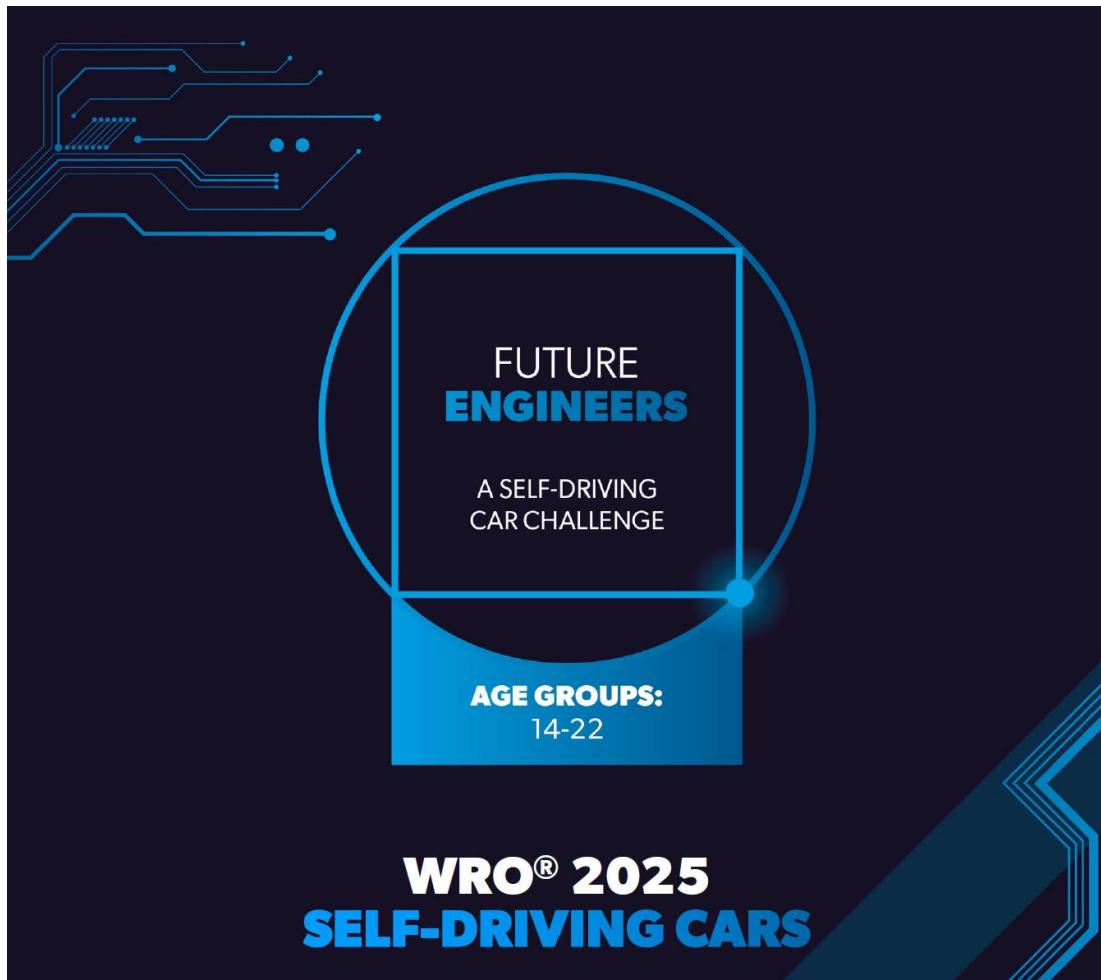


Viva La Vida – WRO FE Open Challenge



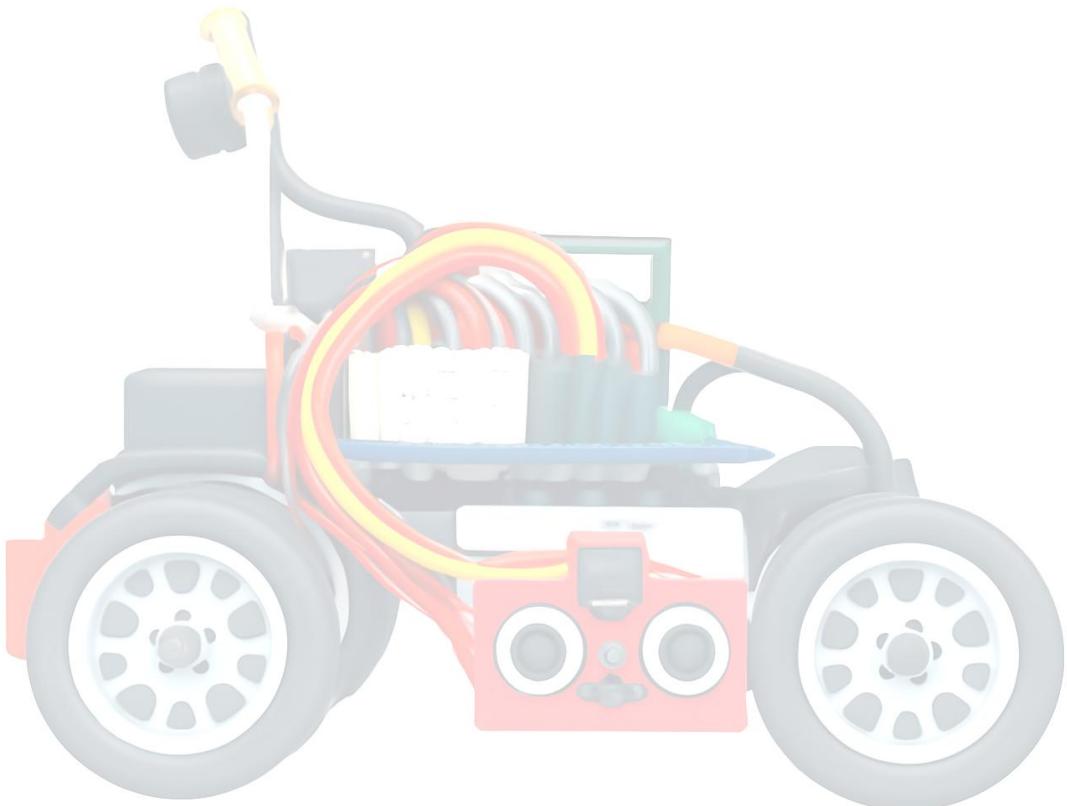
VIVA LA VIDA

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

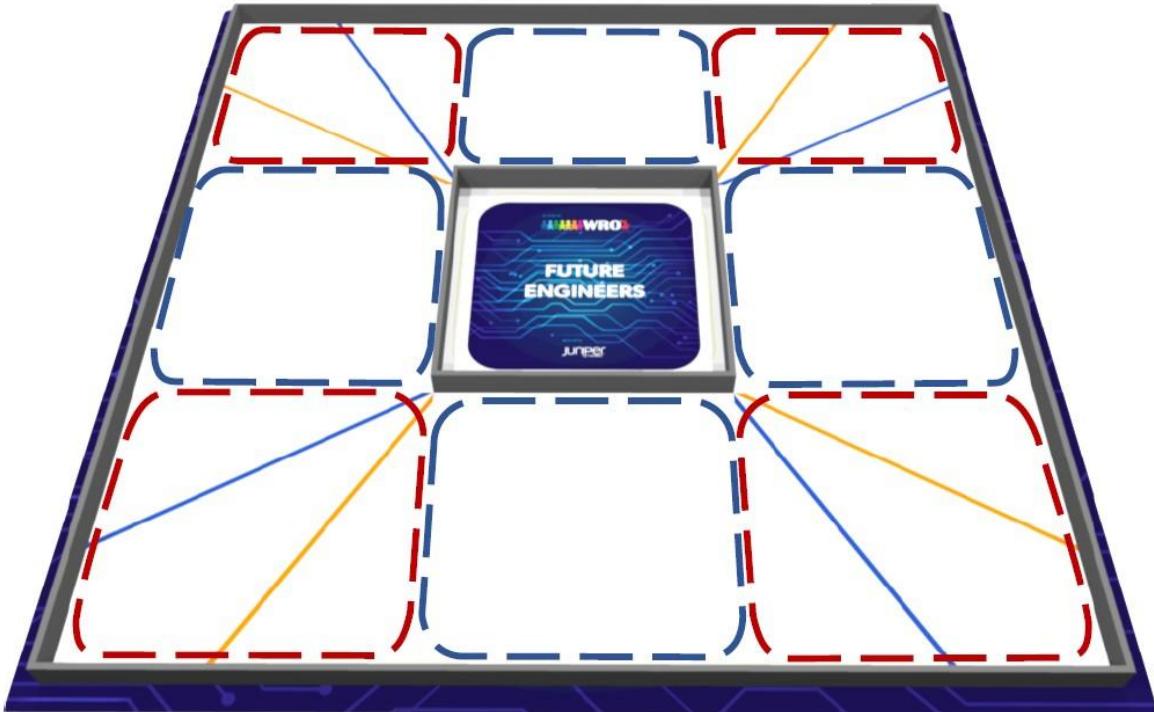
1	Overview & Scope	1
2	The robot - Hardware & Wiring	2
3	System Architecture	3
4	Decision-Making Architecture	6
5	Finite State Machine (FSM)	8
6	Parameters & Profiles	9
7	Diagrams	10
8	Appendix A - Variables	12
9	Appendix B - Source Code Analysis	16
10	Appendix C - Source code	21





1 Overview & Scope

Mission: autonomously complete three laps inside a 3x3 m field containing a randomized 1x1 m inner island that creates corridors. Robot must maintain direction, avoid the walls, and stop safely. Competition rules: single power switch + Start button; no wireless during the run.



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



2 The robot - Hardware & Wiring

2.1 Bill of Materials

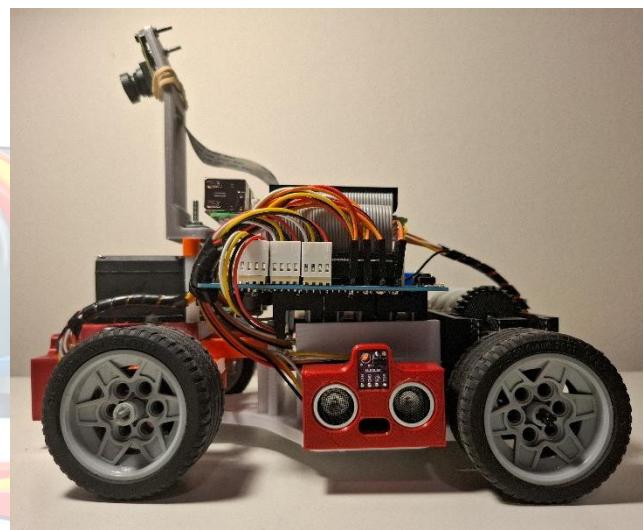
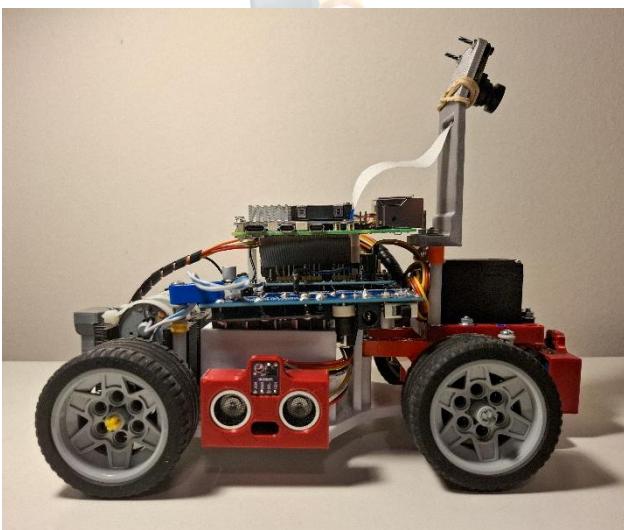
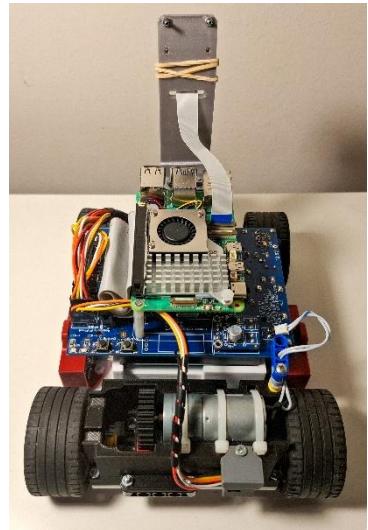
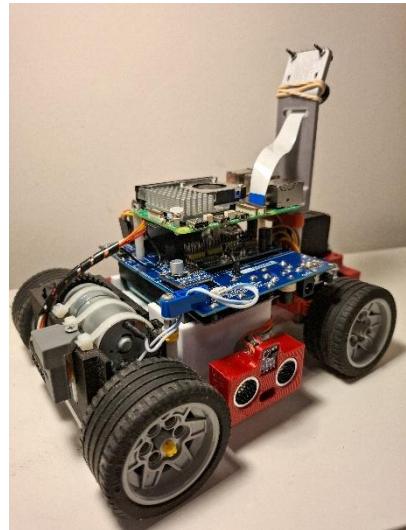
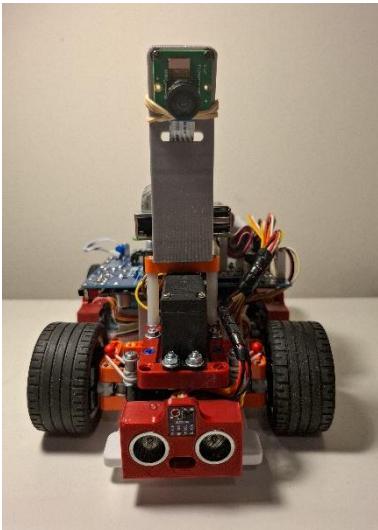
Component	Role	Interface/Notes
Raspberry Pi	Main controller	GPIO/I ² C
PCA9685	Servo + motor PWM	50 Hz PWM; I ² C
MPU6050	Gyro-Z for yaw	I ² C
Ultrasonic x3	Front/Left/Right	Trig/Echo via gpiozero
VL53L0X (opt.)	ToF sensors	I ² C + XSHUT addressing
Steering Servo	Single	1000–2000 µs pulse
Motor + Driver/ESC	Single	PWM ch1/ch2 (FWD/REV)
Start Button	Headless start	GPIO20
Status LEDs	Green / Red	GPIO19 / GPIO13
Battery/Power	5–6 V servo; 5 V logic	Common ground

2.2 Pin Map

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



2.3 Robot photos



3 System Architecture

Code structure:

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



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

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

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

Outline:

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

3.1 Code Structure

3.1.1 Imports

3.1.1.1 External Libraries

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

3.1.1.2 Internal Modules

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

3.1.2 Variables

3.1.2.1 Global Constants

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

3.1.2.2 Robot-Specific Variables

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



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

3.1.3 H/W Setup

3.1.3.1 Hardware Configuration

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

3.1.3.2 Pin Assignments

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

3.1.4 Robot Classes

3.1.4.1 Sensor Handling

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

3.1.4.2 Motor Control

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

3.1.4.3 State Management

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

3.1.5 Robot Loop

3.1.5.1 Main Control Loop

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

3.1.5.2 Decision Making

Turn preconditions: $\text{front} < \text{FRONT_TURN_trigger}$ and $(t - \text{last_turn}) \geq \text{TURN_LOCKOUT}$. Direction requires XOR (`open_left`, `open_right` – only one Input is True), where `open` is None/999 or $> \text{TURN_DECISION_THRESHOLD}$. Optional `LOCK_TURN_DIRECTION` fixes the first chosen side. Turn termination uses $|\Delta\text{yaw} - \text{TARGET_TURN_ANGLE}| \leq \text{tolerance}$, else timeout or max-angle as fail-safes. `STOP_THRESHOLD` forces immediate STOPPED with timed auto-retry.



3.1.5.3 Movement Logic

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

3.1.6 GUI

3.1.6.1 Tkinter Setup

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

3.1.6.2 Sliders & Data Visualization

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

3.1.7 Main

3.1.7.1 Initialization

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

3.1.7.2 Robot Execution

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

4 Decision-Making Architecture

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

4.1 Perception Pipeline

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

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

4.2 Corridor Model & Narrow-Mode Scaling

We treat the track as four straight corridors and four corners. Narrow-mode activates when L+R < NARROW_SUM_THRESHOLD and deactivates above threshold+NARROW_HYSTESIS. 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 \geq TURN_LOCKOUT seconds ago. This prevents premature or back-to-back turns on zig-zag geometries.

4.4 Direction Selection (Symmetry-Aware)

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

4.5 Turn Commit & Steering Command

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

4.6 Termination Strategy (Multi-guard)

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

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

4.7 Straight-Line Corrections

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

4.8 Obstacle Stop & Auto-Retry

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

4.9 Determinism, Hysteresis, and Lockouts

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



5 Finite State Machine (FSM)

5.1 States

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

5.2 State actions

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



6 Parameters & Profiles

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

6.1 Preset Profiles

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

6.2 Operations Runbook

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

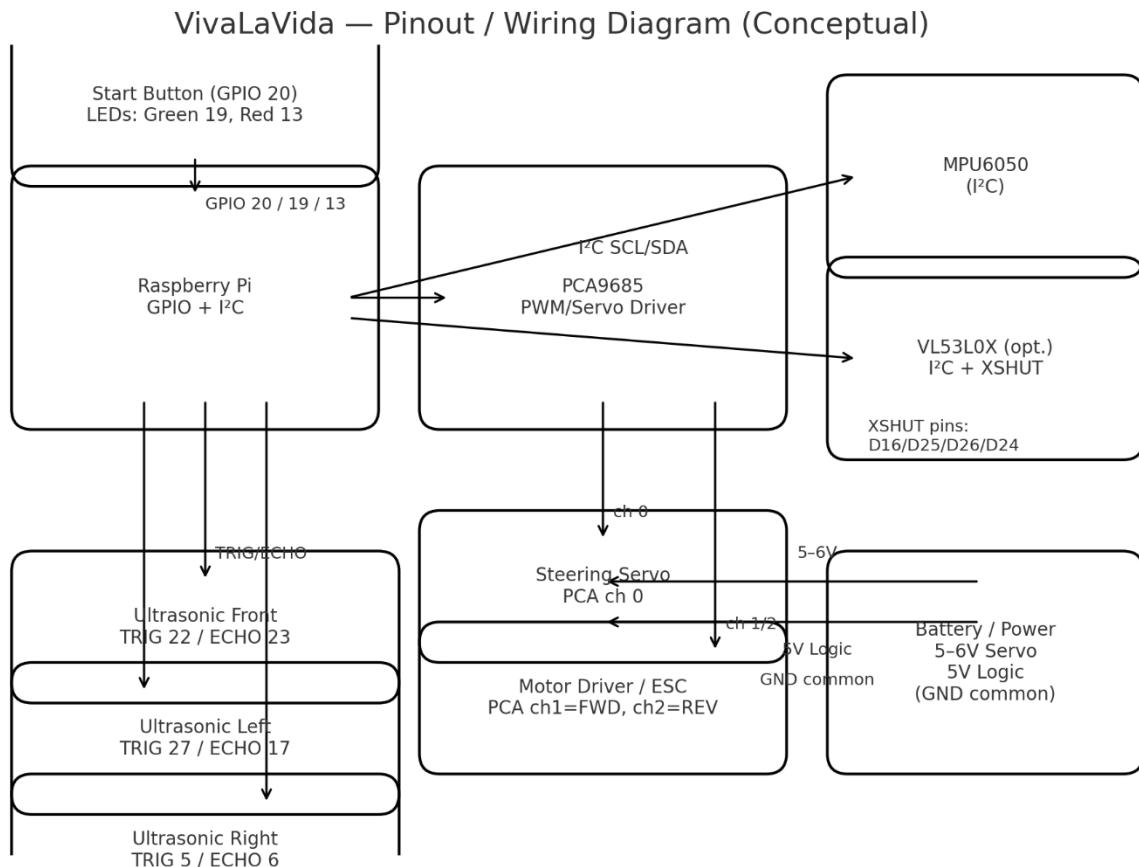
6.3 Boot & Headless Start

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

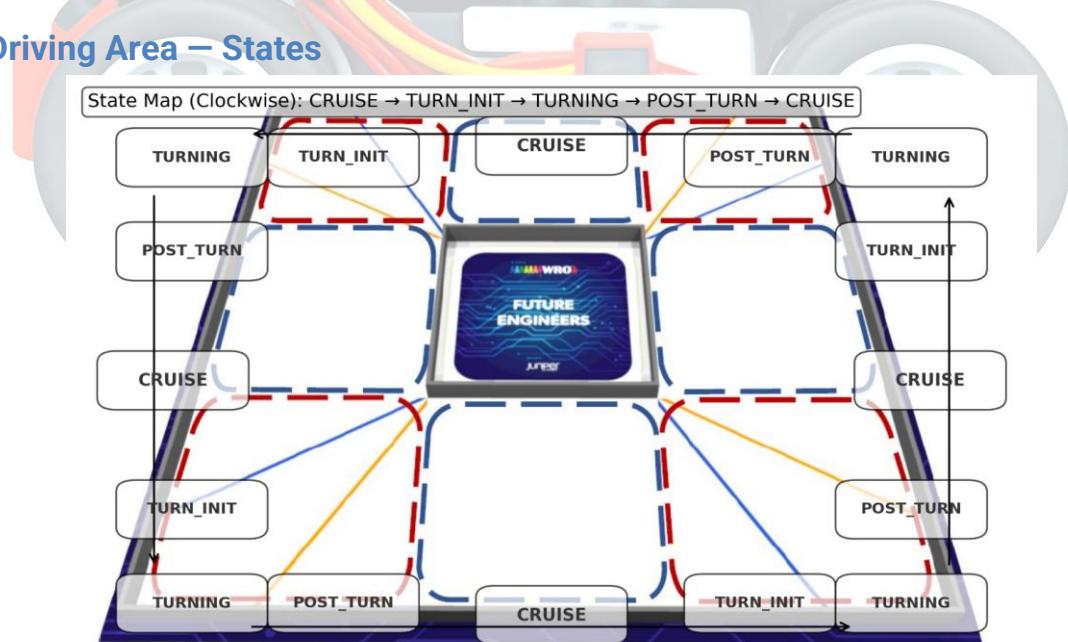


7 Diagrams

7.1 Pinout / Wiring (Conceptual)

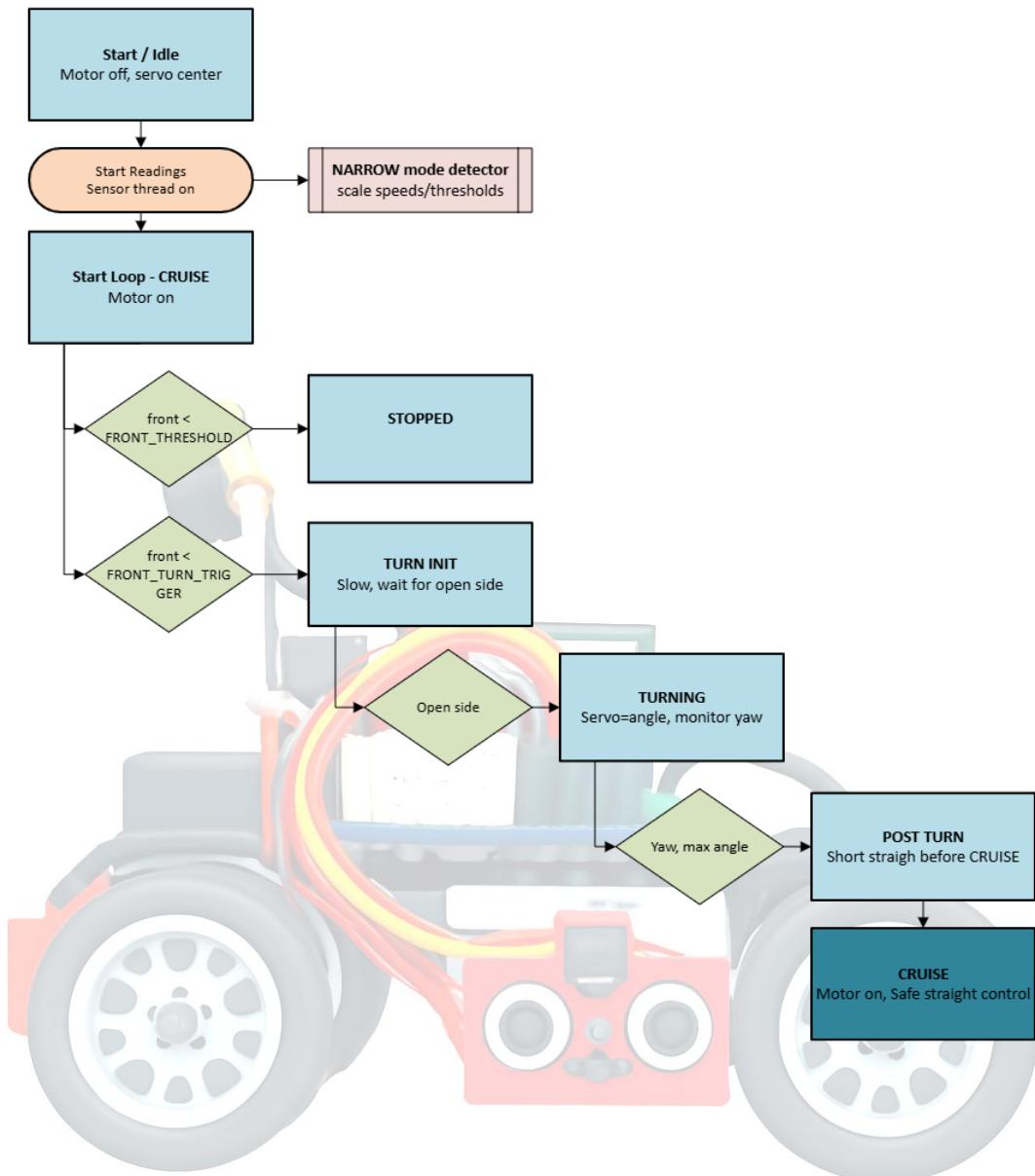


7.2 Driving Area – States





7.3 Decision Flowchart (FSM + Safety + Narrow Mode)





8 Appendix A - Variables

Notes:

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

8.1 Variables

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



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



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



8.2 JSON Override Example

1st_mission_variables.json

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



9 Appendix B - Source Code Analysis

9.1 Imports & global configuration

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

9.2 Hardware Setup (GPIO/I²C, Pin Map)

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



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

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

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

9.4 Data Model & Threading Primitives

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

9.5 RobotController (Low-Level Actuation & Signal Processing)

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



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

9.6 Real-Time Threads

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

9.7 Yaw Integration & Units

- **IMU Input:**
 - `gyro_z` in rad/s, bias-corrected, low-passed (ALPHA), integrated to degrees:
 - $yaw += (\text{gyro}_z_{\text{filtered}} * \text{RAD2DEG}) * dt$, where $\text{RAD2DEG} = 180/\pi$, dt from `monotonic_ns()`

9.8 Finite State Machine (FSM)

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

9.9 Environment Adaptation (Narrow-Mode)

- **Detector:**



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

9.10 GUI Subsystem (For Debugging)

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

9.11 Start/Stop & Lifecycle

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

9.12 Synchronization & Timing

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



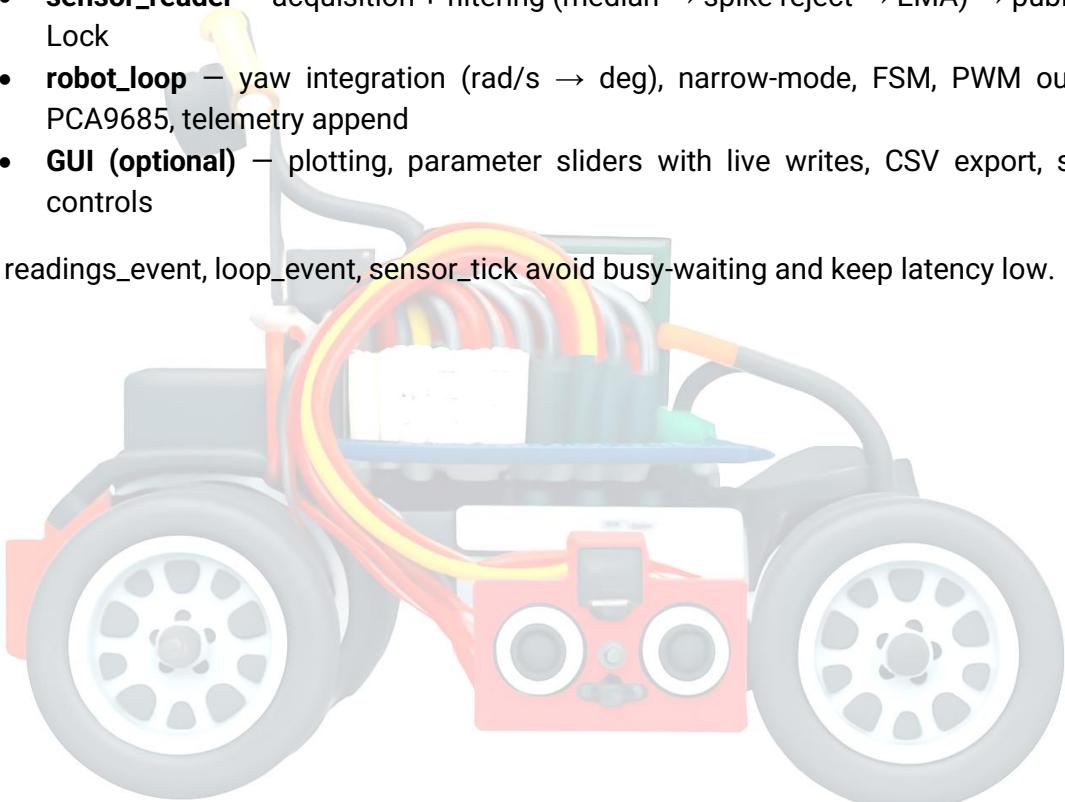
9.13 Safety/Fallbacks

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

9.14 Thread Summary

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

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





10 Appendix C - Source code

```
1: #-----
2: # 1st Mission WRO 2025 FE - VivaLaVida
3: # Final Version
4: #-----
5:
6: # =====
7: # IMPORTS
8: # =====
9:
10: import threading
11: import time
12: from collections import deque
13: import busio
14: from adafruit_pca9685 import PCA9685
15: import adafruit_mpu6050
16: import csv
17: from datetime import datetime
18: import json
19: import os
20: import board
21: import digitalio
22: import adafruit_vl53l0x
23: from enum import Enum, auto
24: import numpy as np
25: from gpiozero import Device
26: from gpiozero.pins.lgpio import LGPIOFactory # Uses /dev/gpiochip*
27: Device.pin_factory = LGPIOFactory()
28: from gpiozero import Button, LED
29: from gpiozero import DistanceSensor
30: from threading import Event
31:
32: BASE_DIR = os.path.dirname(os.path.abspath(__file__)) # Get directory of the running script
33: CONFIG_FILE = os.path.join(BASE_DIR, "1st_mission_variables.json")
34:
35: # =====
36: # CONFIGURATION VARIABLES
37: # =====
38:
39: # ----- Initialization -----
40: USE_GUI = 0 # 1 = Debugging mode (run with GUI), 0 = COMPETITION MODE (run headless, no GUI)
41: DEBUG = 0
42: USE_TOF_SIDES = 0
43: USE_TOF_FRONT = 0
44: USE_TOF_CORNERS = 0
45:
46: NARROW_SUM_THRESHOLD = 60 # cm; left+right distance threshold to decide if in between narrow walls
47: NARROW_HYSTESIS = 10 # cm; prevents rapid toggling
48: NARROW_FACTOR_SPEED = 0.6 # multiply state speeds when in narrow corridors
49: NARROW_FACTOR_DIST = 0.5 # multiply distance-based thresholds/corrections when in narrow corridors
50:
51: # ----- Speeds -----
52: SPEED_IDLE = 0
53: SPEED_STOPPED = 0
54: SPEED_CRUISE = 24 # Motor speed for normal straight driving (0-100%)
55: SPEED_TURN_INIT = 16 # Motor speed while waiting for open side to turn
56: SPEED_TURN = 24 # Motor speed while turning
57: SPEED_POST_TURN = 24 # Motor speed following a turn
58:
59: # ----- Driving -----
60: SOFT_MARGIN = 26 # Distance from wall where small steering corrections start (cm)
61: MAX_CORRECTION = 7 # Maximum servo correction applied for wall-following (degrees)
62: CORR_EPS = 1.5 # cm: treat the side as "steady" if within ±1.5 cm of trigger value
63: CORRECTION_MULTIPLIER = 1.4 # Proportional gain (servo degrees per cm of error 0 default: 2). Higher = snappier; lower = smoother&slower.
64:
65: STOP_THRESHOLD = 20 # Front distance (cm) at which robot stops immediately
66: OBSTACLE_WAIT_TIME = 5.0 # seconds to wait before retrying after a front-stop
```



```
67:  
68: # ----- Turn management -----  
69: FRONT_TURN_TRIGGER = 90          # Front distance (cm) at which a turn is triggered  
70: TURN_DECISION_THRESHOLD = 80      # Minimum side distance (cm) to allow turn in that direction  
71: TURN_ANGLE_LEFT = 61             # Servo angle for left turn  
72: TURN_ANGLE_RIGHT = 119           # Servo angle for right turn  
73: TURN_TIMEOUT = 2.5              # Maximum time allowed for a turn (seconds)  
74: TURN_LOCKOUT = 1.5              # Minimum interval between consecutive turns (seconds)  
75: POST_TURN_DURATION = 0.5        # Time to drive straight after a turn (seconds)  
76: LOCK_TURN_DIRECTION = 1         # 1 = enable turn lock direction after 1st turn, 0 = disable  
77: TARGET_TURN_ANGLE = 85          # Degrees to turn per corner  
78: TURN_ANGLE_TOLERANCE = 6         # Acceptable overshoot (degrees)  
79:  
80: # ----- Turn angle constraints -----  
81: MIN_TURN_ANGLE = 75            # Minimum yaw change (degrees) before turn can stop  
82: MAX_TURN_ANGLE = 105           # Maximum yaw change (degrees) to force stop turn  
83:  
84: # ----- Laps -----  
85: MAX_LAPS = 3                  # Maximum number of laps before stopping (0 = unlimited)  
86: POST_LAP_DURATION = 0.85       # Time to drive forward after final lap before stopping  
(seconds)  
87:  
88: # ----- Sensor data filtering -----  
89: N_READINGS = 5                # Number of readings stored for median filtering  
90: US_QUEUE_LEN = 1              # Queue_len for Ultrasonic gpiozero.DistanceSensor  
91: FILTER_ALPHA = 1.0            # Ultrasonic 0.1 = smoother, 0.5 = faster reaction, 1 to ignore  
filter  
92: FILTER_JUMP = 9999            # Ultrasonic maximum jump (cm) allowed between readings  
93: FILTER_ALPHA_TOF = 1.0         # ToF 0.1 = smoother, 0.5 = faster reaction, 1 to ignore filter  
94: FILTER_JUMP_TOF = 9999         # ToF maximum jump (cm) allowed between readings  
95:  
96: US_MAX_DISTANCE_FRONT = 2.0  # Ultrasonic front max read distance  
97: US_MAX_DISTANCE_SIDE = 1.2   # Ultrasonic side max read distance  
98:  
99: # ----- Loop timing -----  
100: LOOP_DELAY = 0.005            # Delay between main loop iterations (seconds) 0.02  
101: SENSOR_DELAY = 0.01          # Delay between sensor reads  
102:  
103: #-----  
104:  
105: # ----- Plotting -----  
106: MAX_POINTS = 500             # Maximum number of points to store for plotting  
107:  
108: RAD2DEG = 180.0 / 3.141592653589793  
109:  
110: # --- I2C concurrency guard ---  
111: I2C_LOCK = threading.Lock()  
112:  
113: def dprint(*args, **kwargs): #Conditional print - only prints when DEBUG == 1.  
114: if DEBUG:  
115: print(*args, **kwargs)  
116:  
117: # --- Headless-safe placeholders for GUI symbols ---  
118: slider_vars = {}  
119: btn_readings = btn_start = btn_stop = None  
120: lbl_turns = lbl_laps = None  
121: root = canvas = None  
122:  
123: # =====  
124: # LOAD VARIABLES FROM JSON (if exists)  
125: # =====  
126:  
127: def load_variables_from_json(path=CONFIG_FILE): # Override defaults with values from JSON if  
the file exists.  
128: if not os.path.exists(path):  
129: print("⚙️ No external config found - using built-in defaults.")  
130: return  
131: try:  
132: with open(path, "r") as f:  
133: data = json.load(f)  
134: count = 0  
135: for k, v in data.items():  
136: if k in globals():  
137: globals()[k] = v  
138: count += 1  
139: print(f"✅ Loaded {count} variables from {os.path.basename(path)}")  
140: except Exception as e:
```



```
141: print(f"[ERROR] Config load failed: {e} - using built-in defaults.")
142:
143: load_variables_from_json() # Call it right after defining defaults so everything below sees the
overrides
144:
145: if os.environ.get("DISPLAY", "") == "": # If there is no X server, force headless regardless of
config
146: USE_GUI = 0
147:
148: def norm180(a: float) -> float: # Normalize angle to [-180, 180)
149: return (a + 180.0) % 360.0 - 180.0
150:
151: def snap90(a: float) -> float: # Nearest multiple of 90° (...,-180,-90,0,90,180,...)
152: return round(a / 90.0) * 90.0
153:
154: # --- Store immutable base values for scaling ---
155: BASE_SPEED_IDLE = SPEED_IDLE
156: BASE_SPEED_STOPPED = SPEED_STOPPED
157: BASE_SPEED_CRUISE = SPEED_CRUISE
158: BASE_SPEED_TURN_INIT = SPEED_TURN_INIT
159: BASE_SPEED_TURN = SPEED_TURN
160: BASE_SPEED_POST_TURN = SPEED_POST_TURN
161:
162: BASE_MAX_CORRECTION = MAX_CORRECTION
163: BASE_FRONT_TURN_TRIGGER = FRONT_TURN_TRIGGER
164: BASE_STOP_THRESHOLD = STOP_THRESHOLD
165:
166: # --- Global factors (automatically updated) ---
167: SPEED_ENV_FACTOR = 1.0
168: DIST_ENV_FACTOR = 1.0 # for distance-related thresholds
169:
170: # --- Helper functions to get "effective" (scaled) values ---
171: def eff_soft_margin(): #Return scaled soft margin distance (cm) for gentle steering corrections
172: return int(SOFT_MARGIN * DIST_ENV_FACTOR)
173:
174: def eff_max_correction(): #Return scaled maximum steering correction (degrees)
175: return max(1, int(MAX_CORRECTION * DIST_ENV_FACTOR))
176:
177: def eff_front_turn_trigger(): #Return scaled front distance threshold (cm) for initiating a
turn
178: return int(FRONT_TURN_TRIGGER * DIST_ENV_FACTOR)
179:
180: def eff_stop_threshold(): #Return scaled front distance threshold (cm) for immediate stop
181: return int(STOP_THRESHOLD * DIST_ENV_FACTOR)
182:
183: # --- Scaled state speeds ---
184: def state_speed_value(state_name: str) -> int:
185: base = {
186: "IDLE": SPEED_IDLE,
187: "STOPPED": SPEED_STOPPED,
188: "CRUISE": SPEED_CRUISE,
189: "TURN_INIT": SPEED_TURN_INIT,
190: "TURNING": SPEED_TURN,
191: "POST_TURN": SPEED_POST_TURN,
192: }.get(state_name, SPEED_CRUISE)
193: return int(base * SPEED_ENV_FACTOR)
194:
195: def headless_toggle_pause(): #Toggle pause/resume of the driving loop in headless mode
196: global paused_by_button, status_text
197: if loop_event.is_set(): #Pause
198: loop_event.clear()
199: paused_by_button = True
200: robot.stop_motor()
201: robot.set_servo(SERVO_CENTER)
202: RED_LED.off() # paused
203: GREEN_LED.on()
204: status_text = "Paused (button)"
205: else: # RESUME
206: paused_by_button = False
207: loop_event.set()
208: GREEN_LED.off() # running
209: RED_LED.on()
210: status_text = "🚗 Resumed"
211:
212: # =====
213: # HARDWARE SETUP
214: # =====
```



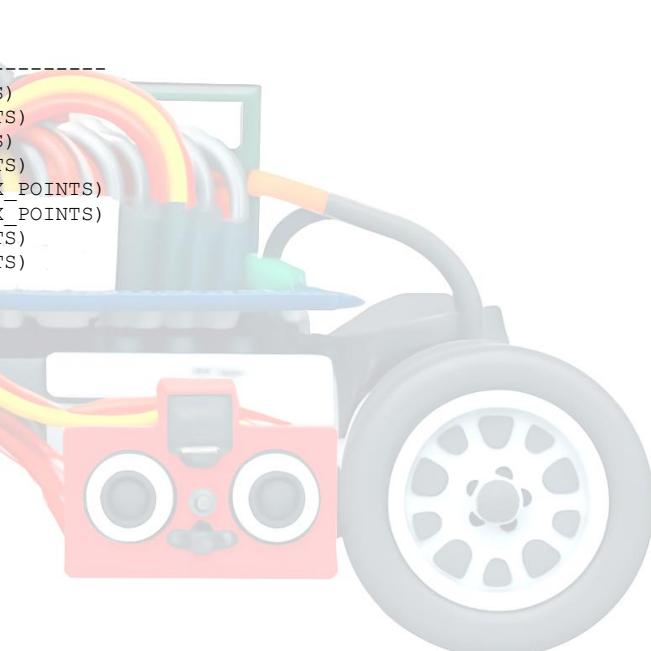
```
215:  
216: i2c = busio.I2C(board.SCL, board.SDA)  
217:  
218: # ----- Button & LEDs (gpiozero-based) -----  
219: START_BTN = Button(20, pull_up=True, bounce_time=0.03)  
220: GREEN_LED = LED(19)  
221: RED_LED = LED(13)  
222:  
223: # ----- Ultrasonic sensor pins -----  
224: TRIG_FRONT, ECHO_FRONT = 22, 23 # GPIO pins for front sensor  
225: TRIG_LEFT, ECHO_LEFT = 27, 17 # GPIO pins for left sensor  
226: TRIG_RIGHT, ECHO_RIGHT = 5, 6 # GPIO pins for right sensor  
227:  
228: # ----- Motor -----  
229: MOTOR_FWD = 1 # PCA9685 channel for forward motor control  
230: MOTOR_REV = 2 # PCA9685 channel for reverse motor control  
231:  
232: # ----- Servo -----  
233: SERVO_CHANNEL = 0 # PCA9685 channel controlling steering servo  
234: SERVO_CENTER = 90 # Neutral servo angle (degrees)  
235: SERVO_MIN_ANGLE = 50 # Minimum physical servo angle (degrees)  
236: SERVO_MAX_ANGLE = 130 # Maximum physical servo angle (degrees)  
237: SERVO_PULSE_MIN = 1000 # Minimum PWM pulse width (microseconds)  
238: SERVO_PULSE_MAX = 2000 # Maximum PWM pulse width (microseconds)  
239: SERVO_PERIOD = 20000 # Servo PWM period (microseconds)  
240:  
241: # ======  
242: # SENSOR INITIALIZATION (ToF + Ultrasonic)  
243: # ======  
244:  
245: vl53_left = vl53_right = vl53_front = vl53_back = None  
246: vl53_front_left = vl53_front_right = None  
247: us_left = us_right = us_front = None  
248:  
249: try:  
250: # ----- ToF sensors -----  
251: if USE_TOF_SIDES or USE_TOF_FRONT or USE_TOF_CORNERS:  
252: print("Initializing VL53L0X ToF sensors...")  
253: # XSHUT pins setup (for powering sensors individually)  
254: xshut_left = digitalio.DigitalInOut(board.D16)  
255: xshut_right = digitalio.DigitalInOut(board.D25)  
256: xshut_front = digitalio.DigitalInOut(board.D26)  
257: xshut_back = digitalio.DigitalInOut(board.D8)  
258: xshut_front_left = digitalio.DigitalInOut(board.D7)  
259: xshut_front_right = digitalio.DigitalInOut(board.D24)  
260: for xshut in [xshut_left, xshut_right, xshut_front, xshut_back,  
261: xshut_front_left, xshut_front_right]:  
262: xshut.direction = digitalio.Direction.OUTPUT  
263: xshut.value = False # power down  
264: time.sleep(0.1)  
265:  
266: # Initialize left ToF if needed  
267: if USE_TOF_SIDES:  
268: xshut_left.value = True  
269: time.sleep(0.05)  
270: vl53_left = adafruit_vl53l0x.VL53L0X(i2c)  
271: vl53_left.set_address(0x30)  
272: vl53_left.measurement_timing_budget = 20000  
273: vl53_left.start_continuous()  
274: print("✅ Left ToF sensor set to address 0x30")  
275:  
276: # Initialize right ToF if needed  
277: if USE_TOF_SIDES:  
278: xshut_right.value = True  
279: time.sleep(0.05)  
280: vl53_right = adafruit_vl53l0x.VL53L0X(i2c)  
281: vl53_right.set_address(0x31)  
282: vl53_right.measurement_timing_budget = 20000  
283: vl53_right.start_continuous()  
284: print("✅ Right ToF sensor set to address 0x31")  
285:  
286: # Initialize front ToF if needed  
287: if USE_TOF_FRONT:  
288: xshut_front.value = True  
289: time.sleep(0.05)  
290: vl53_front = adafruit_vl53l0x.VL53L0X(i2c)  
291: vl53_front.set_address(0x32)
```



```
292: v153_front.measurement_timing_budget = 20000
293: v153_front.start_continuous()
294: print("✓ Front ToF sensor set to address 0x32")
295:
296: # Initialize corner/front-left ToF (optional)
297: if USE_TOF_CORNERS:
298:     xshut_front_left.value = True
299:     time.sleep(0.05)
300:     v153_front_left = adafruit_vl53l0x.VL53L0X(i2c)
301:     v153_front_left.set_address(0x34)
302:     v153_front_left.measurement_timing_budget = 20000
303:     v153_front_left.start_continuous()
304:     print("✓ Front-Left ToF sensor set to address 0x34")
305:
306: # Initialize corner/front-right ToF (optional)
307: if USE_TOF_CORNERS:
308:     xshut_front_right.value = True
309:     time.sleep(0.05)
310:     v153_front_right = adafruit_vl53l0x.VL53L0X(i2c)
311:     v153_front_right.set_address(0x35)
312:     v153_front_right.measurement_timing_budget = 20000
313:     v153_front_right.start_continuous()
314:     print("✓ Front-Right ToF sensor set to address 0x35")
315:
316: # Initialize back ToF (optional, if you want to use it)
317: xshut_back.value = True
318: time.sleep(0.05)
319: v153_back = adafruit_vl53l0x.VL53L0X(i2c)
320: v153_back.set_address(0x33)
321: v153_back.measurement_timing_budget = 20000
322: v153_back.start_continuous()
323: print("✓ Back ToF sensor set to address 0x33")
324:
325: except Exception as e:
326:     print(f"[ERROR] VL53L0X initialization failed: {e}")
327: # fallback to ultrasonic if ToF fails
328: USE_TOF_SIDES = 0
329: USE_TOF_FRONT = 0
330: USE_TOF_CORNERS = 0
331: v153_left = v153_right = v153_front = v153_back = None
332: v153_front_left = v153_front_right = None
333:
334: time.sleep(0.2)
335:
336: # ----- Ultrasonic sensors -----
337: # Initialize ultrasonic sensors only if that sensor is set to ultrasonic
338: if not USE_TOF_FRONT:
339:     us_front = DistanceSensor(echo=ECHO_FRONT, trigger=TRIG_FRONT,
max_distance=US_MAX_DISTANCE_FRONT, queue_len=US_QUEUE_LEN)
340: if not USE_TOF_SIDES:
341:     us_left = DistanceSensor(echo=ECHO_LEFT, trigger=TRIG_LEFT,
max_distance=US_MAX_DISTANCE_SIDE, queue_len=US_QUEUE_LEN)
342:     us_right = DistanceSensor(echo=ECHO_RIGHT, trigger=TRIG_RIGHT,
max_distance=US_MAX_DISTANCE_SIDE, queue_len=US_QUEUE_LEN)
343:
344: dprint("Sensors initialized:")
345: dprint(f"  Front: {'ToF' if USE_TOF_FRONT else 'Ultrasonic'}")
346: dprint(f"  Left : {'ToF' if USE_TOF_SIDES else 'Ultrasonic'}")
347: dprint(f"  Right: {'ToF' if USE_TOF_SIDES else 'Ultrasonic'}")
348: if USE_TOF_CORNERS:
349:     dprint("  Turn corners: ToF (front_left & front_right)")
350:
351: pca = PCA9685(i2c)
352: pca.frequency = 50
353: mpu = adafruit_mpu6050.MPU6050(i2c)
354:
355: # ----- Calibrate gyro bias at startup -----
356: dprint("Calibrating gyro...")
357: N = 500
358: bias = 0
359: for _ in range(N):
360:     with I2C_LOCK:
361:         bias += mpu.gyro[2]
362:     time.sleep(0.005)
363: bias /= N
364: dprint(f"Gyro bias: {bias}")
```



```
365:  
366: # ======  
367: # CONTROL FLAGS & STORAGE  
368: # ======  
369:  
370: STATE_SPEED = {  
371: "IDLE": SPEED_IDLE,  
372: "STOPPED": SPEED_STOPPED,  
373: "CRUISE": SPEED_CRUISE,  
374: "TURN_INIT": SPEED_TURN_INIT,  
375: "TURNING": SPEED_TURN,  
376: "POST_TURN": SPEED_POST_TURN  
377: }  
378:  
379: # ----- CONTROL FLAGS -----  
380: readings_event = Event()  
381: loop_event = Event()  
382: sensor_tick = Event() # used to wake waits immediately  
383: shutdown_event = Event()  
384: status_text = "Idle"  
385: turn_count = 0  
386: lap_count = 0  
387: stop_reason = None # Reason-aware stopping (USER / OBSTACLE / LAPS)  
388: obstacle_wait_deadline = 0.0  
389: # --- Headless START button toggle / debounce ---  
390: BTN_DEBOUNCE_S = 0.30 # seconds  
391: _btn_prev = 1 # pull-up idle state (button not pressed)  
392: _btn_last_ts = 0.0  
393: paused_by_button = False  
394:  
395: # ----- SENSOR DATA STORAGE -----  
396: time_data = deque(maxlen=MAX_POINTS)  
397: front_data = deque(maxlen=MAX_POINTS)  
398: left_data = deque(maxlen=MAX_POINTS)  
399: right_data = deque(maxlen=MAX_POINTS)  
400: front_left_data = deque(maxlen=MAX_POINTS)  
401: front_right_data = deque(maxlen=MAX_POINTS)  
402: angle_data = deque(maxlen=MAX_POINTS)  
403: state_data = deque(maxlen=MAX_POINTS)  
404:  
405: # ======  
406: # THREADED SENSOR READING  
407: # ======  
408: sensor_data = {  
409: "front": None,  
410: "left": None,  
411: "right": None,  
412: "front_left": None,  
413: "front_right": None,  
414: }  
415: sensor_lock = threading.Lock()  
416:  
417: # ======  
418: # ROBOT CONTROLLER  
419: # ======  
420:  
421: class RobotController:  
422:     def __init__(self, pca):  
423:         self.pca = pca  
424:         self.front_history = deque(maxlen=N_READINGS)  
425:         self.left_history = deque(maxlen=N_READINGS)  
426:         self.right_history = deque(maxlen=N_READINGS)  
427:         self.front_left_history = deque(maxlen=N_READINGS)  
428:         self.front_right_history = deque(maxlen=N_READINGS)  
429:         self.smooth_left = None  
430:         self.smooth_right = None  
431:         self.smooth_front = None  
432:         self.smooth_front_left = None  
433:         self.smooth_front_right = None  
434:         self.d_front = None  
435:         self.d_left = None  
436:         self.d_right = None  
437:         self.d_front_left = None  
438:         self.d_front_right = None  
439:         self.sensor_index = 0  
440:         self.gyro_z_prev = 0  
441:         self._servo_last_angle = SERVO_CENTER
```





```
442: self._servo_last_ns = time.monotonic_ns()
443: # -- Simple safe-straight latch --
444: self.ss = 0          # 0 = inactive, +1 = left triggered, -1 = right triggered
445: self.ss_base = None # distance at trigger
446:
447: def ss_reset(self):
448:     self.ss = 0
449:     self.ss_base = None
450:
451: def set_filter_size(self, n):
452:     self.front_history = deque(list(self.front_history)[-n:], maxlen=n)
453:     self.left_history = deque(list(self.left_history)[-n:], maxlen=n)
454:     self.right_history = deque(list(self.right_history)[-n:], maxlen=n)
455:     self.front_left_history = deque(list(self.front_left_history)[-n:], maxlen=n)
456:     self.front_right_history = deque(list(self.front_right_history)[-n:], maxlen=n)
457:
458: def set_servo(self, angle):
459:     # clamp to physical limits
460:     target = max(SERVO_MIN_ANGLE, min(SERVO_MAX_ANGLE, angle))
461:     # angle → pulse (μs)
462:     pulse = int(
463:         SERVO_PULSE_MIN
464:         + (SERVO_PULSE_MAX - SERVO_PULSE_MIN)
465:         * ((target - SERVO_MIN_ANGLE) / (SERVO_MAX_ANGLE - SERVO_MIN_ANGLE)))
466: )
467: # write to PCA9685
468: with I2C_LOCK:
469:     self.pca.channels[SERVO_CHANNEL].duty_cycle = int(pulse * 65535 / SERVO_PERIOD)
470: return target
471:
472: def rotate_motor(self, speed):
473:     duty_cycle = int(min(max(abs(speed), 0), 100)/100*65535)
474:     with I2C_LOCK:
475:         if speed >= 0:
476:             self.pca.channels[MOTOR_FWD].duty_cycle = duty_cycle
477:             self.pca.channels[MOTOR_REV].duty_cycle = 0
478:         else:
479:             self.pca.channels[MOTOR_FWD].duty_cycle = 0
480:             self.pca.channels[MOTOR_REV].duty_cycle = duty_cycle
481:
482: def stop_motor(self):
483:     with I2C_LOCK:
484:         self.pca.channels[MOTOR_FWD].duty_cycle = 0
485:         self.pca.channels[MOTOR_REV].duty_cycle = 0
486:
487: def set_state_speed(self, state):
488:     # Set motor speed based on FSM state.
489:     speed = state_speed_value(state)
490:     self.rotate_motor(speed)
491:
492: def get_distance(self, sensor):
493:     #sensor: a gpiozero.DistanceSensor instance
494:     try:
495:         d = sensor.distance # returns meters
496:         if d is None:
497:             return None
498:         return d * 100 # convert to cm
499:     except:
500:         return None
501:
502: def stable_filter(self, new_val, prev_val, alpha=FILTER_ALPHA, max_jump=FILTER_JUMP):
503:     #Reject large spikes and apply exponential moving average smoothing.
504:     if new_val is None:
505:         return prev_val # keep previous if invalid
506:     if prev_val is not None and abs(new_val - prev_val) > max_jump:
507:         new_val = prev_val # reject spike
508:     if prev_val is None:
509:         return new_val
510:     return alpha * new_val + (1 - alpha) * prev_val
511:
512: def filtered_distance(self, sensor_obj, history, smooth_attr, sensor_type='us'):
513:     # Fallback to use when ToF is invalid or too far (use US side max, in cm)
514:     TOF_FALLBACK_CM = US_MAX_DISTANCE_SIDE * 100.0 # 1.2 m -> 120 cm
515:     try:
516:         if sensor_type == 'tof':
517:             with I2C_LOCK:
518:                 raw_mm = sensor_obj.range # VL53L0X returns mm
```



```
519: d = (raw_mm / 10.0) if raw_mm is not None else None # -> cm
520: invalid_far = (d is None) or (d >= 800.0) # ≥ 8 m or None
521: d_for_history = None if invalid_far else d # keep filter clean
522: else:
523: d = sensor_obj.distance * 100.0 # meters -> cm
524: invalid_far = False
525: d_for_history = d
526: except:
527: d = None
528: invalid_far = (sensor_type == 'tof')
529: d_for_history = None
530:
531: # Update history with valid values only (for stable median/EMA)
532: history.append(d_for_history)
533: valid = [x for x in history if x is not None]
534:
535: # If ToF is invalid/too far, immediately report the side max distance (120 cm)
536: if sensor_type == 'tof' and invalid_far:
537: return TOF_FALLBACK_CM
538:
539: if not valid:
540: # No valid history yet -> for ToF use side max, else keep your sentinel
541: return TOF_FALLBACK_CM if sensor_type == 'tof' else 999
542:
543: median_val = np.median(valid)
544: # avg_val = np.mean(valid) # unused
545: filtered_val = median_val
546:
547: prev_val = getattr(self, smooth_attr)
548: alpha = FILTER_ALPHA_TOF if sensor_type == 'tof' else FILTER_ALPHA
549: max_jump = FILTER_JUMP_TOF if sensor_type == 'tof' else FILTER_JUMP
550: smoothed_val = self.stable_filter(filtered_val, prev_val, alpha, max_jump)
551:
552: setattr(self, smooth_attr, smoothed_val)
553: return smoothed_val
554:
555: def safe_straight_control(self, d_left, d_right):
556: # Treat 999/None as no data
557: d_left = None if (d_left is None or d_left >= 900) else d_left
558: d_right = None if (d_right is None or d_right >= 900) else d_right
559:
560: m = eff_soft_margin()
561: max_corr = eff_max_correction()
562:
563: # ----- ARM if inactive (inside margin by more than CORR_EPS) -----
564: if self.ss == 0:
565: if d_left is not None and d_left < (m - CORR_EPS):
566: self.ss = +1; self.ss_base = d_left
567: elif d_right is not None and d_right < (m - CORR_EPS):
568: self.ss = -1; self.ss_base = d_right
569: else:
570: return SERVO_CENTER
571:
572: # ----- ACTIVE: correct only away from the triggering side -----
573: d = d_left if self.ss > 0 else d_right
574:
575: # Release if sensor lost or clearly recovered beyond margin
576: if d is None or d >= (m + CORR_EPS):
577: self.ss_reset()
578: return SERVO_CENTER
579:
580: # Small deadband near the margin to avoid micro-jitter
581: err = m - d # >0 when too close
582: if err <= CORR_EPS:
583: return SERVO_CENTER
584:
585: corr = self.ss * min(max_corr, CORRECTION_MULTIPLIER * err)
586: angle = SERVO_CENTER + corr
587: return max(SERVO_MIN_ANGLE, min(SERVO_MAX_ANGLE, angle))
588:
589: def turn_decision(self, d_left, d_right, d_fl=None, d_fr=None): #Prefer corner/front sensors
for turn decision; fallback to sides if absent.
590: l_val = d_fl if d_fl is not None and d_fl != 999 else d_left
591: r_val = d_fr if d_fr is not None and d_fr != 999 else d_right
592: left_open = (l_val is None) or (l_val == 999) or (l_val > TURN_DECISION_THRESHOLD)
593: right_open = (r_val is None) or (r_val == 999) or (r_val > TURN_DECISION_THRESHOLD)
594:
```



```
595: # Decide only when exactly one side is open; otherwise keep trying.
596: if left_open ^ right_open:                                #XOR: exactly one is True
597:     return "LEFT" if left_open else "RIGHT"
598: return None # both open or both closed → keep trying
599:
600: robot = RobotController(pca)
601:
602: def sensor_reader():
603:     global sensor_data
604:     while True:
605:         # Left sensor
606:         if vl53_left: # ToF
607:             left = robot.filtered_distance(vl53_left, robot.left_history, "smooth_left", sensor_type='tof')
608:         elif us_left: # ultrasonic
609:             left = robot.filtered_distance(us_left, robot.left_history, "smooth_left", sensor_type='us')
610:         else:
611:             left = None
612:
613:         # Right sensor
614:         if vl53_right:
615:             right = robot.filtered_distance(vl53_right, robot.right_history, "smooth_right",
616:             sensor_type='tof')
617:         elif us_right:
618:             right = robot.filtered_distance(us_right, robot.right_history, "smooth_right",
619:             sensor_type='us')
620:         else:
621:             right = None
622:
623:         # Front sensor
624:         if vl53_front:
625:             front = robot.filtered_distance(vl53_front, robot.front_history, "smooth_front",
626:             sensor_type='tof')
627:         elif us_front:
628:             front = robot.filtered_distance(us_front, robot.front_history, "smooth_front",
629:             sensor_type='us')
630:         else:
631:             front = None
632:
633:         # Corner/front-left sensor
634:         if vl53_front_left:
635:             front_left = robot.filtered_distance(vl53_front_left, robot.front_left_history,
636:             "smooth_front_left", sensor_type='tof')
637:         else:
638:             front_left = None
639:
640:         # Corner/front-right sensor
641:         if vl53_front_right:
642:             front_right = robot.filtered_distance(vl53_front_right, robot.front_right_history,
643:             "smooth_front_right", sensor_type='tof')
644:         else:
645:             front_right = None
646:
647:         # Save readings in global dictionary
648:         with sensor_lock:
649:             sensor_data["front"] = front
650:             sensor_data["left"] = left
651:             sensor_data["right"] = right
652:             sensor_data["front_left"] = front_left
653:             sensor_data["front_right"] = front_right
654:
655:
656: class RobotState(Enum):
657:     IDLE = auto()
658:     CRUISE = auto()
659:     TURN_INIT = auto()
660:     TURNING = auto()
661:     POST_TURN = auto()
662:     STOPPED = auto()
663:
664:     locked_turn_direction = None # Keep a global locked_turn_direction variable to persist across
runs (reset in stop_loop)
```



```
665:  
666: def robot_loop():  
667:     global status_text, turn_count, lap_count, locked_turn_direction, stop_reason,  
obstacle_wait_deadline  
668:     global _btn_prev, _btn_last_ts, paused_by_button  
669:     state = RobotState.IDLE  
670:     direction = None  
671:     yaw = 0.0  
672:     turn_start_yaw = 0.0  
673:     turn_start_time = 0.0  
674:     post_turn_start = 0.0  
675:     last_turn_time = -999  
676:     last_ns = time.monotonic_ns()  
677:     start_ns = last_ns  
678:     turn_target_delta = 0.0 # relative yaw target for this turn (deg)  
679:     robot.stop_motor() # Ensure robot stopped at start  
680:     robot.set_servo(SERVO_CENTER) # Ensure robot stopped at start  
681:  
682:     while True:  
683:         current_ns = time.monotonic_ns()  
684:         dt = (current_ns - last_ns) * 1e-9 # seconds  
685:         last_ns = current_ns  
686:         current_time = current_ns * 1e-9 # float seconds, used by UI/timers  
687:  
688:         # --- Headless START button edge detection (press to PAUSE/RESUME) ---  
689:         if not USE_GUI:  
690:             cur = 0 if START_BTN.is_pressed else 1  
691:             if _btn_prev == 1 and cur == 0 and (current_time - _btn_last_ts) > BTN_DEBOUNCE_S:  
692:                 _btn_last_ts = current_time  
693:                 headless_toggle_pause()  
694:                 _btn_prev = cur  
695:  
696:             if not readings_event.is_set():  
697:                 # if readings not started, keep motors stopped and idle  
698:                 robot.stop_motor()  
699:                 state = RobotState.IDLE  
700:                 status_text = "Idle"  
701:                 sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()  
702:                 continue  
703:  
704:             # -----  
705:             # Sensor reads & gyro integration (always update while readings_flag)  
706:             # -----  
707:             with sensor_lock:  
708:                 d_front = sensor_data["front"]  
709:                 d_left = sensor_data["left"]  
710:                 d_right = sensor_data["right"]  
711:                 d_front_left = sensor_data["front_left"]  
712:                 d_front_right = sensor_data["front_right"]  
713:  
714:                 # Update robot attributes for plotting/logging and decision making  
715:                 robot.d_front = d_front  
716:                 robot.d_left = d_left  
717:                 robot.d_right = d_right  
718:                 robot.d_front_left = d_front_left  
719:                 robot.d_front_right = d_front_right  
720:  
721:             # ---- Narrow corridor detector (sum of side distances) ----  
722:             if not hasattr(robot_loop, "_narrow_mode"):  
723:                 robot_loop._narrow_mode = False  
724:  
725:             # Use None-safe math (ignore invalid readings)  
726:             l = robot.d_left if (robot.d_left is not None and robot.d_left != 999) else None  
727:             r = robot.d_right if (robot.d_right is not None and robot.d_right != 999) else None  
728:             sum_lr = None if (l is None or r is None) else (l + r)  
729:  
730:             enter_thresh = NARROW_SUM_THRESHOLD  
731:             exit_thresh = NARROW_SUM_THRESHOLD + NARROW_HYSTESIS  
732:  
733:             prev_mode = robot_loop._narrow_mode  
734:             if sum_lr is not None:  
735:                 if not robot_loop._narrow_mode and sum_lr < enter_thresh:  
736:                     robot_loop._narrow_mode = True  
737:                 elif robot_loop._narrow_mode and sum_lr > exit_thresh:  
738:                     robot_loop._narrow_mode = False  
739:  
740:             # Apply global factors
```



```
741: global SPEED_ENV_FACTOR, DIST_ENV_FACTOR
742: SPEED_ENV_FACTOR = NARROW_FACTOR_SPEED if robot_loop._narrow_mode else 1.0
743: DIST_ENV_FACTOR = NARROW_FACTOR_DIST if robot_loop._narrow_mode else 1.0
744:
745: # Optional console feedback
746: if robot_loop._narrow_mode != prev_mode:
747: mode_txt = "NARROW mode ON" if robot_loop._narrow_mode else "NARROW mode OFF"
748: dprint(f"[corridor] {mode_txt} (L+R = {sum_lr:.1f} cm)" if sum_lr is not None else f"[corridor] {mode_txt}")
749:
750: with I2C_LOCK:
751: raw_gyro_z = mpu.gyro[2] - bias # rad/s from MPU6050
752: ALPHA = 0.8
753: gyro_z_filtered = ALPHA * raw_gyro_z + (1 - ALPHA) * getattr(robot, 'gyro_z_prev', 0.0)
754: robot.gyro_z_prev = gyro_z_filtered
755: yaw += (gyro_z_filtered * RAD2DEG) * dt
756:
757: # Append to deque for plotting/logging
758: elapsed_time = (current_ns - start_ns) * 1e-9
759: time_data.append(elapsed_time)
760: front_data.append(robot.d_front if robot.d_front is not None else 0)
761: left_data.append(robot.d_left if robot.d_left is not None else 0)
762: right_data.append(robot.d_right if robot.d_right is not None else 0)
763: _fl = getattr(robot, "d_front_left", None)
764: _fr = getattr(robot, "d_front_right", None)
765: front_left_data.append(_fl if _fl is not None else 0)
766: front_right_data.append(_fr if _fr is not None else 0)
767: angle_data.append(yaw)
768: state_data.append(state.name)
769:
770: # If loop_flag is still False, do NOT run FSM/motors yet--just update plots
771: if not loop_event.is_set():
772: robot.stop_motor()
773: state = RobotState.IDLE
774: status_text = "Paused" if paused_by_button else "Sensor readings started"
775: sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
776: continue
777:
778: # -----
779: # FSM transitions & actions
780: # -----
781: if state == RobotState.IDLE:
782: status_text = "Ready (readings started, loop not started)" if readings_event.is_set() else "Idle"
783: robot.stop_motor()
784: robot.set_servo(SERVO_CENTER)
785: robot.ss_reset()
786: if loop_event.is_set():
787: state = RobotState.CRUISE
788: status_text = "Driving (cruise)"
789: for speed in range(0, SPEED_CRUISE + 1):
790: if not loop_event.is_set():
791: robot.stop_motor()
792: break
793: robot.rotate_motor(speed)
794: sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
795:
796: elif state == RobotState.CRUISE:
797: status_text = "Driving (cruise)"
798:
799: # -----
800: # Emergency stop: immediate
801: # -----
802: if robot.d_front is not None and robot.d_front < eff_stop_threshold():
803: robot.stop_motor()
804: robot.set_servo(SERVO_CENTER)
805: status_text = "Stopped! Obstacle ahead"
806: dprint(status_text)
807: state = RobotState.STOPPED
808:
809: stop_reason = "OBSTACLE"
810: obstacle_wait_deadline = current_time + OBSTACLE_WAIT_TIME
811: continue
812:
813: # -----
814: # Check if we can trigger a turn
815: # -----
```



```
816: front_triggered = (robot.d_front is not None and robot.d_front < eff_front_turn_trigger())
817: lockout_ok = (current_time - last_turn_time >= TURN_LOCKOUT)
818:
819: if front_triggered and lockout_ok and loop_event.is_set():
820: # immediately enter TURN_INIT and drop speed
821: state = RobotState.TURN_INIT
822: status_text = "Approaching turn - waiting for side to open"
823: continue
824:
825: # -----
826: # Normal cruise servo control
827: # Safe straight control active only after the 1st turn
828: # -----
829: if turn_count >= 1:
830: desired_servo_angle = robot.safe_straight_control(robot.d_left, robot.d_right)
831: else:
832: desired_servo_angle = SERVO_CENTER
833:
834: robot.set_servo(desired_servo_angle)
835: robot.set_state_speed(state.name)
836:
837: elif state == RobotState.TURN_INIT:
838: # Stay slow and keep wheels mostly straight while we wait for a side to open.
839: # If you prefer slight wall-following here, use safe_straight_control().
840: status_text = "Turn init - waiting for open side"
841: robot.set_state_speed(state.name)
842: robot.ss_reset()
843:
844: # fail-safe: if front becomes safe again, return to cruise
845: if robot.d_front is not None and robot.d_front >= eff_front_turn_trigger():
846: robot.set_servo(SERVO_CENTER)
847: state = RobotState.CRUISE
848: status_text = "Driving (cruise)"
849: # brief yield
850: sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
851: continue
852:
853: # keep the wheels centered (or gentle straight correction)
854: if turn_count >= 1:
855: desired_servo_angle = robot.safe_straight_control(robot.d_left, robot.d_right)
856: else:
857: # No correction before first turn: keep the wheels centered (or gentle straight correction)
858: desired_servo_angle = SERVO_CENTER
859: robot.set_servo(desired_servo_angle)
860:
861: # Decide direction only when exactly one side is open
862: proposed_direction = robot.turn_decision(
863: robot.d_left, robot.d_right,
864: d_fl=robot.d_front_left, d_fr=robot.d_front_right
865: )
866:
867: # If neither or both are open, keep waiting at TURN_INIT speed
868: if proposed_direction is None:
869: sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
870: continue
871:
872: # Turn lock mechanic (unchanged logic, just applied here)
873: if LOCK_TURN_DIRECTION == 1:
874: if locked_turn_direction is None:
875: locked_turn_direction = proposed_direction
876: dprint(f"Turn direction locked to {locked_turn_direction}")
877: elif proposed_direction != locked_turn_direction:
878: # keep waiting at TURN_INIT speed for the locked direction to open
879: status_text = f"Ignoring {proposed_direction} (locked to {locked_turn_direction})"
880: robot.set_servo(desired_servo_angle)
881: sensor_tick.wait(LOOP_DELAY); sensor_tick.clear()
882: continue
883: direction = locked_turn_direction
884: else:
885: direction = proposed_direction
886:
887: # Commit the turn now that one side is open and lock (if any) allows it
888: dprint(f"Turn initiated {direction}. Left: {robot.d_left if robot.d_left is not None else -1:.1f} cm, Right: {robot.d_right if robot.d_right is not None else -1:.1f} cm")
889: turn_start_yaw = yaw
890: turn_start_time = current_time
891:
```



```
892: # how skewed are we vs the nearest corridor axis at entry?
893: entry_skew = norm180(yaw - snap90(yaw)) # + = already rotated to the LEFT
894:
895: # base ±TARGET_TURN_ANGLE, then compensate by entry skew
896: base = TARGET_TURN_ANGLE if direction == "LEFT" else -TARGET_TURN_ANGLE
897: turn_target_delta = base - entry_skew
898:
899: # keep within your safety bounds
900: turn_target_delta = max(-MAX_TURN_ANGLE, min(MAX_TURN_ANGLE, turn_target_delta))
901: if abs(turn_target_delta) < MIN_TURN_ANGLE:
902: turn_target_delta = MIN_TURN_ANGLE if turn_target_delta >= 0 else -MIN_TURN_ANGLE
903:
904: # start turning
905: angle = TURN_ANGLE_LEFT if direction == "LEFT" else TURN_ANGLE_RIGHT
906: robot.set_servo(angle)
907: robot.set_state_speed("TURNING")
908: state = RobotState.TURNING
909:
910: elif state == RobotState.TURNING:
911: # active turning: monitor yaw & safety/time conditions to stop
912: # compute turn angle relative to start
913: turn_angle = yaw - turn_start_yaw # +left, -right
914: target_angle = turn_target_delta # adjusted by entry skew
915: robot.set_state_speed("TURNING")
916: stop_condition = False
917: robot.ss_reset()
918:
919: # Condition A: Turn angle
920: if abs(turn_angle - target_angle) <= TURN_ANGLE_TOLERANCE:
921: dprint("Stop Turn - Target Angle")
922: stop_condition = True
923: # Condition B: timeout
924: if current_time - turn_start_time > TURN_TIMEOUT:
925: dprint("Stop Turn - Max turn time")
926: stop_condition = True
927: # Condition C: Max turn angle
928: if abs(turn_angle) >= MAX_TURN_ANGLE:
929: dprint("Stop Turn - Max turn angle")
930: stop_condition = True
931:
932: # If any stop condition met -> finish turn
933: if stop_condition:
934: robot.stop_motor()
935: robot.set_servo(SERVO_CENTER)
936: last_turn_time = current_time
937: yaw = snap90(yaw)
938: # update counters
939: turn_count += 1
940: if turn_count % 4 == 0:
941: lap_count += 1
942: dprint(f"✓ Lap completed! Total laps: {lap_count}")
943: # if reached max laps -> do final drive then stop (as in original logic)
944: if MAX_LAPS > 0 and lap_count >= MAX_LAPS:
945: # Drive forward a little before stopping
946: robot.set_servo(SERVO_CENTER)
947: robot.set_state_speed(state.name)
948: sensor_tick.wait(POST_LAP_DURATION); sensor_tick.clear()
949: robot.stop_motor()
950: stop_reason = "LAPS"
951: loop_event.clear()
952: sensor_tick.set() # wake waits immediately
953: state = RobotState.STOPPED
954: status_text = f"Stopped - Max Laps ({MAX_LAPS}) reached"
955: continue
956:
957: # prepare post-turn
958: post_turn_start = current_time
959: state = RobotState.POST_TURN
960: status_text = "Driving (post-turn)"
961:
962: elif state == RobotState.POST_TURN:
963: # drive straight for short duration then return to CRUISE
964: if current_time - post_turn_start < POST_TURN_DURATION:
965: robot.set_servo(SERVO_CENTER)
966: robot.set_state_speed(state.name)
967: status_text = "Driving (post-turn)"
968: else:
```



```
969: state = RobotState.CRUISE
970: status_text = "Driving (cruise)"
971:
972: elif state == RobotState.STOPPED:
973: robot.stop_motor()
974: robot.set_servo(SERVO_CENTER)
975: robot.ss_reset()
976:
977: # --- Obstacle-driven stop: auto-retry after OBSTACLE_WAIT_TIME ---
978: if stop_reason == "OBSTACLE":
979: # nice status with countdown
980: remaining = max(0.0, obstacle_wait_deadline - current_time)
981: status_text = f"Stopped (obstacle) - retry in {remaining:.1f}s"
982:
983: # time to retry? re-check front distance
984: if current_time >= obstacle_wait_deadline:
985: # quick re-sample already happens in the sensor thread;
986: # just use the latest filtered reading here:
987: d_front_now = robot.d_front
988:
989: if d_front_now is not None and d_front_now >= eff_stop_threshold():
990: # clear → resume cruise
991: dprint("✓ Obstacle cleared - resuming")
992: status_text = "Driving (cruise)"
993: stop_reason = None
994: state = RobotState.CRUISE
995: robot.set_state_speed("CRUISE")
996: # small fall-through; next loop iteration will continue normally
997: else:
998: # still blocked → schedule another check in OBSTACLE_WAIT_TIME
999: obstacle_wait_deadline = current_time + OBSTACLE_WAIT_TIME
1000:
1001: # yield CPU while waiting
1002: sensor_tick.wait LOOP_DELAY); sensor_tick.clear()
1003: continue
1004:
1005: # --- User or laps stop: stay stopped until user action (original behavior) ---
1006: status_text = "Stopped"
1007: sensor_tick.wait LOOP_DELAY); sensor_tick.clear()
1008: continue
1009:
1010: # =====
1011: # GUI SECTION
1012: # =====
1013: def launch_gui(): #Initialize Tkinter GUI, matplotlib plots, sliders, and status indicators.
1014: import tkinter as tk
1015: from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
1016: import matplotlib
1017: matplotlib.use("TkAgg") # explicit backend for GUI mode
1018: import matplotlib.pyplot as plt
1019: import tkinter.filedialog as fd
1020: global btn_readings, btn_start, btn_stop, root, canvas
1021: global ax_front, ax_side, ax_angle
1022: global front_line, left_line, right_line, angle_line
1023: global status_circle, status_canvas, status_text_id, lbl_status
1024: global lbl_turns, lbl_laps
1025: global sliders_frame, slider_vars, btn_export
1026:
1027: from tkinter import TclError
1028: import sys, os, signal # add
1029:
1030: GUI_CLOSING = False
1031: after_ids = {"status": None, "plot": None}
1032:
1033: # -----
1034: # Export to CSV
1035: # -----
1036: def export_data_csv():
1037: if not USE_GUI:
1038: dprint("Headless: CSV export disabled.")
1039: return
1040: filename = f"viva_data_{datetime.now().strftime('%Y%m%d_%H%M%S')}.csv"
1041: slider_values = {name: var.get() for name, var in slider_vars.items()}
1042:
1043: with open(filename, mode='w', newline='') as file:
1044: writer = csv.writer(file)
1045: header = [
```



```
1046: "Time (s)", "Front (cm)", "Left (cm)", "Right (cm)", "Yaw (deg)",  
1047: "State", "Turns", "Laps", "Servo Angle", "Speed", "Turning", "Direction"  
1048: ]  
1049: header += [name for name in slider_values.keys()]  
1050: writer.writerow(header)  
1051:  
1052: for i in range(len(time_data)):  
1053: row = [  
1054: round(time_data[i], 2),  
1055: round(front_data[i], 2),  
1056: round(left_data[i], 2),  
1057: round(right_data[i], 2),  
1058: round(angle_data[i], 2),  
1059: state_data[i],  
1060: turn_count,  
1061: lap_count,  
1062: globals().get("servo_angle", 0), # latest steering  
1063: globals().get("SPEED_CRUISE", 0),  
1064: "YES" if "Turning" in state_data[i] else "NO",  
1065: globals().get("locked_turn_direction", "")  
1066: ]  
1067: row += [slider_values[name] for name in slider_values.keys()]  
1068: writer.writerow(row)  
1069:  
1070: print(f"✓ Data exported to {filename}")  
1071:  
1072: # -----  
1073: # Save/Load Slider Config  
1074: # -----  
1075: def save_sliders_json():  
1076: if not USE_GUI:  
1077: print("Headless: save_sliders_json disabled.")  
1078: return  
1079: import tkinter.filedialog as fd  
1080: default_filename = f"sliders_config_{datetime.now().strftime('%Y%m%d')}.json"  
1081: file_path = fd.asksaveasfilename(initialdir=BASE_DIR,  
1082: initialfile=default_filename,  
1083: defaultextension=".json",  
1084: filetypes=[("JSON files", "*.json")],  
1085: title="Save Slider Configuration")  
1086: if file_path:  
1087: data = {name: var.get() for name, var in slider_vars.items()}  
1088: with open(file_path, "w") as f:  
1089: json.dump(data, f, indent=4)  
1090: print(f"Slider values saved to {file_path}")  
1091:  
1092: def load_sliders_json():  
1093: if not USE_GUI:  
1094: print("Headless: load_sliders_json disabled.")  
1095: return  
1096: import tkinter.filedialog as fd  
1097: file_path = fd.askopenfilename(initialdir=BASE_DIR,  
1098: defaultextension=".json",  
1099: filetypes=[("JSON files", "*.json")],  
1100: title="Load Slider Configuration")  
1101: if file_path:  
1102: try:  
1103: with open(file_path, "r") as f:  
1104: data = json.load(f)  
1105: for name, value in data.items():  
1106: if name in slider_vars:  
1107: slider_vars[name].set(value)  
1108: globals()[name] = value  
1109: print(f"Sliders restored from {file_path}")  
1110: except Exception as e:  
1111: print(f"Failed to load sliders: {e}")  
1112:  
1113: def gui_alive():  
1114: try:  
1115: return not GUI_CLOSING and root.winfo_exists()  
1116: except Exception:  
1117: return False  
1118:  
1119: def _cancel_afters():  
1120: try:  
1121: if after_ids.get("status"):  
1122: root.after_cancel(after_ids["status"])
```



```
1123: except Exception:  
1124: pass  
1125: try:  
1126: if after_ids.get("plot"):  
1127: root.after_cancel(after_ids["plot"])  
1128: except Exception:  
1129: pass  
1130:  
1131: def _really_exit():  
1132: # ultimate "get me out" in case something else is blocking  
1133: try: root.quit()  
1134: except: pass  
1135: try: root.destroy()  
1136: except: pass  
1137: os._exit(0)  
1138:  
1139:  
1140: root = tk.Tk()  
1141: root.title("VivaLaVida Robot Control")  
1142:  
1143: slider_vars = {}  
1144:  
1145: # ----- Matplotlib Figure -----  
1146: fig, (ax_front, ax_side, ax_angle) = plt.subplots(3, 1, figsize=(6, 8))  
1147: fig.tight_layout(pad=3.0)  
1148:  
1149: ax_front.set_ylim(0, 350)  
1150: ax_front.set_title("Front Sensor")  
1151: ax_front.grid(True)  
1152: front_line, = ax_front.plot([], [], color="blue")  
1153:  
1154: ax_side.set_ylim(-150, 150)  
1155: ax_side.set_title("Left (+Y) vs Right (-Y) Sensors")  
1156: ax_side.grid(True)  
1157: ax_side.axhline(0, color="black")  
1158: left_line, = ax_side.plot([], [], color="#009E73", label="Left")  
1159: right_line, = ax_side.plot([], [], color="#E69F00", label="Right (neg)")  
1160: # New: front-corner ToFs drawn on the same axes  
1161: fl_line, = ax_side.plot([], [], linestyle="--", color="#58D1B1", label="Front-Left")  
1162: fr_line, = ax_side.plot([], [], linestyle="--", color="#F3C553", label="Front-Right (neg)")  
1163: #ax_side.legend(loc="upper right", fontsize=8)  
1164: ax_side.legend(loc="upper center", bbox_to_anchor=(0.5, -0.12), ncol=2, fontsize=8,  
frameon=False)  
1165: fig.subplots_adjust(bottom=0.18) # make room at the bottom  
1166:  
1167:  
1168: ax_angle.set_ylim(-180, 180)  
1169: ax_angle.set_title("Yaw Angle")  
1170: ax_angle.grid(True)  
1171: angle_line, = ax_angle.plot([], [], color="purple")  
1172:  
1173: # ----- Buttons -----  
1174: btn_readings = tk.Button(root, text="Start Readings", command=start_readings,  
width=20, height=2, bg="blue", fg="white")  
1175: btn_start = tk.Button(root, text="Start Loop", command=start_loop,  
width=20, height=2, bg="green", fg="white", state="disabled")  
1176: btn_stop = tk.Button(root, text="Stop Loop", command=stop_loop,  
width=20, height=2, bg="red", fg="white", state="disabled")  
1177:  
1178: # ----- Status Labels -----  
1179: lbl_status = tk.Label(root, text="Idle", font=("Arial", 14))  
1180: lbl_turns = tk.Label(root, text=f"Turns: {turn_count}", font=("Arial", 14))  
1181: lbl_laps = tk.Label(root, text=f"Laps: {lap_count}", font=("Arial", 14))  
1182:  
1183: # ----- Circular Status Indicator -----  
1184: status_canvas = tk.Canvas(root, width=100, height=100, highlightthickness=0,  
bg=root.cget("bg"))  
1185: status_circle = status_canvas.create_oval(10, 10, 90, 90, fill="grey", outline="")  
1186: status_text_id = status_canvas.create_text(50, 50, text="IDLE", fill="white", font=("Arial",  
14, "bold"))  
1187:  
1188: # ----- Sliders Frame -----  
1189: sliders_frame = tk.LabelFrame(root, text="Parameters", padx=10, pady=10)  
1190: sliders_frame.grid(row=0, column=1, rowspan=8, sticky="ns", padx=10, pady=5)  
1191:  
1192: # ----- Column 3 frame for actions -----  
1193: actions_frame = tk.LabelFrame(root, text="Actions", padx=10, pady=10)
```



```
1197: actions_frame.grid(row=0, column=2, rowspan=8, sticky="ns", padx=10, pady=5)
1198:
1199: # ----- Two-column layout -----
1200: btn_readings.grid(row=0, column=0, sticky="ew", padx=5, pady=2)
1201: btn_start.grid(row=1, column=0, sticky="ew", padx=5, pady=2)
1202: btn_stop.grid(row=2, column=0, sticky="ew", padx=5, pady=2)
1203: lbl_status.grid(row=3, column=0, pady=5)
1204: lbl_turns.grid(row=4, column=0, pady=2)
1205: lbl_laps.grid(row=5, column=0, pady=2)
1206: status_canvas.grid(row=6, column=0, pady=5)
1207:
1208: canvas = FigureCanvasTkAgg(fig, master=root)
1209: canvas.get_tk_widget().grid(row=7, column=0, sticky="nsew", padx=5, pady=5)
1210:
1211: root.grid_columnconfigure(0, weight=1) # main buttons + plots
1212: root.grid_columnconfigure(1, weight=0) # sliders
1213: root.grid_columnconfigure(2, weight=0) # actions
1214: root.grid_rowconfigure(7, weight=1) # plot row
1215:
1216: # ----- Slider Definitions -----
1217: slider_groups = {
1218: "Speeds": [
1219: ("Cruise Speed", "SPEED_CRUISE", 0, 100, "int"),
1220: ("Turn Init Speed", "SPEED_TURN_INIT", 0, 100, "int"),
1221: ("Turn Speed", "SPEED_TURN", 0, 100, "int"),
1222: ("Post Turn Speed", "SPEED_POST_TURN", 0, 100, "int")
1223: ],
1224: "Driving & Safety, Wall following, Turns": [
1225: ("Turn angle Left", "TURN_ANGLE_LEFT", 55, 90, "int"),
1226: ("Turn angle Right", "TURN_ANGLE_RIGHT", 90, 125, "int"),
1227: ("Soft side Margin (cm)", "SOFT_MARGIN", 0, 75, "int"),
1228: ("Max Angle Correction dif at Soft Margin (o)", "MAX_CORRECTION", 0, 30, "int"),
1229: ("Stop Threshold (cm)", "STOP_THRESHOLD", 0, 100, "int"),
1230: ("Front Turn Trigger (cm)", "FRONT_TURN_TRIGGER", 50, 150, "int"),
1231: ("Turn Decision Threshold (Left/Right) (cm)", "TURN_DECISION_THRESHOLD", 0, 200, "int"),
1232: ("Turn Timeout (s)", "TURN_TIMEOUT", 0.1, 10, "float"),
1233: ("Max Turn Angle (o)", "MAX_TURN_ANGLE", 90, 120, "int"),
1234: ("Post Turn Duration (s)", "POST_TURN_DURATION", 0, 5, "float"),
1235: ("Turn Lockout (s)", "TURN_LOCKOUT", 0.1, 5, "float")
1236: ],
1237: "Other": [
1238: ("Sensor Filter N (Median)", "N_READINGS", 1, 10, "int"),
1239: ("Max Laps - 0 for infinite", "MAX_LAPS", 0, 20, "int"),
1240: ("Filter Smoothness (alpha)", "FILTER_ALPHA", 0.05, 0.9, "float"),
1241: ("Max Jump (cm)", "FILTER_JUMP", 5, 999, "int"),
1242: ]
1243: }
1244:
1245: # ----- Create Sliders -----
1246: for group_name, sliders in slider_groups.items():
1247: group_frame = tk.LabelFrame(sliders_frame, text=group_name, padx=5, pady=5)
1248: group_frame.pack(fill="x", pady=5)
1249: for label_text, var_name, vmin, vmax, vartype in sliders:
1250: frame = tk.Frame(group_frame)
1251: frame.pack(fill="x", pady=2)
1252:
1253: if vartype == "float":
1254: var = tk.DoubleVar(value=globals()[var_name])
1255: res = 0.1
1256: else:
1257: var = tk.IntVar(value=globals()[var_name])
1258: res = 1
1259:
1260: slider_vars[var_name] = var
1261:
1262: scale = tk.Scale(frame, from_=vmin, to=vmax, orient="horizontal",
1263: variable=var, resolution=res)
1264: scale.pack(side="left", fill="x", expand=True)
1265:
1266: lbl_val = tk.Label(frame, text=f"{label_text}: {var.get()}", width=25, anchor="w")
1267: lbl_val.pack(side="right", padx=5)
1268:
1269: def make_callback(lbl, name, var, label_text):
1270: def callback(value, label_text=label_text):
1271: globals()[name] = var.get()
1272: if name == "MAX_LAPS" and var.get() == 0:
1273: lbl.config(text=f"{label_text}: ∞")
```



```
1274: else:
1275:     lbl.config(text=f"{label_text}: {var.get():.1f}" if isinstance(var.get(), float) else
1276: f"{label_text}: {var.get()}")
1277:     return callback
1278:
1279: scale.config(command=make_callback(lbl_val, var_name, var, label_text))
1280:
1281: # ----- Buttons under sliders -----
1282: btn_export = tk.Button(actions_frame, text="Export CSV", command=export_data_csv,
1283: width=20, height=2, bg="purple", fg="white")
1284: btn_export.pack(pady=10, fill="x")
1285:
1286: btn_save_sliders = tk.Button(actions_frame, text="Save Sliders", command=save_sliders_json,
1287: width=20, height=2, bg="green", fg="white")
1288: btn_save_sliders.pack(pady=5, fill="x")
1289:
1290: btn_load_sliders = tk.Button(actions_frame, text="Load Sliders", command=load_sliders_json,
1291: width=20, height=2, bg="blue", fg="white")
1292: btn_load_sliders.pack(pady=5, fill="x")
1293:
1294: # ===== GUI Update Functions (nested) =====
1295:
1296: def status_color(state: str) -> str:
1297:     if "Stopped" in state:
1298:         return "red"
1299:     elif "Turning" in state:
1300:         return "yellow"
1301:     elif "Driving" in state:
1302:         return "green"
1303:     elif "Loop Started" in state or "Sensor readings started" in state:
1304:         return "blue"
1305:     else:
1306:         return "grey"
1307:
1308: def status_label(state: str) -> str:
1309:     if "Stopped" in state:
1310:         return "STOP"
1311:     elif "Turning" in state:
1312:         return "TURN"
1313:     elif "Driving" in state:
1314:         return "GO"
1315:     elif "Loop Started" in state:
1316:         return "LOOP"
1317:     elif "Sensor readings started" in state:
1318:         return "READ"
1319:     else:
1320:         return "IDLE"
1321:
1322: def update_status():
1323:     if not gui_alive():
1324:         return
1325:     lbl_status.config(text=status_text)
1326:     status_canvas.itemconfig(status_circle, fill=status_color(status_text))
1327:     status_canvas.itemconfig(status_text_id, text=status_label(status_text))
1328:     lbl_turns.config(text=f"Turns: {turn_count}")
1329:     lbl_laps.config(text=f"Laps: {lap_count}" if slider_vars["MAX_LAPS"].get() > 0 else f"Laps:
{lap_count}/∞")
1330:     try:
1331:         after_ids["status"] = root.after(200, update_status)
1332:     except TclError:
1333:         # window is gone; just stop
1334:         return
1335:
1336:     _prev_label_pos = {
1337:         "front": None, "left": None, "right": None,
1338:         "front_left": None, "front_right": None,
1339:         "angle": None
1340:     }
1341:
1342:     def update_plot():
1343:         if not gui_alive():
1344:             return
1345:         try:
1346:             if time_data:
1347:                 # Update lines
1348:                 front_line.set_data(range(len(front_data)), front_data)
```



```
1349: left_line.set_data(range(len(left_data)), left_data)
1350: right_line.set_data(range(len(right_data)), [-v for v in right_data])
1351: fl_line.set_data(range(len(front_left_data)), front_left_data)
1352: fr_line.set_data(range(len(front_right_data)), [-v for v in front_right_data])
1353: angle_line.set_data(range(len(angle_data)), angle_data)
1354:
1355: # Set axis limits
1356: ax_front.set_xlim(0, MAX_POINTS)
1357: ax_side.set_xlim(0, MAX_POINTS)
1358: ax_angle.set_xlim(0, MAX_POINTS)
1359:
1360: # Remove previous text annotations
1361: for ax in [ax_front, ax_side, ax_angle]:
1362:     for t in ax.texts:
1363:         t.remove()
1364:
1365: # Helper for smoothing
1366: def smooth_move(prev, target, alpha=0.3):
1367:     if prev is None:
1368:         return target
1369:     return prev + alpha * (target - prev)
1370:
1371: # -----
1372: # FRONT SENSOR
1373: # -----
1374: if len(front_data) > 0:
1375:     x = len(front_data) - 1
1376:     y_target = front_data[-1]
1377:     y_prev = _prev_label_pos["front"]
1378:     y_smooth = smooth_move(y_prev, y_target)
1379:     _prev_label_pos["front"] = y_smooth
1380:     ax_front.text(x, y_smooth, f"Y: {y_target:.1f} cm", color="blue",
1381:     fontsize=9, fontweight="bold", va="bottom", ha="left")
1382:
1383: # -----
1384: # LEFT SENSOR
1385: # -----
1386: if len(left_data) > 0:
1387:     x = len(left_data) - 1
1388:     y_target = left_data[-1]
1389:     y_prev = _prev_label_pos["left"]
1390:     y_smooth = smooth_move(y_prev, y_target)
1391:     _prev_label_pos["left"] = y_smooth
1392:     ax_side.text(x, y_smooth, f"Y: {y_target:.1f} cm", color="#009E73",
1393:     fontsize=9, fontweight="bold", va="bottom", ha="left")
1394:
1395: # -----
1396: # RIGHT SENSOR
1397: # -----
1398: if len(right_data) > 0:
1399:     x = len(right_data) - 1
1400:     y_target = -right_data[-1]
1401:     y_prev = _prev_label_pos["right"]
1402:     y_smooth = smooth_move(y_prev, y_target)
1403:     _prev_label_pos["right"] = y_smooth
1404:     ax_side.text(x, y_smooth, f"R: {right_data[-1]:.1f} cm", color="#E69F00",
1405:     fontsize=9, fontweight="bold", va="bottom", ha="left")
1406:
1407:
1408: # -----
1409: # FRONT-LEFT SENSOR (corner)
1410: # -----
1411: if len(front_left_data) > 0:
1412:     x = len(front_left_data) - 1
1413:     y_target = front_left_data[-1]
1414:     y_prev = _prev_label_pos["front_left"]
1415:     y_smooth = smooth_move(y_prev, y_target)
1416:     _prev_label_pos["front_left"] = y_smooth
1417:     ax_side.text(x, y_smooth, f"FL: {y_target:.1f} cm", color="#58D1B1", fontsize=9,
fontweight="bold", va="bottom", ha="left")
1418: # -----
1419: # FRONT-RIGHT SENSOR (corner)
1420: # -----
1421: if len(front_right_data) > 0:
1422:     x = len(front_right_data) - 1
1423:     y_target = -front_right_data[-1]
1424:     y_prev = _prev_label_pos["front_right"]
```



```
1425: y_smooth = smooth_move(y_prev, y_target)
1426: _prev_label_pos["front_right"] = y_smooth
1427: ax_side.text(x, y_smooth, f"FR: {front_right_data[-1]:.1f} cm", color="#F3C553", fontsize=9,
fontweight="bold", va="bottom", ha="left")
1428:
1429: # -----
1430: # YAW ANGLE
1431: # -----
1432: if len(angle_data) > 0:
1433: x = len(angle_data) - 1
1434: y_target = angle_data[-1]
1435: y_prev = _prev_label_pos["angle"]
1436: y_smooth = smooth_move(y_prev, y_target)
1437: _prev_label_pos["angle"] = y_smooth
1438: ax_angle.text(x, y_smooth, f"{y_target:.1f}°", color="purple",
1439: fontsize=9, fontweight="bold", va="bottom", ha="left")
1440:
1441: # -----
1442: # Dynamic Y-axis scaling for yaw
1443: # -----
1444: if angle_data:
1445: min_angle = min(angle_data)
1446: max_angle = max(angle_data)
1447: if max_angle - min_angle > 180:
1448: ax_angle.set_ylim(-180, 180)
1449: else:
1450: ax_angle.set_ylim(min_angle - 10, max_angle + 10)
1451:
1452: canvas.draw()
1453: except TclError:
1454: return
1455: try:
1456: after_ids["plot"] = root.after(100, update_plot)
1457: except TclError:
1458: return
1459:
1460: def on_closing():
1461: nonlocal GUI_CLOSING
1462: GUI_CLOSING = True
1463:
1464: # stop scheduling anything new
1465: _cancel_afters()
1466:
1467: # tell loops/threads to stand down
1468: try: loop_event.clear()
1469: except: pass
1470: try: readings_event.clear()
1471: except: pass
1472: try: sensor_tick.set()
1473: except: pass
1474:
1475: # stop robot now
1476: try:
1477: robot.stop_motor()
1478: robot.set_servo(SERVO_CENTER)
1479: except:
1480: pass
1481:
1482: # shut down ToF
1483: for sensor in [vl53_left, vl53_right, vl53_front, vl53_back,
1484: vl53_front_left, vl53_front_right]:
1485: if sensor is not None:
1486: try:
1487: with I2C_LOCK:
1488: sensor.stop_continuous()
1489: except Exception as e:
1490: dprint(f"Warning: could not stop sensor {sensor}: {e}")
1491:
1492: # close DistanceSensors
1493: for us in [us_front, us_left, us_right]:
1494: try:
1495: if us is not None:
1496: us.close()
1497: except:
1498: pass
1499:
1500: # LEDs / button
```



```
1501: try:
1502:     GREEN_LED.off(); RED_LED.off()
1503:     GREEN_LED.close(); RED_LED.close(); START_BTN.close()
1504: except:
1505:     pass
1506:
1507: # IMPORTANT: quit the Tk loop first, then destroy
1508: try: root.quit()
1509: except: pass
1510:
1511: # Some Tk/Matplotlib stacks need the canvas widget gone before destroy
1512: try:
1513:     canvas.get_tk_widget().destroy()
1514: except:
1515:     pass
1516:
1517: try: root.destroy()
1518: except:
1519:     pass
1520:
1521: # Failsafe: if something is still hanging, hard-exit soon
1522: root.after(100, _really_exit)
1523:
1524:
1525: root.protocol("WM_DELETE_WINDOW", on_closing)
1526: root.bind("<Escape>", lambda e: on_closing())
1527: update_status()
1528: update_plot()
1529: root.mainloop()
1530:
1531: # -----
1532: # Start / Stop
1533: # -----
1534:
1535: def start_readings():
1536:     global status_text
1537:     readings_event.set()
1538:     status_text = "Sensor readings started"
1539:     if USE_GUI:
1540:         btn_readings.config(state="disabled")
1541:         btn_start.config(state="normal")
1542:
1543:     # Start background sensor thread once
1544:     if not hasattr(start_readings, "sensor_thread_started"):
1545:         threading.Thread(target=sensor_reader, daemon=True).start()
1546:         start_readings.sensor_thread_started = True
1547:
1548:     if not hasattr(start_readings, "loop_thread_started"):
1549:         threading.Thread(target=robot_loop, daemon=True).start()
1550:         start_readings.loop_thread_started = True
1551:
1552:     sensor_tick.set()
1553:
1554: def start_loop(): #Start the robot driving loop
1555:     global status_text
1556:     if not readings_event.is_set():
1557:         print("Start sensor readings first!")
1558:     return
1559:
1560:     loop_event.set()
1561:     status_text = "🚗 Loop Started"
1562:     if USE_GUI:
1563:         btn_start.config(state="disabled")
1564:         btn_stop.config(state="normal")
1565:
1566:     def stop_loop(): #Stop the robot loop and motor
1567:         global status_text, turn_count, lap_count, locked_turn_direction, stop_reason
1568:         loop_event.clear()
1569:         stop_reason = "USER"
1570:         sensor_tick.set() # wake any waits immediately
1571:         robot.stop_motor()
1572:         status_text = "Stopped"
1573:         turn_count = 0
1574:         lap_count = 0
1575:
1576:     if USE_GUI:
1577:         btn_start.config(state="normal")
```



```
1578: btn_stop.config(state="disabled")
1579: lbl_turns.config(text=f"Turns: {turn_count}")
1580: lbl_laps.config(text=f"Laps: {lap_count}")
1581:
1582: locked_turn_direction = None # Reset direction lock so a new session can re-choose
1583:
1584: # =====
1585: # MAIN
1586: # =====
1587:
1588: if __name__ == "__main__":
1589:     print(f"Starting VivaLaVida Autonomous Drive - GUI mode: {USE_GUI}")
1590:
1591:     if USE_GUI:
1592:         launch_gui()
1593:     else: # Start the sensor reading thread
1594:         #RED_LED.on()
1595:         #GREEN_LED.off()
1596:         sensor_thread = threading.Thread(target=sensor_reader, daemon=True)
1597:         sensor_thread.start()
1598:         print("Headless mode: waiting for START button...")
1599:         RED_LED.on() # turn on red when ready
1600:         GREEN_LED.blink(on_time=0.5, off_time=0.5, background=True)
1601:         #GREEN_LED.on() # green = ready to press the button
1602:         try: # Wait for button press (GPIO input goes LOW when pressed)
1603:             while not START_BTN.is_pressed:
1604:                 time.sleep(0.05)
1605:             print("START button pressed! Beginning autonomous loop...")
1606:             readings_event.set() #readings_flag = True
1607:             GREEN_LED.blink(on_time=0.1, off_time=0.1, background=True)
1608:             RED_LED.off()
1609:             time.sleep(2) # wait 2 seconds before starting loop
1610:             loop_event.set() #loop_flag = True
1611:             GREEN_LED.on() # running
1612:             print("Starting main robot loop...")
1613:             robot_loop() # Start robot loop in this thread (blocking)
1614:         except KeyboardInterrupt:
1615:             print("\nKeyboard interrupt received. Stopping robot loop.")
1616:             try: loop_event.clear() # Stop loops/threads
1617:             except: pass
1618:             try: readings_event.clear()
1619:             except: pass
1620:             try: sensor_tick.set()
1621:             except: pass
1622:             try: robot.stop_motor() # Motors & servo to safe state
1623:             except: pass
1624:             try: robot.set_servo(SERVO_CENTER)
1625:             except: pass
1626:             try: # Stop any ToF sensors cleanly (if enabled)
1627:                 for s in [vl53_left, vl53_right, vl53_front, vl53_back, vl53_front_left, vl53_front_right]:
1628:                     if s is not None:
1629:                         try:
1630:                             with I2C_LOCK:
1631:                                 s.stop_continuous()
1632:                         except Exception as e:
1633:                             dprint(f"[KeyboardInterrupt] ToF stop warning: {e}")
1634:                         except:
1635:                             pass
1636:                             try: # Zero PCA9685 outputs
1637:                                 with I2C_LOCK:
1638:                                     for ch in [MOTOR_FWD, MOTOR_REV, SERVO_CHANNEL]:
1639:                                         pca.channels[ch].duty_cycle = 0
1640:                                     except:
1641:                                         pass
1642:                                         try:
1643:                                             GREEN_LED.off()
1644:                                             RED_LED.off()
1645:                                             GREEN_LED.close()
1646:                                             RED_LED.close()
1647:                                             START_BTN.close()
1648:                                         except:
1649:                                             pass
1650:
1651: # End
```