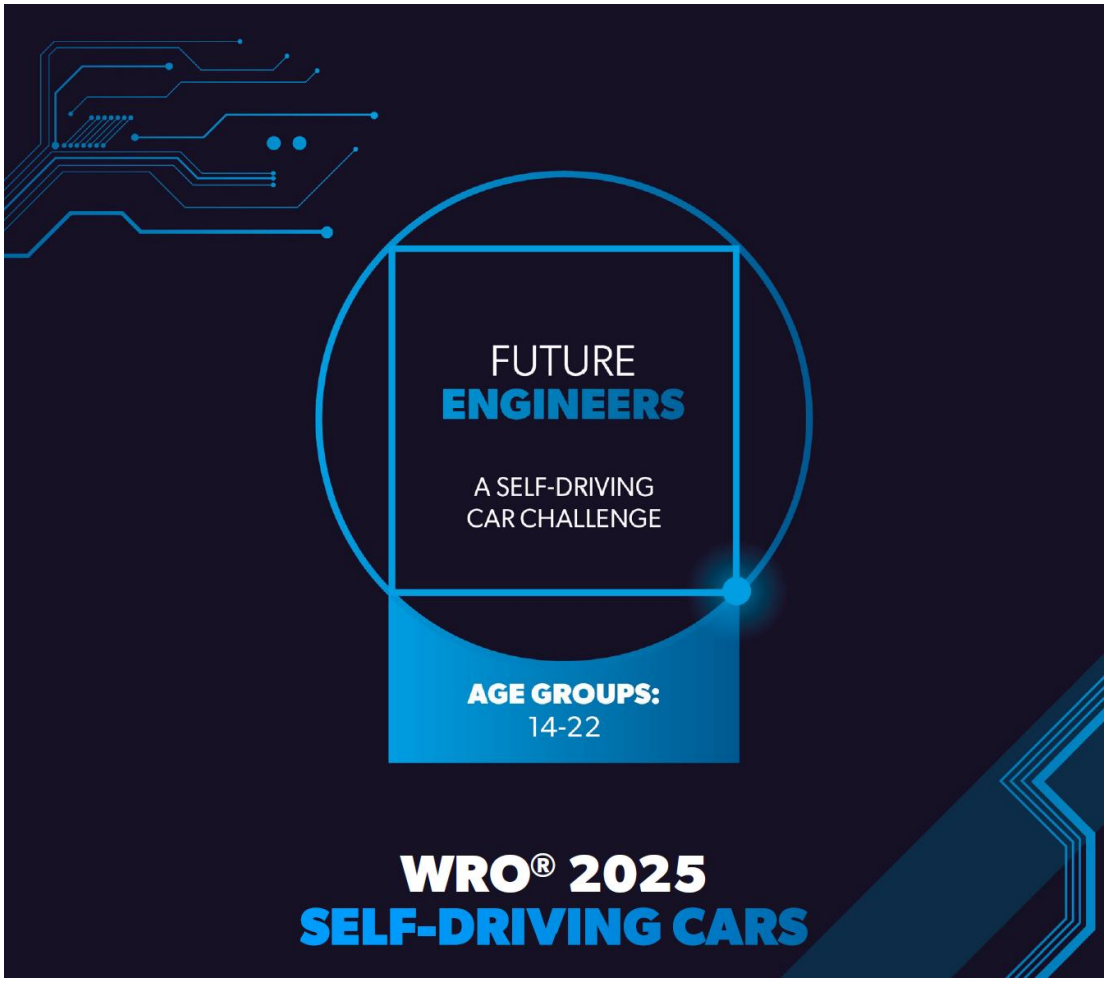
# Viva La Vida — WRO FE Open Challenge



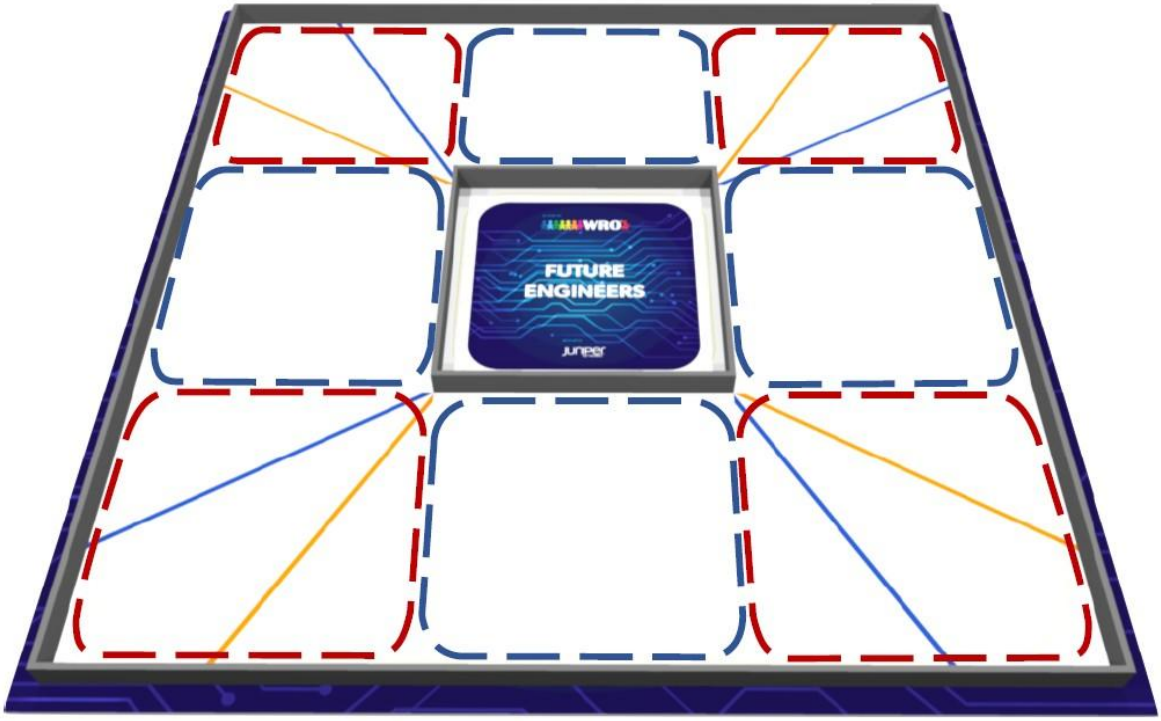

**VIVA ΛA VIΔA**

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.




# 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 actuator | 1000–2000 µs pulse |
| **Motor + Driver/ESC** | Propulsion | PWM ch1/ch2 (FWD/REV) |
| **Start Button** | Headless start | GPIO20 (pull-up; active LOW) |
| **Status LEDs** | Green ready / Red status | GPIO19 / GPIO13 |
| **Battery/Power** | 5–6 V servo; 5 V logic | Common ground |

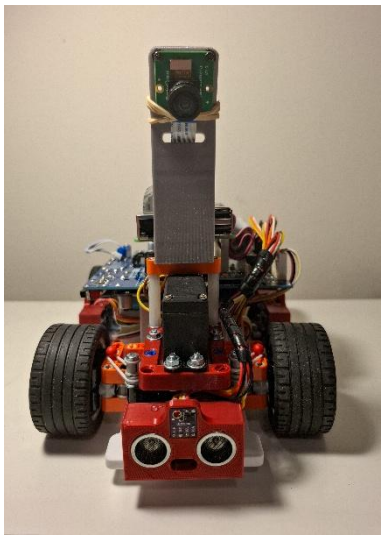
## 2.2 Pin Map

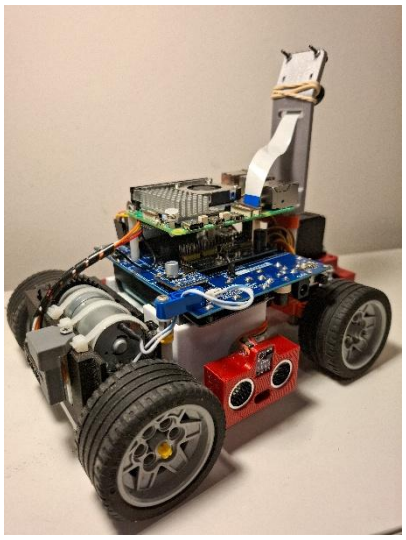
| Subsystem | Signal | Pin/Channel | Dir | Notes |
|---|---|---|---|---|
| **Start Button** | START_BTN | GPIO 20 | IN | Pull-up; active LOW |
| **LEDs** | GREEN/RED | GPIO 19 / 13 | OUT | Status/ready |
| **US Front** | TRIG / ECHO | 22 / 23 | OUT / IN | max_distance 3.43 m |
| **US Left** | TRIG / ECHO | 27 / 17 | OUT / IN | queue_len 5 |
| **US Right** | TRIG / ECHO | 5 / 6 | OUT / IN | queue_len 5 |
| **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 | D16 / D25 / D26 / D24 | OUT | Addr 0x30/31/32/33 |

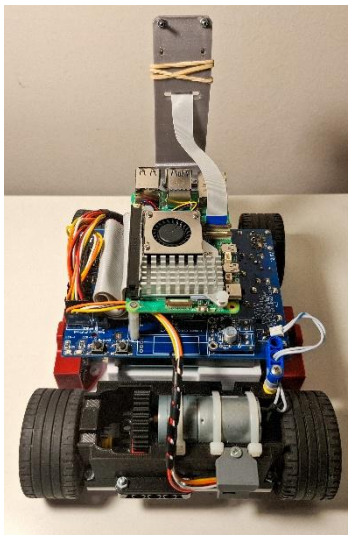
## 2.3 Boot & Headless Start

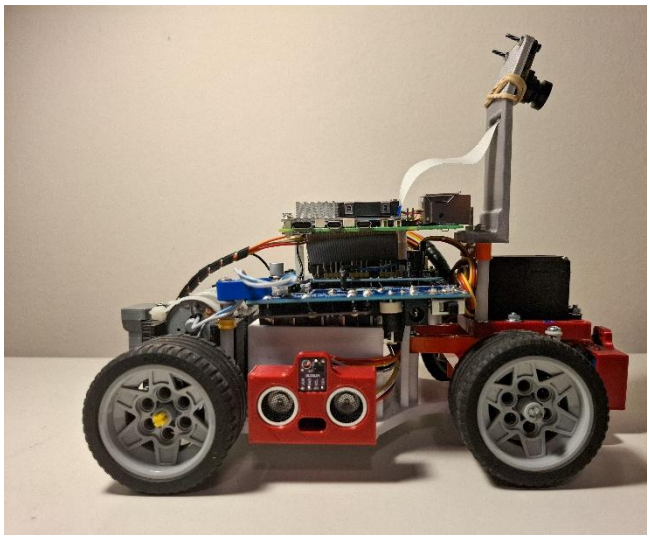
Boot sequence: RED LED on → sensors init → GREEN LED on (ready). Headless: press Start (GPIO20 goes LOW) to arm readings_event and then loop_event.

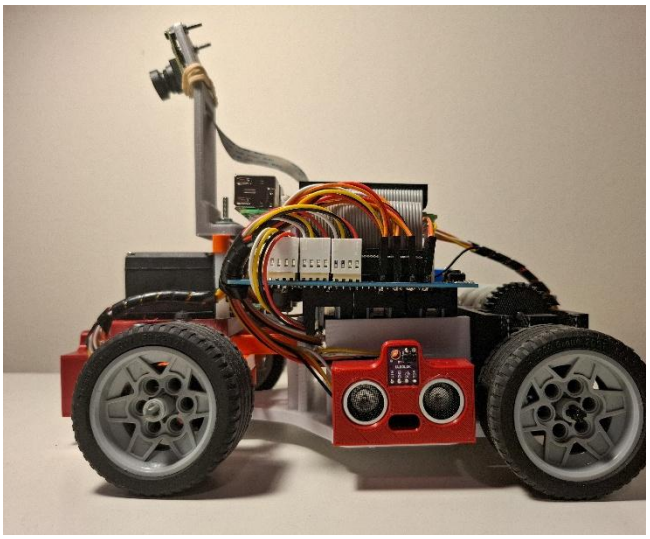
## 2.4 Robot photos (test version)











## 3 System Architecture

Code structure at a glance:

- 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 with optional slew limiting, 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. A servo slew limiter (deg/s) may be enabled to reduce oscillation on straights.

## 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..0x33) 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 via board.SCL/board.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), 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; each correction is time-limited (CORRECTION_DURATION) and can renew if still needed. 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 GPIO.cleanup().

#### *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) 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. If servo slew limiting is enabled, the command rate is constrained to SERVO_SLEW_DPS to avoid mechanical shock and to reduce IMU spikes.

### 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, time-boxed corrections when min(left,right) < SOFT_MARGIN. The correction magnitude saturates at MAX_CORRECTION for CORRECTION_DURATION; 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)

| 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(open side) | Direction chosen → TURNING; Front re-safe → CRUISE |
| **TURNING** | Direction committed | Servo setpoint; track | Angle tolerance OR |

| | | | |
|---|---|---|---|
| | | yaw; guards | 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 |

# 6 Parameters & Profiles

| Category | Key Parameters (defaults) | Purpose |
|---|---|---|
| **Speeds** | SPEED_CRUISE/TURN_INIT/TURN/POST_TURN = 25/17/20/20 | State-specific duty |
| **Driving** | SOFT_MARGIN=30 cm; MAX_CORRECTION=7°; CORRECTION_DURATION=0.25 s | Wall-following |
| **Safety** | STOP_THRESHOLD=20 cm; TURN_TIMEOUT=4.5 s; TURN_LOCKOUT=1.5 s | Guards |
| **Turns** | FRONT_TURN_TRIGGER=90 cm; TURN_DECISION_THRESHOLD=100 cm | Eligibility & direction |
| **Yaw** | TARGET_TURN_ANGLE=85°; TURN_ANGLE_TOLERANCE=5°; MAX_TURN_ANGLE=120° | Termination |
| **Narrow** | NARROW_SUM_THRESHOLD=60 cm; NARROW_HYSTERESIS=10 cm | Scaling & chatter-free toggling |
| **Filters** | FILTER_ALPHA(_TOF)=1.0; FILTER_JUMP(_TOF)=999; N_READINGS=1 | Smoothing & spike reject |
| **Timing** | LOOP_DELAY=0.001 s; SENSOR_DELAY=0.005 s | Loop pacing |

## 6.1 Preset Profiles

| Profile | Cruise | Turn | Target° | Tol° | Front Trigger | Lockout |
|---|---|---|---|---|---|---|
| **Safe** | 12 | 12 | 84 | 6 | 100 cm | 2.0 s |
| **Balanced (default)** | 15 | 15 | 86 | 5 | 90 cm | 1.5 s |
| **Aggressive** | 18 | 16 | 88 | 4 | 80 cm | 1.2 s |

# 7 Calibration & Ops

- Gyro: keep robot still for 500-sample bias calibration at boot.
- Servo: verify SERVO_CENTER and endpoints to avoid binding.
- Ranges: validate 20/50/100 cm; adjust FILTER_ALPHA/JUMP if noisy.
- Front trigger: increase until turns start safely without false positives.
- Yaw target: mark square corners; adjust TARGET_TURN_ANGLE, then refine with tolerance.

### 7.1 Operations Runbook

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

## 8 KPIs & Test Protocol

| KPI | Target | Measurement |
|---|---|---|
| **Laps completed** | 3/3 | lap_count; CSV |
| **Avg yaw error/turn** | ≤ 5° | compare Δyaw vs target |
| **False emergency stops** | ≤ 1 per run | STOP without obstacle |
| **Narrow-mode stability** | No chatter | L+R hysteresis behavior |
| **Lap time variance** | ≤ 10% | std. dev. across laps |

Protocol: straight-line baseline → single-corner tuning → full-lap with varying corridor widths → obstacle stop/retry → 3-lap verification.

## 9 Compliance (Rules Snapshot)

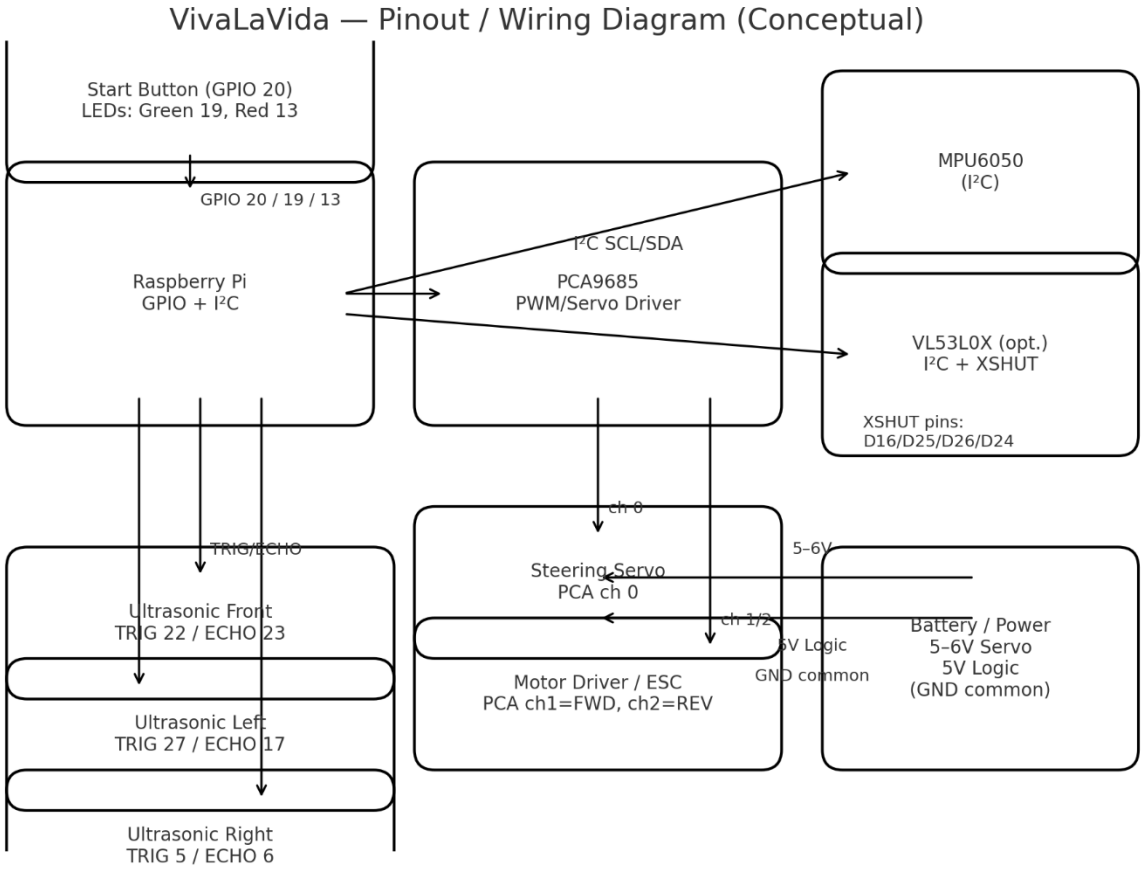
| Rule | Implementation | Status |
|---|---|---|
| **3 autonomous laps** | lap_count; MAX_LAPS; POST_LAP_DURATION | Compliant |
| **Single Start button; no wireless during run** | Headless GPIO20 start; GUI optional before run | Compliant |
| **Respect driving direction; no wall movement** | LOCK_TURN_DIRECTION; no wall control pins | Compliant |
| **Obstacle stop** | STOP_THRESHOLD + retry window | Compliant |

## 10 Risks & Mitigations

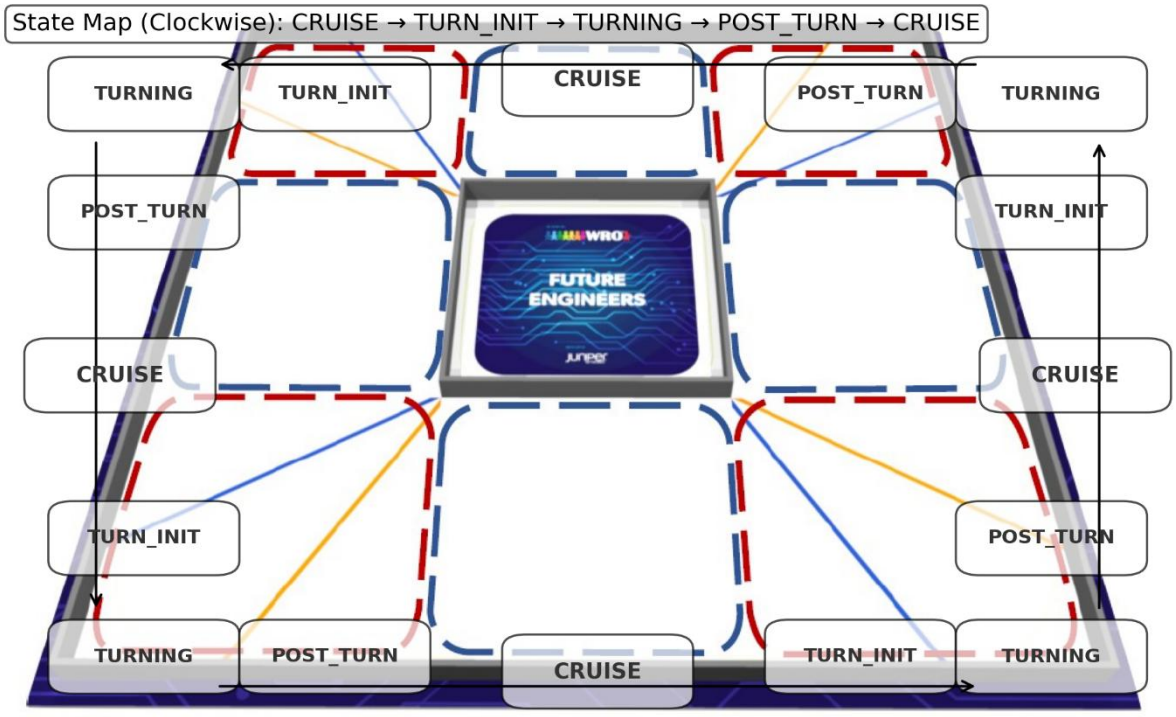
Chattering near thresholds → add hysteresis or filtering; under/over turns → adjust target°/tolerance and verify IMU bias; late turns → raise front trigger or reduce turn-init speed; oscillations on straights → lower MAX_CORRECTION or enable servo slew; sensor dropouts → check power/ground and fall back to ultrasonics if ToF fails.

# 11 Diagrams

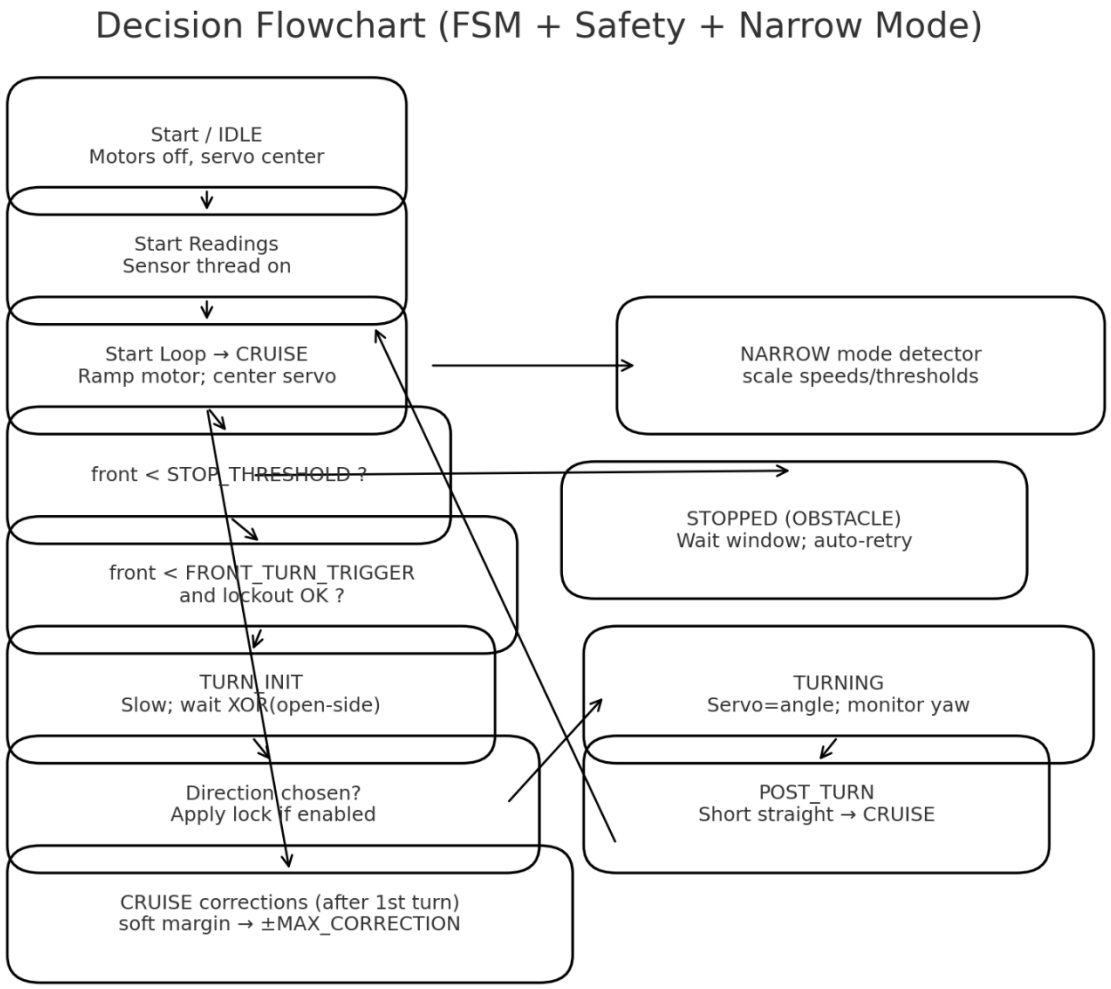
## 11.1 Pinout / Wiring (Conceptual)




## 11.2 Driving Area — State Map (Abstract)



## 11.3 Decision Flowchart (FSM + Safety + Narrow Mode)




# 12 Appendix A — JSON Override Example

```
{
  "SPEED_CRUISE": 25, "SPEED_TURN_INIT": 17, "SPEED_TURN": 20,
"SPEED_POST_TURN": 20,
  "FRONT_TURN_TRIGGER": 90, "TURN_DECISION_THRESHOLD": 100,
  "TURN_ANGLE_LEFT": 60, "TURN_ANGLE_RIGHT": 120,
  "TARGET_TURN_ANGLE": 85, "TURN_ANGLE_TOLERANCE": 5,
  "TURN_TIMEOUT": 4.5, "TURN_LOCKOUT": 1.5,
  "SOFT_MARGIN": 30, "MAX_CORRECTION": 7, "STOP_THRESHOLD": 20,
  "NARROW_SUM_THRESHOLD": 60, "NARROW_HYSTERESIS": 10, "SERVO_SLEW_DPS":
0
}
```

# 13 Appendix B — Source Code Placeholder

## 13.1 Analysis

**1) Imports & global configuration**

- **External libs:** RPi.GPIO, gpiozero.DistanceSensor (ultrasonic, meters), adafruit_pca9685 (PWM @ 50 Hz), adafruit_mpu6050 (gyro Z in rad/s), adafruit_vl53l0x (ToF in mm), board/digitalio (I²C + XSHUT), tkinter + matplotlib (GUI/plots), numpy (median/mean), threading, collections.deque, csv, json, time.
- **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.

**2) Hardware setup (GPIO/I²C, pin map)**

- **GPIO mode:** BCM; LEDs on 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 ch 0; motor FWD/REV on PCA ch 1/2.
- **IMU bias:** 500 samples averaged at boot to compute Z-gyro bias.

**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, then start_continuous() with a 20 ms budget.
- **Fallback:** any ToF init exception disables ToF flags and uses ultrasonics for the affected directions.
- **Mixed mode:** front/side ToF enablement is independent; you can run US on sides and ToF on front (or vice-versa).

**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.

**5) RobotController (low-level actuation & signal processing)**

- **Servo command:** set_servo(angle) clamps to [SERVO_MIN_ANGLE, SERVO_MAX_ANGLE]; optional slew limiting:

angle_step ≤ SERVO_SLEW_DPS * dt, where dt is computed via monotonic_ns(). 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.
- **Distance filtering:** filtered_distance(sensor, history, attr, sensor_type):
  - Read (us: distance * 100 cm; tof: range/10 cm); append to bounded history.
  - Median over last N_READINGS; reject spikes >|FILTER_JUMP| vs previous; EMA: y = α·x + (1−α)·y_prev.
  - Returns smoothed cm or 999 sentinel if no valid 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) where “open” is (None or 999 or > TURN_DECISION_THRESHOLD).

**6) Real-time threads**

- **sensor_reader (period ≈ SENSOR_DELAY):**
  - Acquire left/right/front from ToF or US, run the filter, write 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 not set → hold IDLE.
  - On each tick: copy snapshot, update yaw, narrow-mode, telemetry, then run FSM if loop_event is set.

**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().

**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); 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) 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.

**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, GPIO.cleanup().

**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.

**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).

**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.
- **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. **Servo slew** (SERVO_SLEW_DPS) is optional; when enabled it caps Δangle/Δt to damp oscillations and reduce IMU shock during aggressive steering.

## 13.2 Source code

```
0001 | """
0002 | ----------------------------------------------------------------------------------------------------
-------------------------
0003 | Autonomous drive 1st Mission WRO 2025 FE - VivaLaVida
0004 | v5.7
0005 | ----------------------------------------------------------------------------------------------------
-------------------------
0006 | """
0007 |
0008 | # ================================
0009 | # IMPORTS
0010 | # ================================
0011 |
0012 | import threading
0013 | import time
0014 | import tkinter as tk
0015 | from collections import deque
0016 | import RPi.GPIO as GPIO
0017 | from board import SCL, SDA
0018 | import busio
0019 | from adafruit_pca9685 import PCA9685
0020 | import adafruit_mpu6050
0021 | import matplotlib.pyplot as plt
0022 | from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
0023 | import csv
0024 | from datetime import datetime
0025 | import json
0026 | import os
0027 | import tkinter.filedialog as fd
0028 | import board
0029 | import digitalio
0030 | import adafruit_vl53l0x
0031 | from enum import Enum, auto
0032 | import numpy as np
0033 | from gpiozero import DistanceSensor
0034 | #from gpiozero import Device
0035 | #from gpiozero.pins.pigpio import PiGPIOFactory
0036 | from threading import Event
0037 |
0038 | BASE_DIR = os.path.dirname(os.path.abspath(__file__)) # Get directory of the running script
0039 |
0040 | # ================================
0041 | # CONFIGURATION VARIABLES
0042 | # ================================
0043 |
0044 | # ---------- Initialization ----------
0045 | USE_GUI = 1                   # 1 = Debugging mode (run with GUI), 0 = COMPETITION MODE (run
headless, no GUI)
```

```
0046 | USE_TOF_SIDES = 0             # Side sensors:  0 = Ultrasonic, 1 = ToF
0047 | USE_TOF_FRONT = 0             # Front sensor:  0 = Ultrasonic, 1 = ToF
0048 |
0049 | NARROW_SUM_THRESHOLD = 60     # cm; left+right distance threshold to decide if in between
narrow walls
0050 | NARROW_HYSTERESIS = 10        # cm; prevents rapid toggling
0051 | NARROW_FACTOR = 1.0           # multiply by this when narrow, eg speed * factor
0052 | NARROW_FACTOR_SPEED = 1.0     # multiply state speeds when in narrow corridors
0053 | NARROW_FACTOR_DIST  = 1.0     # multiply distance-based thresholds/corrections when in narrow
corridors
0054 |
0055 | # ---------- Speeds ----------
0056 | SPEED_IDLE = 0
0057 | SPEED_STOPPED = 0
0058 | SPEED_CRUISE = 25            # Motor speed for normal straight driving (0-100%)
0059 | SPEED_TURN_INIT = 17          # Motor speed while waiting for open side to turn
0060 | SPEED_TURN = 20               # Motor speed while turning
0061 | SPEED_POST_TURN = 20          # Motor speed following a turn
0062 |
0063 | # ---------- Driving ----------
0064 | SOFT_MARGIN = 30              # Distance from wall where small steering corrections start
(cm)
0065 | MAX_CORRECTION = 7            # Maximum servo correction applied for wall-following (degrees)
0066 | CORRECTION_DURATION = 0.25    # How long a side-correction is held (seconds)
0067 | STOP_THRESHOLD = 20           # Front distance (cm) at which robot stops immediately
0068 | OBSTACLE_WAIT_TIME = 5.0      # seconds to wait before retrying after a front-stop
0069 |
0070 | # ---------- Turn management ----------
0071 | FRONT_TURN_TRIGGER = 90       # Front distance (cm) at which a turn is triggered
0072 | TURN_DECISION_THRESHOLD = 100  # Minimum side distance (cm) to allow turn in that direction
0073 | TURN_ANGLE_LEFT = 60          # Servo angle for left turn
0074 | TURN_ANGLE_RIGHT = 120        # Servo angle for right turn
0075 | FRONT_SAFE_DISTANCE = 160     # Front distance considered safe to end a turn (cm)
0076 | SIDE_SAFE_DISTANCE = 30       # Side distance considered safe to end a turn (cm)
0077 | TARGET_DISTANCE = 30          # Desired distance from side walls (cm)
0078 | TURN_TIMEOUT = 4.5            # Maximum time allowed for a turn (seconds)
0079 | TURN_LOCKOUT = 1.5            # Minimum interval between consecutive turns (seconds)
0080 | POST_TURN_DURATION = 0.5      # Time to drive straight after a turn (seconds)
0081 | LOCK_TURN_DIRECTION = 1       # 1 = enable turn lock direction after 1st turn, 0 = disable
0082 | TARGET_TURN_ANGLE = 85        # Degrees to turn per corner
0083 | TURN_ANGLE_TOLERANCE = 5      # Acceptable overshoot (degrees)
0084 |
0085 | SERVO_SLEW_DPS = 0        # max degrees/second the servo command may change (0 = no limit)
0086 | # Reduce to ~200 for gentler turns, increase to ~450 for sharper response.
0087 | # Set to 0 to disable limiting and go back to "as fast as the servo can".
0088 |
0089 | # ---------- Turn angle constraints ----------
0090 | MIN_TURN_ANGLE = 65           # Minimum yaw change (degrees) before turn can stop
0091 | MAX_TURN_ANGLE = 120          # Maximum yaw change (degrees) to force stop turn
0092 |
0093 | # ---------- Laps ----------
0094 | MAX_LAPS = 3                  # Maximum number of laps before stopping (0 = unlimited)
0095 | POST_LAP_DURATION = 1       # Time to drive forward after final lap before stopping (seconds)
0096 |
0097 | # ---------- Sensor data filtering ----------
0098 | N_READINGS = 1                # Number of readings stored for median filtering
0099 | US_QUEUE_LEN = 5              # Queue_len for Ultrasonic gpiozero.DistanceSensor
0100 | FILTER_ALPHA = 1.0            # Ultrasonic 0.1 = smoother, 0.5 = faster reaction, 1 to ignore
filter
0101 | FILTER_JUMP = 999             # Ultrasonic maximum jump (cm) allowed between readings
0102 | FILTER_ALPHA_TOF = 1.0        # ToF 0.1 = smoother, 0.5 = faster reaction, 1 to ignore filter
0103 | FILTER_JUMP_TOF = 999         # ToF maximum jump (cm) allowed between readings
0104 | US_MAX_DISTANCE_FRONT = 3.43  # Ultrasonic front max read distance
0105 | US_MAX_DISTANCE_SIDE = 1.72   # Ultrasonic side max read distance
0106 |
0107 | # ---------- Loop timing ----------
0108 | LOOP_DELAY = 0.001            # Delay between main loop iterations (seconds) 0.02
0109 | SENSOR_DELAY = 0.005          # Delay between sensor reads
0110 |
0111 | #-------------------------------------------------------------------------------------
0112 |
0113 | # ---------- Plotting ----------
0114 | MAX_POINTS = 500               # Maximum number of points to store for plotting
0115 |
0116 | RAD2DEG = 180.0 / 3.141592653589793
0117 |
0118 | # ================================
```

```
# LOAD VARIABLES FROM JSON (if exists)
# ================================

CONFIG_FILE = os.path.join(BASE_DIR, "1st_mission_variables.json")

def load_variables_from_json(path=CONFIG_FILE):
    """
    Override defaults with values from JSON if the file exists.
    If missing or invalid, keep the built-in defaults from the code.
    """
    if not os.path.exists(path):
        print("⚙️ No external config found — using built-in defaults.")
        return
    try:
         with open(path, "r") as f:
              data = json.load(f)
         count = 0
         for k, v in data.items():
             if k in globals():
                 globals()[k] = v
                 count += 1
         print(f"✅ Loaded {count} variables from {os.path.basename(path)}")
     except Exception as e:
         print(f"[ERROR] Config load failed: {e} — using built-in defaults.")

# Call it right after defining defaults so everything below sees the overrides
load_variables_from_json()

# --- Store immutable base values for scaling ---
BASE_SPEED_IDLE        = SPEED_IDLE
BASE_SPEED_STOPPED     = SPEED_STOPPED
BASE_SPEED_CRUISE      = SPEED_CRUISE
BASE_SPEED_TURN_INIT   = SPEED_TURN_INIT
BASE_SPEED_TURN        = SPEED_TURN
BASE_SPEED_POST_TURN   = SPEED_POST_TURN

BASE_MAX_CORRECTION     = MAX_CORRECTION
BASE_FRONT_TURN_TRIGGER = FRONT_TURN_TRIGGER
BASE_STOP_THRESHOLD     = STOP_THRESHOLD

# --- Global factors (automatically updated) ---
SPEED_ENV_FACTOR = 1.0
DIST_ENV_FACTOR  = 1.0  # for distance-related thresholds

# --- Helper functions to get "effective" (scaled) values ---
def eff_soft_margin():
    """Return scaled soft margin distance (cm) for gentle steering corrections."""
    return int(SOFT_MARGIN * DIST_ENV_FACTOR)

def eff_max_correction():
    """Return scaled maximum steering correction (degrees)."""
    return max(1, int(MAX_CORRECTION * DIST_ENV_FACTOR))

def eff_front_turn_trigger():
    """Return scaled front distance threshold (cm) for initiating a turn."""
    return int(FRONT_TURN_TRIGGER * DIST_ENV_FACTOR)

def eff_stop_threshold():
    """Return scaled front distance threshold (cm) for immediate stop."""
    return int(STOP_THRESHOLD * DIST_ENV_FACTOR)

# --- Scaled state speeds ---
def state_speed_value(state_name: str) -> int:
    base = {
        "IDLE":      SPEED_IDLE,
        "STOPPED":   SPEED_STOPPED,
        "CRUISE":    SPEED_CRUISE,
        "TURN_INIT": SPEED_TURN_INIT,
        "TURNING":   SPEED_TURN,
        "POST_TURN": SPEED_POST_TURN,
    }.get(state_name, SPEED_CRUISE)
    return int(base * SPEED_ENV_FACTOR)

# ================================
# HARDWARE SETUP
# ================================

```

```
0196 | GPIO.setmode(GPIO.BCM)
0197 | GPIO.setwarnings(False)
0198 | i2c = busio.I2C(board.SCL, board.SDA)
0199 |
0200 | START_BTN = 20                    #Button No1
0201 | GPIO.setup(START_BTN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
0202 |
0203 | # ---------- LEDs ----------
0204 | GREEN_LED = 19
0205 | RED_LED   = 13
0206 | GPIO.setup(GREEN_LED, GPIO.OUT, initial=GPIO.LOW)
0207 | GPIO.setup(RED_LED,   GPIO.OUT, initial=GPIO.LOW)
0208 |
0209 | # ---------- Ultrasonic sensor pins ----------
0210 | TRIG_FRONT, ECHO_FRONT = 22, 23  # GPIO pins for front sensor
0211 | TRIG_LEFT, ECHO_LEFT = 27, 17    # GPIO pins for left sensor
0212 | TRIG_RIGHT, ECHO_RIGHT = 5, 6    # GPIO pins for right sensor
0213 |
0214 | # ---------- Motor ----------
0215 | MOTOR_FWD = 1                # PCA9685 channel for forward motor control
0216 | MOTOR_REV = 2                # PCA9685 channel for reverse motor control
0217 |
0218 | # ---------- Servo ----------
0219 | SERVO_CHANNEL = 0            # PCA9685 channel controlling steering servo
0220 | SERVO_CENTER = 88            # Neutral servo angle (degrees)
0221 | SERVO_MIN_ANGLE = 50         # Minimum physical servo angle (degrees)
0222 | SERVO_MAX_ANGLE = 130        # Maximum physical servo angle (degrees)
0223 | SERVO_PULSE_MIN = 1000       # Minimum PWM pulse width (microseconds)
0224 | SERVO_PULSE_MAX = 2000       # Maximum PWM pulse width (microseconds)
0225 | SERVO_PERIOD = 20000         # Servo PWM period (microseconds)
0226 |
0227 | # ================================
0228 | # SENSOR INITIALIZATION (ToF + Ultrasonic, flexible for mixed scenarios)
0229 | # ================================
0230 |
0231 | vl53_left = vl53_right = vl53_front = vl53_back = None
0232 | us_left = us_right = us_front = None
0233 |
0234 | try:
0235 |     # --------- ToF sensors ----------
0236 |     if USE_TOF_SIDES or USE_TOF_FRONT:
0237 |         print("Initializing VL53L0X ToF sensors...")
0238 |         # XSHUT pins setup (for powering sensors individually)
0239 |         xshut_left  = digitalio.DigitalInOut(board.D16)
0240 |         xshut_right = digitalio.DigitalInOut(board.D25)
0241 |         xshut_front = digitalio.DigitalInOut(board.D26)
0242 |         xshut_back  = digitalio.DigitalInOut(board.D24)
0243 |         for xshut in [xshut_left, xshut_right, xshut_front, xshut_back]:
0244 |             xshut.direction = digitalio.Direction.OUTPUT
0245 |             xshut.value = False  # power down
0246 |         time.sleep(0.1)
0247 |
0248 |         # Initialize left ToF if needed
0249 |         if USE_TOF_SIDES:
0250 |             xshut_left.value = True
0251 |             time.sleep(0.05)
0252 |             vl53_left = adafruit_vl53l0x.VL53L0X(i2c)
0253 |             vl53_left.set_address(0x30)
0254 |             vl53_left.measurement_timing_budget = 20000
0255 |             vl53_left.start_continuous()
0256 |             print("✅ Left ToF sensor set to address 0x30")
0257 |
0258 |         # Initialize right ToF if needed
0259 |         if USE_TOF_SIDES:
0260 |             xshut_right.value = True
0261 |             time.sleep(0.05)
0262 |             vl53_right = adafruit_vl53l0x.VL53L0X(i2c)
0263 |             vl53_right.set_address(0x31)
0264 |             vl53_right.measurement_timing_budget = 20000
0265 |             vl53_right.start_continuous()
0266 |             print("✅ Right ToF sensor set to address 0x31")
0267 |
0268 |         # Initialize front ToF if needed
0269 |         if USE_TOF_FRONT:
0270 |             xshut_front.value = True
0271 |             time.sleep(0.05)
0272 |             vl53_front = adafruit_vl53l0x.VL53L0X(i2c)
```

```
0273 |             vl53_front.set_address(0x32)
0274 |             vl53_front.measurement_timing_budget = 20000
0275 |             vl53_front.start_continuous()
0276 |             print("✅ Front ToF sensor set to address 0x32")
0277 |
0278 |         # Initialize back ToF (optional, if you want to use it)
0279 |         xshut_back.value = True
0280 |         time.sleep(0.05)
0281 |         vl53_back = adafruit_vl53l0x.VL53L0X(i2c)
0282 |         vl53_back.set_address(0x33)
0283 |         vl53_back.measurement_timing_budget = 20000
0284 |         vl53_back.start_continuous()
0285 |         print("✅ Back ToF sensor set to address 0x33")
0286 |
0287 | except Exception as e:
0288 |     print(f"[ERROR] VL53L0X initialization failed: {e}")
0289 |     # fallback to ultrasonic if ToF fails
0290 |     USE_TOF_SIDES = 0
0291 |     USE_TOF_FRONT = 0
0292 |     vl53_left = vl53_right = vl53_front = vl53_back = None
0293 |
0294 | time.sleep(0.2)
0295 |
0296 | # --------- Ultrasonic sensors ----------
0297 | # Initialize ultrasonic sensors only if that sensor is set to ultrasonic
0298 | if not USE_TOF_FRONT:
0299 |     us_front = DistanceSensor(echo=ECHO_FRONT, trigger=TRIG_FRONT,
max_distance=US_MAX_DISTANCE_FRONT, queue_len=US_QUEUE_LEN)
0300 | if not USE_TOF_SIDES:
0301 |     us_left  = DistanceSensor(echo=ECHO_LEFT,  trigger=TRIG_LEFT,
max_distance=US_MAX_DISTANCE_SIDE, queue_len=US_QUEUE_LEN)
0302 |     us_right = DistanceSensor(echo=ECHO_RIGHT, trigger=TRIG_RIGHT,
max_distance=US_MAX_DISTANCE_SIDE, queue_len=US_QUEUE_LEN)
0303 |
0304 | print("Sensors initialized:")
0305 | print(f"  Front: {'ToF' if USE_TOF_FRONT else 'Ultrasonic'}")
0306 | print(f"  Left : {'ToF' if USE_TOF_SIDES else 'Ultrasonic'}")
0307 | print(f"  Right: {'ToF' if USE_TOF_SIDES else 'Ultrasonic'}")
0308 |
0309 | pca = PCA9685(i2c)
0310 | pca.frequency = 50
0311 | mpu = adafruit_mpu6050.MPU6050(i2c)
0312 |
0313 | # ---------- Calibrate gyro bias at startup ----------
0314 | print("Calibrating gyro...")
0315 | N = 500
0316 | bias = 0
0317 | for _ in range(N):
0318 |     bias += mpu.gyro[2]
0319 |     time.sleep(0.005)
0320 | bias /= N
0321 | #print(f"Gyro bias: {bias}")
0322 |
0323 | # ==============================
0324 | # CONTROL FLAGS & STORAGE
0325 | # ==============================
0326 |
0327 | STATE_SPEED = {
0328 |     "IDLE": SPEED_IDLE,
0329 |     "STOPPED": SPEED_STOPPED,
0330 |     "CRUISE": SPEED_CRUISE,
0331 |     "TURN_INIT": SPEED_TURN_INIT,
0332 |     "TURNING": SPEED_TURN,
0333 |     "POST_TURN": SPEED_POST_TURN
0334 | }
0335 |
0336 | # ---------- CONTROL FLAGS ----------
0337 | readings_event = Event()
0338 | loop_event = Event()
0339 | sensor_tick = Event()   # used to wake waits immediately
0340 | status_text = "Idle"
0341 | turn_count = 0
0342 | lap_count = 0
0343 | # Reason-aware stopping (USER / OBSTACLE / LAPS)
0344 | stop_reason = None
0345 | obstacle_wait_deadline = 0.0
0346 |
```

```
# ---------- SENSOR DATA STORAGE ----------
time_data = deque(maxlen=MAX_POINTS)
front_data = deque(maxlen=MAX_POINTS)
left_data = deque(maxlen=MAX_POINTS)
right_data = deque(maxlen=MAX_POINTS)
angle_data = deque(maxlen=MAX_POINTS)
state_data = deque(maxlen=MAX_POINTS)

# ================================
# THREADED SENSOR READING
# ================================
sensor_data = {
    "front": None,
    "left": None,
    "right": None
}
sensor_lock = threading.Lock()

# ================================
# ROBOT CONTROLLER
# ================================

class RobotController:
    def __init__(self, pca):
        self.pca = pca
        self.front_history = deque(maxlen=N_READINGS)
        self.left_history = deque(maxlen=N_READINGS)
        self.right_history = deque(maxlen=N_READINGS)
        self.smooth_left = None
        self.smooth_right = None
        self.smooth_front = None
        self.d_front = None
        self.d_left = None
        self.d_right = None
        self.sensor_index = 0
        self.gyro_z_prev = 0
        self._servo_last_angle = SERVO_CENTER
        self._servo_last_ns = time.monotonic_ns()

    def set_filter_size(self, n):
        self.front_history = deque(list(self.front_history)[-n:], maxlen=n)
        self.left_history = deque(list(self.left_history)[-n:], maxlen=n)
        self.right_history = deque(list(self.right_history)[-n:], maxlen=n)

    #def set_servo(self, angle):
    #    angle = max(SERVO_MIN_ANGLE, min(SERVO_MAX_ANGLE, angle))
    #    pulse = int(SERVO_PULSE_MIN + (SERVO_PULSE_MAX - SERVO_PULSE_MIN) *
    #                ((angle - SERVO_MIN_ANGLE) / (SERVO_MAX_ANGLE - SERVO_MIN_ANGLE)))
    #    self.pca.channels[SERVO_CHANNEL].duty_cycle = int(pulse * 65535 / SERVO_PERIOD)

    def set_servo(self, angle):
        # 1) clamp target
        target = max(SERVO_MIN_ANGLE, min(SERVO_MAX_ANGLE, angle))

        # 2) simple built-in slew limiter using one knob: SERVO_SLEW_DPS
        now_ns = time.monotonic_ns()
        dt = max(0.0, (now_ns - self._servo_last_ns) * 1e-9)
        if SERVO_SLEW_DPS > 0 and dt > 0:
            max_step = SERVO_SLEW_DPS * dt
            diff = target - self._servo_last_angle
            if abs(diff) > max_step:
                target = self._servo_last_angle + (max_step if diff > 0 else -max_step)

        # 3) remember for next call
        self._servo_last_angle = target
        self._servo_last_ns = now_ns

        # 4) convert to pulse & send (unchanged)
        pulse = int(SERVO_PULSE_MIN + (SERVO_PULSE_MAX - SERVO_PULSE_MIN) *
                    ((target - SERVO_MIN_ANGLE) / (SERVO_MAX_ANGLE - SERVO_MIN_ANGLE)))
        self.pca.channels[SERVO_CHANNEL].duty_cycle = int(pulse * 65535 / SERVO_PERIOD)

        return target  # handy if you want to log the actual command

    def rotate_motor(self, speed):
        duty_cycle = int(min(max(abs(speed), 0), 100)/100*65535)
        if speed >= 0:
```

```
0424 |             self.pca.channels[MOTOR_FWD].duty_cycle = duty_cycle
0425 |             self.pca.channels[MOTOR_REV].duty_cycle = 0
0426 |         else:
0427 |             self.pca.channels[MOTOR_FWD].duty_cycle = 0
0428 |             self.pca.channels[MOTOR_REV].duty_cycle = duty_cycle
0429 |
0430 |     def stop_motor(self):
0431 |         self.pca.channels[MOTOR_FWD].duty_cycle = 0
0432 |         self.pca.channels[MOTOR_REV].duty_cycle = 0
0433 |
0434 |     def set_state_speed(self, state):
0435 |         # Set motor speed based on FSM state.
0436 |         #speed = STATE_SPEED.get(state, SPEED_CRUISE)  # default to cruise if unknown
0437 |         speed = state_speed_value(state)
0438 |         self.rotate_motor(speed)
0439 |
0440 |     def get_distance(self, sensor):
0441 |         #sensor: a gpiozero.DistanceSensor instance
0442 |         try:
0443 |             d = sensor.distance  # returns meters
0444 |             if d is None:
0445 |                 return None
0446 |             return d * 100  # convert to cm
0447 |         except:
0448 |             return None
0449 |
0450 |     def stable_filter(self, new_val, prev_val, alpha=FILTER_ALPHA, max_jump=FILTER_JUMP):
0451 |     #Reject large spikes and apply exponential moving average smoothing.
0452 |         if new_val is None:
0453 |             return prev_val  # keep previous if invalid
0454 |         if prev_val is not None and abs(new_val - prev_val) > max_jump:
0455 |             new_val = prev_val  # reject spike
0456 |         if prev_val is None:
0457 |             return new_val
0458 |         return alpha * new_val + (1 - alpha) * prev_val
0459 |
0460 |     def filtered_distance(self, sensor_obj, history, smooth_attr, sensor_type='us'):
0461 |          try:
0462 |              d = sensor_obj.range / 10.0 if sensor_type == 'tof' else sensor_obj.distance *
100.0
0463 |          except:
0464 |              d = None
0465 |
0466 |          history.append(d)
0467 |          valid = [x for x in history if x is not None]
0468 |          if not valid:
0469 |              return 999
0470 |
0471 |          median_val = np.median(valid)
0472 |          avg_val = np.mean(valid)
0473 |          #filtered_val = 0.7 * median_val + 0.3 * avg_val
0474 |          filtered_val = median_val
0475 |
0476 |          prev_val = getattr(self, smooth_attr)
0477 |          alpha = FILTER_ALPHA_TOF if sensor_type == 'tof' else FILTER_ALPHA
0478 |          max_jump = FILTER_JUMP_TOF if sensor_type == 'tof' else FILTER_JUMP
0479 |          smoothed_val = self.stable_filter(filtered_val, prev_val, alpha, max_jump)
0480 |
0481 |          setattr(self, smooth_attr, smoothed_val)
0482 |          return smoothed_val
0483 |
0484 |
0485 |     def safe_straight_control(self, d_left, d_right):
0486 |         correction = 0
0487 |         if d_left is not None and d_left < eff_soft_margin():
0488 |             correction = (eff_soft_margin() - d_left)*2
0489 |         elif d_right is not None and d_right < eff_soft_margin():
0490 |             correction = -(eff_soft_margin() - d_right)*2
0491 |         #correction = max(-MAX_CORRECTION, min(MAX_CORRECTION, correction))
0492 |         correction = max(-eff_max_correction(), min(eff_max_correction(), correction))
0493 |         #return SERVO_CENTER + correction
0494 |         angle = SERVO_CENTER + correction
0495 |         angle = max(SERVO_MIN_ANGLE, min(SERVO_MAX_ANGLE, angle))
0496 |         return angle
0497 |
0498 |     def turn_decision(self, d_left, d_right):
```

```
0499 |         left_open  = (d_left  is None) or (d_left  == 999) or (d_left  >
TURN_DECISION_THRESHOLD)
0500 |         right_open = (d_right is None) or (d_right == 999) or (d_right >
TURN_DECISION_THRESHOLD)
0501 |
0502 |         # Decide only when exactly one side is open; otherwise keep trying.
0503 |         if left_open ^ right_open:              #XOR: exactly one is True
0504 |             return "LEFT" if left_open else "RIGHT"
0505 |         return None # both open or both closed → keep trying
0506 |
0507 | robot = RobotController(pca)
0508 |
0509 | def sensor_reader():
0510 |     global sensor_data
0511 |     while True:
0512 |         # Left sensor
0513 |         if vl53_left:  # ToF
0514 |             left = robot.filtered_distance(vl53_left, robot.left_history, "smooth_left")
0515 |         elif us_left:  # ultrasonic
0516 |             left = robot.filtered_distance(us_left, robot.left_history, "smooth_left")
0517 |         else:
0518 |             left = None
0519 |
0520 |         # Right sensor
0521 |         if vl53_right:
0522 |             right = robot.filtered_distance(vl53_right, robot.right_history, "smooth_right")
0523 |         elif us_right:
0524 |             right = robot.filtered_distance(us_right, robot.right_history, "smooth_right")
0525 |         else:
0526 |             right = None
0527 |
0528 |         # Front sensor
0529 |         if vl53_front:
0530 |             front = robot.filtered_distance(vl53_front, robot.front_history, "smooth_front")
0531 |         elif us_front:
0532 |             front = robot.filtered_distance(us_front, robot.front_history, "smooth_front")
0533 |         else:
0534 |             front = None
0535 |
0536 |         # Save readings in global dictionary
0537 |         with sensor_lock:
0538 |             sensor_data["front"] = front
0539 |             sensor_data["left"] = left
0540 |             sensor_data["right"] = right
0541 |
0542 |         #time.sleep(SENSOR_DELAY)
0543 |         sensor_tick.wait(SENSOR_DELAY)
0544 |         sensor_tick.clear()
0545 |
0546 | # ================================
0547 | # FINITE STATE MACHINE: robot_loop
0548 | # ================================
0549 | class RobotState(Enum):
0550 |     IDLE = auto()
0551 |     CRUISE = auto()
0552 |     TURN_INIT = auto()
0553 |     TURNING = auto()
0554 |     POST_TURN = auto()
0555 |     STOPPED = auto()
0556 |
0557 | # Keep a global locked_turn_direction variable to persist across runs (reset in stop_loop)
0558 | locked_turn_direction = None
0559 |
0560 | def robot_loop():
0561 |
0562 |     #global loop_flag, readings_flag, status_text, turn_count, lap_count,
locked_turn_direction
0563 |     global status_text, turn_count, lap_count, locked_turn_direction, stop_reason,
obstacle_wait_deadline
0564 |     state = RobotState.IDLE
0565 |     direction = None
0566 |     yaw = 0.0
0567 |     turn_start_yaw = 0.0
0568 |     turn_start_time = 0.0
0569 |     post_turn_start = 0.0
0570 |     #last_turn_time = -TURN_LOCKOUT
0571 |     last_turn_time = -999
```

```
0572 |      #last_time = time.monotonic()
0573 |      #start_time = time.monotonic()
0574 |      last_ns = time.monotonic_ns()
0575 |      start_ns = last_ns
0576 |      use_post_turn_ref = False
0577 |      post_turn_ref_diff = None
0578 |      correction_active = False
0579 |      correction_start_time = 0.0
0580 |      correction_angle = SERVO_CENTER
0581 |      current_servo_angle = SERVO_CENTER
0582 |
0583 |      # Ensure robot stopped at start
0584 |      robot.stop_motor()
0585 |      robot.set_servo(SERVO_CENTER)
0586 |
0587 |      while True:
0588 |          #current_time = time.monotonic()
0589 |          #dt = current_time - last_time
0590 |          #last_time = current_time
0591 |          current_ns = time.monotonic_ns()
0592 |          dt = (current_ns - last_ns) * 1e-9   # seconds
0593 |          last_ns = current_ns
0594 |          current_time = current_ns * 1e-9     # float seconds, used by UI/timers
0595 |
0596 |          #if not readings_flag or not loop_flag:
0597 |          #if not readings_flag:
0598 |          if not readings_event.is_set():
0599 |              # if readings not started, keep motors stopped and idle
0600 |              robot.stop_motor()
0601 |              state = RobotState.IDLE
0602 |              status_text = "Idle"
0603 |              #time.sleep(LOOP_DELAY)
0604 |              sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0605 |              continue
0606 |
0607 |          # ----------------------------
0608 |          # Sensor reads & gyro integration (always update while readings_flag)
0609 |          # ----------------------------
0610 |          with sensor_lock:
0611 |              d_front = sensor_data["front"]
0612 |              d_left = sensor_data["left"]
0613 |              d_right = sensor_data["right"]
0614 |
0615 |          # Update robot attributes for plotting/logging and decision making
0616 |          robot.d_front = d_front
0617 |          robot.d_left = d_left
0618 |          robot.d_right = d_right
0619 |
0620 |          # ---- Narrow corridor detector (sum of side distances) ----
0621 |          if not hasattr(robot_loop, "_narrow_mode"):
0622 |              robot_loop._narrow_mode = False
0623 |
0624 |          # Use None-safe math (ignore invalid readings)
0625 |          l = robot.d_left  if (robot.d_left  is not None and robot.d_left  != 999) else None
0626 |          r = robot.d_right if (robot.d_right is not None and robot.d_right != 999) else None
0627 |          sum_lr = None if (l is None or r is None) else (l + r)
0628 |
0629 |          enter_thresh = NARROW_SUM_THRESHOLD
0630 |          exit_thresh  = NARROW_SUM_THRESHOLD + NARROW_HYSTERESIS
0631 |
0632 |          prev_mode = robot_loop._narrow_mode
0633 |          if sum_lr is not None:
0634 |              if not robot_loop._narrow_mode and sum_lr < enter_thresh:
0635 |                  robot_loop._narrow_mode = True
0636 |              elif robot_loop._narrow_mode and sum_lr > exit_thresh:
0637 |                  robot_loop._narrow_mode = False
0638 |
0639 |          # Apply global factors
0640 |          global SPEED_ENV_FACTOR, DIST_ENV_FACTOR
0641 |          SPEED_ENV_FACTOR = NARROW_FACTOR_SPEED if robot_loop._narrow_mode else 1.0
0642 |          DIST_ENV_FACTOR  = NARROW_FACTOR_DIST  if robot_loop._narrow_mode else 1.0
0643 |
0644 |          # Optional console feedback
0645 |          if robot_loop._narrow_mode != prev_mode:
0646 |              mode_txt = "NARROW mode ON" if robot_loop._narrow_mode else "NARROW mode OFF"
0647 |              print(f"[corridor] {mode_txt} (L+R = {sum_lr:.1f} cm)" if sum_lr is not None else
f"[corridor] {mode_txt}")
```

```
0648 |
0649 |         # Gyro integration
0650 |         #raw_gyro_z = mpu.gyro[2] - bias
0651 |         #ALPHA = 0.8
0652 |         #gyro_z_filtered = ALPHA * raw_gyro_z + (1 - ALPHA) * getattr(robot, 'gyro_z_prev',
0)
0653 |         #robot.gyro_z_prev = gyro_z_filtered
0654 |         #gyro_z_deg = gyro_z_filtered * (180 / 3.14159265)
0655 |         #yaw += gyro_z_deg * dt
0656 |
0657 |         raw_gyro_z = mpu.gyro[2] - bias      # rad/s from MPU6050
0658 |         ALPHA = 0.8
0659 |         gyro_z_filtered = ALPHA * raw_gyro_z + (1 - ALPHA) * getattr(robot, 'gyro_z_prev',
0.0)
0660 |         robot.gyro_z_prev = gyro_z_filtered
0661 |         yaw += (gyro_z_filtered * RAD2DEG) * dt
0662 |
0663 |
0664 |         # Append to deques for plotting/logging
0665 |         #elapsed_time = current_time - start_time
0666 |         elapsed_time = (current_ns - start_ns) * 1e-9
0667 |         time_data.append(elapsed_time)
0668 |         front_data.append(robot.d_front if robot.d_front is not None else 0)
0669 |         left_data.append(robot.d_left if robot.d_left is not None else 0)
0670 |         right_data.append(robot.d_right if robot.d_right is not None else 0)
0671 |         angle_data.append(yaw)
0672 |         state_data.append(state.name)
0673 |
0674 |         # If loop_flag is still False, do NOT run FSM/motors yet—just update plots
0675 |         #if not loop_flag:
0676 |         if not loop_event.is_set():
0677 |             robot.stop_motor()
0678 |             state = RobotState.IDLE
0679 |             status_text = "Sensor readings started"
0680 |             #time.sleep(LOOP_DELAY)
0681 |             sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0682 |             continue
0683 |
0684 |         # ----------------------------
0685 |         # FSM transitions & actions
0686 |         # ----------------------------
0687 |         if state == RobotState.IDLE:
0688 |             #status_text = "Ready (readings started, loop not started)" if readings_flag else
"Idle"
0689 |             status_text = "Ready (readings started, loop not started)" if
readings_event.is_set() else "Idle"
0690 |             robot.stop_motor()
0691 |             robot.set_servo(SERVO_CENTER)
0692 |             correction_active = False
0693 |             #if loop_flag:
0694 |             if loop_event.is_set():
0695 |                 state = RobotState.CRUISE
0696 |                 status_text = "Driving (cruise)"
0697 |                 correction_active = False  # Reset correction when transitioning to CRUISE
0698 |                 for speed in range(0, SPEED_CRUISE + 1):
0699 |                     #if not loop_flag:
0700 |                     if not loop_event.is_set():
0701 |                         robot.stop_motor()
0702 |                         break
0703 |                     robot.rotate_motor(speed)
0704 |                     #time.sleep(0.01)
0705 |                     sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0706 |
0707 |         elif state == RobotState.CRUISE:
0708 |             status_text = "Driving (cruise)"
0709 |
0710 |             # -------------------------------
0711 |             # Emergency stop: immediate
0712 |             # -------------------------------
0713 |             if robot.d_front is not None and robot.d_front < eff_stop_threshold():
0714 |                 robot.stop_motor()
0715 |                 robot.set_servo(SERVO_CENTER)
0716 |                 status_text = "Stopped! Obstacle ahead"
0717 |                 print(status_text)
0718 |                 state = RobotState.STOPPED
0719 |
0720 |                 stop_reason = "OBSTACLE"
```

```
0721 |                 obstacle_wait_deadline = current_time + OBSTACLE_WAIT_TIME
0722 |                 continue
0723 |
0724 |             # -------------------------------
0725 |             # Check if we can trigger a turn
0726 |             # -------------------------------
0727 |             front_triggered = (robot.d_front is not None and robot.d_front <
eff_front_turn_trigger())
0728 |             lockout_ok = (current_time - last_turn_time >= TURN_LOCKOUT)
0729 |
0730 |             if front_triggered and lockout_ok and loop_event.is_set():
0731 |                 # immediately enter TURN_INIT and drop speed
0732 |                 state = RobotState.TURN_INIT
0733 |                 status_text = "Approaching turn - waiting for side to open"
0734 |                 continue
0735 |
0736 |             # -------------------------------
0737 |             # Normal cruise servo control
0738 |             # Safe straight control active only after the 1st turn
0739 |             # -------------------------------
0740 |             if turn_count >= 1:
0741 |             # Wall-following correction after the first turn
0742 |             desired_servo_angle = robot.safe_straight_control(robot.d_left, robot.d_right)
0743 |
0744 |                 # Start correction if robot is too close to a wall
0745 |                 if (robot.d_left is not None and robot.d_left < eff_soft_margin()) or
(robot.d_right is not None and robot.d_right < eff_soft_margin()):
0746 |                     if not correction_active:  # Start correction if not already active
0747 |                         correction_active = True
0748 |                         correction_start_time = current_time  # Record the start time of
correction
0749 |                         status_text = "Driving (correction)"
0750 |                         print(f"↔ Correction started. Servo angle: {desired_servo_angle}")
0751 |
0752 |                 # If correction is active, apply the correction angle
0753 |                 if correction_active:
0754 |                     # If correction duration is up, stop correction
0755 |                     if current_time - correction_start_time >= CORRECTION_DURATION:
0756 |                         correction_active = False
0757 |                         desired_servo_angle = SERVO_CENTER  # Reset to center once correction
is done
0758 |                         status_text = "Driving (correction ended)"
0759 |                         print(f"↔ Correction ended. Servo angle: {desired_servo_angle}")
0760 |
0761 |                 robot.set_servo(desired_servo_angle)  # Apply the desired servo angle
(corrected or not)
0762 |
0763 |             else:
0764 |                 # Before the first turn, keep the servo centered and don't apply correction
0765 |                 desired_servo_angle = SERVO_CENTER
0766 |                 correction_active = False
0767 |                 robot.set_servo(desired_servo_angle)  # Keep centered
0768 |
0769 |             # -------------------------------
0770 |             # Timed servo correction behavior
0771 |             # -------------------------------
0772 |             if correction_active:
0773 |                 if current_time - correction_start_time >= CORRECTION_DURATION:
0774 |                     # Re-evaluate desired angle; if still not center, renew correction
0775 |                     if desired_servo_angle != SERVO_CENTER:
0776 |                         correction_start_time = current_time
0777 |                         correction_angle = desired_servo_angle
0778 |                         robot.set_servo(correction_angle)
0779 |                         status_text = "Driving (renewed correction)"
0780 |                         print(f"↔ Straight correction renewed. Servo angle:
{correction_angle}")
0781 |                     else:
0782 |                         correction_active = False
0783 |
0784 |             #robot.rotate_motor(SPEED_CRUISE)
0785 |             robot.set_state_speed(state.name)
0786 |
0787 |         elif state == RobotState.TURN_INIT:
0788 |             # Stay slow and keep wheels mostly straight while we wait for a side to open.
0789 |             # If you prefer slight wall-following here, use safe_straight_control().
0790 |             status_text = "Turn init - waiting for open side"
0791 |             robot.set_state_speed(state.name)
```

```
0792 |             correction_active = False  # Reset when entering TURN_INIT
0793 |
0794 |             # fail-safe: if front becomes safe again, return to cruise
0795 |             if robot.d_front is not None and robot.d_front >= eff_front_turn_trigger():
0796 |                 robot.set_servo(SERVO_CENTER)
0797 |                 state = RobotState.CRUISE
0798 |                 status_text = "Driving (cruise)"
0799 |                 # brief yield
0800 |                 sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0801 |                 continue
0802 |
0803 |             # keep the wheels centered (or gentle straight correction)
0804 |             #desired_servo_angle = robot.safe_straight_control(robot.d_left, robot.d_right)
if turn_count >= 1 else SERVO_CENTER
0805 |             #robot.set_servo(desired_servo_angle)
0806 |             if turn_count >= 1:
0807 |                  desired_servo_angle = robot.safe_straight_control(robot.d_left,
robot.d_right)
0808 |             else:
0809 |                  # No correction before first turn: keep the wheels centered (or gentle
straight correction)
0810 |                  desired_servo_angle = SERVO_CENTER
0811 |             robot.set_servo(desired_servo_angle)
0812 |
0813 |             # Decide direction only when exactly one side is open
0814 |             proposed_direction = robot.turn_decision(robot.d_left, robot.d_right)
0815 |
0816 |             # If neither or both are open, keep waiting at TURN_INIT speed
0817 |             if proposed_direction is None or proposed_direction == 999:
0818 |                 sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0819 |                 continue
0820 |
0821 |             # Turn lock mechanic (unchanged logic, just applied here)
0822 |             if LOCK_TURN_DIRECTION == 1:
0823 |                 if locked_turn_direction is None:
0824 |                     locked_turn_direction = proposed_direction
0825 |                     print(f"Turn direction locked to {locked_turn_direction}")
0826 |                 elif proposed_direction != locked_turn_direction:
0827 |                     # keep waiting at TURN_INIT speed for the locked direction to open
0828 |                     status_text = f"Ignoring {proposed_direction} (locked to
{locked_turn_direction})"
0829 |                     robot.set_servo(desired_servo_angle)
0830 |                     sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
0831 |                     continue
0832 |                 direction = locked_turn_direction
0833 |             else:
0834 |                 direction = proposed_direction
0835 |
0836 |             # Commit the turn now that one side is open and lock (if any) allows it
0837 |             print(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")
0838 |             turn_start_yaw = yaw
0839 |             turn_start_time = current_time
0840 |             angle = TURN_ANGLE_LEFT if direction == "LEFT" else TURN_ANGLE_RIGHT
0841 |             robot.set_servo(angle)
0842 |             robot.set_state_speed("TURNING")
0843 |             state = RobotState.TURNING
0844 |
0845 |         elif state == RobotState.TURNING:
0846 |             # active turning: monitor yaw & safety/time conditions to stop
0847 |             # compute turn angle relative to start
0848 |             turn_angle = yaw - turn_start_yaw  # positive for LEFT, negative for RIGHT
0849 |             target_angle = TARGET_TURN_ANGLE if direction == "LEFT" else -TARGET_TURN_ANGLE
0850 |             robot.set_state_speed("TURNING")
0851 |             stop_condition = False
0852 |
0853 |             # Condition A: side distance convergence (monitor opposite wall)
0854 |             #side_distance_monitor = robot.d_left if direction == "RIGHT" else robot.d_right
0855 |             #MIN_SIDE_CHANGE = 5  # cm
0856 |             #if side_distance_monitor is not None and turn_reference_side_distance is not
None:
0857 |             #    if abs(side_distance_monitor - turn_reference_side_distance) >
MIN_SIDE_CHANGE:
0858 |             #        lower_bound = turn_reference_side_distance * 0.95
0859 |             #        upper_bound = turn_reference_side_distance * 1.05
0860 |             #        if lower_bound <= side_distance_monitor <= upper_bound:
0861 |             #            print("Stop Turn - Opposite distance")
```

```
0862 |              #            stop_condition = True
0863 |
0864 |              # Condition B: front + side safe
0865 |              #side_distance = robot.d_left if direction == "LEFT" else robot.d_right
0866 |              #if robot.d_front is not None and robot.d_front > FRONT_SAFE_DISTANCE and
side_distance is not None and side_distance > SIDE_SAFE_DISTANCE:
0867 |              #    print("Stop Turn - Front distance")
0868 |              #    stop_condition = True
0869 |
0870 |              # Condition C: reached target angle ± tolerance
0871 |              if abs(turn_angle - target_angle) <= TURN_ANGLE_TOLERANCE:
0872 |                  print("Stop Turn - Target Angle")
0873 |                  stop_condition = True
0874 |
0875 |              # Condition D: timeout or extreme yaw
0876 |              if current_time - turn_start_time > TURN_TIMEOUT:
0877 |                  print("Stop Turn - Max turn time")
0878 |                  stop_condition = True
0879 |              if abs(turn_angle) >= MAX_TURN_ANGLE:
0880 |                  print("Stop Turn - Max turn angle")
0881 |                  stop_condition = True
0882 |
0883 |              # If any stop condition met -> finish turn
0884 |              if stop_condition:
0885 |                  robot.stop_motor()
0886 |                  robot.set_servo(SERVO_CENTER)
0887 |                  current_servo_angle = SERVO_CENTER
0888 |                  last_turn_time = current_time
0889 |                  # Snap yaw to target for consistency
0890 |                  yaw = turn_start_yaw + target_angle
0891 |                  # update counters
0892 |                  turn_count += 1
0893 |                  if turn_count % 4 == 0:
0894 |                      lap_count += 1
0895 |                      print(f"✅ Lap completed! Total laps: {lap_count}")
0896 |                      # if reached max laps -> do final drive then stop (as in original logic)
0897 |                      if MAX_LAPS > 0 and lap_count >= MAX_LAPS:
0898 |                          # Drive forward a little before stopping
0899 |                          robot.set_servo(SERVO_CENTER)
0900 |                          robot.set_state_speed(state.name)
0901 |                          #time.sleep(POST_LAP_DURATION)
0902 |                          sensor_tick.wait(POST_LAP_DURATION); sensor_tick.clear()
0903 |                          robot.stop_motor()
0904 |                          stop_reason = "LAPS"
0905 |                          #loop_flag = False
0906 |                          #state = RobotState.STOPPED
0907 |                          #status_text = f"Stopped - Max Laps ({MAX_LAPS}) reached"
0908 |                          loop_event.clear()
0909 |                          sensor_tick.set()  # wake waits immediately
0910 |                          state = RobotState.STOPPED
0911 |                          status_text = f"Stopped - Max Laps ({MAX_LAPS}) reached"
0912 |                          continue
0913 |
0914 |                  # prepare post-turn
0915 |                  post_turn_start = current_time
0916 |                  state = RobotState.POST_TURN
0917 |                  status_text = "Driving (post-turn)"
0918 |
0919 |          elif state == RobotState.POST_TURN:
0920 |              # drive straight for short duration then return to CRUISE
0921 |              if current_time - post_turn_start < POST_TURN_DURATION:
0922 |                  robot.set_servo(SERVO_CENTER)
0923 |                  robot.set_state_speed(state.name)
0924 |                  status_text = "Driving (post-turn)"
0925 |              else:
0926 |                  state = RobotState.CRUISE
0927 |                  status_text = "Driving (cruise)"
0928 |
0929 |          #elif state == RobotState.STOPPED:
0930 |              # motors must be stopped; remain stopped until loop_flag cleared by GUI or user
0931 |          elif state == RobotState.STOPPED:
0932 |              robot.stop_motor()
0933 |              robot.set_servo(SERVO_CENTER)
0934 |
0935 |              # --- Obstacle-driven stop: auto-retry after OBSTACLE_WAIT_TIME ---
0936 |              if stop_reason == "OBSTACLE":
0937 |                  # 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
                        print("✅ 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."""
    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
    global lbl_turns, lbl_laps
    global sliders_frame, slider_vars, btn_export

    root = tk.Tk()
    root.title("VivaLaVida Robot Control")

    slider_vars = {}

    # ---------- Matplotlib Figure ----------
    fig, (ax_front, ax_side, ax_angle) = plt.subplots(3, 1, figsize=(6, 8))
    fig.tight_layout(pad=3.0)

    ax_front.set_ylim(0, 350)
    ax_front.set_title("Front Sensor")
    ax_front.grid(True)
    front_line, = ax_front.plot([], [], color="blue")

    ax_side.set_ylim(-150, 150)
    ax_side.set_title("Left (+Y) vs Right (-Y) Sensors")
    ax_side.grid(True)
    ax_side.axhline(0, color="black")
    left_line, = ax_side.plot([], [], color="green")
    right_line, = ax_side.plot([], [], color="orange")

    ax_angle.set_ylim(-180, 180)
    ax_angle.set_title("Yaw Angle")
    ax_angle.grid(True)
    angle_line, = ax_angle.plot([], [], color="purple")

    # ---------- Buttons ----------
    btn_readings = tk.Button(root, text="Start Readings", command=start_readings,
                             width=20, height=2, bg="blue", fg="white")
    btn_start = tk.Button(root, text="Start Loop", command=start_loop,
                          width=20, height=2, bg="green", fg="white", state="disabled")
    btn_stop = tk.Button(root, text="Stop Loop", command=stop_loop,
                         width=20, height=2, bg="red", fg="white", state="disabled")

    # ---------- Status Labels ----------
```

```
1015 |     lbl_status = tk.Label(root, text="Idle", font=("Arial", 14))
1016 |     lbl_turns = tk.Label(root, text=f"Turns: {turn_count}", font=("Arial", 14))
1017 |     lbl_laps = tk.Label(root, text=f"Laps: {lap_count}", font=("Arial", 14))
1018 |
1019 |     # ---------- Circular Status Indicator ----------
1020 |     status_canvas = tk.Canvas(root, width=100, height=100, highlightthickness=0,
bg=root.cget("bg"))
1021 |     status_circle = status_canvas.create_oval(10, 10, 90, 90, fill="grey", outline="")
1022 |     status_text_id = status_canvas.create_text(50, 50, text="IDLE", fill="white",
font=("Arial", 14, "bold"))
1023 |
1024 |     # ---------- Sliders Frame ----------
1025 |     sliders_frame = tk.LabelFrame(root, text="Parameters", padx=10, pady=10)
1026 |     sliders_frame.grid(row=0, column=1, rowspan=8, sticky="ns", padx=10, pady=5)
1027 |
1028 |     # ---------- Column 3 frame for actions ----------
1029 |     actions_frame = tk.LabelFrame(root, text="Actions", padx=10, pady=10)
1030 |     actions_frame.grid(row=0, column=2, rowspan=8, sticky="ns", padx=10, pady=5)
1031 |
1032 |     # ---------- Two-column layout ----------
1033 |     btn_readings.grid(row=0, column=0, sticky="ew", padx=5, pady=2)
1034 |     btn_start.grid(row=1, column=0, sticky="ew", padx=5, pady=2)
1035 |     btn_stop.grid(row=2, column=0, sticky="ew", padx=5, pady=2)
1036 |     lbl_status.grid(row=3, column=0, pady=5)
1037 |     lbl_turns.grid(row=4, column=0, pady=2)
1038 |     lbl_laps.grid(row=5, column=0, pady=2)
1039 |     status_canvas.grid(row=6, column=0, pady=5)
1040 |
1041 |     canvas = FigureCanvasTkAgg(fig, master=root)
1042 |     canvas.get_tk_widget().grid(row=7, column=0, sticky="nsew", padx=5, pady=5)
1043 |
1044 |     root.grid_columnconfigure(0, weight=1)  # main buttons + plots
1045 |     root.grid_columnconfigure(1, weight=0)  # sliders
1046 |     root.grid_columnconfigure(2, weight=0)  # actions
1047 |     root.grid_rowconfigure(7, weight=1)     # plot row
1048 |
1049 |     # ---------- Slider Definitions ----------
1050 |     slider_groups = {
1051 |         "Speeds": [
1052 |             ("Cruise Speed", "SPEED_CRUISE", 0, 100, "int"),
1053 |             ("Turn Init Speed", "SPEED_TURN_INIT", 0, 100, "int"),
1054 |             ("Turn Speed", "SPEED_TURN", 0, 100, "int"),
1055 |             ("Post Turn Speed", "SPEED_POST_TURN", 0, 100, "int")
1056 |         ],
1057 |         "Driving & Safety, Wall following, Turns": [
1058 |             ("Turn angle Left", "TURN_ANGLE_LEFT", 55, 90, "int"),
1059 |             ("Turn angle Right", "TURN_ANGLE_RIGHT", 90, 125, "int"),
1060 |             ("Soft side Margin (cm)", "SOFT_MARGIN", 0, 75, "int"),
1061 |             ("Max Angle Correction dif at Soft Margin (o)", "MAX_CORRECTION", 0, 30, "int"),
1062 |             ("Stop Threshold (cm)", "STOP_THRESHOLD", 0, 100, "int"),
1063 |             ("Front Turn Trigger (cm)", "FRONT_TURN_TRIGGER", 50, 150, "int"),
1064 |             ("Turn Decision Threshold (Left/Right) (cm)", "TURN_DECISION_THRESHOLD", 0, 200,
"int"),
1065 |             ("Turn Timeout (s)", "TURN_TIMEOUT", 0.1, 10, "float"),
1066 |             ("Max Turn Angle (o)", "MAX_TURN_ANGLE", 90, 120, "int"),
1067 |             ("Post Turn Duration (s)", "POST_TURN_DURATION", 0, 5, "float"),
1068 |             ("Turn Lockout (s)", "TURN_LOCKOUT", 0.1, 5, "float")
1069 |         ],
1070 |         "Other": [
1071 |             ("Sensor Filter N (Median)", "N_READINGS", 1, 10, "int"),
1072 |             ("Max Laps - 0 for infinite", "MAX_LAPS", 0, 20, "int"),
1073 |             ("Filter Smoothness (alpha)", "FILTER_ALPHA", 0.05, 0.9, "float"),
1074 |             ("Max Jump (cm)", "FILTER_JUMP", 5, 999, "int"),
1075 |         ]
1076 |     }
1077 |
1078 |     # ---------- Create Sliders ----------
1079 |     for group_name, sliders in slider_groups.items():
1080 |         group_frame = tk.LabelFrame(sliders_frame, text=group_name, padx=5, pady=5)
1081 |         group_frame.pack(fill="x", pady=5)
1082 |         for label_text, var_name, vmin, vmax, vartype in sliders:
1083 |             frame = tk.Frame(group_frame)
1084 |             frame.pack(fill="x", pady=2)
1085 |
1086 |             if vartype == "float":
1087 |                 var = tk.DoubleVar(value=globals()[var_name])
1088 |                 res = 0.1
```

```
1089 |             else:
1090 |                 var = tk.IntVar(value=globals()[var_name])
1091 |                 res = 1
1092 |
1093 |             slider_vars[var_name] = var
1094 |
1095 |             scale = tk.Scale(frame, from_=vmin, to=vmax, orient="horizontal",
1096 |                              variable=var, resolution=res)
1097 |             scale.pack(side="left", fill="x", expand=True)
1098 |
1099 |             lbl_val = tk.Label(frame, text=f"{label_text}: {var.get()}", width=25,
anchor="w")
1100 |             lbl_val.pack(side="right", padx=5)
1101 |
1102 |             def make_callback(lbl, name, var, label_text):
1103 |                 def callback(value, label_text=label_text):
1104 |                     globals()[name] = var.get()
1105 |                     if name == "MAX_LAPS" and var.get() == 0:
1106 |                         lbl.config(text=f"{label_text}: ∞")
1107 |                     else:
1108 |                         lbl.config(text=f"{label_text}: {var.get():.1f}" if
isinstance(var.get(), float) else f"{label_text}: {var.get()}")
1109 |                 return callback
1110 |
1111 |             scale.config(command=make_callback(lbl_val, var_name, var, label_text))
1112 |
1113 |     # ---------- Buttons under sliders ----------
1114 |     btn_export = tk.Button(actions_frame, text="Export CSV", command=export_data_csv,
1115 |                            width=20, height=2, bg="purple", fg="white")
1116 |     btn_export.pack(pady=10, fill="x")
1117 |
1118 |     btn_save_sliders = tk.Button(actions_frame, text="Save Sliders",
command=save_sliders_json,
1119 |                                  width=20, height=2, bg="green", fg="white")
1120 |     btn_save_sliders.pack(pady=5, fill="x")
1121 |
1122 |     btn_load_sliders = tk.Button(actions_frame, text="Load Sliders",
command=load_sliders_json,
1123 |                                  width=20, height=2, bg="blue", fg="white")
1124 |     btn_load_sliders.pack(pady=5, fill="x")
1125 |
1126 |     # ================================
1127 |     # GUI Update Functions (nested)
1128 |     # ================================
1129 |     def status_color(state: str) -> str:
1130 |         if "Stopped" in state:
1131 |             return "red"
1132 |         elif "Turning" in state:
1133 |             return "yellow"
1134 |         elif "Driving" in state:
1135 |             return "green"
1136 |         elif "Loop Started" in state or "Sensor readings started" in state:
1137 |             return "blue"
1138 |         else:
1139 |             return "grey"
1140 |
1141 |     def status_label(state: str) -> str:
1142 |         if "Stopped" in state:
1143 |             return "STOP"
1144 |         elif "Turning" in state:
1145 |             return "TURN"
1146 |         elif "Driving" in state:
1147 |             return "GO"
1148 |         elif "Loop Started" in state:
1149 |             return "LOOP"
1150 |         elif "Sensor readings started" in state:
1151 |             return "READ"
1152 |         else:
1153 |             return "IDLE"
1154 |
1155 |     def update_status():
1156 |         lbl_status.config(text=status_text)
1157 |         status_canvas.itemconfig(status_circle, fill=status_color(status_text))
1158 |         status_canvas.itemconfig(status_text_id, text=status_label(status_text))
1159 |         lbl_turns.config(text=f"Turns: {turn_count}")
1160 |         lbl_laps.config(text=f"Laps: {lap_count}" if slider_vars["MAX_LAPS"].get() > 0 else
f"Laps: {lap_count}/∞")
```

```
1161 |         root.after(200, update_status)
1162 |
1163 |     _prev_label_pos = {"front": None, "left": None, "right": None, "angle": None}
1164 |     def update_plot():
1165 |         #global _prev_label_pos
1166 |
1167 |         if time_data:
1168 |             # Update lines
1169 |             front_line.set_data(range(len(front_data)), front_data)
1170 |             left_line.set_data(range(len(left_data)), left_data)
1171 |             right_line.set_data(range(len(right_data)), [-v for v in right_data])
1172 |             angle_line.set_data(range(len(angle_data)), angle_data)
1173 |
1174 |             # Set axis limits
1175 |             ax_front.set_xlim(0, MAX_POINTS)
1176 |             ax_side.set_xlim(0, MAX_POINTS)
1177 |             ax_angle.set_xlim(0, MAX_POINTS)
1178 |
1179 |             # Remove previous text annotations
1180 |             for ax in [ax_front, ax_side, ax_angle]:
1181 |                 for t in ax.texts:
1182 |                     t.remove()
1183 |
1184 |             # Helper for smoothing
1185 |             def smooth_move(prev, target, alpha=0.3):
1186 |                 if prev is None:
1187 |                     return target
1188 |                 return prev + alpha * (target - prev)
1189 |
1190 |             # ----------------------------
1191 |             # FRONT SENSOR
1192 |             # ----------------------------
1193 |             if len(front_data) > 0:
1194 |                 x = len(front_data) - 1
1195 |                 y_target = front_data[-1]
1196 |                 y_prev = _prev_label_pos["front"]
1197 |                 y_smooth = smooth_move(y_prev, y_target)
1198 |                 _prev_label_pos["front"] = y_smooth
1199 |                 ax_front.text(x, y_smooth, f"{y_target:.1f} cm", color="blue",
1200 |                               fontsize=9, fontweight="bold", va="bottom", ha="left")
1201 |
1202 |             # ----------------------------
1203 |             # LEFT SENSOR
1204 |             # ----------------------------
1205 |             if len(left_data) > 0:
1206 |                 x = len(left_data) - 1
1207 |                 y_target = left_data[-1]
1208 |                 y_prev = _prev_label_pos["left"]
1209 |                 y_smooth = smooth_move(y_prev, y_target)
1210 |                 _prev_label_pos["left"] = y_smooth
1211 |                 ax_side.text(x, y_smooth, f"L: {y_target:.1f} cm", color="green",
1212 |                              fontsize=9, fontweight="bold", va="bottom", ha="left")
1213 |
1214 |             # ----------------------------
1215 |             # RIGHT SENSOR
1216 |             # ----------------------------
1217 |             if len(right_data) > 0:
1218 |                 x = len(right_data) - 1
1219 |                 y_target = -right_data[-1]
1220 |                 y_prev = _prev_label_pos["right"]
1221 |                 y_smooth = smooth_move(y_prev, y_target)
1222 |                 _prev_label_pos["right"] = y_smooth
1223 |                 ax_side.text(x, y_smooth, f"R: {right_data[-1]:.1f} cm", color="orange",
1224 |                              fontsize=9, fontweight="bold", va="bottom", ha="left")
1225 |
1226 |             # ----------------------------
1227 |             # YAW ANGLE
1228 |             # ----------------------------
1229 |             if len(angle_data) > 0:
1230 |                 x = len(angle_data) - 1
1231 |                 y_target = angle_data[-1]
1232 |                 y_prev = _prev_label_pos["angle"]
1233 |                 y_smooth = smooth_move(y_prev, y_target)
1234 |                 _prev_label_pos["angle"] = y_smooth
1235 |                 ax_angle.text(x, y_smooth, f"{y_target:.1f}°", color="purple",
1236 |                               fontsize=9, fontweight="bold", va="bottom", ha="left")
1237 |
```

```
            # ----------------------------
            # Dynamic Y-axis scaling for yaw
            # ----------------------------
            if angle_data:
                min_angle = min(angle_data)
                max_angle = max(angle_data)
                if max_angle - min_angle > 180:
                    ax_angle.set_ylim(-180, 180)
                else:
                    ax_angle.set_ylim(min_angle - 10, max_angle + 10)

            # Draw updated plots
            canvas.draw()

        root.after(100, update_plot)

    def on_closing():
        stop_loop()
        # Stop all ToF sensors cleanly
        for sensor in [vl53_left, vl53_right, vl53_front, vl53_back]:
            if sensor is not None:
                try:
                    sensor.stop_continuous()
                except Exception as e:
                    print(f"Warning: could not stop sensor {sensor}: {e}")

        GPIO.cleanup()
        root.destroy()

    root.protocol("WM_DELETE_WINDOW", on_closing)
    update_status()
    update_plot()
    root.mainloop()

# -------------------------
# Start / Stop
# -------------------------

def start_readings():
    #global readings_flag, status_text
    #readings_flag = True
    global status_text
    readings_event.set()
    status_text = "Sensor readings started"
    btn_readings.config(state="disabled")
    btn_start.config(state="normal")

    # Start background sensor thread once
    if not hasattr(start_readings, "sensor_thread_started"):
        threading.Thread(target=sensor_reader, daemon=True).start()
        #threading.Thread(target=robot_loop, daemon=True).start()
        #start_readings.thread_started = True
        start_readings.sensor_thread_started = True

    if not hasattr(start_readings, "loop_thread_started"):
        threading.Thread(target=robot_loop, daemon=True).start()
        start_readings.loop_thread_started = True


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

    #loop_flag = True
    loop_event.set()
    status_text = "🚗 Loop Started"
    btn_start.config(state="disabled")
    btn_stop.config(state="normal")

    #if not hasattr(start_loop, "loop_thread_started"):
    #  threading.Thread(target=robot_loop, daemon=True).start()
    #  start_loop.loop_thread_started = True
```

```
1315 |
1316 | def stop_loop():
1317 |     """Stop the robot loop and motor."""
1318 |     #global loop_flag, status_text, turn_count, lap_count, locked_turn_direction
1319 |     #loop_flag = False
1320 |     global status_text, turn_count, lap_count, locked_turn_direction, stop_reason
1321 |     loop_event.clear()
1322 |     stop_reason = "USER"
1323 |     sensor_tick.set()  # wake any waits immediately
1324 |     robot.stop_motor()
1325 |     status_text = "Stopped"
1326 |     btn_start.config(state="normal")
1327 |     btn_stop.config(state="disabled")
1328 |
1329 |     # Reset counters
1330 |     turn_count = 0
1331 |     lap_count = 0
1332 |     lbl_turns.config(text=f"Turns: {turn_count}")
1333 |     lbl_laps.config(text=f"Laps: {lap_count}")
1334 |
1335 |     # Reset direction lock so a new session can re-choose
1336 |     locked_turn_direction = None
1337 |
1338 | # -------------------------
1339 | # Export to CSV
1340 | # -------------------------
1341 | def export_data_csv():
1342 |     filename = f"viva_data_{datetime.now().strftime('%Y%m%d_%H%M%S')}.csv"
1343 |     slider_values = {name: var.get() for name, var in slider_vars.items()}
1344 |
1345 |     with open(filename, mode='w', newline='') as file:
1346 |         writer = csv.writer(file)
1347 |         header = [
1348 |             "Time (s)", "Front (cm)", "Left (cm)", "Right (cm)", "Yaw (deg)",
1349 |             "State", "Turns", "Laps", "Servo Angle", "Speed", "Turning", "Direction"
1350 |         ]
1351 |         header += [name for name in slider_values.keys()]
1352 |         writer.writerow(header)
1353 |
1354 |         for i in range(len(time_data)):
1355 |             row = [
1356 |                 round(time_data[i], 2),
1357 |                 round(front_data[i], 2),
1358 |                 round(left_data[i], 2),
1359 |                 round(right_data[i], 2),
1360 |                 round(angle_data[i], 2),
1361 |                 state_data[i],
1362 |                 turn_count,
1363 |                 lap_count,
1364 |                 globals().get("servo_angle", 0),  # latest steering
1365 |                 globals().get("SPEED_CRUISE", 0),
1366 |                 "YES" if "Turning" in state_data[i] else "NO",
1367 |                 globals().get("locked_turn_direction", "")
1368 |             ]
1369 |             row += [slider_values[name] for name in slider_values.keys()]
1370 |             writer.writerow(row)
1371 |
1372 |     print(f"✅ Data exported to {filename}")
1373 |
1374 | # -------------------------
1375 | # Save/Load Slider Config
1376 | # -------------------------
1377 | def save_sliders_json():
1378 |     default_filename = f"sliders_config_{datetime.now().strftime('%Y%m%d')}.json"
1379 |     file_path = fd.asksaveasfilename(initialdir=BASE_DIR,
1380 |                                      initialfile=default_filename,
1381 |                                      defaultextension=".json",
1382 |                                      filetypes=[("JSON files", "*.json")],
1383 |                                      title="Save Slider Configuration")
1384 |     if file_path:
1385 |         data = {name: var.get() for name, var in slider_vars.items()}
1386 |         with open(file_path, "w") as f:
1387 |             json.dump(data, f, indent=4)
1388 |         print(f"Slider values saved to {file_path}")
1389 |
1390 | def load_sliders_json():
1391 |     file_path = fd.askopenfilename(initialdir=BASE_DIR,
```

```
                                    defaultextension=".json",
                                    filetypes=[("JSON files", "*.json")],
                                    title="Load Slider Configuration")
    if file_path:
        try:
            with open(file_path, "r") as f:
                data = json.load(f)
            for name, value in data.items():
                if name in slider_vars:
                    slider_vars[name].set(value)
                    globals()[name] = value
            print(f"Sliders restored from {file_path}")
        except Exception as e:
            print(f"Failed to load sliders: {e}")

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

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

    if USE_GUI:
        launch_gui()
    else:
        # Start the sensor reading thread
        GPIO.output(RED_LED, GPIO.HIGH)
        GPIO.output(GREEN_LED, GPIO.LOW)
        sensor_thread = threading.Thread(target=sensor_reader, daemon=True)
        sensor_thread.start()
        print("Headless mode: waiting for START button...")
        GPIO.output(RED_LED, GPIO.LOW)     # turn off red when ready
        GPIO.output(GREEN_LED, GPIO.HIGH)  # green = ready to press the button
        try:
            # Wait for button press (GPIO input goes LOW when pressed)
            while GPIO.input(START_BTN) == 1:
                time.sleep(0.05)
            print("✅ START button pressed! Beginning autonomous loop...")
            #readings_flag = True
            readings_event.set()
            time.sleep(2)  # wait 2 seconds before starting loop
            #loop_flag = True
            loop_event.set()
            print("Starting main robot loop...")
            # Start robot loop in this thread (blocking)
            robot_loop()
        except KeyboardInterrupt:
            print("
❌ Keyboard interrupt received. Stopping robot loop.")
            #loop_flag = False
            loop_event.clear()
            sensor_tick.set()
            GPIO.cleanup()
```