
CORRECTION_MULTIPLIER	Proportional gain (servo degrees per cm of error 0 default: 2).	Float; higher = snappier, lower = smoother & slower.
STOP_THRESHOLD	Immediate stop if front distance < STOP_THRESHOLD.	cm; typical 10–50.
OBSTACLE_WAIT_TIME	Time to wait while stopped before retrying after an obstacle stop.	seconds; typical 2–8s.
— Turn Management —		
FRONT_TURN_TRIGGER	Entering TURN_INIT when front < threshold and lockout allows.	cm; typical 70–130.
TURN_DECISION_THRESHOLD	Side considered open if None/999 or > threshold.	cm; typical 60–150.
TURN_ANGLE_LEFT	Servo target to initiate left turn in TURN_INIT.	degrees; ~55–90° (servo scale).
TURN_ANGLE_RIGHT	Servo target to initiate right turn in TURN_INIT.	degrees; ~90–125° (servo scale).
FRONT_SAFE_DISTANCE	Front distance considered safe to end a turn (kept for reuse; not active).	cm; 120–200+.
SIDE_SAFE_DISTANCE	Side distance considered safe to end a turn (kept for reuse; not active).	cm; 20–60.
TARGET_DISTANCE	Desired distance from walls (for potential centering).	cm; typical ~30.
TURN_TIMEOUT	Maximum duration allowed for TURNING before force stop.	seconds; 2–6s.
TURN_LOCKOUT	Minimum interval between consecutive turns.	seconds; 0.5–3s.
POST_TURN_DURATION	Straight driving duration after a turn.	seconds; 0.2–1.5s.
LOCK_TURN_DIRECTION	Lock first valid turn direction and keep using it.	0 = off, 1 = on.
TARGET_TURN_ANGLE	Desired yaw change; reaching within tolerance ends the turn (Condition C).	degrees; 80–95°.
TURN_ANGLE_TOLERANCE	Allowed deviation from target to stop the turn.	degrees; 3–10°.

MIN_TURN_ANGLE	Minimum yaw change before permitting stop (not currently enforced).	degrees; 60–80°.
MAX_TURN_ANGLE	Maximum yaw change; exceeding forces stop (Condition D).	degrees; 100–130°.
— Laps —		
MAX_LAPS	Max laps before final stop (every 4 turns = 1 lap). 0 = unlimited.	Integer ≥ 0; 0 = ∞.
POST_LAP_DURATION	After last lap, drive forward briefly before stopping.	seconds; 0.5–2s.
— Sensor Filtering & Ranges —		
N_READINGS	Median window size for sensor samples.	Integer ≥ 1; 1–7 typical.
US_QUEUE_LEN	gpiozero.DistanceSensor queue length for ultrasonic.	Integer ≥ 1; default 5.
FILTER_ALPHA	EMA smoothing for ultrasonic after median.	0–1; 0.1 smooth, 1.0 none.
FILTER_JUMP	Max allowed jump between ultrasonic readings; bigger jumps treated as spikes.	cm; 10–200+, 999 disables.
FILTER_ALPHA_TOF	EMA smoothing for ToF after median.	0–1; same as above.
FILTER_JUMP_TOF	Max allowed jump between ToF readings before spike rejection.	cm; 10–200+, 999 disables.
US_MAX_DISTANCE_FRONT	Ultrasonic max distance (meters) for front sensor (library parameter).	meters; ~2–4m.
US_MAX_DISTANCE_SIDE	Ultrasonic max distance (meters) for side sensors.	meters; ~1–3m.
— Loop & Plotting —		
LOOP_DELAY	Main FSM loop delay; smaller = more responsive but more CPU.	seconds; 0.001–0.02.
SENSOR_DELAY	Sensor thread pacing; also used with sensor_tick.	seconds; 0.003–0.02.
— Config & Scaling Bases —		
CONFIG_FILE	External JSON used to override defaults at startup. Only keys present in globals() are applied.	Path string; default BASE_DIR/1st_mission_variables.json.

8.2 JSON Override Example

1st_mission_variables.json


```
{
  "USE_GUI": 0,
  "DEBUG": 0,
  "USE_TOF_SIDES": 0,
  "USE_TOF_FRONT": 0,
  "USE_TOF_CORNERS": 0,
  "NARROW_SUM_THRESHOLD": 60,
  "NARROW_HYSTERESIS": 10,
  "NARROW_FACTOR_SPEED": 0.6,
  "NARROW_FACTOR_DIST": 0.5,
  "SPEED_IDLE": 0,
  "SPEED_STOPPED": 0,
  "SPEED_CRUISE": 24,
  "SPEED_TURN_INIT": 16,
  "SPEED_TURN": 24,
  "SPEED_POST_TURN": 24,
  "SOFT_MARGIN": 26,
  "MAX_CORRECTION": 7,
  "CORR_EPS": 1.5,
  "CORRECTION_MULTIPLIER": 1.4,
  "STOP_THRESHOLD": 20,
  "OBSTACLE_WAIT_TIME": 5.0,
  "FRONT_TURN_TRIGGER": 90,
  "TURN_DECISION_THRESHOLD": 80,
  "TURN_ANGLE_LEFT": 61,
  "TURN_ANGLE_RIGHT": 119,
  "TURN_TIMEOUT": 2.5,
  "TURN_LOCKOUT": 1.5,
  "POST_TURN_DURATION": 0.5,
  "LOCK_TURN_DIRECTION": 1,
  "TARGET_TURN_ANGLE": 85,
  "TURN_ANGLE_TOLERANCE": 6,
  "MIN_TURN_ANGLE": 75,
  "MAX_TURN_ANGLE": 105,
  "MAX_LAPS": 3,
  "POST_LAP_DURATION": 0.85,
  "N_READINGS": 5,
  "US_QUEUE_LEN": 1,
  "FILTER_ALPHA": 1.0,
  "FILTER_JUMP": 9999,
  "FILTER_ALPHA_TOF": 1.0,
  "FILTER_JUMP_TOF": 9999,
  "US_MAX_DISTANCE_FRONT": 2.0,
  "US_MAX_DISTANCE_SIDE": 1.2,
  "LOOP_DELAY": 0.005,
  "SENSOR_DELAY": 0.01,
  "MAX_POINTS": 500
}
```

9 Appendix B - Source Code Analysis

9.1 Imports & global configuration

- **External Libraries**:
  - **gpiozero**: DistanceSensor for ultrasonic sensors (distance in meters)
  - **adafruit_pca9685**: PWM control for servos and motors (50 Hz frequency)
  - **adafruit_mpu6050**: Gyroscope (Z-axis in rad/s)
  - **adafruit_vl53l0x**: Time-of-Flight (ToF) sensors (distance in mm)
  - **board/digitalio**: I²C support and XSHUT pin control for individual sensor power cycling
  - **tkinter + matplotlib**: GUI and plots for real-time data visualization
  - **numpy**: Median and mean calculations for sensor data smoothing
  - **threading**: For running parallel tasks
  - **collections.deque**: To store and manage sensor readings efficiently
  - **csv**: For exporting data to CSV format
  - **json**: For reading/writing configuration files
  - **time**: For time-related operations and delays
- **Configuration Constants**:
  - Speed setpoints per FSM state: SPEED_*
  - Turn thresholds: FRONT_TURN_TRIGGER, TURN_DECISION_THRESHOLD
  - Safety guards: STOP_THRESHOLD, TURN_TIMEOUT, TURN_LOCKOUT, MAX_TURN_ANGLE
  - Yaw targets: TARGET_TURN_ANGLE, TURN_ANGLE_TOLERANCE
  - Narrow-mode thresholds: NARROW_SUM_THRESHOLD, NARROW_HYSTERESIS
  - Filtering: FILTER_ALPHA(_TOF), FILTER_JUMP(_TOF), N_READINGS
  - Timing: LOOP_DELAY, SENSOR_DELAY
  - Servo geometry and pulses: SERVO_*, 1000–2000 µs
- **Overrides**:
  - load_variables_from_json() runs immediately after defaults so all subsequent logic uses injected values. Only keys already present in globals() are applied.

9.2 Hardware Setup (GPIO/I²C, Pin Map)

- **GPIO Mode**: BCM
- **LEDs**: GPIO19 (green), GPIO13 (red)
- **Start Button**: GPIO20 (pull-up, active-LOW)
- **I²C Devices**:
  - PCA9685 (pca.frequency = 50)
  - MPU6050
  - Optional VL53L0X sensors
- **Ultrasonic Pins**:
  - Front: TRIG22, ECHO23
  - Left: 27, 17

  - Right: 5, 6
  - via gpiozero.DistanceSensor (distance in meters, max_distance set per front/side)
- **PWM Channels**:
  - Servo on PCA channel 0
  - Motor FWD/REV on PCA channels 1/2
- **IMU Bias**:
  - 500 samples averaged at boot to compute Z-gyro bias

9.3 Sensor Initialization (ToF w/ XSHUT, Ultrasonic Fallback)

- **ToF Addressing**:
  - Per-sensor XSHUT pins (D16/D25/D26/D24) power-cycle and set unique I²C addresses (0x30–0x33)
  - Start continuous with a 20 ms budget for each sensor
- **Fallback**:
  - Any ToF initialization exception disables ToF flags and uses ultrasonics for the affected directions
- **Mixed Mode**:
  - Front/side ToF enablement is independent. For example, you can run ultrasonic on sides and ToF on the front (or vice-versa).

9.4 Data Model & Threading Primitives

- **Shared Sensor Snapshot**:
  - sensor_data = {"front":..., "left":..., "right":...} guarded by a Lock
- **Events**:
  - readings_event (sensors armed)
  - loop_event (FSM armed)
  - sensor_tick (timed wait with immediate wake for quick responsiveness)
- **Telemetry**:
  - Deques (time_data, front/left/right_data, angle_data, state_data) with MAX_POINTS cap for plots/CSV export

9.5 RobotController (Low-Level Actuation & Signal Processing)

- **Servo Command**:
  - set_servo(angle) clamps to [SERVO_MIN_ANGLE, SERVO_MAX_ANGLE]
  - Pulse conversion:
    - pulse_us = SERVO_PULSE_MIN + (SERVO_PULSE_MAX - SERVO_PULSE_MIN) * (angle - MIN) / (MAX - MIN) → PCA duty (0..65535)
- **Motor Command**:
  - rotate_motor(speed) maps 0–100% to PCA duty; selects FWD vs REV channel by sign
  - stop_motor() zeros both PWM channels
- **Distance Filtering**:
  - filtered_distance(sensor, history, attr, sensor_type) applies median over last N_READINGS, rejects spikes greater than |FILTER_JUMP| vs previous readings, and applies an Exponential Moving Average (EMA):
    - $y = \alpha \cdot x + (1-\alpha) \cdot y_{prev}$
  - Returns smoothed distance in cm or 999 sentinel for invalid data

- **Lane-Keeping**:
  - safe_straight_control(l,r) computes bounded correction using eff_soft_margin() and eff_max_correction() (scaled by narrow-mode factors), returns a target servo angle.
- **Turn Helper**:
  - turn_decision(l,r) → "LEFT", "RIGHT", or None using XOR (open_left, open_right - only one Input is True) where “open” is (None or 999 or > TURN_DECISION_THRESHOLD).

9.6 Real-Time Threads

- **sensor_reader (period ≈ SENSOR_DELAY)**:
  - Acquires left/right/front sensor data (either ToF or US), applies filters, writes to sensor_data under Lock
  - Sleeps with sensor_tick.wait(SENSOR_DELAY) (can be preemptively woken)
- **robot_loop (soft-real-time main loop)**:
  - If readings_event is not set → hold IDLE state.
  - On each tick: copy snapshot, update yaw, narrow-mode, telemetry, then run FSM if loop_event is set.

9.7 Yaw Integration & Units

- **IMU Input**:
  - gyro_z in rad/s, bias-corrected, low-passed (ALPHA), integrated to degrees:
    - yaw += (gyro_z_filtered * RAD2DEG) * dt, where RAD2DEG = 180/π, dt from monotonic_ns()

9.8 Finite State Machine (FSM)

- **States**:
- IDLE → CRUISE → TURN_INIT → TURNING → POST_TURN → STOPPED
- **Key Transitions & Guards**:
- **Emergency Stop (Anywhere)**: front < eff_stop_threshold() → STOPPED, schedule auto-retry for OBSTACLE_WAIT_TIME
- **Turn Eligibility (CRUISE → TURN_INIT)**: front < eff_front_turn_trigger() and (now − last_turn_time) ≥ TURN_LOCKOUT
- **Direction Selection (TURN_INIT)**: Require XOR (open_left, open_right - only one Input is True); if LOCK_TURN_DIRECTION=1, fix first chosen direction for the run; else, keep waiting at SPEED_TURN_INIT
- **Turn Commit (→ TURNING)**: Set servo to TURN_ANGLE_LEFT/RIGHT, record turn_start_yaw/time, run at SPEED_TURN
- **Turn Termination (TURNING → POST_TURN)**:
- Primary: |Δyaw − TARGET_TURN_ANGLE| ≤ TURN_ANGLE_TOLERANCE
- Failsafes: (now − turn_start_time) > TURN_TIMEOUT or |Δyaw| ≥ MAX_TURN_ANGLE
- On stop: center servo, last_turn_time = now, optionally snap yaw to exact target for continuity, increment turn_count; every 4 turns → lap_count++ (stop after MAX_LAPS with POST_LAP_DURATION forward).
- **Post-Turn (POST_TURN → CRUISE)**: Hold straight for POST_TURN_DURATION.

9.9 Environment Adaptation (Narrow-Mode)

- **Detector**:

  - Compute sum_lr = left + right for valid readings; enter if < NARROW_SUM_THRESHOLD, exit if > threshold + NARROW_HYSTERESIS
- **Scaling**:
  - Set SPEED_ENV_FACTOR and DIST_ENV_FACTOR; all eff_* helpers (e.g., eff_soft_margin(), eff_front_turn_trigger()) and state_speed_value() consume these to adapt behavior without changing base constants.

9.10 GUI Subsystem (For Debugging)

- **Tkinter UI**:
  - Buttons (Start Readings, Start Loop, Stop Loop), status label, circular state indicator, turn/lap counters.
- **Plots**:
  - 3 Matplotlib subplots (front, side [left vs −right], yaw); dynamic x-window, last-value annotations, and after() updates (100–200 ms)
- **Sliders**:
  - Grouped controls for speeds, margins, triggers, timeouts; write through to globals()
  - Save/load JSON presets
- **Export**:
  - export_data_csv() writes time series + current state + slider snapshot for reproducibility
- **Shutdown**:
  - on_closing() calls stop_loop(), stops ToF continuous mode, cleanup

9.11 Start/Stop & Lifecycle

- **Handlers**:
  - start_readings() spawns sensor_reader (daemon) and robot_loop (daemon) once
  - start_loop() sets loop_event
  - stop_loop() clears loop, records stop_reason, resets counters, unlocks direction, centers servo, stops motor
- **Headless Path (__main__)**:
  - LEDs signal readiness; wait for physical Start (active-LOW); then set events and run robot_loop() in the foreground.
  - KeyboardInterrupt and GUI close lead to clean shutdown.

9.12 Synchronization & Timing

- **Progress Gating**:
  - readings_event must be set before any motion
  - loop_event controls whether FSM drives motors (plots still update while off)
- **Wait Strategy**:
  - sensor_tick.wait(…) used instead of time.sleep() so stop_loop() can set() to break out immediately
- **Typical Periods**:
  - SENSOR_DELAY ≈ 5 ms (up to ~200 Hz raw read cadence)
  - Main loop LOOP_DELAY ≈ 1 ms cap (effective rate bounded by processing and GUI)

9.13 Safety/Fallbacks

- **Bounds**:
  - All servo angles clamped
  - Motor duty bounded to 0–100%
- **Guards**:
  - Emergency stop, turn timeout, max turn angle, turn lockout
  - Narrow-mode scaling to reduce risk in tight corridors
- **Sensor Resilience**:
  - Jump-rejection, EMA smoothing, 999 sentinel for invalid
  - Automatic ToF→US fallback on init failure

9.14 Thread Summary

- **sensor_reader** — acquisition + filtering (median → spike reject → EMA) → publish under Lock
- **robot_loop** — yaw integration (rad/s → deg), narrow-mode, FSM, PWM outputs via PCA9685, telemetry append
- **GUI (optional)** — plotting, parameter sliders with live writes, CSV export, start/stop controls

Events: readings_event, loop_event, sensor_tick avoid busy-waiting and keep latency low.

10 Appendix C - Source code

```
0010: #-----------------------------------------------------------------------------------------------
0011: # 1st Mission WRO 2025 FE - VivaLaVida
0012: # Final Version
0013: #-----------------------------------------------------------------------------------------------
0014:
0015: # ================================
0016: # IMPORTS
0017: # ================================
0018:
0019:
0020: import os, sys
0021:
0022: VENV_PY = "/home/stem/env/bin/python"  # exact path to Python in your venv
0023:
0024: if sys.executable != VENV_PY and os.path.exists(VENV_PY):
0025:     print(f"[INFO] Relaunching under virtual environment: {VENV_PY}", flush=True)
0026:     os.execv(VENV_PY, [VENV_PY] + sys.argv)
0027:
0028: # --- your normal robot code below ---
0029: print(f"[INFO] Now running inside: {sys.executable}")
0030:
0031:
0032: import threading
0033: import time
0034: from collections import deque
0035: import busio
0036: from adafruit_pca9685 import PCA9685
0037: import adafruit_mpu6050
0038: import csv
0039: from datetime import datetime
0040: import json
0041:
0042: import board
0043: import digitalio
0044: import adafruit_vl53l0x
0045: from enum import Enum, auto
0046: import numpy as np
0047: from gpiozero import Device
0048: from gpiozero.pins.lgpio import LGPIOFactory   # Uses /dev/gpiochip*
0049: Device.pin_factory = LGPIOFactory()
0050: from gpiozero import Button, LED
0051: from gpiozero import DistanceSensor
0052: from threading import Event
0053:
0054: BASE_DIR = os.path.dirname(os.path.abspath(__file__)) # Get directory of the running script
0055: CONFIG_FILE = os.path.join(BASE_DIR, "1st_mission_variables.json")
0056:
0057: # ================================
0058: # CONFIGURATION VARIABLES
0059: # ================================
0060:
0061: # ---------- Initialization ----------
0062: USE_GUI = 0                   # 1 = Debugging mode (run with GUI), 0 = COMPETITION MODE (run
headless, no GUI)
0063: DEBUG = 0                     # 1 = enable prints, 0 = disable all prints
0064: USE_TOF_SIDES = 0             # Side sensors:  0 = Ultrasonic, 1 = ToF
0065: USE_TOF_FRONT = 0             # Front sensor:  0 = Ultrasonic, 1 = ToF
0066: USE_TOF_CORNERS = 0           # Front side sensor: 0 = disable, 1 = enable
0067:
0068: NARROW_SUM_THRESHOLD = 60     # cm; left+right distance threshold to decide if in between
narrow walls
0069: NARROW_HYSTERESIS = 10        # cm; prevents rapid toggling
0070: NARROW_FACTOR_SPEED = 0.6     # multiply state speeds when in narrow corridors
0071: NARROW_FACTOR_DIST  = 0.5     # multiply distance-based thresholds/corrections when in narrow
corridors
0072:
0073: # ---------- Speeds ----------
0074: SPEED_IDLE = 0
0075: SPEED_STOPPED = 0
0076: SPEED_CRUISE = 24             # Motor speed for normal straight driving (0-100%)
0077: SPEED_TURN_INIT = 16          # Motor speed while waiting for open side to turn
0078: SPEED_TURN = 24               # Motor speed while turning
0079: SPEED_POST_TURN = 24          # Motor speed following a turn
```

```
0080:
0081: # ---------- Driving ----------
0082: SOFT_MARGIN = 26              # Distance from wall where small steering corrections start (cm)
0083: MAX_CORRECTION = 7            # Maximum servo correction applied for wall-following (degrees)
0084: CORR_EPS = 1.5                # cm: treat the side as "steady" if within ±1.5 cm of trigger
value
0085: CORRECTION_MULTIPLIER = 1.4   # Proportional gain (servo degrees per cm of error 0 default:
2). Higher = snappier; lower = smoother&slower.
0086:
0087: STOP_THRESHOLD = 20           # Front distance (cm) at which robot stops immediately
0088: OBSTACLE_WAIT_TIME = 5.0      # seconds to wait before retrying after a front-stop
0089:
0090: # ---------- Turn management ----------
0091: FRONT_TURN_TRIGGER = 90       # Front distance (cm) at which a turn is triggered
0092: TURN_DECISION_THRESHOLD = 80  # Minimum side distance (cm) to allow turn in that direction
0093: TURN_ANGLE_LEFT = 61          # Servo angle for left turn
0094: TURN_ANGLE_RIGHT = 119        # Servo angle for right turn
0095: TURN_TIMEOUT = 2.5            # Maximum time allowed for a turn (seconds)
0096: TURN_LOCKOUT = 1.5            # Minimum interval between consecutive turns (seconds)
0097: POST_TURN_DURATION = 0.5      # Time to drive straight after a turn (seconds)
0098: LOCK_TURN_DIRECTION = 1       # 1 = enable turn lock direction after 1st turn, 0 = disable
0099: TARGET_TURN_ANGLE = 85        # Degrees to turn per corner
0100: TURN_ANGLE_TOLERANCE = 6      # Acceptable overshoot (degrees)
0101:
0102: # ---------- Turn angle constraints ----------
0103: MIN_TURN_ANGLE = 75           # Minimum yaw change (degrees) before turn can stop
0104: MAX_TURN_ANGLE = 105          # Maximum yaw change (degrees) to force stop turn
0105:
0106: # ---------- Laps ----------
0107: MAX_LAPS = 3                  # Maximum number of laps before stopping (0 = unlimited)
0108: POST_LAP_DURATION = 0.85        # Time to drive forward after final lap before stopping
(seconds)
0109:
0110: # ---------- Sensor data filtering ----------
0111: N_READINGS = 5                # Number of readings stored for median filtering
0112: US_QUEUE_LEN = 1              # Queue_len for Ultrasonic gpiozero.DistanceSensor
0113: FILTER_ALPHA = 1.0            # Ultrasonic 0.1 = smoother, 0.5 = faster reaction, 1 to ignore
filter
0114: FILTER_JUMP = 9999            # Ultrasonic maximum jump (cm) allowed between readings
0115: FILTER_ALPHA_TOF = 1.0        # ToF 0.1 = smoother, 0.5 = faster reaction, 1 to ignore filter
0116: FILTER_JUMP_TOF = 9999        # ToF maximum jump (cm) allowed between readings
0117:
0118: US_MAX_DISTANCE_FRONT = 2.0  # Ultrasonic front max read distance
0119: US_MAX_DISTANCE_SIDE = 1.2   # Ultrasonic side max read distance
0120:
0121: # ---------- Loop timing ----------
0122: LOOP_DELAY = 0.005           # Delay between main loop iterations (seconds) 0.02
0123: SENSOR_DELAY = 0.01          # Delay between sensor reads
0124:
0125: #-------------------------------------------------------------------------------------------
0126:
0127: # ---------- Plotting ----------
0128: MAX_POINTS = 500              # Maximum number of points to store for plotting
0129:
0130: RAD2DEG = 180.0 / 3.141592653589793
0131:
0132: # --- I2C concurrency guard ---
0133: I2C_LOCK = threading.Lock()
0134:
0135: def dprint(*args, **kwargs): #Conditional print — only prints when DEBUG == 1.
0136:     if DEBUG:
0137:         print(*args, **kwargs)
0138:
0139: # --- Headless-safe placeholders for GUI symbols ---
0140: slider_vars = {}
0141: btn_readings = btn_start = btn_stop = None
0142: lbl_turns = lbl_laps = None
0143: root = canvas = None
0144:
0145: # ================================
0146: # LOAD VARIABLES FROM JSON (if exists)
0147: # ================================
0148:
0149: def load_variables_from_json(path=CONFIG_FILE): # Override defaults with values from JSON if
the file exists.
0150:     if not os.path.exists(path):
0151:         print("⚙️ No external config found — using built-in defaults.")
```

```
0152:         return
0153:     try:
0154:          with open(path, "r") as f:
0155:               data = json.load(f)
0156:          count = 0
0157:          for k, v in data.items():
0158:              if k in globals():
0159:                  globals()[k] = v
0160:                  count += 1
0161:          print(f"✅ Loaded {count} variables from {os.path.basename(path)}")
0162:     except Exception as e:
0163:          print(f"[ERROR] Config load failed: {e} — using built-in defaults.")
0164:
0165: load_variables_from_json() # Call it right after defining defaults so everything below sees
the overrides
0166:
0167: if os.environ.get("DISPLAY", "") == "": # If there is no X server, force headless regardless
of config
0168:     USE_GUI = 0
0169:
0170: def norm180(a: float) -> float: # Normalize angle to [-180, 180)
0171:     return (a + 180.0) % 360.0 - 180.0
0172:
0173: def snap90(a: float) -> float: # Nearest multiple of 90° (…,-180,-90,0,90,180,…)
0174:     return round(a / 90.0) * 90.0
0175:
0176: # --- Store immutable base values for scaling ---
0177: BASE_SPEED_IDLE         = SPEED_IDLE
0178: BASE_SPEED_STOPPED      = SPEED_STOPPED
0179: BASE_SPEED_CRUISE       = SPEED_CRUISE
0180: BASE_SPEED_TURN_INIT    = SPEED_TURN_INIT
0181: BASE_SPEED_TURN         = SPEED_TURN
0182: BASE_SPEED_POST_TURN    = SPEED_POST_TURN
0183:
0184: BASE_MAX_CORRECTION     = MAX_CORRECTION
0185: BASE_FRONT_TURN_TRIGGER = FRONT_TURN_TRIGGER
0186: BASE_STOP_THRESHOLD     = STOP_THRESHOLD
0187:
0188: # --- Global factors (automatically updated) ---
0189: SPEED_ENV_FACTOR = 1.0
0190: DIST_ENV_FACTOR  = 1.0  # for distance-related thresholds
0191:
0192: # --- Helper functions to get "effective" (scaled) values ---
0193: def eff_soft_margin(): #Return scaled soft margin distance (cm) for gentle steering
corrections
0194:     return int(SOFT_MARGIN * DIST_ENV_FACTOR)
0195:
0196: def eff_max_correction(): #Return scaled maximum steering correction (degrees)
0197:     return max(1, int(MAX_CORRECTION * DIST_ENV_FACTOR))
0198:
0199: def eff_front_turn_trigger(): #Return scaled front distance threshold (cm) for initiating a
turn
0200:     return int(FRONT_TURN_TRIGGER * DIST_ENV_FACTOR)
0201:
0202: def eff_stop_threshold(): #Return scaled front distance threshold (cm) for immediate stop
0203:     return int(STOP_THRESHOLD * DIST_ENV_FACTOR)
0204:
0205: # --- Scaled state speeds ---
0206: def state_speed_value(state_name: str) -> int:
0207:     base = {
0208:         "IDLE":      SPEED_IDLE,
0209:         "STOPPED":   SPEED_STOPPED,
0210:         "CRUISE":    SPEED_CRUISE,
0211:         "TURN_INIT": SPEED_TURN_INIT,
0212:         "TURNING":   SPEED_TURN,
0213:         "POST_TURN": SPEED_POST_TURN,
0214:     }.get(state_name, SPEED_CRUISE)
0215:     return int(base * SPEED_ENV_FACTOR)
0216:
0217: def headless_toggle_pause(): #Toggle pause/resume of the driving loop in headless mode
0218:     global paused_by_button, status_text
0219:     if loop_event.is_set(): #Pause
0220:         loop_event.clear()
0221:         paused_by_button = True
0222:         robot.stop_motor()
0223:         robot.set_servo(SERVO_CENTER)
0224:         RED_LED.off()      # paused
```

```
0225:         GREEN_LED.on()
0226:         status_text = "Paused (button)"
0227:     else:  # RESUME
0228:         paused_by_button = False
0229:         loop_event.set()
0230:         GREEN_LED.off()   # running
0231:         RED_LED.on()
0232:         status_text = "🚗 Resumed"
0233:
0234: # ================================
0235: # HARDWARE SETUP
0236: # ================================
0237:
0238: i2c = busio.I2C(board.SCL, board.SDA)
0239:
0240: # ---------- Button & LEDs (gpiozero-based) ----------
0241: START_BTN = Button(20, pull_up=True, bounce_time=0.03)
0242: GREEN_LED = LED(19)
0243: RED_LED   = LED(13)
0244: # ✅ Turn RED on as soon as the program starts
0245: RED_LED.on()
0246:
0247: # ---------- Ultrasonic sensor pins ----------
0248: TRIG_FRONT, ECHO_FRONT = 22, 23  # GPIO pins for front sensor
0249: TRIG_LEFT, ECHO_LEFT = 27, 17    # GPIO pins for left sensor
0250: TRIG_RIGHT, ECHO_RIGHT = 5, 6    # GPIO pins for right sensor
0251:
0252: # ---------- Motor ----------
0253: MOTOR_FWD = 1                 # PCA9685 channel for forward motor control
0254: MOTOR_REV = 2                 # PCA9685 channel for reverse motor control
0255:
0256: # ---------- Servo ----------
0257: SERVO_CHANNEL = 0             # PCA9685 channel controlling steering servo
0258: SERVO_CENTER = 90             # Neutral servo angle (degrees)
0259: SERVO_MIN_ANGLE = 50          # Minimum physical servo angle (degrees)
0260: SERVO_MAX_ANGLE = 130         # Maximum physical servo angle (degrees)
0261: SERVO_PULSE_MIN = 1000        # Minimum PWM pulse width (microseconds)
0262: SERVO_PULSE_MAX = 2000        # Maximum PWM pulse width (microseconds)
0263: SERVO_PERIOD = 20000          # Servo PWM period (microseconds)
0264:
0265: # ================================
0266: # SENSOR INITIALIZATION (ToF + Ultrasonic)
0267: # ================================
0268:
0269: vl53_left = vl53_right = vl53_front = vl53_back = None
0270: vl53_front_left = vl53_front_right = None
0271: us_left = us_right = us_front = None
0272:
0273: try:
0274:     # --------- ToF sensors ----------
0275:     if USE_TOF_SIDES or USE_TOF_FRONT or USE_TOF_CORNERS:
0276:         print("Initializing VL53L0X ToF sensors...")
0277:         # XSHUT pins setup (for powering sensors individually)
0278:         xshut_left  = digitalio.DigitalInOut(board.D16)
0279:         xshut_right = digitalio.DigitalInOut(board.D25)
0280:         xshut_front = digitalio.DigitalInOut(board.D26)
0281:         xshut_back  = digitalio.DigitalInOut(board.D8)
0282:         xshut_front_left  = digitalio.DigitalInOut(board.D7)
0283:         xshut_front_right = digitalio.DigitalInOut(board.D24)
0284:         for xshut in [xshut_left, xshut_right, xshut_front, xshut_back,
0285:                       xshut_front_left, xshut_front_right]:
0286:             xshut.direction = digitalio.Direction.OUTPUT
0287:             xshut.value = False  # power down
0288:         time.sleep(0.1)
0289:
0290:         # Initialize left ToF if needed
0291:         if USE_TOF_SIDES:
0292:             xshut_left.value = True
0293:             time.sleep(0.05)
0294:             vl53_left = adafruit_vl53l0x.VL53L0X(i2c)
0295:             vl53_left.set_address(0x30)
0296:             vl53_left.measurement_timing_budget = 20000
0297:             vl53_left.start_continuous()
0298:             print("✅ Left ToF sensor set to address 0x30")
0299:
0300:         # Initialize right ToF if needed
0301:         if USE_TOF_SIDES:
```

```
0302:             xshut_right.value = True
0303:             time.sleep(0.05)
0304:             vl53_right = adafruit_vl53l0x.VL53L0X(i2c)
0305:             vl53_right.set_address(0x31)
0306:             vl53_right.measurement_timing_budget = 20000
0307:             vl53_right.start_continuous()
0308:             print("✅ Right ToF sensor set to address 0x31")
0309:
0310:         # Initialize front ToF if needed
0311:         if USE_TOF_FRONT:
0312:             xshut_front.value = True
0313:             time.sleep(0.05)
0314:             vl53_front = adafruit_vl53l0x.VL53L0X(i2c)
0315:             vl53_front.set_address(0x32)
0316:             vl53_front.measurement_timing_budget = 20000
0317:             vl53_front.start_continuous()
0318:             print("✅ Front ToF sensor set to address 0x32")
0319:
0320:         # Initialize corner/front-left ToF (optional)
0321:         if USE_TOF_CORNERS:
0322:             xshut_front_left.value = True
0323:             time.sleep(0.05)
0324:             vl53_front_left = adafruit_vl53l0x.VL53L0X(i2c)
0325:             vl53_front_left.set_address(0x34)
0326:             vl53_front_left.measurement_timing_budget = 20000
0327:             vl53_front_left.start_continuous()
0328:             print("✅ Front-Left ToF sensor set to address 0x34")
0329:
0330:         # Initialize corner/front-right ToF (optional)
0331:         if USE_TOF_CORNERS:
0332:             xshut_front_right.value = True
0333:             time.sleep(0.05)
0334:             vl53_front_right = adafruit_vl53l0x.VL53L0X(i2c)
0335:             vl53_front_right.set_address(0x35)
0336:             vl53_front_right.measurement_timing_budget = 20000
0337:             vl53_front_right.start_continuous()
0338:             print("✅ Front-Right ToF sensor set to address 0x35")
0339:
0340:         # Initialize back ToF (optional, if you want to use it)
0341:         xshut_back.value = True
0342:         time.sleep(0.05)
0343:         vl53_back = adafruit_vl53l0x.VL53L0X(i2c)
0344:         vl53_back.set_address(0x33)
0345:         vl53_back.measurement_timing_budget = 20000
0346:         vl53_back.start_continuous()
0347:         print("✅ Back ToF sensor set to address 0x33")
0348:
0349: except Exception as e:
0350:     print(f"[ERROR] VL53L0X initialization failed: {e}")
0351:     # fallback to ultrasonic if ToF fails
0352:     USE_TOF_SIDES = 0
0353:     USE_TOF_FRONT = 0
0354:     USE_TOF_CORNERS = 0
0355:     vl53_left = vl53_right = vl53_front = vl53_back = None
0356:     vl53_front_left = vl53_front_right = None
0357:
0358: time.sleep(0.2)
0359:
0360: # --------- Ultrasonic sensors ----------
0361: # Initialize ultrasonic sensors only if that sensor is set to ultrasonic
0362: if not USE_TOF_FRONT:
0363:     us_front = DistanceSensor(echo=ECHO_FRONT, trigger=TRIG_FRONT,
max_distance=US_MAX_DISTANCE_FRONT, queue_len=US_QUEUE_LEN)
0364: if not USE_TOF_SIDES:
0365:     us_left  = DistanceSensor(echo=ECHO_LEFT,  trigger=TRIG_LEFT,
max_distance=US_MAX_DISTANCE_SIDE, queue_len=US_QUEUE_LEN)
0366:     us_right = DistanceSensor(echo=ECHO_RIGHT, trigger=TRIG_RIGHT,
max_distance=US_MAX_DISTANCE_SIDE, queue_len=US_QUEUE_LEN)
0367:
0368: dprint("Sensors initialized:")
0369: dprint(f"  Front: {'ToF' if USE_TOF_FRONT else 'Ultrasonic'}")
0370: dprint(f"  Left : {'ToF' if USE_TOF_SIDES else 'Ultrasonic'}")
0371: dprint(f"  Right: {'ToF' if USE_TOF_SIDES else 'Ultrasonic'}")
0372: if USE_TOF_CORNERS:
0373:     dprint("  Turn corners: ToF (front_left & front_right)")
0374:
```

```
0375: pca = PCA9685(i2c)
0376: pca.frequency = 50
0377: mpu = adafruit_mpu6050.MPU6050(i2c)
0378:
0379: # ---------- Calibrate gyro bias at startup ----------
0380: dprint("Calibrating gyro...")
0381: N = 500
0382: bias = 0
0383: for _ in range(N):
0384:     with I2C_LOCK:
0385:         bias += mpu.gyro[2]
0386:     time.sleep(0.005)
0387: bias /= N
0388: dprint(f"Gyro bias: {bias}")
0389:
0390: # ================================
0391: # CONTROL FLAGS & STORAGE
0392: # ================================
0393:
0394: STATE_SPEED = {
0395:     "IDLE": SPEED_IDLE,
0396:     "STOPPED": SPEED_STOPPED,
0397:     "CRUISE": SPEED_CRUISE,
0398:     "TURN_INIT": SPEED_TURN_INIT,
0399:     "TURNING": SPEED_TURN,
0400:     "POST_TURN": SPEED_POST_TURN
0401: }
0402:
0403: # ---------- CONTROL FLAGS ----------
0404: readings_event = Event()
0405: loop_event = Event()
0406: sensor_tick = Event()   # used to wake waits immediately
0407: shutdown_event = Event()
0408: status_text = "Idle"
0409: turn_count = 0
0410: lap_count = 0
0411: stop_reason = None # Reason-aware stopping (USER / OBSTACLE / LAPS)
0412: obstacle_wait_deadline = 0.0
0413: # --- Headless START button toggle / debounce ---
0414: BTN_DEBOUNCE_S = 0.30   # seconds
0415: _btn_prev = 1           # pull-up idle state (button not pressed)
0416: _btn_last_ts = 0.0
0417: paused_by_button = False
0418:
0419: # ---------- SENSOR DATA STORAGE ----------
0420: time_data = deque(maxlen=MAX_POINTS)
0421: front_data = deque(maxlen=MAX_POINTS)
0422: left_data = deque(maxlen=MAX_POINTS)
0423: right_data = deque(maxlen=MAX_POINTS)
0424: front_left_data  = deque(maxlen=MAX_POINTS)
0425: front_right_data  = deque(maxlen=MAX_POINTS)
0426: angle_data = deque(maxlen=MAX_POINTS)
0427: state_data = deque(maxlen=MAX_POINTS)
0428:
0429: # ================================
0430: # THREADED SENSOR READING
0431: # ================================
0432: sensor_data = {
0433:     "front": None,
0434:     "left": None,
0435:     "right": None,
0436:     "front_left": None,
0437:     "front_right": None,
0438: }
0439: sensor_lock = threading.Lock()
0440:
0441: # ================================
0442: # ROBOT CONTROLLER
0443: # ================================
0444:
0445: class RobotController:
0446:     def __init__(self, pca):
0447:         self.pca = pca
0448:         self.front_history = deque(maxlen=N_READINGS)
0449:         self.left_history = deque(maxlen=N_READINGS)
0450:         self.right_history = deque(maxlen=N_READINGS)
0451:         self.front_left_history = deque(maxlen=N_READINGS)
```

```
0452:         self.front_right_history = deque(maxlen=N_READINGS)
0453:         self.smooth_left = None
0454:         self.smooth_right = None
0455:         self.smooth_front = None
0456:         self.smooth_front_left = None
0457:         self.smooth_front_right = None
0458:         self.d_front = None
0459:         self.d_left = None
0460:         self.d_right = None
0461:         self.d_front_left = None
0462:         self.d_front_right = None
0463:         self.sensor_index = 0
0464:         self.gyro_z_prev = 0
0465:         self._servo_last_angle = SERVO_CENTER
0466:         self._servo_last_ns = time.monotonic_ns()
0467:         # -- Simple safe-straight latch --
0468:         self.ss = 0          # 0 = inactive, +1 = left triggered, -1 = right triggered
0469:         self.ss_base = None  # distance at trigger
0470:
0471:     def ss_reset(self):
0472:         self.ss = 0
0473:         self.ss_base = None
0474:
0475:     def set_filter_size(self, n):
0476:         self.front_history = deque(list(self.front_history)[-n:], maxlen=n)
0477:         self.left_history = deque(list(self.left_history)[-n:], maxlen=n)
0478:         self.right_history = deque(list(self.right_history)[-n:], maxlen=n)
0479:         self.front_left_history = deque(list(self.front_left_history)[-n:], maxlen=n)
0480:         self.front_right_history = deque(list(self.front_right_history)[-n:], maxlen=n)
0481:
0482:     def set_servo(self, angle):
0483:         # clamp to physical limits
0484:         target = max(SERVO_MIN_ANGLE, min(SERVO_MAX_ANGLE, angle))
0485:         # angle → pulse (µs)
0486:         pulse = int(
0487:             SERVO_PULSE_MIN
0488:             + (SERVO_PULSE_MAX - SERVO_PULSE_MIN)
0489:             * ((target - SERVO_MIN_ANGLE) / (SERVO_MAX_ANGLE - SERVO_MIN_ANGLE))
0490:         )
0491:         # write to PCA9685
0492:         with I2C_LOCK:
0493:             self.pca.channels[SERVO_CHANNEL].duty_cycle = int(pulse * 65535 / SERVO_PERIOD)
0494:         return target
0495:
0496:     def rotate_motor(self, speed):
0497:         duty_cycle = int(min(max(abs(speed), 0), 100)/100*65535)
0498:         with I2C_LOCK:
0499:             if speed >= 0:
0500:                 self.pca.channels[MOTOR_FWD].duty_cycle = duty_cycle
0501:                 self.pca.channels[MOTOR_REV].duty_cycle = 0
0502:             else:
0503:                 self.pca.channels[MOTOR_FWD].duty_cycle = 0
0504:                 self.pca.channels[MOTOR_REV].duty_cycle = duty_cycle
0505:
0506:     def stop_motor(self):
0507:         with I2C_LOCK:
0508:             self.pca.channels[MOTOR_FWD].duty_cycle = 0
0509:             self.pca.channels[MOTOR_REV].duty_cycle = 0
0510:
0511:     def set_state_speed(self, state):
0512:         # Set motor speed based on FSM state.
0513:         speed = state_speed_value(state)
0514:         self.rotate_motor(speed)
0515:
0516:     def get_distance(self, sensor):
0517:         #sensor: a gpiozero.DistanceSensor instance
0518:         try:
0519:             d = sensor.distance  # returns meters
0520:             if d is None:
0521:                 return None
0522:             return d * 100  # convert to cm
0523:         except:
0524:             return None
0525:
0526:     def stable_filter(self, new_val, prev_val, alpha=FILTER_ALPHA, max_jump=FILTER_JUMP):
0527:     #Reject large spikes and apply exponential moving average smoothing.
0528:         if new_val is None:
```

```
0529:             return prev_val  # keep previous if invalid
0530:         if prev_val is not None and abs(new_val - prev_val) > max_jump:
0531:             new_val = prev_val  # reject spike
0532:         if prev_val is None:
0533:             return new_val
0534:         return alpha * new_val + (1 - alpha) * prev_val
0535:
0536:     def filtered_distance(self, sensor_obj, history, smooth_attr, sensor_type='us'):
0537:         # Fallback to use when ToF is invalid or too far (use US side max, in cm)
0538:         TOF_FALLBACK_CM = US_MAX_DISTANCE_SIDE * 100.0  # 1.2 m -> 120 cm
0539:         try:
0540:             if sensor_type == 'tof':
0541:                 with I2C_LOCK:
0542:                     raw_mm = sensor_obj.range  # VL53L0X returns mm
0543:                 d = (raw_mm / 10.0) if raw_mm is not None else None  # -> cm
0544:                 invalid_far = (d is None) or (d >= 800.0)            # ≥ 8 m or None
0545:                 d_for_history = None if invalid_far else d           # keep filter clean
0546:             else:
0547:                 d = sensor_obj.distance * 100.0   # meters -> cm
0548:                 invalid_far = False
0549:                 d_for_history = d
0550:         except:
0551:             d = None
0552:             invalid_far = (sensor_type == 'tof')
0553:             d_for_history = None
0554:
0555:         # Update history with valid values only (for stable median/EMA)
0556:         history.append(d_for_history)
0557:         valid = [x for x in history if x is not None]
0558:
0559:         # If ToF is invalid/too far, immediately report the side max distance (120 cm)
0560:         if sensor_type == 'tof' and invalid_far:
0561:             return TOF_FALLBACK_CM
0562:
0563:         if not valid:
0564:             # No valid history yet -> for ToF use side max, else keep your sentinel
0565:             return TOF_FALLBACK_CM if sensor_type == 'tof' else 999
0566:
0567:         median_val = np.median(valid)
0568:         # avg_val = np.mean(valid)  # unused
0569:         filtered_val = median_val
0570:
0571:         prev_val = getattr(self, smooth_attr)
0572:         alpha = FILTER_ALPHA_TOF if sensor_type == 'tof' else FILTER_ALPHA
0573:         max_jump = FILTER_JUMP_TOF if sensor_type == 'tof' else FILTER_JUMP
0574:         smoothed_val = self.stable_filter(filtered_val, prev_val, alpha, max_jump)
0575:
0576:         setattr(self, smooth_attr, smoothed_val)
0577:         return smoothed_val
0578:
0579:     def safe_straight_control(self, d_left, d_right):
0580:         # Treat 999/None as no data
0581:         d_left  = None if (d_left  is None or d_left  >= 900) else d_left
0582:         d_right = None if (d_right is None or d_right >= 900) else d_right
0583:
0584:         m = eff_soft_margin()
0585:         max_corr = eff_max_correction()
0586:
0587:         # -------- ARM if inactive (inside margin by more than CORR_EPS) --------
0588:         if self.ss == 0:
0589:             if d_left is not None and d_left < (m - CORR_EPS):
0590:                 self.ss = +1; self.ss_base = d_left
0591:             elif d_right is not None and d_right < (m - CORR_EPS):
0592:                 self.ss = -1; self.ss_base = d_right
0593:             else:
0594:                 return SERVO_CENTER
0595:
0596:         # -------- ACTIVE: correct only away from the triggering side --------
0597:         d = d_left if self.ss > 0 else d_right
0598:
0599:         # Release if sensor lost or clearly recovered beyond margin
0600:         if d is None or d >= (m + CORR_EPS):
0601:             self.ss_reset()
0602:             return SERVO_CENTER
0603:
0604:         # Small deadband near the margin to avoid micro-jitter
0605:         err = m - d   # >0 when too close
```

```
0606:         if err <= CORR_EPS:
0607:             return SERVO_CENTER
0608:
0609:         corr = self.ss * min(max_corr, CORRECTION_MULTIPLIER * err)
0610:         angle = SERVO_CENTER + corr
0611:         return max(SERVO_MIN_ANGLE, min(SERVO_MAX_ANGLE, angle))
0612:
0613:     def turn_decision(self, d_left, d_right, d_fl=None, d_fr=None): #Prefer corner/front
sensors for turn decision; fallback to sides if absent.
0614:         l_val = d_fl if d_fl is not None and d_fl != 999 else d_left
0615:         r_val = d_fr if d_fr is not None and d_fr != 999 else d_right
0616:         left_open  = (l_val is None) or (l_val == 999) or (l_val > TURN_DECISION_THRESHOLD)
0617:         right_open = (r_val is None) or (r_val == 999) or (r_val > TURN_DECISION_THRESHOLD)
0618:
0619:         # Decide only when exactly one side is open; otherwise keep trying.
0620:         if left_open ^ right_open:                #XOR: exactly one is True
0621:             return "LEFT" if left_open else "RIGHT"
0622:         return None # both open or both closed → keep trying
0623:
0624: robot = RobotController(pca)
0625:
0626: def sensor_reader():
0627:     global sensor_data
0628:     while True:
0629:         # Left sensor
0630:         if vl53_left:  # ToF
0631:             left = robot.filtered_distance(vl53_left, robot.left_history, "smooth_left",
sensor_type='tof')
0632:         elif us_left:  # ultrasonic
0633:             left = robot.filtered_distance(us_left, robot.left_history, "smooth_left",
sensor_type='us')
0634:         else:
0635:             left = None
0636:
0637:         # Right sensor
0638:         if vl53_right:
0639:             right = robot.filtered_distance(vl53_right, robot.right_history, "smooth_right",
sensor_type='tof')
0640:         elif us_right:
0641:             right = robot.filtered_distance(us_right, robot.right_history, "smooth_right",
sensor_type='us')
0642:         else:
0643:             right = None
0644:
0645:         # Front sensor
0646:         if vl53_front:
0647:             front = robot.filtered_distance(vl53_front, robot.front_history, "smooth_front",
sensor_type='tof')
0648:         elif us_front:
0649:             front = robot.filtered_distance(us_front, robot.front_history, "smooth_front",
sensor_type='us')
0650:         else:
0651:             front = None
0652:
0653:         # Corner/front-left sensor
0654:         if vl53_front_left:
0655:             front_left = robot.filtered_distance(vl53_front_left, robot.front_left_history,
"smooth_front_left", sensor_type='tof')
0656:         else:
0657:             front_left = None
0658:         # Corner/front-right sensor
0659:         if vl53_front_right:
0660:             front_right = robot.filtered_distance(vl53_front_right, robot.front_right_history,
"smooth_front_right", sensor_type='tof')
0661:         else:
0662:             front_right = None
0663:
0664:         # Save readings in global dictionary
0665:         with sensor_lock:
0666:             sensor_data["front"] = front
0667:             sensor_data["left"] = left
0668:             sensor_data["right"] = right
0669:             sensor_data["front_left"] = front_left
0670:             sensor_data["front_right"] = front_right
0671:
0672:         #time.sleep(SENSOR_DELAY)
0673:         sensor_tick.wait(SENSOR_DELAY)
```

```
0674:         sensor_tick.clear()
0675:
0676: # ================================
0677: # FINITE STATE MACHINE: robot_loop
0678: # ================================
0679:
0680: class RobotState(Enum):
0681:     IDLE = auto()
0682:     CRUISE = auto()
0683:     TURN_INIT = auto()
0684:     TURNING = auto()
0685:     POST_TURN = auto()
0686:     STOPPED = auto()
0687:
0688: locked_turn_direction = None # Keep a global locked_turn_direction variable to persist across
runs (reset in stop_loop)
0689:
0690: def robot_loop():
0691:     global status_text, turn_count, lap_count, locked_turn_direction, stop_reason,
obstacle_wait_deadline
0692:     global _btn_prev, _btn_last_ts, paused_by_button
0693:     state = RobotState.IDLE
0694:     direction = None
0695:     yaw = 0.0
0696:     turn_start_yaw = 0.0
0697:     turn_start_time = 0.0
0698:     post_turn_start = 0.0
0699:     last_turn_time = -999
0700:     last_ns = time.monotonic_ns()
0701:     start_ns = last_ns
0702:     turn_target_delta = 0.0  # relative yaw target for this turn (deg)
0703:     robot.stop_motor()                  # Ensure robot stopped at start
0704:     robot.set_servo(SERVO_CENTER)       # Ensure robot stopped at start
0705:
0706:     while True:
0707:         current_ns = time.monotonic_ns()
0708:         dt = (current_ns - last_ns) * 1e-9   # seconds
0709:         last_ns = current_ns
0710:         current_time = current_ns * 1e-9     # float seconds, used by UI/timers
0711:
0712:         # --- Headless START button edge detection (press to PAUSE/RESUME) ---
0713:         if not USE_GUI:
0714:             cur = 0 if START_BTN.is_pressed else 1
0715:             if _btn_prev == 1 and cur == 0 and (current_time - _btn_last_ts) > BTN_DEBOUNCE_S:
0716:                 _btn_last_ts = current_time
0717:                 headless_toggle_pause()
0718:             _btn_prev = cur
0719:
0720:         if not readings_event.is_set():
0721:             # if readings not started, keep motors stopped and idle
0722:             robot.stop_motor()
0723:             state = RobotState.IDLE
0724:             status_text = "Idle"
0725:             sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0726:             continue
0727:
0728:         # ----------------------------
0729:         # Sensor reads & gyro integration (always update while readings_flag)
0730:         # ----------------------------
0731:         with sensor_lock:
0732:             d_front = sensor_data["front"]
0733:             d_left = sensor_data["left"]
0734:             d_right = sensor_data["right"]
0735:             d_front_left = sensor_data["front_left"]
0736:             d_front_right = sensor_data["front_right"]
0737:
0738:         # Update robot attributes for plotting/logging and decision making
0739:         robot.d_front = d_front
0740:         robot.d_left = d_left
0741:         robot.d_right = d_right
0742:         robot.d_front_left = d_front_left
0743:         robot.d_front_right = d_front_right
0744:
0745:         # ---- Narrow corridor detector (sum of side distances) ----
0746:         if not hasattr(robot_loop, "_narrow_mode"):
0747:             robot_loop._narrow_mode = False
0748:
```

```
0749:         # Use None-safe math (ignore invalid readings)
0750:         l = robot.d_left  if (robot.d_left  is not None and robot.d_left  != 999) else None
0751:         r = robot.d_right if (robot.d_right is not None and robot.d_right != 999) else None
0752:         sum_lr = None if (l is None or r is None) else (l + r)
0753:
0754:         enter_thresh = NARROW_SUM_THRESHOLD
0755:         exit_thresh  = NARROW_SUM_THRESHOLD + NARROW_HYSTERESIS
0756:
0757:         prev_mode = robot_loop._narrow_mode
0758:         if sum_lr is not None:
0759:             if not robot_loop._narrow_mode and sum_lr < enter_thresh:
0760:                 robot_loop._narrow_mode = True
0761:             elif robot_loop._narrow_mode and sum_lr > exit_thresh:
0762:                 robot_loop._narrow_mode = False
0763:
0764:         # Apply global factors
0765:         global SPEED_ENV_FACTOR, DIST_ENV_FACTOR
0766:         SPEED_ENV_FACTOR = NARROW_FACTOR_SPEED if robot_loop._narrow_mode else 1.0
0767:         DIST_ENV_FACTOR  = NARROW_FACTOR_DIST  if robot_loop._narrow_mode else 1.0
0768:
0769:         # Optional console feedback
0770:         if robot_loop._narrow_mode != prev_mode:
0771:             mode_txt = "NARROW mode ON" if robot_loop._narrow_mode else "NARROW mode OFF"
0772:             dprint(f"[corridor] {mode_txt} (L+R = {sum_lr:.1f} cm)" if sum_lr is not None else
f"[corridor] {mode_txt}")
0773:
0774:         with I2C_LOCK:
0775:             raw_gyro_z = mpu.gyro[2] - bias      # rad/s from MPU6050
0776:             ALPHA = 0.8
0777:             gyro_z_filtered = ALPHA * raw_gyro_z + (1 - ALPHA) * getattr(robot, 'gyro_z_prev',
0.0)
0778:             robot.gyro_z_prev = gyro_z_filtered
0779:             yaw += (gyro_z_filtered * RAD2DEG) * dt
0780:
0781:         # Append to deques for plotting/logging
0782:         elapsed_time = (current_ns - start_ns) * 1e-9
0783:         time_data.append(elapsed_time)
0784:         front_data.append(robot.d_front if robot.d_front is not None else 0)
0785:         left_data.append(robot.d_left if robot.d_left is not None else 0)
0786:         right_data.append(robot.d_right if robot.d_right is not None else 0)
0787:         _fl = getattr(robot, "d_front_left", None)
0788:         _fr = getattr(robot, "d_front_right", None)
0789:         front_left_data.append(_fl if _fl is not None else 0)
0790:         front_right_data.append(_fr if _fr is not None else 0)
0791:         angle_data.append(yaw)
0792:         state_data.append(state.name)
0793:
0794:         # If loop_flag is still False, do NOT run FSM/motors yet—just update plots
0795:         if not loop_event.is_set():
0796:             robot.stop_motor()
0797:             state = RobotState.IDLE
0798:             status_text = "Paused" if paused_by_button else "Sensor readings started"
0799:             sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0800:             continue
0801:
0802:         # ----------------------------
0803:         # FSM transitions & actions
0804:         # ----------------------------
0805:         if state == RobotState.IDLE:
0806:             status_text = "Ready (readings started, loop not started)" if
readings_event.is_set() else "Idle"
0807:             robot.stop_motor()
0808:             robot.set_servo(SERVO_CENTER)
0809:             robot.ss_reset()
0810:             if loop_event.is_set():
0811:                 state = RobotState.CRUISE
0812:                 status_text = "Driving (cruise)"
0813:                 for speed in range(0, SPEED_CRUISE + 1):
0814:                     if not loop_event.is_set():
0815:                         robot.stop_motor()
0816:                         break
0817:                     robot.rotate_motor(speed)
0818:                     sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0819:
0820:         elif state == RobotState.CRUISE:
0821:             status_text = "Driving (cruise)"
0822:
```

```
0823:             # --------------------------------
0824:             # Emergency stop: immediate
0825:             # --------------------------------
0826:             if robot.d_front is not None and robot.d_front < eff_stop_threshold():
0827:                 robot.stop_motor()
0828:                 robot.set_servo(SERVO_CENTER)
0829:                 status_text = "Stopped! Obstacle ahead"
0830:                 dprint(status_text)
0831:                 state = RobotState.STOPPED
0832:
0833:                 stop_reason = "OBSTACLE"
0834:                 obstacle_wait_deadline = current_time + OBSTACLE_WAIT_TIME
0835:                 continue
0836:
0837:             # --------------------------------
0838:             # Check if we can trigger a turn
0839:             # --------------------------------
0840:             front_triggered = (robot.d_front is not None and robot.d_front <
eff_front_turn_trigger())
0841:             lockout_ok = (current_time - last_turn_time >= TURN_LOCKOUT)
0842:
0843:             if front_triggered and lockout_ok and loop_event.is_set():
0844:                 # immediately enter TURN_INIT and drop speed
0845:                 state = RobotState.TURN_INIT
0846:                 status_text = "Approaching turn - waiting for side to open"
0847:                 continue
0848:
0849:             # --------------------------------
0850:             # Normal cruise servo control
0851:             # Safe straight control active only after the 1st turn
0852:             # --------------------------------
0853:             if turn_count >= 1:
0854:                 desired_servo_angle = robot.safe_straight_control(robot.d_left, robot.d_right)
0855:             else:
0856:                 desired_servo_angle = SERVO_CENTER
0857:
0858:             robot.set_servo(desired_servo_angle)
0859:             robot.set_state_speed(state.name)
0860:
0861:         elif state == RobotState.TURN_INIT:
0862:             # Stay slow and keep wheels mostly straight while we wait for a side to open.
0863:             # If you prefer slight wall-following here, use safe_straight_control().
0864:             status_text = "Turn init - waiting for open side"
0865:             robot.set_state_speed(state.name)
0866:             robot.ss_reset()
0867:
0868:             # fail-safe: if front becomes safe again, return to cruise
0869:             if robot.d_front is not None and robot.d_front >= eff_front_turn_trigger():
0870:                 robot.set_servo(SERVO_CENTER)
0871:                 state = RobotState.CRUISE
0872:                 status_text = "Driving (cruise)"
0873:                 # brief yield
0874:                 sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0875:                 continue
0876:
0877:             # keep the wheels centered (or gentle straight correction)
0878:             if turn_count >= 1:
0879:                  desired_servo_angle = robot.safe_straight_control(robot.d_left,
robot.d_right)
0880:             else:
0881:                  # No correction before first turn: keep the wheels centered (or gentle
straight correction)
0882:                  desired_servo_angle = SERVO_CENTER
0883:             robot.set_servo(desired_servo_angle)
0884:
0885:             # Decide direction only when exactly one side is open
0886:             proposed_direction = robot.turn_decision(
0887:               robot.d_left, robot.d_right,
0888:               d_fl=robot.d_front_left, d_fr=robot.d_front_right
0889:             )
0890:
0891:             # If neither or both are open, keep waiting at TURN_INIT speed
0892:             if proposed_direction is None:
0893:                 sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0894:                 continue
0895:
0896:             # Turn lock mechanic (unchanged logic, just applied here)
```

```
0897:             if LOCK_TURN_DIRECTION == 1:
0898:                 if locked_turn_direction is None:
0899:                     locked_turn_direction = proposed_direction
0900:                     dprint(f"Turn direction locked to {locked_turn_direction}")
0901:                 elif proposed_direction != locked_turn_direction:
0902:                     # keep waiting at TURN_INIT speed for the locked direction to open
0903:                     status_text = f"Ignoring {proposed_direction} (locked to
{locked_turn_direction})"
0904:                     robot.set_servo(desired_servo_angle)
0905:                     sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0906:                     continue
0907:                 direction = locked_turn_direction
0908:             else:
0909:                 direction = proposed_direction
0910:
0911:             # Commit the turn now that one side is open and lock (if any) allows it
0912:             dprint(f"🔁 Turn initiated {direction}. Left: {robot.d_left if robot.d_left is not
None else -1:.1f} cm, Right: {robot.d_right if robot.d_right is not None else -1:.1f} cm")
0913:             turn_start_yaw = yaw
0914:             turn_start_time = current_time
0915:
0916:             # how skewed are we vs the nearest corridor axis at entry?
0917:             entry_skew = norm180(yaw - snap90(yaw))  # + = already rotated to the LEFT
0918:
0919:             # base ±TARGET_TURN_ANGLE, then compensate by entry skew
0920:             base = TARGET_TURN_ANGLE if direction == "LEFT" else -TARGET_TURN_ANGLE
0921:             turn_target_delta = base - entry_skew
0922:
0923:             # keep within your safety bounds
0924:             turn_target_delta = max(-MAX_TURN_ANGLE, min(MAX_TURN_ANGLE, turn_target_delta))
0925:             if abs(turn_target_delta) < MIN_TURN_ANGLE:
0926:                 turn_target_delta = MIN_TURN_ANGLE if turn_target_delta >= 0 else -
MIN_TURN_ANGLE
0927:
0928:             # start turning
0929:             angle = TURN_ANGLE_LEFT if direction == "LEFT" else TURN_ANGLE_RIGHT
0930:             robot.set_servo(angle)
0931:             robot.set_state_speed("TURNING")
0932:             state = RobotState.TURNING
0933:
0934:         elif state == RobotState.TURNING:
0935:             # active turning: monitor yaw & safety/time conditions to stop
0936:             # compute turn angle relative to start
0937:             turn_angle = yaw - turn_start_yaw  # +left, -right
0938:             target_angle = turn_target_delta   # adjusted by entry skew
0939:             robot.set_state_speed("TURNING")
0940:             stop_condition = False
0941:             robot.ss_reset()
0942:
0943:             # Condition A: Turn angle
0944:             if abs(turn_angle - target_angle) <= TURN_ANGLE_TOLERANCE:
0945:                 dprint("Stop Turn - Target Angle")
0946:                 stop_condition = True
0947:             # Condition B: timeout
0948:             if current_time - turn_start_time > TURN_TIMEOUT:
0949:                 dprint("Stop Turn - Max turn time")
0950:                 stop_condition = True
0951:             # Condition C: Max turn angle
0952:             if abs(turn_angle) >= MAX_TURN_ANGLE:
0953:                 dprint("Stop Turn - Max turn angle")
0954:                 stop_condition = True
0955:
0956:             # If any stop condition met -> finish turn
0957:             if stop_condition:
0958:                 robot.stop_motor()
0959:                 robot.set_servo(SERVO_CENTER)
0960:                 last_turn_time = current_time
0961:                 yaw = snap90(yaw)
0962:                 # update counters
0963:                 turn_count += 1
0964:                 if turn_count % 4 == 0:
0965:                     lap_count += 1
0966:                     dprint(f"✅ Lap completed! Total laps: {lap_count}")
0967:                     # if reached max laps -> do final drive then stop (as in original logic)
0968:                     if MAX_LAPS > 0 and lap_count >= MAX_LAPS:
0969:                         # Drive forward a little before stopping
0970:                         robot.set_servo(SERVO_CENTER)
```

```
                        robot.set_state_speed(state.name)
                        sensor_tick.wait(POST_LAP_DURATION); sensor_tick.clear()
                        robot.stop_motor()
                        stop_reason = "LAPS"
                        loop_event.clear()
                        sensor_tick.set()  # wake waits immediately
                        state = RobotState.STOPPED
                        status_text = f"Stopped - Max Laps ({MAX_LAPS}) reached"
                        continue

                # prepare post-turn
                post_turn_start = current_time
                state = RobotState.POST_TURN
                status_text = "Driving (post-turn)"

        elif state == RobotState.POST_TURN:
            # drive straight for short duration then return to CRUISE
            if current_time - post_turn_start < POST_TURN_DURATION:
                robot.set_servo(SERVO_CENTER)
                robot.set_state_speed(state.name)
                status_text = "Driving (post-turn)"
            else:
                state = RobotState.CRUISE
                status_text = "Driving (cruise)"

        elif state == RobotState.STOPPED:
            robot.stop_motor()
            robot.set_servo(SERVO_CENTER)
            robot.ss_reset()

            # --- Obstacle-driven stop: auto-retry after OBSTACLE_WAIT_TIME ---
            if stop_reason == "OBSTACLE":
                # nice status with countdown
                remaining = max(0.0, obstacle_wait_deadline - current_time)
                status_text = f"Stopped (obstacle) - retry in {remaining:.1f}s"

                # time to retry? re-check front distance
                if current_time >= obstacle_wait_deadline:
                    # quick re-sample already happens in the sensor thread;
                    # just use the latest filtered reading here:
                    d_front_now = robot.d_front

                    if d_front_now is not None and d_front_now >= eff_stop_threshold():
                        # clear → resume cruise
                        dprint("✅ Obstacle cleared — resuming")
                        status_text = "Driving (cruise)"
                        stop_reason = None
                        state = RobotState.CRUISE
                        robot.set_state_speed("CRUISE")
                        # small fall-through; next loop iteration will continue normally
                    else:
                        # still blocked → schedule another check in OBSTACLE_WAIT_TIME
                        obstacle_wait_deadline = current_time + OBSTACLE_WAIT_TIME

                # yield CPU while waiting
                sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
                continue

            # --- User or laps stop: stay stopped until user action (original behavior) ---
            status_text = "Stopped"
            sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
            continue

# ================================
# GUI SECTION
# ================================
def launch_gui(): #Initialize Tkinter GUI, matplotlib plots, sliders, and status indicators.
    import tkinter as tk
    from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
    import matplotlib
    matplotlib.use("TkAgg")  # explicit backend for GUI mode
    import matplotlib.pyplot as plt
    import tkinter.filedialog as fd
    global btn_readings, btn_start, btn_stop, root, canvas
    global ax_front, ax_side, ax_angle
    global front_line, left_line, right_line, angle_line
    global status_circle, status_canvas, status_text_id, lbl_status
```

```
1048:     global lbl_turns, lbl_laps
1049:     global sliders_frame, slider_vars, btn_export
1050:
1051:     from tkinter import TclError
1052:     import sys, os, signal  # add
1053:
1054:     GUI_CLOSING = False
1055:     after_ids = {"status": None, "plot": None}
1056:
1057:     # -------------------------
1058:     # Export to CSV
1059:     # -------------------------
1060:     def export_data_csv():
1061:         if not USE_GUI:
1062:             dprint("Headless: CSV export disabled.")
1063:             return
1064:         filename = f"viva_data_{datetime.now().strftime('%Y%m%d_%H%M%S')}.csv"
1065:         slider_values = {name: var.get() for name, var in slider_vars.items()}
1066:
1067:         with open(filename, mode='w', newline='') as file:
1068:             writer = csv.writer(file)
1069:             header = [
1070:                 "Time (s)", "Front (cm)", "Left (cm)", "Right (cm)", "Yaw (deg)",
1071:                 "State", "Turns", "Laps", "Servo Angle", "Speed", "Turning", "Direction"
1072:             ]
1073:             header += [name for name in slider_values.keys()]
1074:             writer.writerow(header)
1075:
1076:             for i in range(len(time_data)):
1077:                 row = [
1078:                     round(time_data[i], 2),
1079:                     round(front_data[i], 2),
1080:                     round(left_data[i], 2),
1081:                     round(right_data[i], 2),
1082:                     round(angle_data[i], 2),
1083:                     state_data[i],
1084:                     turn_count,
1085:                     lap_count,
1086:                     globals().get("servo_angle", 0),  # latest steering
1087:                     globals().get("SPEED_CRUISE", 0),
1088:                     "YES" if "Turning" in state_data[i] else "NO",
1089:                     globals().get("locked_turn_direction", "")
1090:                 ]
1091:                 row += [slider_values[name] for name in slider_values.keys()]
1092:                 writer.writerow(row)
1093:
1094:         print(f"✅ Data exported to {filename}")
1095:
1096:     # -------------------------
1097:     # Save/Load Slider Config
1098:     # -------------------------
1099:     def save_sliders_json():
1100:         if not USE_GUI:
1101:             print("Headless: save_sliders_json disabled.")
1102:             return
1103:         import tkinter.filedialog as fd
1104:         default_filename = f"sliders_config_{datetime.now().strftime('%Y%m%d')}.json"
1105:         file_path = fd.asksaveasfilename(initialdir=BASE_DIR,
1106:                                          initialfile=default_filename,
1107:                                          defaultextension=".json",
1108:                                          filetypes=[("JSON files", "*.json")],
1109:                                          title="Save Slider Configuration")
1110:         if file_path:
1111:             data = {name: var.get() for name, var in slider_vars.items()}
1112:             with open(file_path, "w") as f:
1113:                 json.dump(data, f, indent=4)
1114:             print(f"Slider values saved to {file_path}")
1115:
1116:     def load_sliders_json():
1117:         if not USE_GUI:
1118:             print("Headless: load_sliders_json disabled.")
1119:             return
1120:         import tkinter.filedialog as fd
1121:         file_path = fd.askopenfilename(initialdir=BASE_DIR,
1122:                                        defaultextension=".json",
1123:                                        filetypes=[("JSON files", "*.json")],
1124:                                        title="Load Slider Configuration")
```

```
1125:         if file_path:
1126:             try:
1127:                 with open(file_path, "r") as f:
1128:                     data = json.load(f)
1129:                 for name, value in data.items():
1130:                     if name in slider_vars:
1131:                         slider_vars[name].set(value)
1132:                         globals()[name] = value
1133:                 print(f"Sliders restored from {file_path}")
1134:             except Exception as e:
1135:                 print(f"Failed to load sliders: {e}")
1136:
1137:     def gui_alive():
1138:         try:
1139:             return not GUI_CLOSING and root.winfo_exists()
1140:         except Exception:
1141:             return False
1142:
1143:     def _cancel_afters():
1144:         try:
1145:             if after_ids.get("status"):
1146:                 root.after_cancel(after_ids["status"])
1147:         except Exception:
1148:             pass
1149:         try:
1150:             if after_ids.get("plot"):
1151:                 root.after_cancel(after_ids["plot"])
1152:         except Exception:
1153:             pass
1154:
1155:     def _really_exit():
1156:         # ultimate "get me out" in case something else is blocking
1157:         try: root.quit()
1158:         except: pass
1159:         try: root.destroy()
1160:         except: pass
1161:         os._exit(0)
1162:
1163:
1164:     root = tk.Tk()
1165:     root.title("VivaLaVida Robot Control")
1166:
1167:     slider_vars = {}
1168:
1169:     # ---------- Matplotlib Figure ----------
1170:     fig, (ax_front, ax_side, ax_angle) = plt.subplots(3, 1, figsize=(6, 8))
1171:     fig.tight_layout(pad=3.0)
1172:
1173:     ax_front.set_ylim(0, 350)
1174:     ax_front.set_title("Front Sensor")
1175:     ax_front.grid(True)
1176:     front_line, = ax_front.plot([], [], color="blue")
1177:
1178:     ax_side.set_ylim(-150, 150)
1179:     ax_side.set_title("Left (+Y) vs Right (-Y) Sensors")
1180:     ax_side.grid(True)
1181:     ax_side.axhline(0, color="black")
1182:     left_line,  = ax_side.plot([], [], color="#009E73", label="Left")
1183:     right_line, = ax_side.plot([], [], color="#E69F00", label="Right (neg)")
1184:     # New: front-corner ToFs drawn on the same axes
1185:     fl_line,    = ax_side.plot([], [], linestyle="--", color="#58D1B1", label="Front-Left")
1186:     fr_line,    = ax_side.plot([], [], linestyle="--", color="#F3C553", label="Front-Right
(neg)")
1187:     #ax_side.legend(loc="upper right", fontsize=8)
1188:     ax_side.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12), ncol=2, fontsize=8,
frameon=False)
1189:     fig.subplots_adjust(bottom=0.18)  # make room at the bottom
1190:
1191:
1192:     ax_angle.set_ylim(-180, 180)
1193:     ax_angle.set_title("Yaw Angle")
1194:     ax_angle.grid(True)
1195:     angle_line, = ax_angle.plot([], [], color="purple")
1196:
1197:     # ---------- Buttons ----------
1198:     btn_readings = tk.Button(root, text="Start Readings", command=start_readings,
1199:                              width=20, height=2, bg="blue", fg="white")
```

```
1200:     btn_start = tk.Button(root, text="Start Loop", command=start_loop,
1201:                           width=20, height=2, bg="green", fg="white", state="disabled")
1202:     btn_stop = tk.Button(root, text="Stop Loop", command=stop_loop,
1203:                          width=20, height=2, bg="red", fg="white", state="disabled")
1204:
1205:     # ---------- Status Labels ----------
1206:     lbl_status = tk.Label(root, text="Idle", font=("Arial", 14))
1207:     lbl_turns = tk.Label(root, text=f"Turns: {turn_count}", font=("Arial", 14))
1208:     lbl_laps = tk.Label(root, text=f"Laps: {lap_count}", font=("Arial", 14))
1209:
1210:     # ---------- Circular Status Indicator ----------
1211:     status_canvas = tk.Canvas(root, width=100, height=100, highlightthickness=0,
bg=root.cget("bg"))
1212:     status_circle = status_canvas.create_oval(10, 10, 90, 90, fill="grey", outline="")
1213:     status_text_id = status_canvas.create_text(50, 50, text="IDLE", fill="white",
font=("Arial", 14, "bold"))
1214:
1215:     # ---------- Sliders Frame ----------
1216:     sliders_frame = tk.LabelFrame(root, text="Parameters", padx=10, pady=10)
1217:     sliders_frame.grid(row=0, column=1, rowspan=8, sticky="ns", padx=10, pady=5)
1218:
1219:     # ---------- Column 3 frame for actions ----------
1220:     actions_frame = tk.LabelFrame(root, text="Actions", padx=10, pady=10)
1221:     actions_frame.grid(row=0, column=2, rowspan=8, sticky="ns", padx=10, pady=5)
1222:
1223:     # ---------- Two-column layout ----------
1224:     btn_readings.grid(row=0, column=0, sticky="ew", padx=5, pady=2)
1225:     btn_start.grid(row=1, column=0, sticky="ew", padx=5, pady=2)
1226:     btn_stop.grid(row=2, column=0, sticky="ew", padx=5, pady=2)
1227:     lbl_status.grid(row=3, column=0, pady=5)
1228:     lbl_turns.grid(row=4, column=0, pady=2)
1229:     lbl_laps.grid(row=5, column=0, pady=2)
1230:     status_canvas.grid(row=6, column=0, pady=5)
1231:
1232:     canvas = FigureCanvasTkAgg(fig, master=root)
1233:     canvas.get_tk_widget().grid(row=7, column=0, sticky="nsew", padx=5, pady=5)
1234:
1235:     root.grid_columnconfigure(0, weight=1)  # main buttons + plots
1236:     root.grid_columnconfigure(1, weight=0)  # sliders
1237:     root.grid_columnconfigure(2, weight=0)  # actions
1238:     root.grid_rowconfigure(7, weight=1)     # plot row
1239:
1240:     # ---------- Slider Definitions ----------
1241:     slider_groups = {
1242:         "Speeds": [
1243:             ("Cruise Speed", "SPEED_CRUISE", 0, 100, "int"),
1244:             ("Turn Init Speed", "SPEED_TURN_INIT", 0, 100, "int"),
1245:             ("Turn Speed", "SPEED_TURN", 0, 100, "int"),
1246:             ("Post Turn Speed", "SPEED_POST_TURN", 0, 100, "int")
1247:         ],
1248:         "Driving & Safety, Wall following, Turns": [
1249:             ("Turn angle Left", "TURN_ANGLE_LEFT", 55, 90, "int"),
1250:             ("Turn angle Right", "TURN_ANGLE_RIGHT", 90, 125, "int"),
1251:             ("Soft side Margin (cm)", "SOFT_MARGIN", 0, 75, "int"),
1252:             ("Max Angle Correction dif at Soft Margin (o)", "MAX_CORRECTION", 0, 30, "int"),
1253:             ("Stop Threshold (cm)", "STOP_THRESHOLD", 0, 100, "int"),
1254:             ("Front Turn Trigger (cm)", "FRONT_TURN_TRIGGER", 50, 150, "int"),
1255:             ("Turn Decision Threshold (Left/Right) (cm)", "TURN_DECISION_THRESHOLD", 0, 200,
"int"),
1256:             ("Turn Timeout (s)", "TURN_TIMEOUT", 0.1, 10, "float"),
1257:             ("Max Turn Angle (o)", "MAX_TURN_ANGLE", 90, 120, "int"),
1258:             ("Post Turn Duration (s)", "POST_TURN_DURATION", 0, 5, "float"),
1259:             ("Turn Lockout (s)", "TURN_LOCKOUT", 0.1, 5, "float")
1260:         ],
1261:         "Other": [
1262:             ("Sensor Filter N (Median)", "N_READINGS", 1, 10, "int"),
1263:             ("Max Laps - 0 for infinite", "MAX_LAPS", 0, 20, "int"),
1264:             ("Filter Smoothness (alpha)", "FILTER_ALPHA", 0.05, 0.9, "float"),
1265:             ("Max Jump (cm)", "FILTER_JUMP", 5, 999, "int"),
1266:         ]
1267:     }
1268:
1269:     # ---------- Create Sliders ----------
1270:     for group_name, sliders in slider_groups.items():
1271:         group_frame = tk.LabelFrame(sliders_frame, text=group_name, padx=5, pady=5)
1272:         group_frame.pack(fill="x", pady=5)
1273:         for label_text, var_name, vmin, vmax, vartype in sliders:
```

```
1274:             frame = tk.Frame(group_frame)
1275:             frame.pack(fill="x", pady=2)
1276:
1277:             if vartype == "float":
1278:                 var = tk.DoubleVar(value=globals()[var_name])
1279:                 res = 0.1
1280:             else:
1281:                 var = tk.IntVar(value=globals()[var_name])
1282:                 res = 1
1283:
1284:             slider_vars[var_name] = var
1285:
1286:             scale = tk.Scale(frame, from_=vmin, to=vmax, orient="horizontal",
1287:                              variable=var, resolution=res)
1288:             scale.pack(side="left", fill="x", expand=True)
1289:
1290:             lbl_val = tk.Label(frame, text=f"{label_text}: {var.get()}", width=25, anchor="w")
1291:             lbl_val.pack(side="right", padx=5)
1292:
1293:             def make_callback(lbl, name, var, label_text):
1294:                 def callback(value, label_text=label_text):
1295:                     globals()[name] = var.get()
1296:                     if name == "MAX_LAPS" and var.get() == 0:
1297:                         lbl.config(text=f"{label_text}: ∞")
1298:                     else:
1299:                         lbl.config(text=f"{label_text}: {var.get():.1f}" if
isinstance(var.get(), float) else f"{label_text}: {var.get()}")
1300:                 return callback
1301:
1302:             scale.config(command=make_callback(lbl_val, var_name, var, label_text))
1303:
1304:     # ---------- Buttons under sliders ----------
1305:     btn_export = tk.Button(actions_frame, text="Export CSV", command=export_data_csv,
1306:                            width=20, height=2, bg="purple", fg="white")
1307:     btn_export.pack(pady=10, fill="x")
1308:
1309:     btn_save_sliders = tk.Button(actions_frame, text="Save Sliders",
command=save_sliders_json,
1310:                                  width=20, height=2, bg="green", fg="white")
1311:     btn_save_sliders.pack(pady=5, fill="x")
1312:
1313:     btn_load_sliders = tk.Button(actions_frame, text="Load Sliders",
command=load_sliders_json,
1314:                                  width=20, height=2, bg="blue", fg="white")
1315:     btn_load_sliders.pack(pady=5, fill="x")
1316:
1317:     # ==============================
1318:     # GUI Update Functions (nested)
1319:     # ==============================
1320:     def status_color(state: str) -> str:
1321:         if "Stopped" in state:
1322:             return "red"
1323:         elif "Turning" in state:
1324:             return "yellow"
1325:         elif "Driving" in state:
1326:             return "green"
1327:         elif "Loop Started" in state or "Sensor readings started" in state:
1328:             return "blue"
1329:         else:
1330:             return "grey"
1331:
1332:     def status_label(state: str) -> str:
1333:         if "Stopped" in state:
1334:             return "STOP"
1335:         elif "Turning" in state:
1336:             return "TURN"
1337:         elif "Driving" in state:
1338:             return "GO"
1339:         elif "Loop Started" in state:
1340:             return "LOOP"
1341:         elif "Sensor readings started" in state:
1342:             return "READ"
1343:         else:
1344:             return "IDLE"
1345:
1346:     def update_status():
1347:         if not gui_alive():
```

```
            return
        lbl_status.config(text=status_text)
        status_canvas.itemconfig(status_circle, fill=status_color(status_text))
        status_canvas.itemconfig(status_text_id, text=status_label(status_text))
        lbl_turns.config(text=f"Turns: {turn_count}")
        lbl_laps.config(text=f"Laps: {lap_count}" if slider_vars["MAX_LAPS"].get() > 0 else
f"Laps: {lap_count}/∞")
        try:
            after_ids["status"] = root.after(200, update_status)
        except TclError:
            # window is gone; just stop
            return

    _prev_label_pos = {
        "front": None, "left": None, "right": None,
        "front_left": None, "front_right": None,
        "angle": None
    }

    def update_plot():
        if not gui_alive():
            return
        try:
            if time_data:
                # Update lines
                front_line.set_data(range(len(front_data)), front_data)
                left_line.set_data(range(len(left_data)), left_data)
                right_line.set_data(range(len(right_data)), [-v for v in right_data])
                fl_line.set_data(range(len(front_left_data)), front_left_data)
                fr_line.set_data(range(len(front_right_data)), [-v for v in front_right_data])
                angle_line.set_data(range(len(angle_data)), angle_data)

                # Set axis limits
                ax_front.set_xlim(0, MAX_POINTS)
                ax_side.set_xlim(0, MAX_POINTS)
                ax_angle.set_xlim(0, MAX_POINTS)

                # Remove previous text annotations
                for ax in [ax_front, ax_side, ax_angle]:
                    for t in ax.texts:
                        t.remove()

                # Helper for smoothing
                def smooth_move(prev, target, alpha=0.3):
                    if prev is None:
                        return target
                    return prev + alpha * (target - prev)

                # ----------------------------
                # FRONT SENSOR
                # ----------------------------
                if len(front_data) > 0:
                    x = len(front_data) - 1
                    y_target = front_data[-1]
                    y_prev = _prev_label_pos["front"]
                    y_smooth = smooth_move(y_prev, y_target)
                    _prev_label_pos["front"] = y_smooth
                    ax_front.text(x, y_smooth, f"{y_target:.1f} cm", color="blue",
                                  fontsize=9, fontweight="bold", va="bottom", ha="left")

                # ----------------------------
                # LEFT SENSOR
                # ----------------------------
                if len(left_data) > 0:
                    x = len(left_data) - 1
                    y_target = left_data[-1]
                    y_prev = _prev_label_pos["left"]
                    y_smooth = smooth_move(y_prev, y_target)
                    _prev_label_pos["left"] = y_smooth
                    ax_side.text(x, y_smooth, f"L: {y_target:.1f} cm", color="#009E73",
                                  fontsize=9, fontweight="bold", va="bottom", ha="left")

                # ----------------------------
                # RIGHT SENSOR
                # ----------------------------
                if len(right_data) > 0:
                    x = len(right_data) - 1
```

```
1424:                     y_target = -right_data[-1]
1425:                     y_prev = _prev_label_pos["right"]
1426:                     y_smooth = smooth_move(y_prev, y_target)
1427:                     _prev_label_pos["right"] = y_smooth
1428:                     ax_side.text(x, y_smooth, f"R: {right_data[-1]:.1f} cm", color="#E69F00",
1429:                                  fontsize=9, fontweight="bold", va="bottom", ha="left")
1430:
1431:
1432:                 # ----------------------------
1433:                 # FRONT-LEFT SENSOR (corner)
1434:                 # ----------------------------
1435:                 if len(front_left_data) > 0:
1436:                     x = len(front_left_data) - 1
1437:                     y_target = front_left_data[-1]
1438:                     y_prev = _prev_label_pos["front_left"]
1439:                     y_smooth = smooth_move(y_prev, y_target)
1440:                     _prev_label_pos["front_left"] = y_smooth
1441:                     ax_side.text(x, y_smooth, f"FL: {y_target:.1f} cm", color="#58D1B1",
fontsize=9, fontweight="bold", va="bottom", ha="left")
1442:                 # ----------------------------
1443:                 # FRONT-RIGHT SENSOR (corner)
1444:                 # ----------------------------
1445:                 if len(front_right_data) > 0:
1446:                     x = len(front_right_data) - 1
1447:                     y_target = -front_right_data[-1]
1448:                     y_prev = _prev_label_pos["front_right"]
1449:                     y_smooth = smooth_move(y_prev, y_target)
1450:                     _prev_label_pos["front_right"] = y_smooth
1451:                     ax_side.text(x, y_smooth, f"FR: {front_right_data[-1]:.1f} cm",
color="#F3C553", fontsize=9, fontweight="bold", va="bottom", ha="left")
1452:
1453:                 # ----------------------------
1454:                 # YAW ANGLE
1455:                 # ----------------------------
1456:                 if len(angle_data) > 0:
1457:                     x = len(angle_data) - 1
1458:                     y_target = angle_data[-1]
1459:                     y_prev = _prev_label_pos["angle"]
1460:                     y_smooth = smooth_move(y_prev, y_target)
1461:                     _prev_label_pos["angle"] = y_smooth
1462:                     ax_angle.text(x, y_smooth, f"{y_target:.1f}°", color="purple",
1463:                                   fontsize=9, fontweight="bold", va="bottom", ha="left")
1464:
1465:                 # ----------------------------
1466:                 # Dynamic Y-axis scaling for yaw
1467:                 # ----------------------------
1468:                 if angle_data:
1469:                     min_angle = min(angle_data)
1470:                     max_angle = max(angle_data)
1471:                     if max_angle - min_angle > 180:
1472:                         ax_angle.set_ylim(-180, 180)
1473:                     else:
1474:                         ax_angle.set_ylim(min_angle - 10, max_angle + 10)
1475:
1476:                 canvas.draw()
1477:             except TclError:
1478:                 return
1479:             try:
1480:                 after_ids["plot"] = root.after(100, update_plot)
1481:             except TclError:
1482:                 return
1483:
1484:         def on_closing():
1485:             nonlocal GUI_CLOSING
1486:             GUI_CLOSING = True
1487:
1488:             # stop scheduling anything new
1489:             _cancel_afters()
1490:
1491:             # tell loops/threads to stand down
1492:             try: loop_event.clear()
1493:             except: pass
1494:             try: readings_event.clear()
1495:             except: pass
1496:             try: sensor_tick.set()
1497:             except: pass
1498:
```

```
1499:         # stop robot now
1500:         try:
1501:             robot.stop_motor()
1502:             robot.set_servo(SERVO_CENTER)
1503:         except:
1504:             pass
1505:
1506:         # shut down ToF
1507:         for sensor in [vl53_left, vl53_right, vl53_front, vl53_back,
1508:                        vl53_front_left, vl53_front_right]:
1509:             if sensor is not None:
1510:                 try:
1511:                     with I2C_LOCK:
1512:                         sensor.stop_continuous()
1513:                 except Exception as e:
1514:                     dprint(f"Warning: could not stop sensor {sensor}: {e}")
1515:
1516:         # close DistanceSensors
1517:         for us in [us_front, us_left, us_right]:
1518:             try:
1519:                 if us is not None:
1520:                     us.close()
1521:             except:
1522:                 pass
1523:
1524:         # LEDs / button
1525:         try:
1526:             GREEN_LED.off(); RED_LED.off()
1527:             GREEN_LED.close(); RED_LED.close(); START_BTN.close()
1528:         except:
1529:             pass
1530:
1531:         # IMPORTANT: quit the Tk loop first, then destroy
1532:         try: root.quit()
1533:         except: pass
1534:
1535:         # Some Tk/Matplotlib stacks need the canvas widget gone before destroy
1536:         try:
1537:             canvas.get_tk_widget().destroy()
1538:         except:
1539:             pass
1540:
1541:         try: root.destroy()
1542:         except:
1543:             pass
1544:
1545:         # Failsafe: if something is still hanging, hard-exit soon
1546:         root.after(100, _really_exit)
1547:
1548:
1549:     root.protocol("WM_DELETE_WINDOW", on_closing)
1550:     root.bind("<Escape>", lambda e: on_closing())
1551:     update_status()
1552:     update_plot()
1553:     root.mainloop()
1554:
1555: # -------------------------
1556: # Start / Stop
1557: # -------------------------
1558:
1559: def start_readings():
1560:     global status_text
1561:     readings_event.set()
1562:     status_text = "Sensor readings started"
1563:     if USE_GUI:
1564:         btn_readings.config(state="disabled")
1565:         btn_start.config(state="normal")
1566:
1567:     # Start background sensor thread once
1568:     if not hasattr(start_readings, "sensor_thread_started"):
1569:         threading.Thread(target=sensor_reader, daemon=True).start()
1570:         start_readings.sensor_thread_started = True
1571:
1572:     if not hasattr(start_readings, "loop_thread_started"):
1573:         threading.Thread(target=robot_loop, daemon=True).start()
1574:         start_readings.loop_thread_started = True
1575:
```

```
    sensor_tick.set()

def start_loop(): #Start the robot driving loop
    global status_text
    if not readings_event.is_set():
        print("Start sensor readings first!")
        return

    loop_event.set()
    status_text = "🚗 Loop Started"
    if USE_GUI:
        btn_start.config(state="disabled")
        btn_stop.config(state="normal")

def stop_loop(): #Stop the robot loop and motor
    global status_text, turn_count, lap_count, locked_turn_direction, stop_reason
    loop_event.clear()
    stop_reason = "USER"
    sensor_tick.set()  # wake any waits immediately
    robot.stop_motor()
    status_text = "Stopped"
    turn_count = 0
    lap_count = 0

    if USE_GUI:
        btn_start.config(state="normal")
        btn_stop.config(state="disabled")
        lbl_turns.config(text=f"Turns: {turn_count}")
        lbl_laps.config(text=f"Laps: {lap_count}")

    locked_turn_direction = None # Reset direction lock so a new session can re-choose

# ================================
# MAIN
# ================================

if __name__ == "__main__":
    print(f"Starting VivaLaVida Autonomous Drive - GUI mode: {USE_GUI}")

    # Force headless if no display variable is present
    if os.environ.get("DISPLAY", "") == "":
        USE_GUI = 0

    if USE_GUI:
        started = launch_gui()
        if not started:
            print("⚠️ GUI not available. Falling back to headless.")
            USE_GUI = 0

    if not USE_GUI: # Start the sensor reading thread
        sensor_thread = threading.Thread(target=sensor_reader, daemon=True)
        sensor_thread.start()
        print("Headless mode: waiting for START button...")
        RED_LED.on()     # turn on red when ready
        GREEN_LED.blink(on_time=0.5, off_time=0.5, background=True)
        #GREEN_LED.on()  # green = ready to press the button
        try: # Wait for button press (GPIO input goes LOW when pressed)
            while not START_BTN.is_pressed:
                time.sleep(0.05)
            print("✅ START button pressed! Beginning autonomous loop...")
            readings_event.set() #readings_flag = True
            GREEN_LED.blink(on_time=0.1, off_time=0.1, background=True)
            RED_LED.off()

            #time.sleep(2)  # wait 2 seconds before starting loop
            #loop_event.set() #loop_flag = True
            # Warm-up: wait (max 2s) until front/left/right have at least one usable value
            deadline = time.time() + 2.0
            while time.time() < deadline:
                with sensor_lock:
                    f = sensor_data["front"]
                    l = sensor_data["left"]
                    r = sensor_data["right"]
                if all(v is not None and v != 999 for v in (f, l, r)):
                    break
                sensor_tick.set()          # nudge the sensor thread
                time.sleep(0.05)
```

```
1653:
1654:             loop_event.set()
1655:
1656:             GREEN_LED.on()  # running
1657:             print("Starting main robot loop...")
1658:             robot_loop() # Start robot loop in this thread (blocking)
1659:         except KeyboardInterrupt:
1660:             print("\n❌ Keyboard interrupt received. Stopping robot loop.")
1661:             try: loop_event.clear() # Stop loops/threads
1662:             except: pass
1663:             try: readings_event.clear()
1664:             except: pass
1665:             try: sensor_tick.set()
1666:             except: pass
1667:             try: robot.stop_motor() # Motors & servo to safe state
1668:             except: pass
1669:             try: robot.set_servo(SERVO_CENTER)
1670:             except: pass
1671:             try: # Stop any ToF sensors cleanly (if enabled)
1672:                 for s in [vl53_left, vl53_right, vl53_front, vl53_back, vl53_front_left,
vl53_front_right]:
1673:                     if s is not None:
1674:                         try:
1675:                             with I2C_LOCK:
1676:                                 s.stop_continuous()
1677:                         except Exception as e:
1678:                             dprint(f"[KeyboardInterrupt] ToF stop warning: {e}")
1679:             except:
1680:                 pass
1681:             try: # Zero PCA9685 outputs
1682:                 with I2C_LOCK:
1683:                     for ch in [MOTOR_FWD, MOTOR_REV, SERVO_CHANNEL]:
1684:                         pca.channels[ch].duty_cycle = 0
1685:             except:
1686:                 pass
1687:             try:
1688:                 GREEN_LED.off()
1689:                 RED_LED.off()
1690:                 GREEN_LED.close()
1691:                 RED_LED.close()
1692:                 START_BTN.close()
1693:             except:
1694:                 pass
1695:
1696: # End
```