

École polytechnique de Louvain

Dancing Robots

**Experiments in Domotics with Balancing Robots
and High-Performance Sensor Fusion**

Author: **Fozan SHAHID**

Supervisor: **Peter VAN ROY**

Readers: **Helene VERHAEGHE, Peer STRITZINGER**

Academic year 2025–2026

Master [120] in Computer Science

Abstract

Domestic automation has been quietly accumulating sensor-rich devices that share indoor space with the people living there, and mobile robotics is converging on the same set of problems. Two-wheeled self-balancing robots sit in the middle of that space: compact, mechanically unstable, and constrained in a way that forces a particular kind of design discipline. This thesis builds on top of an inherited self-balancing chassis, brought to a working balancing state on a GRiSP-2 board running Erlang/OTP by a previous master thesis, and asks what to do with the robot once it is standing.

The longer-term picture is a small robot that follows a second, taller robot around a home without GPS, without external cameras, and without beacons of its own. To get there, three intertwined problems are taken up: a path-following layer that traces a smooth curve through received waypoints, a wireless waypoint transmission protocol that shares the inherited single-byte radio link without breaking the existing manual control, and an onboard pose estimator that works only from wheel speeds.

Four pieces of work are reported, none of which modifies the inherited balance loop. A spline+tracker stack provides smooth waypoint following, with a small sub-state machine for new waypoint arrivals. An over-the-air protocol carries waypoints from a sender. An empirical study compares three onboard localization configurations for yaw (wheel odometry alone, yaw based on the integrated gyro, and yaw based on tilt-compensated magnetometer) on the same chassis and the same arena; and a lifecycle finite-state machine that manages the overall trajectory state and with a colour-coded LED scheme makes the system state legible to external observers.

Acknowledgements

Many people helped me carry this master thesis to its end, and without them it would probably not have been possible, so I would like to thank them, in particular:

***Prof. Peter Van Roy**, my supervisor, for his support, his guidance, his good advice, and his enthusiasm for this project.*

***Cédric Ponsard** and **François Goens**, for the detailed documentation of the inherited robot that highlighted their design choices so clearly.*

*My **family** and **friends**, for their continued support throughout my masters.*

AI use disclaimer

Large language models were used throughout the work done on this master's thesis. Google Search AI was used to speed up background research. Gemini was used to accelerate documentation, help with bug discovery, and assist in building the user interface used for testing. QuillBot was used to rephrase and improve some passages of this document.

Contents

I	Introduction and Background	1
1	Introduction	2
1.1	Context	2
1.2	Objective	3
1.3	Contributions	3
2	The Inherited Platform	5
2.1	Mechanical and electrical stack	5
2.2	The wire to the LilyGo co-processor	6
2.3	The balance cascade	7
3	Background and Related Work	8
3.1	The GRiSP-2 platform, Erlang, and Hera	8
3.2	Self-balancing two-wheeled devices	9
3.3	Inertial sensing and complementary filtering	9
3.4	Path representation	10
3.5	Path-tracking controllers	11
3.6	Indoor localization without external tracking or external reference . . .	11
4	Objectives, Problem Statement, and Approach	13
4.1	Functional goal and platform constraint	13
4.2	The four pieces	13
4.3	The localization gap	14
4.4	Methodological rules	14
II	Design and Implementation	16
5	System Architecture	17
5.1	Erlang module map	17
5.2	Tick anatomy	18
5.3	Ownership rules	19
5.4	The velocity mux	19
5.5	Logging	20
6	Path Representation: Catmull-Rom Splines	21
6.1	Why Catmull-Rom over Bezier / cubic spline	21
6.2	Mathematical formulation	22

6.3	Computing the curve	22
6.4	Mid-cruise seed-based rebuild	23
7	Pure Pursuit Controller	25
7.1	Geometric formulation	25
7.2	Lookahead distance choice	26
7.3	Path-advance: pruning behind the robot	27
7.4	Curvature-modulated cruise speed	28
7.5	Tuning ratios	29
7.6	Problems faced	30
7.7	Putting the three pieces together	31
8	Trajectory Sub-state Machine	32
8.1	The three sub-states	32
8.2	Entry and exit conditions	33
8.3	Stop tuning history	34
9	Lifecycle FSM and User Interaction	36
9.1	The three lifecycle atoms and the single-state reset funnel <code>go_idle/3</code>	36
9.2	Transition table	37
9.3	Pre-flight waypoint queueing	38
10	Over-the-Air Waypoint Transmission Protocol and Planning UI	39
10.1	Why a one-byte protocol	39
10.2	Frame layout	39
10.3	The command set	40
10.4	The ADD sequence	41
10.5	Receiver state machine and stability gate	41
10.6	The planning UI as a test harness	42
III	Experimentation and Evaluation	44
11	Experimental Methodology	45
11.1	The three candidate yaw sources	45
11.2	Test paths and physical setting	46
11.3	Logging mechanics	46
11.4	Ground truth and measurement uncertainty	47
12	Wheel Odometry Baseline	48
12.1	Kinematic model	48
12.2	Why it drifts	49
12.3	Measured final-pose error	49
13	Gyro Yaw Fusion Experiment	51
13.1	Gyro Drift Theory	51
13.2	What the data showed	52
13.3	Outcome	52

14 Magnetometer-Based Heading Experiment	53
14.1 Hard iron, soft iron, and why calibration is necessary	53
14.2 The calibration journey	54
14.3 Why it still did not work	54
14.4 Retirement	55
15 Results and Evaluation	56
15.1 The three estimators on one plot	56
15.2 Why odometry won	57
16 System Evaluation	59
16.1 What was delivered	59
16.2 Further testing	60
16.3 LED signals	62
IV Discussion and Conclusion	64
17 Discussion	65
17.1 The four-piece decomposition in retrospect	65
17.2 What the localization journey actually taught	65
17.3 Erlang at the measured loop rate, in hindsight	66
17.4 Levers not pulled	66
18 Future Work	67
18.1 Where this work sits in a longer chain	67
18.2 A second robot and shared UWB localization	67
18.3 Pushing the existing tuning further	68
18.4 Hardware upkeep	68
19 Conclusion	69
Bibliography	70
V Appendices	76
A Code Listings	77
A.1 GRiSP code (Erlang)	77
A.1.1 balancing_robot.erl	77
A.1.2 balancing_robot_sup.erl	78
A.1.3 hera_interface.erl	78
A.1.4 main_loop.erl	80
A.1.5 traj_planner.erl	91
A.1.6 spline.erl	97
A.1.7 pure_pursuit.erl	98
A.1.8 odometry.erl	99
A.1.9 stability_engine.erl	100
A.1.10 robot.erl	101

A.1.11	input_combo.erl	107
A.1.12	led_control.erl	108
A.1.13	csv_logger.erl	109
A.1.14	numerl.erl	109
A.2	Trajectory logger (Python)	111
A.2.1	traj_log.py	111
A.3	Waypoint planner and RC UI (Python)	114
A.3.1	RC_robot_UI.py	114
A.4	LilyGo robot-side firmware (Arduino/C++)	136
A.4.1	lilygo_bot.ino	136
A.4.2	motor_engine.h	139
A.4.3	motor_engine.cpp	140
A.5	LilyGo sender-side firmware (Arduino/C++)	144
A.5.1	liygo_user.ino	144
A.5.2	board_def.h	146
B	Wire Protocol Full Specification	148
B.1	Scope and transport stack	148
B.2	Control byte format	148
B.3	Command set	149
B.4	Data-frame encoding	150
B.5	ADD transaction	150
B.6	CANCEL_N transaction	151
B.7	Receiver state machine	152
B.8	Guards, error handling, and feedback	153
B.9	Constants summary	154
C	Reproducing the Experiments	155
C.1	Rotation rig	155
C.2	Straight-line rig	156
D	Code Repositories	158

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Context

Domestic automation has been quietly accumulating sensor-rich devices that operate in the same physical space as the people living there: adaptive lights, vacuum robots, personal-mobility platforms. Mobile robotics and the Internet of Things have been converging on the same set of indoor problems for a while now [1]. Two-wheeled self-balancing robots are an interesting middle ground in that converging space. They are compact and mechanically unstable in a way that forces a particular kind of design discipline [2], [3], [4]. The platform was not built from scratch. The robot used here was brought to a working balancing state by a previous master thesis, on a GRiSP-2 board running Erlang/OTP through the Hera framework [5], [6], [7]. That is the starting point.

A balancing robot that stands still is not very interesting on its own. The next question is what to do with it once it is standing, and the question taken up here is multi-robot: can one robot follow another robot indoors, without GPS, without external cameras, and without beacons [8]? Three problems sit underneath, planning, path tracking, and knowing where you are, and on this hardware they tangle. A self-balancing chassis cannot pivot on the spot without punishing its own balance loop, so straight-line waypoint chains are out. GPS does not work indoors, so the pose has to come from onboard sensing alone: wheel speeds and a 9-DoF IMU, and working out how to put those sensors to use for pose estimation and traversal is what this thesis ended up spending most of its time on.

The platform itself comes from the master thesis of Goens and Ponsard, who built the chassis and wrote the balance loop on the GRiSP [9]. Their thesis sits at the end of a longer line of GRiSP-based master theses, which collectively grew Hera from a sensor-fusion proof of concept into a fault-tolerant inertial navigation framework [7], [10], [11], [12], [13], [14]. This thesis adds the next layer above that work; Chapter 2 reproduces the parts of the inherited platform that matter for what follows.

1.2 Objective

The main goal was to take the inherited robot, a self-balancing chassis with manual arrow-key remote control from a laptop over LoRa, and add to it the ability to follow a smooth curve through waypoints arriving over the same link, while keeping a usable pose estimate the whole time. The intended sender is a second robot streaming its own position; the sender used here is a desktop stub on that same wire. The goal broke down into three smaller ones, and the three turned out to depend on each other in awkward ways.

First, a path follower layer was needed. That meant a parametric curve through the waypoints and a controller that tracks it. The waypoints had to be interpolated rather than approximated, because the robot is small and deviation of more than 10 cm could mean the robot never actually hit the coordinate. The expectation is that the robot goes through each one. Second, a way to send waypoints was needed over the same wireless link the inherited platform already uses for manual control: laptop, serial to a LilyGo LoRa board on the user side, LoRa over the air to a second LilyGo on the robot side, I²C between that board and the GRiSP. The waypoint protocol had to share the single-byte wire without breaking the existing manual control. Third, and most open-ended, a way for the robot to know where it is was needed. Wheel odometry alone drifts. A gyro might help. A magnetometer might help. Or maybe neither helps and the simplest thing is best. That last question is what the experimentation part of this thesis sets out to answer with actual experiments on actual hardware [1], [15].

The long term idea behind this is a domestic companion robot that follows a person around the home, could also coordinate with other robots and orchestrate a wide variety of specialized robots. A taller variant of the same chassis exists, above one meter tall and built for a butler role, and a sister thesis is developing ultra-wideband ranging that would let the tall robot lock onto a badge worn at the user's waist and track the person directly. The smaller robot, the one used here, would then follow the tall one without needing UWB hardware of its own: it would read the leader's position over the same wireless link this thesis defines, possibly cross-checked by an ultrasonic ranger on the front. None of that is in scope here. It is the reason, though, that the wireless protocol carries waypoints in a generic way, the sender does not need to be the laptop UI built for development; it could just as easily be the another robot streaming its own pose.

1.3 Contributions

There are four pieces of work in this thesis. None of them touches the inherited balance loop.

Smooth waypoint following. A Catmull-Rom spline through the trajectory waypoints [16], plus a Pure Pursuit lookahead controller [17], plus a small sub-state machine running *cruise*, *braking*, and *dwelling* at each waypoint.

An over-the-air waypoint protocol. A protocol layered on top of the robot's existing LoRa and I²C link carries waypoint coordinates from a sender to the robot. The receiver does not care what the sender is: in this thesis it is the same desktop UI that drove the inherited robot with arrow keys, now extended with grid and waypoint controls; the

longer-term sender is a second robot streaming its own position.

An empirical look at localization. Three configurations tested, one at a time, using the same robot and the same test setup: wheel odometry only, odometry plus a gyro-fused yaw, and odometry plus a magnetometer based yaw. The experimentation part works through each one and reports what happened. Both gyro based fused yaw and magnetometer based yaw were eventually rejected, which is itself the central finding.

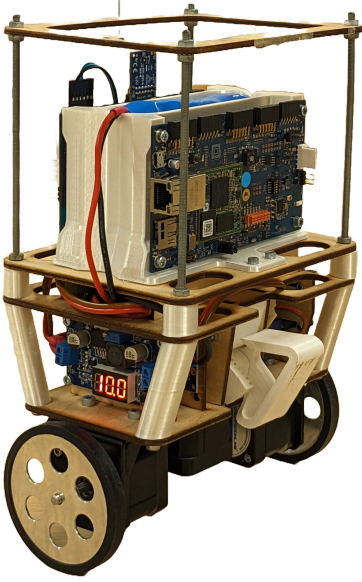
A lifecycle finite-state machine. The robot now has an `idle/running/paused` cycle with a single reset funnel and a LED(GRIiSP onboard) based indicator so system state can be viewed without connecting.

Chapter 2

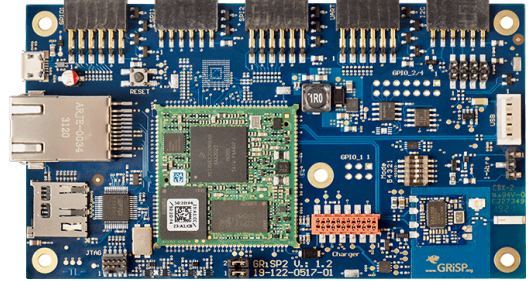
The Inherited Platform

2.1 Mechanical and electrical stack

The robot is a 3D-printed two-wheeled chassis with an 18.5 cm wheelbase, two stepper motors, a lithium-polymer battery, a GRiSP-2 single-board computer [5], [18] with on-board LEDs, a LilyGo TTGO board (ESP32 plus on-board LoRa) [19], and a Pmod NAV nine-axis IMU [20].



(a) The assembled platform. The GRiSP-2 board and Pmod NAV sit on the upper deck; the LilyGo, motor drivers, and battery are on the lower deck.



(b) The GRiSP-2 board. Pmod connectors on the left edge carry the SPI link to the Pmod NAV and the I²C link to the co-processor.

Figure 2.1: Inherited hardware. Both figures reproduced from [9].

2.2 The wire to the LilyGo co-processor

The two boards share the work in the obvious way. The LilyGo handles time-critical tasks: PWM generation for the motors, the LoRa radio, and reading the operator buttons. The GRiSP handles everything else in Erlang. They talk over an I²C bus at address 0x40, GRiSP as master. Every control tick is one read of five bytes followed by one write of three bytes. The read returns two wheel-speed measurements encoded as IEEE 754 half-precision floats [21] plus a one-byte control word whose bits carry the arm state, the button signals, and a protocol flag repurposed for the wireless waypoint protocol later. The write carries two half-precision wheel-speed setpoints and a one-byte output word. The full bit layout is in Appendix B. Figure 2.2 shows how the two boards sit relative to the sensors and motors.

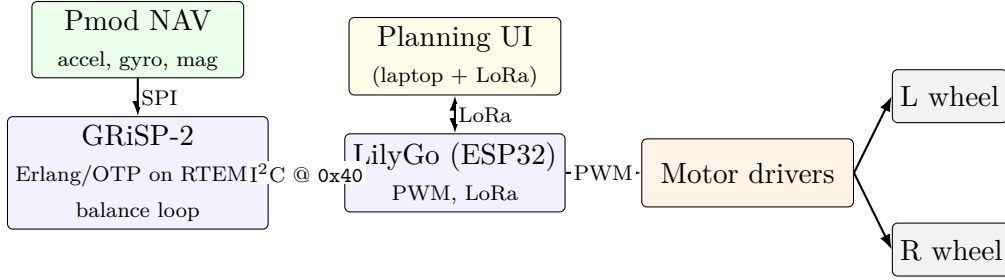


Figure 2.2: Inherited hardware layout. The GRiSP reads the Pmod NAV over SPI and exchanges a 5-byte read / 3-byte write with the LilyGo every tick. The LilyGo owns the PWM, the LoRa radio, and the operator buttons. Waypoint frames from the laptop travel through LoRa and into the same control byte the buttons already use.

2.3 The balance cascade

The robot stays upright on the two-stage cascade that [9] built. An inner PI loop tracks a commanded wheel velocity. An outer PD loop tilts the robot by demanding the wheel velocity that brings the chassis back upright. The tilt estimate comes from an Extended Kalman Filter running inside Hera [7], fed by the accelerometer and gyroscope channels of the Pmod NAV; a complementary-filter version is kept as a configurable fallback [22], [23]. The whole cascade lives in two Erlang modules, `stability_engine` and `pid_controller`. For the rest of this thesis, those two are treated as a black box. No gain was retuned, no filter coefficient was changed, and no part of them was rewritten.

Chapter 3

Background and Related Work

3.1 The GRiSP-2 platform, Erlang, and Hera

The GRiSP-2 is a small embedded board from Stritzinger GmbH [5], [18]. It runs Erlang/OTP directly on top of RTEMS [6], [24], which gives a deterministic real-time OS underneath and the fault-tolerant process model of Erlang on top. The board exposes six Pmod connectors; the Pmod NAV used for inertial sensing connects over SPI [20], [25], and a separate I²C line goes to the ESP32 co-processor [19]. Heavy number crunching is slow in the Erlang VM, so the platform delegates matrix and vector work to Native Implemented Functions (NIFs) written in C, a convention introduced for Hera in earlier theses [12], [13]. Figure 3.1 shows the board.

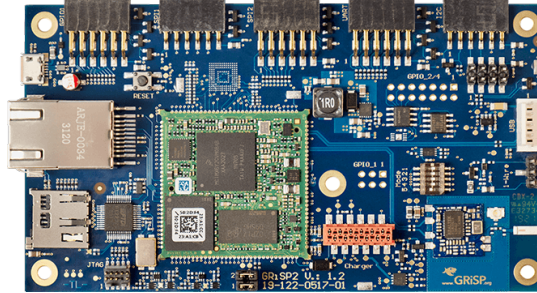


Figure 3.1: The GRiSP-2 single-board computer. The Pmod connectors (left edge) carry the SPI link to the Pmod NAV and the I²C link to the LilyGo co-processor. Reproduced from [9].

Erlang/OTP is an unusual host runtime for an embedded robotic controller. The closest precedent is ROSiE [26], [27], a port of the Robot Operating System to Erlang. The Hera paper [7] positions itself against standard robotic middlewares of the time [28], [29], [30], [31], [32]. The GRiSP master-thesis line forms a small research thread of its own: sensor fusion and fault tolerance built up step by step inside Hera [10], [11], [12], [13], [14]. The compositional benefits of OTP, process isolation, supervisor restarts, hot-code loading, show up throughout the platform.

Hera was introduced by Kalbusch et al. [7] and extended by several master theses since [10], [11], [12], [13], [14]. The framework is a supervision tree with one data process, one

communication process, and any number of measurement processes sharing a common behaviour [33]. Measurements come in asynchronously, get cached in the data process, and are read by downstream consumers without a strict global clock. Hera also ships with a Kalman filter implementation over arbitrary state vectors, which the inherited robot uses for its tilt estimate.

3.2 Self-balancing two-wheeled devices

The Segway transporter [3], the iBOT and Kim-e platforms [4], [34], and the Handle and evoBOT logistic robots from Boston Dynamics and Fraunhofer [2], [35] all exploit the same property: the inverted-pendulum geometry is compact, and that compactness is the source of its instability. The previous thesis catalogued a large set of research and education platforms grouped by control board family: Arduino-style boards [36], [37], [38], [39], [40], [41], [42], [43], LEGO Mindstorms [44], [45], [46], and small embedded computers [47]. Across that corpus, the controller of choice is almost always a cascaded PID. A handful use LQG [45], ADRC [47], or fuzzy-PID hybrids [41], and a few extend the inverted-pendulum study to reaction-wheel or double-pendulum variants [39], [40].

The robot used here is unusual in this group in one way: it runs Erlang/OTP over RTEMS on a GRiSP-2 board, while almost everything else in the literature is C or C++.

3.3 Inertial sensing and complementary filtering

The Pmod NAV gives a three-axis accelerometer, a three-axis gyroscope, and a three-axis magnetometer [20] (Figure 3.2). The first two feed the tilt estimate that drives the inner balance loop. The accelerometer picks up vibrations and is noisy at high frequency. The gyroscope integrates angular rate over time and drifts slowly at low frequency. Because the two noise spectra are complementary, the standard trick is to combine them with a complementary filter, a pair of transfer functions that sum to one, keeping each sensor in the band where it is trustworthy [22], [23]. The same recombination can be done by an Extended Kalman Filter that adapts the weights on the fly, and that is what the inherited robot runs [7], [48], [49]. The derivations, the prediction-update recursion, and the Gaussian-product picture are all in the previous thesis.

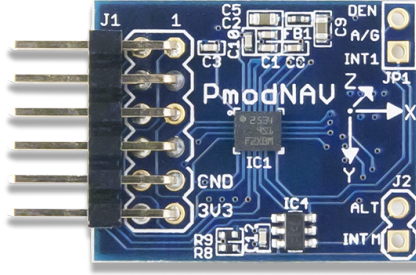


Figure 3.2: The Digilent Pmod NAV module, housing the LSM9DS1 nine-axis IMU (accelerometer, gyroscope, magnetometer). It connects to the GRiSP-2 over SPI.

The magnetometer is on the same board, but is treated separately. Its calibration and use as a heading reference are talked about later.

3.4 Path representation

Turning a list of waypoints into a smooth drivable curve has been studied since the early days of autonomous-vehicle research, and several families of curve are in common use.

Approximating curves. B-splines and Bézier curves take the given points as control points rather than knots, so the final curve passes near them but not through them [50]. The smoothness is excellent and the curvature is easy to bound but the penalty is that the resulting path does not pass through the points that were specified.

Interpolating curves. Cubic splines, Hermite splines, and Catmull-Rom splines all pass through every waypoint on the curve [16], [51]. Cubic and Hermite need extra tangent information at each knot but the Catmull-Rom derives its tangents from neighbouring waypoints automatically, so the only input is the list of points. All three give C^1 continuity at segment boundaries.

Clothoids and arc-line composites. Clothoid (Euler) spirals give linear curvature growth and are the textbook answer for road design, where steering rate is the comfort metric [52]. Dubins and Reeds-Shepp paths compose straight segments and circular arcs with fixed turning radius, which is the standard formulation for car-like vehicles with steering limits [53], [54].

Polynomial interpolation. A single polynomial through all n points is mathematically the simplest method on interpolating, but Runge’s phenomenon makes it oscillate wildly between knots for $n \geq 6$ [55]. Piecewise splines exist specifically to fix this issue.

The robot used here is small and slow, and the waypoints come from exact positions the robot should visit. That makes interpolating curves the natural family to look at, not approximating ones: passing through the specified points is the expected behaviour.

3.5 Path-tracking controllers

Given a smooth curve and a pose, a geometric or model-based controller is needed to follow the curve. The literature on differential drive and car like path tracking offers a small set of standard answers.

Pure Pursuit. Coulter’s geometric tracker picks a point a fixed arc distance ahead on the path then builds a circular arc through the robot pose and that point, and commands its curvature [17]. It needs no dynamical model and only one tuning parameter (the lookahead distance), which makes it the default starting point for slow indoor robots.

Stanley controller. Developed for the Stanford DARPA Grand Challenge entry, the Stanley controller combines a cross-track error term and a heading error term and is sensitive to high-speed lateral correction [56]. It works best for car-like vehicles at road speeds.

Model Predictive Control (MPC). MPC formulates the tracking task as a constrained optimisation over a finite horizon and solves it at each tick [57], [58]. It handles actuator limits and predictive horizons naturally, at the cost of needing an explicit kinematic or dynamic model and the compute budget to solve the optimisation in real time.

Linear Quadratic Regulator (LQR). LQR linearises the system around the reference path and minimises a quadratic cost on state error and control effort [59]. It is closed-form once the cost weights are chosen, but the linearisation assumes small deviations and a known model.

Vector pursuit and follow-the-carrot. These are simpler geometric trackers that aim the robot at a moving reference point on the path without the circular-arc geometry of Pure Pursuit [60]. They cut corners more aggressively at high curvature.

Pure Pursuit and follow-the-carrot are the cheapest. Stanley assumes a car-like front-steering geometry. MPC and LQR need an explicit platform model and a compute budget that does not always sit comfortably alongside a 300 Hz balance loop on an embedded board.

3.6 Indoor localization without external tracking or external reference

Indoor robot localization has been surveyed several times recently [1], [8], [15], [61]. The point those surveys keep making is that no single sensor wins. The candidate sensors split into a few families.

Proprioceptive (wheel odometry). Integrating wheel speeds gives an estimate of (x, y, θ) without any external infrastructure [62]. The foundational analysis of dead reckoning error on differential drive robots is Borenstein and Feng’s UMBmark [63], which separates systematic from non-systematic odometric error. The main weakness is that errors accumulate unboundedly with travelled distance.

Inertial sensing (IMU). Gyroscopes give angular rate, accelerometers give specific force [64]. Integrating gyroscope rate gives a yaw estimate that drifts slowly, but

does not depend on wheel slip. Double-integrating accelerometer signal for position is well-known to diverge quickly on small consumer-grade chips.

Magnetic heading. A three-axis magnetometer measures the local magnetic field and gives an absolute heading reference after calibration [65], [66]. The standard procedure is hard iron and soft iron correction using ellipsoid fitting. However, motor-current distortion and structural rebar in indoor environments can cause distortions[1].

Sensor fusion. Gyro or odometry can be fused with the magnetometer heading to combine the absolute reference of the magnetic direction with the short-term precision of the IMU. The fused estimate inherits the slow drift of the integrated terms but the absolute reference of the magnetic term can be used for correction, when they agree.

External-infrastructure approaches. UWB ranging, sonar ranging, LiDAR with a known map, and visual odometry with on-board cameras all need either fixed infrastructure (anchors, fingerprint maps) or computationally heavy processing [1], [8]. GNSS does not work indoors at all. These approaches are powerful but step outside the constraint of using only what is already on the robot.

The robot in this thesis has only what is on the GRiSP-2 board: wheel-speed feedback from the ESP32 co-processor, and the 9-DoF Pmod NAV. Anything past that needs new hardware. The options that fit within that constraint are the first four above.

Chapter 4

Objectives, Problem Statement, and Approach

4.1 Functional goal and platform constraint

The real goal was to get two robots to follow each other. A second robot periodically sends its position; this robot receives those positions as waypoints and must follow the resulting path, so the pair stays close without any shared planner.

During development there is no second robot. A Python UI on the laptop is used to simulate the second robot. It allows waypoints to be placed manually and streamed over the existing LoRa link, which is enough to check whether the path following logic works before a real sender is brought in the system.

The platform puts one hard constraint on the path shape. A self-balancing robot that pivots in place punishes its own balance loop; the earlier prototype from [9] showed this directly. The path through the waypoints must be a smooth curve, not a chain of straight segments with in place turns. Three things make “smooth” achievable: no kinks at segment boundaries, speed that drops on tight corners and rises on straight stretches, and a final approach that settles without oscillating around the target heading.

4.2 The four pieces

The new work is split into four pieces, layered above the inherited balance cascade and orchestrated from a new `main_loop` process on the GRiSP.

Planning is a laptop-side UI that shows a top-down view of the test grid, accepts and previews waypoints, and sends committed waypoints to the robot over LoRa. In this thesis the UI is the sender. In future the sender will be a second robot streaming its own position.

Path representation is a Catmull-Rom spline through the waypoint list, with phantom endpoints reflected across the first and last point so the curve enters and leaves with sensible tangents [16]. The detailed reasoning behind picking Catmull-Rom over the other interpolating and approximating families is in the path representation chapter.

Path tracking is a Pure Pursuit lookahead controller [17] whose output is a curvature command, with cruise speed modulated by that curvature, and a sub-state machine running **cruise**, **braking**, and **dwelling** at each waypoint. Pure Pursuit was picked from the candidate set in the background chapter; the per-chapter reasoning is in the Pure Pursuit chapter.

State estimation is whatever runs on the robot to keep track of where it is in real time. The selection of this piece happens after experimentation, so the part that addresses it lives in the experimentation part of the thesis.

The four pieces talk to each other at only three points. The laptop sends framed bytes and the wireless decoder reads them and updates the waypoint list. The spline emits a dense set of sub-points and the tracker reads them. The state estimator emits a single **#pose** record carrying (x, y, θ) in the launch frame, the spline rebuilder and the tracker both read it. No shared globals, no callbacks, no implicit ordering beyond the tick-level dataflow.

The line between inherited and new is sharp. The LilyGo firmware, the motor drivers, the I²C wire format, the balance cascade, and the velocity tracking loops are all kept as Goens and Ponsard delivered them [9]. Everything added is new Erlang modules on the GRiSP, layered above the cascade. From the LilyGo’s point of view, this added work looks like an operator issuing an unusually smooth and predictable stream of drive commands.

4.3 The localization gap

Of the four sub-problems, planning, path representation, and path tracking each have a small set of well-understood candidate techniques in the literature, presented earlier in the background chapter. The state-estimation sub-problem is the open one. The hardware gives three things to work with. The wheels report their own speeds, so (x, y, θ) can be integrated tick by tick from differential-drive kinematics [62]. The IMU has a z -axis gyroscope whose integration tracks changes in θ . The same IMU has a three-axis magnetometer that, once calibrated for hard and soft iron, gives an absolute heading relative to magnetic north [65], [66]. Each one comes with its own way of going wrong: wheel odometry drifts without bound, gyro integration drifts even faster because bias accumulates, and the magnetometer deflects sideways every time the motors draw current. Which of these to use, and how to combine them, could not be decided from the literature alone [1], [63]. The thesis treats it as an experimental question, with localization as the only variable and the other three sub-problems fixed.

4.4 Methodological rules

A handful of rules came out of the implementation along the way and were written down in a running log. They explain choices later in the document that would otherwise look arbitrary.

Don’t touch the cascade. Everything added lives above the inherited stability loop, in extra Erlang modules. The motor firmware, the half-precision wire format, and the

cascade itself stay exactly as Goens and Ponsard left them. That boundary holds for the whole document.

Closed loop in every tick. The tracker rereads the pose at every control tick and recomputes its curvature command from scratch. If the robot is bumped, the next tick corrects for it. If the joystick is pushed mid-trajectory, the joystick input adds to the controller output rather than overriding it.

One copy of the pose. The pose lives in the main control loop and nowhere else. Other modules ask for it; they do not keep their own copy. The same rule applies to the waypoint list, which lives only in the trajectory planner.

Simplest baseline first. For the localization investigation, the wheel based odometry only configuration was built, logged, and characterised before anything else was layered on top. The fused configurations are measured against this initial baseline, not against an idealised expectation.

Make it A/B-testable. Each sensor configuration is gated by a compile time flag. The same binary can run with odometry only, with odometry plus gyro, or with odometry plus magnetometer. Switching configurations does not change the controller or the planner, only the yaw value added to the logs.

If it fails, show it. The LEDs color combo on the GRiSP Board reported calibration failures(during the testing), lifecycle resets, and sensor anomalies as colours an operator can see across the room. No silent retries, no hidden warnings in logs.

Drift is measured from the starting point of trajectory. The robot's pose is referenced to its position at the last transition into **running**, not to the geomagnetic frame. There is no declination correction, no global frame to maintain, and the magnetometer experiment later in the thesis is judged on how well its value reflects the actual relative drift, not on whether it points to true north.

Part II

Design and Implementation

Chapter 5

System Architecture

5.1 Erlang module map

My additions on the GRiSP are a handful of Erlang modules sitting on top of the inherited `stability_engine` and `pid_controller`. Figure 5.1 shows how they connect.

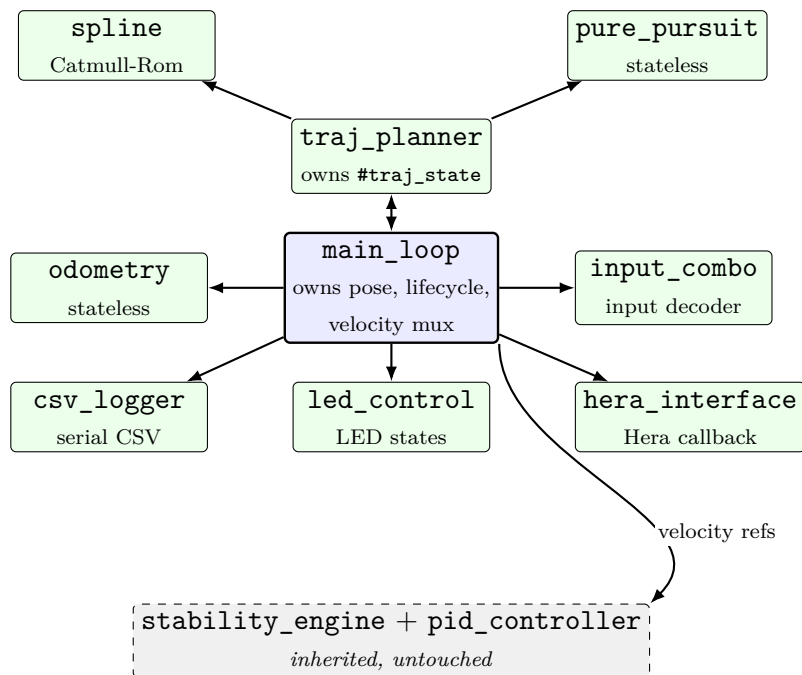


Figure 5.1: Modules added to the GRiSP. Filled boxes are added work. The dashed box at the bottom is the inherited balance cascade, which is not modified. `main_loop` is the only module with mutable state worth talking about (it owns the pose, the lifecycle atom, and the velocity mux). `traj_planner` owns its own waypoint and sub-state record but nothing else. Everything else is either stateless math (`odometry`, `spline`, `pure_pursuit`, `input_combo`) or a small wrapper around an output (`led_control`, `csv_logger`, `hera_interface`).

The list is short on purpose. `main_loop` is the orchestrator. `traj_planner` runs the path-following sub-state machine and owns the waypoint queue. `spline` builds the

Catmull-Rom curve, `pure_pursuit` picks the lookahead point and computes curvature, and both are pure functions. `odometry` integrates wheel speeds into a new pose every tick and is also pure. `input_combo` reads two control-word bits that normally carry forward and backward manual control, but when pressed together are decoded as tap and hold edges for lifecycle transitions. `led_control` maps the current state to LED colours. `csv_logger` emits a CSV line on the serial port every tenth tick. `hera_interface` is the bridge that lets Hera schedule the main loop as a measurement process.

5.2 Tick anatomy

The whole controller is one big loop. The rate is set by `main_loop:modify_frequency/1` with a target of 300 Hz, which gives each tick roughly 3.3 ms to do all of its work before the rate limiter at the bottom pulls it back to schedule. The measured rate on the GRiSP under typical load is closer to 180 Hz, so the per-tick budget in practice is about 5.6 ms. With the now-retired magnetometer read in the per-tick SPI burst the measured rate dropped a further 51 Hz to about 129 Hz, and that drop is one of the costs that contributed to the retirement decision discussed later. The order of operations inside one tick is the same every time, regardless of the achieved rate. Figure 5.2 sketches the timeline; the same flow is described below.

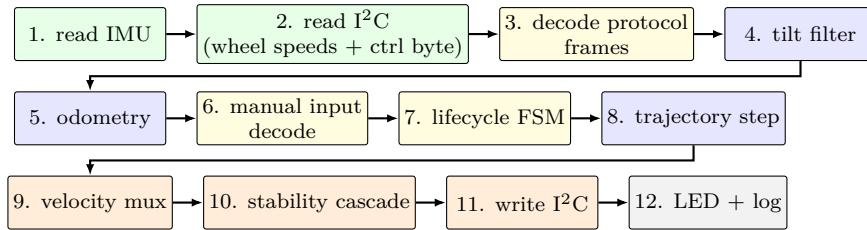


Figure 5.2: One tick of the main loop, reading left-to-right and top-to-bottom. Green stages are sensor reads. Yellow stages are FSM and protocol logic that only run when needed. Blue stages are estimation. Orange stages produce the actuator output. The gray stage at the end handles indicators and logs.

One line per stage:

1. **Read IMU.** Sample accelerometer and gyroscope over SPI.
2. **Read I²C.** Fetch the two wheel speeds and the one-byte control word from the LilyGo.
3. **Decode protocol frames.** If the protocol bit is set, route the byte to the wireless waypoint decoder; otherwise treat it as manual input.
4. **Tilt filter.** Update the pitch estimate from the IMU sample.
5. **Odometry.** Integrate the wheel speeds into a new pose.
6. **Manual input decode.** Extract tap and hold edges from the manual bits.
7. **Lifecycle FSM.** Advance the `idle/running/paused` atom.

8. **Trajectory step.** If the lifecycle is running, `traj_planner` computes the next forward and turn velocity from the spline and the current pose.
9. **Velocity mux.** Merge the trajectory output with whatever joystick command is active.
10. **Stability cascade.** Turn the velocity goal into a wheel-speed setpoint.
11. **Write I²C.** Send the setpoints back to the LilyGo.
12. **LED + log.** Update the LED and, every tenth tick, emit a CSV line on the serial port.

Everything in this loop is single-threaded. There is no message-passing inside the hot path, no `gen_server` calls per tick, no shared mutable state read by anyone else. The state of the system is threaded through as arguments and a recursive tail call closes the loop.

5.3 Ownership rules

main_loop owns the pose, the lifecycle, and the velocity mux. The single `#pose` record lives in the loop's arguments. Nobody else keeps a copy. If a downstream module needs the pose, it gets it as an argument; it does not query a process.

traj_planner owns the waypoint queue and the sub-state. The `#traj_state` record carries the path, the waypoints not yet visited, the waypoints already passed, the current sub-state (`cruise`, `braking`, `dwelling`, `final_align` (used only during testing)), and a couple of counters. No other module pattern-matches on the inside of this record; the public API of `traj_planner` is the only way in.

odometry is stateless. It is a single function that take the current pose, the two wheel speeds, and Δt ; return the new pose. It runs every tick whether or not the trajectory layer is active, because the pose has to keep tracking reality during pause and idle too.

Lifecycle is an atom, not a process. The lifecycle (`idle`, `running`, `paused`) is one atom inside `main_loop`. It is not a separate Erlang process and it does not have its own mailbox. Transitions are explicit function calls.

Events at the edge, state machines where the events arrive. Button events live in `main_loop`, because that is where the I²C byte arrives. Waypoint-frame events live in `main_loop` too, for the same reason; the actual mutation of the waypoint queue is then delegated to `traj_planner`.

5.4 The velocity mux

The mux is the small piece of code that decides what velocity reference goes into the stability cascade on each tick. It has two inputs and one output. The first input is the pair (`advtraj`, `turntraj`) from `traj_planner`, the forward and turn speeds the path-follower wants. The second input is the pair (`advman`, `turnman`) from the joystick buttons. The rule is:

$$(\text{adv}_{\text{goal}}, \text{turn}_{\text{goal}}) = \begin{cases} (\text{adv}_{\text{traj}} + \text{adv}_{\text{man}}, \text{turn}_{\text{traj}} + \text{turn}_{\text{man}}) & \text{if lifecycle is } \mathbf{running} \\ (\text{adv}_{\text{man}}, \text{turn}_{\text{man}}) & \text{otherwise.} \end{cases}$$

Additive, not exclusive. If the trajectory is cruising and the manual input is received, the input adds to the cruise rather than overriding it and the closed loop reacts on the next tick. Outside **running**, the trajectory contribution is zero and only the manual input is in charge. A hard-switching mux would block whichever one of the source and not very useful if the purpose is manual correction of the trajectory.

5.5 Logging

`csv_logger` emits one CSV line every tenth tick (about 18 Hz at the measured loop rate) on the serial port, prefixed with `TLOG,`. A small Python script on the laptop captures the stream and plots it live. The column order is t_{ms} , lifecycle, sub-state, x , y , speed, adv_v , turn_v , distance to next waypoint, waypoints remaining, and wheel-odometry yaw. The last column is the heading from `#pose`; the experimentation part explains why it ended up being the only yaw column.

Chapter 6

Path Representation: Catmull-Rom Splines

This chapter is about how a list of waypoints is turned into a smooth, drivable curve. The curve has to pass through every waypoint that was added, be smooth enough that the balance loop is not punished by it, and be cheap to rebuild every time the waypoint list is mutated mid-cruise.

6.1 Why Catmull-Rom over Bezier / cubic spline

An *interpolating* curve passes through every control point. An *approximating* curve uses them as hints. A waypoint is added with the expectation that the robot goes through it, not near it, so the choice was immediate: an interpolating curve.

That eliminated cubic Bézier, which only passes through its first and last control points. Among other interpolating options, plain cubic splines are globally coupled (changing one waypoint warps the curve everywhere). Only Catmull-Rom is *locally controlled* which means a waypoint only affects the two segments on either side. That property matters because a waypoint can be cancelled or moved while the robot is cruising, a globally coupled spline would re-warp every previously visited point. Figure 6.1 shows the three curves through the same waypoints.

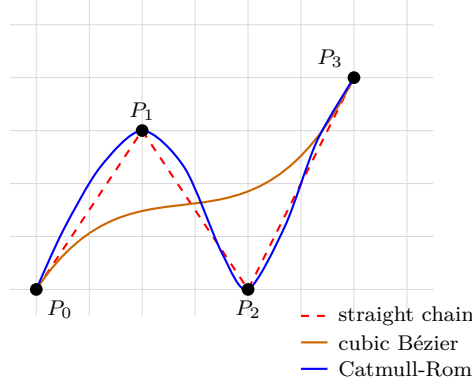


Figure 6.1: Three ways to draw a path through four waypoints. The dashed red chain is the earlier point-to-point prototype. The orange Bézier misses the middle waypoints. The blue Catmull-Rom passes through every waypoint.

6.2 Mathematical formulation

A Catmull-Rom segment is defined by four control points P_0, P_1, P_2, P_3 and a parameter $t \in [0, 1]$. The curve runs from P_1 to P_2 as t goes from 0 to 1; the two outer points only influence the tangent at each end. The polynomial form, which `spline.erl` evaluates directly, is

$$P(t) = \frac{1}{2} \left[2P_1 + (-P_0 + P_2)t + (2P_0 - 5P_1 + 4P_2 - P_3)t^2 + (-P_0 + 3P_1 - 3P_2 + P_3)t^3 \right]. \quad (6.1)$$

Plugging in $t = 0$ gives $P(0) = P_1$ and $t = 1$ gives $P(1) = P_2$, so the segment passes through the inner two points. The derivative at the endpoints is $P'(0) = \frac{1}{2}(P_2 - P_0)$ and $P'(1) = \frac{1}{2}(P_3 - P_1)$: each waypoint's tangent is the half-difference of its two neighbours, which is what gives the curve C^1 continuity at segment joins.

6.3 Computing the curve

The input is an ordered list of waypoints $\{P_0, P_1, \dots, P_n\}$ in centimetres, expressed in the launch-pose frame. The first and last waypoints have only one real neighbour, so neither has a defined tangent. The trick is to invent a phantom point at each end by reflecting the second-from-the-end point across the actual endpoint:

$$P_{-1} = 2P_0 - P_1, \quad (6.2)$$

$$P_{n+1} = 2P_n - P_{n-1}. \quad (6.3)$$

The phantom sits the same distance on the far side of the endpoint as its real neighbour sits on the near side. The tangent at P_0 then becomes $P_1 - P_0$, meaning the curve enters P_0 pointing straight at P_1 . That is what a human draws and it avoids the kinks that other endpoint conventions (zero tangent, doubled point) introduce.

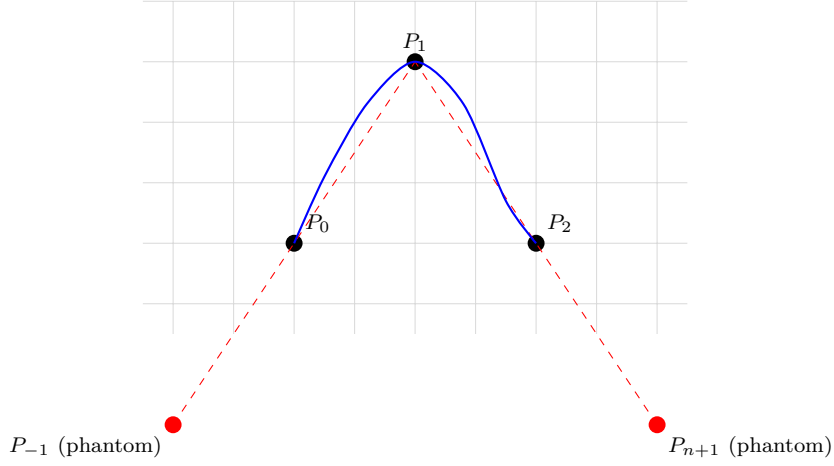


Figure 6.2: Phantom-point construction: the phantom (red) sits opposite the real neighbour, with the endpoint as midpoint.

With phantoms in place, the tangent at any waypoint P_i is:

$$\mathbf{m}_i = \frac{1}{2} (P_{i+1} - P_{i-1}). \quad (6.4)$$

This is implicit in equation (6.1), the polynomial form folds it in. For each interior segment i the window $(P_{i-1}, P_i, P_{i+1}, P_{i+2})$ is fed to the equation (6.1) and sampled at 30 sub-points per segment, uniformly in t at $t = 0/30, \dots, 29/30$. The endpoint $t = 1$ is dropped to avoid duplicates at segment joins. With waypoints around 50 cm apart, this puts one sub-point every 1.5–2 cm, finer than the 12 cm lookahead distance and coarse enough that walking the sub-point list per tick stays cheap.

6.4 Mid-cruise seed-based rebuild

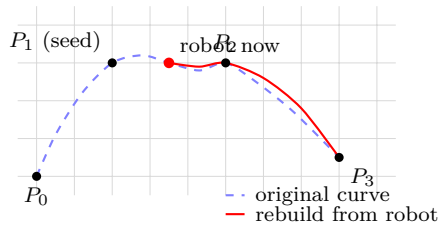


Figure 6.3: Mid-cruise rebuild. The robot (red marker) was cruising on the dashed blue spline; a waypoint is cancelled and the new red spline is built using P_1 as the seed phantom. The tangent through P_2 matches, so the lookahead target does not step.

When the waypoint list is mutated mid-cruise (`append`, `cancel-last`, `cancel-n`, `clear-all`), the new spline must start at the robot’s current pose, not at the launch point. The continuing-rebuild variant takes a *seed* point together with the remaining waypoints, prepends the seed as a real control point, and reflects P_1 around the seed to synthesise the leading phantom,

$$P_{-1} = 2 \cdot \text{Seed} - P_1. \quad (6.5)$$

The first sampled segment then runs from the seed to P_1 , so the path actually begins where the robot stands. The tangent at P_1 stays $\frac{1}{2}(P_1 - \text{Seed})$, identical to the direction the robot was already heading, so the splice is invisible and the lookahead point does not step. The seed is taken as the second-from-last visited waypoint mid-cruise (so tangent continuity is preserved through a point the robot has demonstrably reached), and as the current pose itself on a cold start when no waypoint has yet been visited. The same construction is therefore invoked on every run start as well, not only on mid-cruise mutations. The reason for that is recorded in Section 7.6.

Chapter 7

Pure Pursuit Controller

The previous chapter turned a list of waypoints into a smooth curve. This chapter is about how the robot actually drives along that curve. The controller is Pure Pursuit, written by R. Craig Coulter at the CMU Robotics Institute in 1992 [17], and it has been the default geometric path-follower on small mobile robots for thirty years for one reason: it is simple. The whole implementation of the controller is in `pure_pursuit.erl`. The rest of the chapter explains how those three functions work, why the lookahead distance came out to 12 cm (it can be adjusted to requirements), how the controller is kept from chasing points that are already behind the robot, and the small additional rule that slows the robot down on tight corners.

7.1 Geometric formulation

Pure Pursuit treats path following as a geometry problem. The robot has a pose (x, y, θ) in the launch frame and a path consisting of a list of (x, y) sub-points. The controller picks one of those sub-points, the *lookahead point*, and draws a circular arc that starts at the robot, is tangent to the robot's current heading, and ends at the lookahead point. The curvature of that arc is the steering command. Then the next tick the controller picks a new lookahead point and does it again.

The arc is uniquely determined by three constraints: it passes through the robot, it passes through the lookahead point, and it is tangent to the heading at the robot. To compute its curvature, rotate the world into the robot's frame so the robot sits at the origin with its heading along the positive x -axis. If the lookahead point is at $(\Delta x, \Delta y)$ in the world frame, then in the robot frame it sits at

$$\ell_x = \Delta x \cos \theta + \Delta y \sin \theta, \quad \ell_y = -\Delta x \sin \theta + \Delta y \cos \theta. \quad (7.1)$$

An arc through the origin that is tangent to the x -axis at the origin has its centre on the y -axis at some height $r = 1/\kappa$ above (or below) the robot. For that arc to also pass through (ℓ_x, ℓ_y) , the centre has to be equidistant from both the origin and the lookahead point. A bit of algebra (or one diagram) gives

$$\boxed{\kappa = \frac{2\ell_y}{L^2}}, \quad L^2 = \ell_x^2 + \ell_y^2. \quad (7.2)$$

This is the only formula Pure Pursuit really needs. Positive κ is a left turn, negative is a right turn. The lateral offset ℓ_y measures how far off-axis the lookahead is, and dividing by L^2 normalises that offset to a curvature (with units of cm^{-1}). Figure 7.1 shows the geometry.

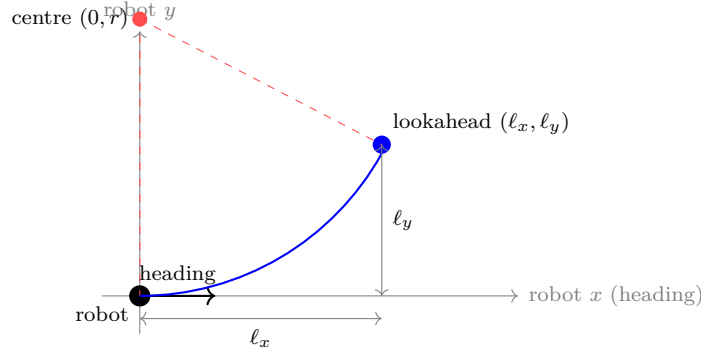


Figure 7.1: Pure Pursuit geometry in the robot frame. The robot sits at the origin with its heading along the x -axis. The lookahead point is at (ℓ_x, ℓ_y) . A circular arc through the origin, tangent to the heading, and through the lookahead point has its centre at $(0, r)$ on the y -axis. The curvature of that arc is $\kappa = 1/r = 2\ell_y/L^2$.

From curvature to wheel speeds. The curvature κ tells the robot how tightly to turn, but the inner balance loop wants velocity setpoints, not curvatures. For a differential drive robot moving at forward speed v on an arc of curvature κ , the angular rate is $\omega = \kappa v$ and the differential between left and right wheel speeds is $\kappa v b$, where b is the wheelbase. The stability engine takes a single *turn velocity* reference v_{turn} that is added on one side and subtracted on the other, so v_{turn} is half the differential:

$$v_{\text{turn}} = \kappa \cdot v \cdot \frac{b}{2}. \quad (7.3)$$

With $v = 14 \text{ cm/s}$ and $b = 18.5 \text{ cm}$, the constant of proportionality is $vb/2 = 129.5 \text{ cm/s} \cdot \text{cm}$. A curvature of $\kappa = 0.1 \text{ cm}^{-1}$ (10 cm turning radius) gives $v_{\text{turn}} = 12.95 \text{ cm/s}$ of differential per side. The clamp to `MAX_TURN_V` = 18 cm/s catches the cases where the lookahead point is unusually close and κ blows up.

7.2 Lookahead distance choice

The lookahead distance is the one parameter Pure Pursuit really has. If you pick it too small, the controller looks only at the next centimetre or two of path, oscillates around the path centreline and is generally jittery. If you pick it too big, the controller stops noticing curvature changes until the robot has already passed them, so corners get cut. The sweet spot depends on the platform's responsiveness, its forward speed, and the wheelbase.

The number that actually matters is not the lookahead distance on its own but the *preview time* it gives the controller, $\tau_{\text{preview}} = \text{LOOKAHEAD}/\text{MAX_SPEED}$. A healthy preview time on a balancing platform sits between 0.5 s and 1.5 s: shorter than 0.5 s and the balance loop is fighting curvature changes it has barely seen; longer than 1.5 s and the lookahead reaches past the corner the robot is still inside, and the controller starts cutting it. At $\text{MAX_SPEED} = 14 \text{ cm/s}$, that band translates to a lookahead between 7 cm and 21 cm. $\text{LOOKAHEAD} = 12 \text{ cm}$ puts the preview time at $12/14 \approx 0.86 \text{ s}$, comfortably in the middle of the band and close to the time-to-stop $\text{MAX_SPEED}/\text{BRAKE_RATE} \approx 0.93 \text{ s}$, which is the rule-of-thumb target for matching preview to deceleration. The whole list of ratios behind these choices is in Section 7.5.

Why 12 cm and not the wheelbase. A common rule of thumb is to set the lookahead to roughly one wheelbase. On this platform the wheelbase is 18.5 cm, which would give a longer lookahead. It was tried. The robot started cutting the inside of every corner. The reason is that at 14 cm/s, a 18.5-cm lookahead corresponds to roughly 1.3 seconds of forward motion, and the paths with tighter corners than that can be problematic. The lookahead has to be short enough that the curvature change at a corner is visible inside it. The published rule of thumb assumes paths with curvature radii much larger than the wheelbase, which is not the case here.

Why the lookahead does not have to be continuously variable. A more elaborate version of Pure Pursuit scales the lookahead with the current speed, on the grounds that a faster robot needs more preview. Since the robot’s nominal cruise speed is fixed at 14 cm/s and the speed modulation in Section 7.4 only ever slows the robot down (never speeds it up), a constant lookahead works without paying for the extra complexity. The corner cases where the speed-scaled version would help, fast straight stretches where a longer lookahead would catch upcoming corners earlier, do not exist in this operating envelope.

7.3 Path-advance: pruning behind the robot

The spline produces a list of sub-points that starts at the first waypoint and ends at the last. As the robot moves forward along the path, the sub-points it has already passed are no longer relevant. Worse, if they are kept in the list, the lookahead search starts from the beginning every tick and will find old sub-points first, points that are behind the robot, and the controller will dutifully try to turn around and drive to them. `pure_pursuit:advance/4` prevents that.

The function walks the front of the path list and drops every point whose x -coordinate *in the robot frame* is negative. “In the robot frame” is the important part. A point may be behind the robot in the world frame after the robot has turned, but if the robot is now heading away from it, that point is also behind the robot in the robot frame, and that is the criterion that matters. The transformation is the same rotation from equation (7.1):

$$\ell_x = (P_x - x) \cos \theta + (P_y - y) \sin \theta. \quad (7.4)$$

If $\ell_x < 0$, the point is behind the robot. `advance/4` drops every leading point that satisfies this and returns the rest. Once a point survives the check, all the remaining points are kept, `advance/4` does not iterate past the first surviving point, so a path that loops back near the robot is not pruned in the middle. That matches the operating assumption that the spline is monotonically progressing along the path, which Catmull-Rom guarantees as long as the waypoints are arranged in a sensible order.

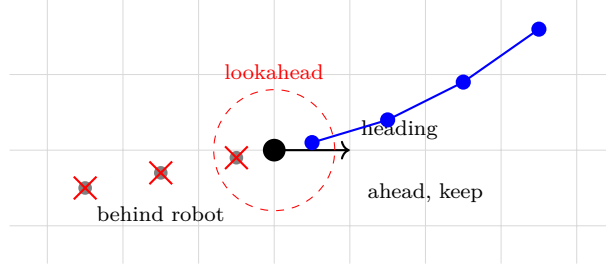


Figure 7.2: Path-advance pruning. Sub-points behind the robot (red crosses) are dropped from the head of the path list because their ℓ_x in the robot frame is negative. The surviving sub-points (blue) are what `find_lookahead/4` then walks through to pick the first one at least 12 cm from the robot.

After `advance/4` prunes the path, `find_lookahead/4` walks forward through the survivors and returns the first one whose Euclidean distance from the robot is at least 12 cm. If the robot is within 12 cm of every remaining point (which happens near the last waypoint), the function returns the very last point of the path. That fallback is what lets the controller continue commanding a sensible direction even when the lookahead can no longer be satisfied, the controller aims at the goal, and the sub-state machine described next takes over for the actual stopping logic.

7.4 Curvature-modulated cruise speed

If the cruise speed stayed at 14 cm/s through every corner, the robot would overshoot tight turns because Pure Pursuit cannot bend the trajectory tighter than what the lookahead and curvature geometry allows. The fix is small but important: the forward speed is slowed whenever the commanded curvature is high.

The actual rule, in the trajectory planner, is

$$v_{\text{forward}} = v_{\text{max}} \cdot \text{clamp}\left(1 - |\kappa| \cdot b, \frac{v_{\text{corner}}}{v_{\text{max}}}, 1\right), \quad (7.5)$$

where $v_{\text{max}} = 14$ cm/s, $b = 18.5$ cm is the wheelbase, and $v_{\text{corner}} = 2.5$ cm/s is the floor that keeps the robot from coming to a complete stop on a very tight corner. Geometrically, $|\kappa| \cdot b$ is the ratio of wheelbase to turning radius. When the turning radius is much larger than the wheelbase, the factor $1 - |\kappa| \cdot b$ is close to 1 and the robot cruises at full speed. When the turning radius approaches the wheelbase, the factor heads toward zero and the robot crawls. The clamp at $v_{\text{corner}}/v_{\text{max}} \approx 0.18$ keeps it from stalling out.

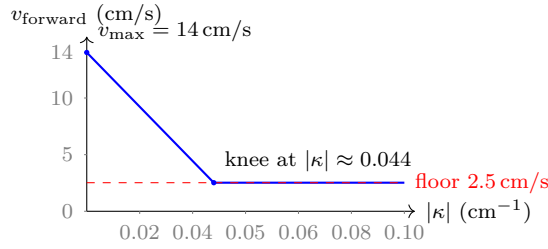


Figure 7.3: Forward speed as a function of $|\kappa|$. On a straight path ($\kappa = 0$), the robot cruises at $v_{\max} = 14$ cm/s. As curvature increases, speed drops linearly with $|\kappa| \cdot b$, hits the floor of 2.5 cm/s at $|\kappa| \approx 0.044$ cm $^{-1}$, and stays there. The floor exists so the robot does not stall in the middle of a tight corner.

Why the wheelbase shows up in the speed formula. It is not a coincidence that the same number (b) governs both the differential-to-curvature mapping in equation (7.3) and the speed reduction in equation (7.5). The wheelbase is the natural length scale of the robot: turns that are larger than the wheelbase are easy, turns that are smaller than the wheelbase are mechanically near-impossible for a balancing chassis. Tying the speed reduction to $|\kappa| \cdot b$ makes the speed roll off exactly when the turn starts to push the chassis beyond what it can do.

Why the floor matters. In the very first prototype, before the floor was added, the robot would slow down on a tight corner, lose enough forward velocity that the balance loop started compensating with an unwelcome backward lean, and eventually stop completely halfway through the turn. The floor of 2.5 cm/s is slow enough to be safe but fast enough that the balance loop keeps the chassis upright. The exact floor was tuned by running figure-of-eight trajectories with increasingly tight inner radii until the robot stopped falling over. A more rigorous derivation could probably be done from the linearised balance dynamics, but the empirical value worked and was left alone.

7.5 Tuning ratios

The Pure Pursuit constants do not stand on their own. Each one is sized in proportion to `MAX_SPEED`, the wheelbase, or one of the balance-cascade ramp rates, and the ratios that come out of those proportions are what keep the controller in a usable envelope. Table 7.1 lists the four that matter.

Ratio	Formula	Healthy	Current
Preview time	LOOKAHEAD/MAX_SPEED	0.5–1.5 s	0.86 s
Corner-speed floor	MIN_CORNER_SPEED/MAX_SPEED	0.15–0.25	0.18
Time to cruise	MAX_SPEED/ACCEL_RATE	0.5–1.5 s	1.17 s
Time to stop	MAX_SPEED/BRAKE_RATE	0.5–1.5 s	0.93 s

Table 7.1: The four ratios that constrain the Pure Pursuit and trajectory constants. Each one names a limitation: preview time is the controller’s reaction window, the corner-speed floor is how slowly the balance loop can keep cruising, and the cruise and brake times are the durations the tilt PID has to absorb a forward or backward lean during acceleration or deceleration. Crossing the lower end of any band makes the controller jittery or stalls the robot; crossing the upper end makes it cut corners or tip on a hard start.

The point of the table is the design rule it implies. If `MAX_SPEED` is changed (say, to test how the same paths look at a higher cruise rate), the other constants do not move independently: `LOOKAHEAD`, `MIN_CORNER_SPEED`, `ACCEL_RATE`, and `BRAKE_RATE` all rescale to keep their respective ratios inside the healthy band. The values used throughout the rest of the thesis (`MAX_SPEED` = 14, `LOOKAHEAD` = 12, `MIN_CORNER_SPEED` = 2.5, `ACCEL_RATE` = 12, `BRAKE_RATE` = 15) were picked together, not one at a time, because the ratios force them to move together.

7.6 Problems faced

Two pathologies were observed during integration testing, both rooted in geometric assumptions that Pure Pursuit makes silently and that the present platform happened to violate. Both have since been addressed; this section is reserved for the bugs themselves and the reasoning that led to the patches, since the patches are described at the points where they belong (Section 6.4 for the first, the paragraph below for the second).

The first-waypoint miss. On a fresh run from the launch pose $(0, 0, \theta = 0)$, the robot was repeatedly seen to sail past P_1 on a wide outside arc and to track the rest of the path from the side. The cause was a mismatch between the robot’s actual position and the start of the sampled path. With the original phantom construction (Section 6.3) the first sampled segment spans $P_1 \rightarrow P_2$, so the path itself did not begin at the robot; it began several centimetres away at P_1 . The pursuit controller was therefore asked to steer toward a lookahead sub-point sitting in open space, and the curvature returned was the one needed to reach that point on a constant-radius arc, not the tighter one that would have been needed to both reach P_1 and end up tangent to the onward path. Under-turning was the natural consequence: the resulting wide arc never re-entered the 4 cm close-enough radius, so P_1 was never declared visited and the run continued tracking the spline from the side. This is a documented limitation of pure-pursuit-style controllers at the path entry, when the platform pose is not co-located with the path origin [67]. The fix is the seed-based construction described in Section 6.4, which is now invoked at every run start as well, with the seed taken from the current pose, so that the path is anchored to where the robot actually stands and the entry tangent points

straight at P_1 .

Spinning around the final waypoint. On approach to the last waypoint, the robot was observed to enter a tight rotation around it rather than coming to a clean stop. The mechanism is the lookahead-search fallback noted in Section 7.3. Once the robot crosses inside the lookahead radius of the last point, no remaining sub-point lies 12 cm away, so the fallback returns the last point itself. The lateral offset ℓ_y in the robot frame then ceases to shrink with distance, while L^2 continues to shrink, and the commanded curvature $\kappa = 2\ell_y/L^2$ grows without bound. The turn velocity saturates at its cap, and the robot pivots in place around a point it can never reach by chasing it. This end-of-path failure mode is well-known in the pure-pursuit literature [17], [67]. The fix is a guard placed at the planner level around the call into Pure Pursuit: when the distance to the final waypoint is below the lookahead radius, the pursuit call is bypassed entirely, the turn-velocity reference is forced to zero, and the forward speed is tapered linearly toward the corner-speed floor. The braking trigger of Section 7.4 is left to take over inside the same radius, so the clean two-phase *coast-then-brake* approach to the final waypoint is recovered.

7.7 Putting the three pieces together

The three pieces, `advance/4`, `find_lookahead/4`, and the curvature-plus-speed pair, together make a complete path-following controller. The cruise sub-state of the trajectory state machine described next calls all three in order on every tick when the robot is cruising, except inside the final-approach guard described above, where the pursuit call is skipped. The other sub-states (braking, dwelling, final-align) override the speed and turn commands directly and bypass Pure Pursuit entirely, because once the robot is decelerating into a waypoint, the path-following geometry is no longer the right model.

Chapter 8

Trajectory Sub-state Machine

Pure Pursuit, on its own, is a cruise controller. It does not know when to slow down for a corner that ends in a waypoint, it does not know how to stop at the goal, and it has no concept of “arrived”. Those rules live one layer up, in a small state machine inside `traj_planner.erl`. The state machine has three states and is reset every time the lifecycle enters `running`.

8.1 The three sub-states

The trajectory state machine has three states: `cruise`, `braking`, and `dwelling`. The whole life of a run from `idle` to the final stopped robot passes through them in that order, with an optional intermediate-waypoint pass-through described below. Figure 8.1 shows the transitions.

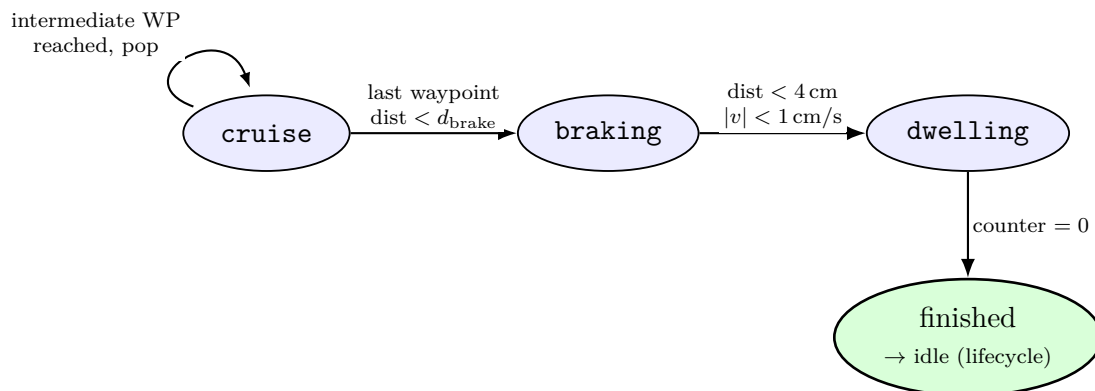


Figure 8.1: The three sub-states of `traj_planner` and the transitions between them. The robot starts in `cruise` and stays there until the distance to the final waypoint drops below the speed-dependent brake distance. `braking` commands zero forward and zero turn and waits for the chassis to actually stop. `dwelling` sits at the waypoint for 600 ticks (about 3.3s at the measured loop rate) and then signals *finished* to the main loop, which transitions the lifecycle back to `idle`. Intermediate waypoints are popped silently from inside `cruise` without a state change.

cruise. This is the working state. On every tick it calls the three Pure Pursuit helpers

from the previous chapter in order (`advance`, `find_lookahead`, `curvature`), computes the forward and turn velocity setpoints from the curvature based speed rule, and emits them into the velocity mux. At each tick, intermediate waypoints are handled without a state change such that if the current waypoint is not the last and the robot is within `WAYPOINT_REACHED = 4 cm` of it, the waypoint is popped from the queue. The next waypoint becomes the active target such that the state remains `cruise` and the tick recurses to evaluate against the new waypoint. The only exit from `cruise` is the brake trigger, which applies only to the last waypoint and is described in the next section.

braking. The braking state commands zero forward velocity and zero turn velocity. It does not actively decelerate the robot with reverse torque, the stability cascade’s velocity-ramp limiter (Section 2.3) handles the deceleration ramp at 15 cm/s^2 for forward and 25 cm/s^2 for turn. The robot coasts to a stop. `braking` stays put until two conditions are both true: the robot is within `WAYPOINT_REACHED = 4 cm` of the waypoint and its current speed is below `SETTLE_VEL = 1 cm/s`. The dual check is important. Without the distance check, the robot would declare “stopped” anywhere along the brake ramp. Without the speed check, the robot might be inside the 4 cm ring but still drifting through it.

dwelling. Once both conditions are met, the machine enters `dwelling` with a counter set to `DWELL_TICKS = 600`. Each tick decrements the counter. While the counter is positive, the state emits zero forward and zero turn velocities and the robot holds its position. When the counter reaches zero, `step/3` returns `finished = true`, the main loop reads that, prints the final pose, and transitions the lifecycle from `running` to `idle`. The 600 tick dwell is roughly 3.3 seconds at the measured loop rate. In normal operation where waypoints are received from an external source, a dwell state would not be necessary. It is retained here because the development setup uses a UI based simulator with a defined end of trajectory event and the pause provides a clear visual confirmation that the robot has stopped. The state also has a potential future use. In a real deployment, it could be used to hold position safely in the event of a communication failure before the trajectory is reset.

What about a fourth state for final alignment? During experimentation, a fourth state, `final_align`, was used to visually verify the yaw error at the end of the path by pivoting the robot to face 0° in the launch frame. It was removed after experimentation once it became clear that for the navigation tasks involved, the final heading was not a meaningful quantity; what mattered was that the robot stopped at the waypoint.

8.2 Entry and exit conditions

Among the entry and exit conditions of the sub-state machine, the brake trigger uses a physics-derived stopping distance rather than a fixed threshold. The brake distance is speed-adaptive:

$$d_{\text{brake}} = \frac{v^2}{2 a_{\text{brake}}} + d_{\text{margin}}, \quad (8.1)$$

where v is the current forward speed, $a_{\text{brake}} = 15 \text{ cm/s}^2$ is the velocity-ramp limit imposed by the stability cascade, and $d_{\text{margin}} = 2.5 \text{ cm}$ is a safety pad that absorbs the residual coast distance the linearised kinematic formula misses (motor lag, PID overshoot, wheel slip on floor). At cruise speed $v = 10 \text{ cm/s}$, equation (8.1) gives $d_{\text{brake}} = 100/30 + 2.5 \approx 5.8 \text{ cm}$. At standstill, it collapses to the margin alone, 2.5 cm . The state machine compares this against the straight-line distance to the last waypoint on every tick and triggers **braking** as soon as the distance drops below.

The margin term compensates for the systematic error the kinematic formula cannot capture. The formula alone is exact for a point mass decelerating at constant a . The robot routinely overshoot by $\sim 3 \text{ cm}$ because the stability cascade does not decelerate at exactly a_{brake} instead it ramps, and the ramp is slightly slower in the last few cm/s than in the first. Adding 2.5 cm of early trigger absorbs that lag without any further tuning of the cascade.

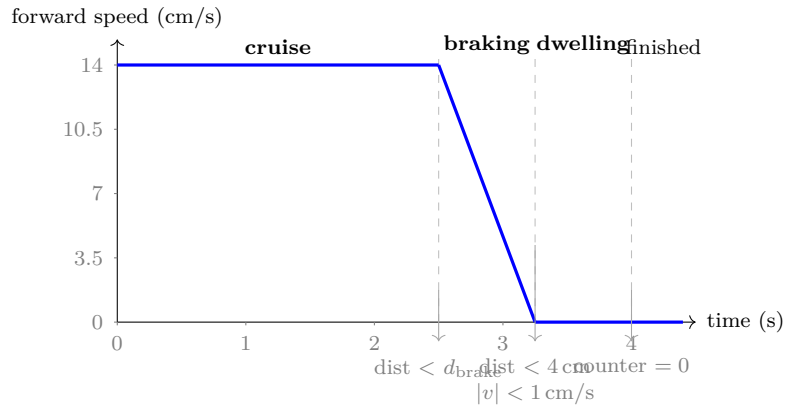


Figure 8.2: A typical stop sequence at the final waypoint, with the forward-speed trace and the sub-state boundaries marked. The robot cruises at 14 cm/s , hits the brake trigger about 9.0 cm before the waypoint, and the velocity ramp inside the stability cascade pulls the speed down to zero over roughly $14/15 \approx 0.93 \text{ s}$. The robot then dwells for 2 s and the machine signals finished.

8.3 Stop tuning history

The constants that drive the sub-state machine were not chosen arbitrarily. They came out of an iterative tuning loop in which the robot was run on a fixed waypoint sequence, the stopping behaviour observed, one or two values changed, and the run repeated. Table 8.1 records where each constant started, what intermediate values were tried, and where it landed. The story behind the numbers is in the decision log; the table is just the punchline.

Constant	Initial	Iterations	Final
BRAKE_DIST	10 cm	3, 6	formula (Eq. 8.1)
WAYPOINT_REACHED	4 cm	—	4 cm
SETTLE_VEL	0.5 cm/s	1.0	1.0 cm/s
DWELL_TICKS	600 (2 s)	100, 600	600
MAX_SPEED	5 cm/s	10	14 cm/s
MIN_CORNER_SPEED	2 cm/s	1	2.5 cm/s
MAX_TURN_V	5 cm/s	6, 12, 18	18 cm/s
LOOKAHEAD	20 cm	12, 8	12 cm
BRAKE_MARGIN	0 cm	1.5, 2.5	2.5 cm

Table 8.1: Tuning history of the trajectory-related constants. Some speed-related constants required more iterations: **MAX_TURN_V** started at 5 cm/s and was raised three times because the robot was taking more than ten seconds to complete a 90° turn, and **LOOKAHEAD** started at 20 cm, was reduced to 8 cm, and raised back to 12 cm when **MAX_SPEED** reached 14 cm/s. **BRAKE_DIST** became a formula rather than a constant, with the margin term replacing the static value.

MAX_TURN_V, from 5 to 18 cm/s. With a wheel speed difference of 5 cm/s, the robot rotates at about 15.5° per second. A 90° turn then takes around six seconds, and close to nine seconds once the start and stop ramps are added. The cap was raised to 18 cm/s over three iterations. At 18 cm/s the rotation rate becomes about 56° per second, so the same 90° turn now takes around 1.7 s plus ramp time. The forward speed is automatically reduced during sharp turns (Section 7.4), so the faster turn cap does not cause the robot to slide outward on corners.

LOOKAHEAD: 20 → 8 → 12 cm. At 20 cm the robot visibly tracked a different path, swinging 15 cm wide on corners. Reducing to 8 cm brought the path following into a visually recognisable shape. The later increase to 12 cm came when **MAX_SPEED** was raised to 14 cm/s. At 14 cm/s that speed an 8 cm lookahead gave only 570 ms of preview, which is too little for the cascade to ramp smoothly into corners. With 12 cm the preview is about 860 ms.

DWELL_TICKS: 600 → 100 → 600. This is purely arbitrarily selected, leaving time for the robot to come to a complete stop. It was used in testing path traversal without spline curved to judge the turning accuracy. It can be changed as now it is only effective at the end of trajectory.

BRAKE_MARGIN: 0 → 2.5 cm. Without the margin, the robot overshoot the waypoint by 3 cm on every stop. The value 2.5 cm was set to compensate it, which brought stops inside the 4 cm ring on the first attempt.

Chapter 9

Lifecycle FSM and User Interaction

The trajectory sub-state machine described earlier only runs when something tells it to. That something is a second, higher-level state machine that lives in `main_loop.erl` and decides whether the robot is allowed to drive at all. It has three states. It is driven by Start, Pause, and Reset commands received over the wire protocol — currently issued by a simulator, but the same interface can be used by another robot to command trajectory following or cancel it — by the robot’s tilt sensor (was it knocked over?), by a clear-all command, and by the trajectory layer reporting it is done.

9.1 The three lifecycle atoms and the single-state reset funnel `go_idle/3`

The lifecycle has three atoms: `idle`, `running`, and `paused`. The Erlang record `#rstate{}` carries the current atom in its `lifecycle` field and `lifecycle_step/4` in `main_loop.erl` is the only function that mutates it. Figure 9.1 shows the full machine.

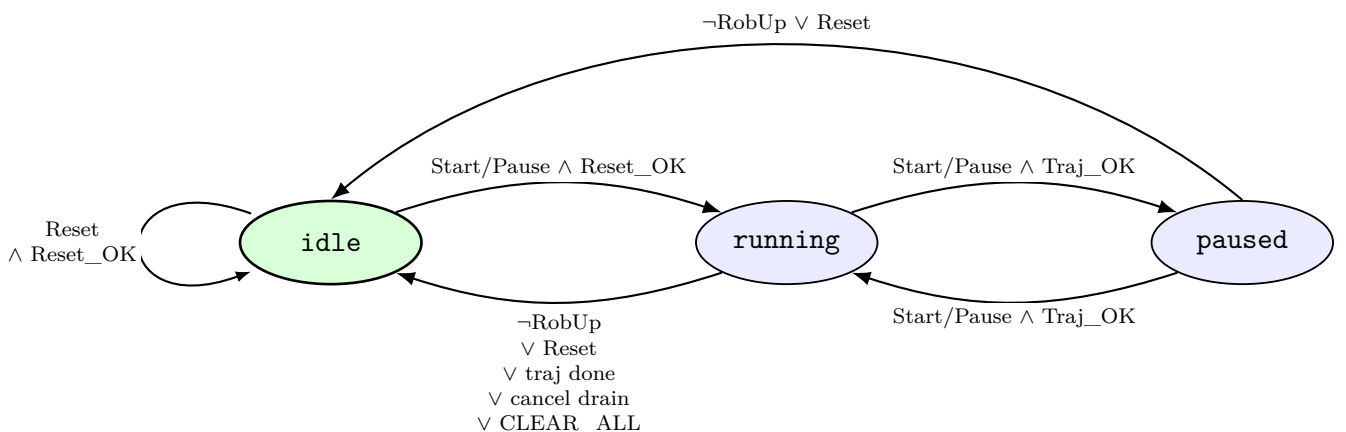


Figure 9.1: The lifecycle FSM. `idle` is the post-boot state and the single reset destination. From `idle`, a Start/Pause command arms the trajectory layer and the robot enters `running`. From `running`, a Start/Pause command toggles `paused`; a Reset command, a knock-over, a finished trajectory, a drained cancel, or a `CLEAR_ALL` frame all collapse to `idle`. `paused` reaches `idle` only via Reset or knock-over.

idle. The robot balances in place but ignores trajectory work. Waypoints that arrive are queued (Section 9.3); no spline is built, no Pure Pursuit runs, and the trajectory velocity refs into the velocity mux are zero.

running. The trajectory layer is live. Every tick passes through `traj_planner:step/3`, which returns forward and turn velocity refs that go into the additive velocity mux from the system architecture. The sub-state machine described earlier runs underneath.

paused. The trajectory state is held with the queue while the spline and the sub-state kept untouched, but the velocity refs from `traj_planner` are not pushed into the mux. The robot balances in place. A Start/Pause command resumes from exactly the same path point, without losing the run.

Five different events can take the machine back to `idle`: a Reset command (manual abort), a Fall, the trajectory layer reporting `finished = true`, a cancel that drains the last remaining waypoint, and a `CLEAR_ALL` frame. All five are routed through a single helper, `go_idle/3`: the waypoint queue is wiped, the integrated pose is reset to the origin, the lifecycle atom is set to `idle`, and the reset clock is stamped. The post-reset state is identical regardless of which event triggered it.

9.2 Transition table

Table 9.1 enumerates every transition currently in `lifecycle_step/4`. The three named guards are: `Reset_OK = (Now - last_reset_ms) ≥ 1000 ms`, `Traj_OK = (Now - last_traj_acti ≥ 500 ms`, and `RobUp = the tilt sensor reports the chassis is upright`.

From	Trigger	Guard	To
<code>idle</code>	Start/Pause	<code>Reset_OK</code>	<code>running</code>
<code>idle</code>	Reset	<code>Reset_OK</code>	<code>idle</code> (wipe)
<code>running</code>	Start/Pause	<code>Traj_OK</code>	<code>paused</code>
<code>running</code>	Reset	—	<code>idle</code>
<code>running</code>	<code>¬RobUp</code>	—	<code>idle</code>
<code>running</code>	<code>traj finished</code>	—	<code>idle</code>
<code>running</code>	cancel drains queue	—	<code>idle</code>
<code>paused</code>	Start/Pause	<code>Traj_OK</code>	<code>running</code>
<code>paused</code>	Reset	—	<code>idle</code>
<code>paused</code>	<code>¬RobUp</code>	—	<code>idle</code>
<code>any</code>	<code>CLEAR_ALL</code> frame	—	<code>idle</code>

Table 9.1: Every lifecycle transition in `main_loop.erl`. The last four rows in the `idle` column all route through `go_idle/3` so the post-transition state is identical regardless of source.

Two deliberate omissions are worth noting. First, `idle` has no `¬RobUp` handler. In case the robot is knocked over while sitting in `idle`, the lifecycle is unaffected. The reason is in Section 9.3. Second, `paused` cannot reach `running` except by an explicit Start/Pause command. If the robot is paused and left unattended, it waits.

9.3 Pre-flight waypoint queueing

Waypoints are accepted at any lifecycle state — `idle`, `running`, or `paused`. A complete run can therefore be loaded while the robot sits in `idle`, and then started with a `Start` command. This is called pre-flight queueing.

Two design choices follow. The first is the deliberate omission of a `¬RobUp` handler from the `idle` branch in `lifecycle_step/4`. If waypoints are queued and the robot is knocked over while in `idle`, the queue is not wiped as reloading every waypoint after an accidental nudge would be frustrating. No movement is possible in `idle` (velocity refs into the mux are zero), so a fall there carries no safety consequence. The only deliberate reset paths from `idle` are a `Reset` command and a full clear with a `CLEAR_ALL` frame.

The second is that `go_idle/3` *does* clear the queue. Preflight waypoints survive accidental falls in `idle`, but anything that transitions the robot back to `idle` from `running` or `paused` for example a fall, a `Reset` command, a completed trajectory, or a drained cancel, is treated as a clean slate. Knocking over a parked robot is an accident, knocking over a running robot is the considered end of that run.

Chapter 10

Over-the-Air Waypoint Transmission Protocol and Planning UI

A path-following robot is useless without a way to tell it where to go. On this platform, the path is a list of waypoints, and the question is how those waypoints are sent from where they are generated to the running Erlang process on the GRiSP. The answer in this thesis is a one-byte wireless protocol that piggybacks on the same drive-control byte the robot was already reading every tick. The Python desktop app built to drive it is a test harness; it exists to exercise the protocol. The protocol is the part that matters; it is what a second robot or a swarm coordinator would use to talk to this one. This chapter is about the protocol first, the desktop UI second.

10.1 Why a one-byte protocol

The carrier was decided by the inherited hardware. The LilyGo co-processor sits between the radio and the GRiSP and forwards one byte at a time over the I²C bus (described in the inherited platform chapter). That byte is for manual control. With bit fields for the four direction buttons and a few status flags. Changing the LilyGo firmware to expose a wider channel was out of scope so was switching the link from LoRa to anything else. Whatever was sent had to be inside that single byte.

A one-byte channel is a constraint, but a workable one. The point of having a protocol is that any sender producing the right byte stream can inject waypoints. The laptop UI is one sender; a second robot running the same radio stack is another. Both produce identical bytes, and the receiver does not know or care which. The planning UI is one sender; a second robot is another.

10.2 Frame layout

Every byte the GRiSP reads from I²C carries eight bits with the same fixed meaning. The relevant ones for the protocol are summarised in Table 10.1.

Bit	Name	Meaning
b7	Arm_Ready	Set by the LilyGo when the chassis is armed.
b6	PROTO	1 = protocol frame, 0 = normal drive byte.
b5	KIND	Only meaningful if PROTO = 1: 1 = data, 0 = ctrl.
b4..b0	payload	5 bits: either a control opcode or 5 data bits.

Table 10.1: Shared control byte bit layout. b7 and b6 are always interpreted; b5..b0 depend on PROTO.

When $\text{PROTO} = 0$, the byte is a drive command which means the lower bits map to direction inputs. When $\text{PROTO} = 1$, the byte is a protocol frame. $\text{KIND} = 0$ means *ctrl*. The payload in *ctrl* is a 5-bit opcode (ADD_HEADER, commit, abort, and so on). $\text{KIND} = 1$ means *data*. The payload in *data* carries 5 raw bits, half of a signed 10-bit coordinate.

Giving up b6 as a mode switch from earlier keeps the channel safe to share. The drive code never sets b6, the protocol always does. Neither side needs to know about the other.

10.3 The command set

The protocol has exactly nine reserved 5-bit opcodes. They split into three groups: simple commands that complete in a single frame, multi-frame transactions that build up a value, and separators that act as commit gates inside a transaction. Table 10.2 lists them all.

Opcode (b4..b0)	Name	Group	Effect
00000	ABORT	single	cancel any in-progress transaction
10000	CANCEL_LAST	single	pop the last waypoint from the queue
10001	CLEAR_ALL	single	drop the queue, reset pose, $\rightarrow$ idle
10010	CANCEL_N_HDR	transaction	start a cancel-N (expects 1 data + commit)
10011	CANCEL_N_COMMIT	transaction	commit the N from the previous data frame
11000	ADD_HEADER	transaction	start an ADD (expects 4 data + 3 separators +
11010	HI_TO_LO	separator	guard between the hi and lo half of ΔX or ΔY
11100	X_TO_Y	separator	guard between the ΔX pair and the ΔY pair
11110	ADD_COMMIT	transaction	commit the accumulated (ΔX , ΔY)

Table 10.2: The full command set. All opcodes share the same 5-bit field; the receiver decides whether a given opcode is legal by checking which state of the receiver FSM it is in (Section 10.5). The opcode bit-patterns were chosen so that no two are off by a single bit-flip, which keeps single-bit channel errors from silently turning one valid command into another.

10.4 The ADD sequence

ADD is the main transaction. A waypoint is a pair of signed offsets (ΔX , ΔY) in centimetres, relative to the previous waypoint or to the current pose if the queue is empty. Each axis is clamped to ± 50 cm; longer moves are chained ADDs. Each axis is encoded as a 10-bit value (offset by 50 so the wire value stays in $[0, 100]$) and split into two 5-bit halves.

The full nine frame sequence is:

```
ADD_HEADER, X_hi, HI_TO_LO, X_lo, X_TO_Y, Y_hi, HI_TO_LO, Y_lo,  
ADD_COMMIT
```

The four data frames carry the 5-bit halves for each axis. The three separator frames (HI_TO_LO, X_TO_Y, HI_TO_LO) exist to break runs of identical byte data frames. The stability gate (Section 10.5) requires two consecutive identical bytes before accepting a frame, which absorbs single byte glitches but is fooled when two consecutive data frames carry the same value. The separator forces a different byte between any two data frames that could collide.

Frames are paced at 30 ms each, well above the two-tick stability threshold (about 11 ms). A full ADD takes about 270 ms end to end. A sender generating one waypoint per second has roughly four times that window between transactions, so the timing fits well within normal inter waypoint spacing.

10.5 Receiver state machine and stability gate

The receiver is a small state machine inside `main_loop.erl` called `cmd_rx_step/2`. It runs every tick, takes the current control byte as its only input, and walks a small state record (`#cmd_rx{}`) through the ADD or CANCEL_N transaction. Figure 10.1 shows the ADD branch.

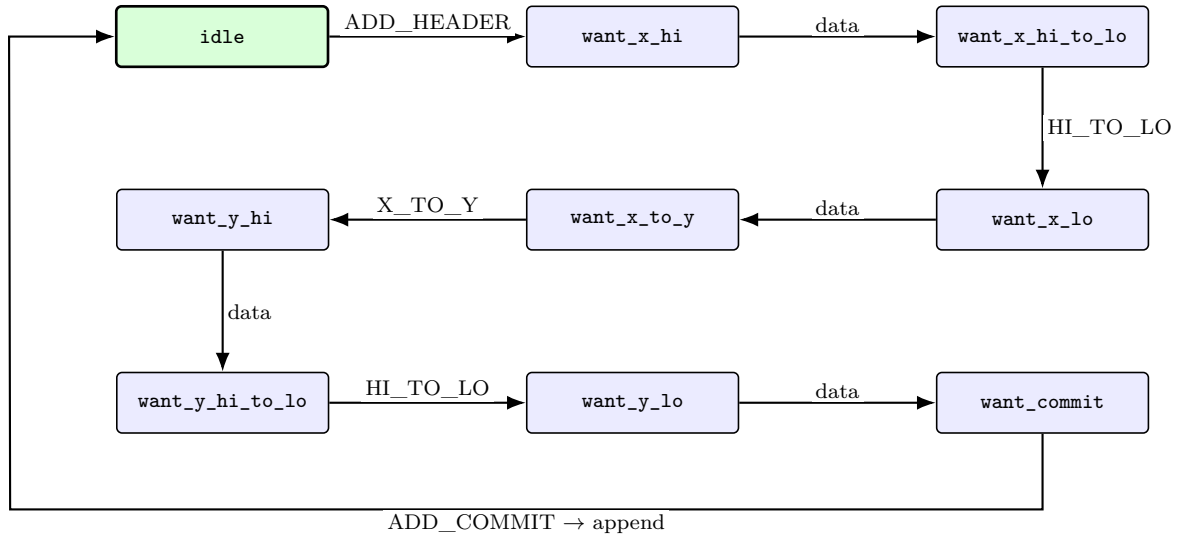


Figure 10.1: The ADD branch of the receiver FSM. Each arrow consumes one frame and advances by one state on success. Any unexpected frame in any non-idle state resets the machine to `idle` and fires the error LED; the error edges are omitted here for legibility, and the complete state machine with every error transition is given in Appendix B.

Three guards sit in front of the FSM dispatch:

Stability gate. The decoder requires two consecutive ticks with the same byte before it accepts the byte as a frame. At the measured loop rate, two ticks is roughly an 11 ms debounce. Without it, a single noisy tick during a LoRa retransmission could fire a stray transition. The sender’s 30 ms inter-frame pacing leaves plenty of room for the gate to settle.

Commit-edge gate. A byte that has already been committed will not commit again until something different comes through. This is what stops a held-still wire from re-firing the same frame every 6.7 ms. The gate is dropped when a stable drive byte appears ($\text{PROTO} = 0$), so the next protocol frame counts fresh.

Sign-range check. On `ADD_COMMIT`, the accumulated 10-bit halves are reassembled and the offset of -50 is subtracted. The result must be within in $[-50, 50]$ on both axes. Out-of-range values are dropped and the waypoint is not appended.

When the commit succeeds, the receiver returns a high-level action (`{append, DX, DY}`, `cancel_last`, `{cancel_n, N}`, or `clear_all`), which the main loop applies to the trajectory state in the same tick. ADD takes effect immediately. If the lifecycle state is `running` then the spline is rebuilt in the middle of cruise as described in the path representation chapter.

10.6 The planning UI as a test harness

The Python desktop UI is a thin send only client. It draws a grid, accepts waypoints, encodes each one into the nine frame ADD sequence above, and pushes the frames out over LoRa at the same 30 ms pacing. There is no return channel. The UI never reads

anything from the robot. The only feedback that a frame landed is the LED on the GRiSP board: white for “frame accepted”, white-held for “committed”, red for “error”. That is enough for testing.

Part III

Experimentation and Evaluation

Chapter 11

Experimental Methodology

Runs were logged over serial using a Python script. Floor tape marks were placed at the trajectory waypoints as physical references. For each test, the robot was aligned to the starting waypoint and commanded to traverse the full path and return to the origin. The final physical position and heading were then compared to the pose the robot had integrated internally. Throughout all experiments, wheel-odometry yaw was used as the active channel in the path-following layer.

11.1 The three candidate yaw sources

All three yaw estimates run on the same robot, on the same tick, reading from the same physical sensors. They differ only in which physical signals they touch.

Wheel odometry yaw (`theta_deg` in the log) comes from differential wheel speed integration. The left and right wheel velocities are derived on the LilyGo from motor step counts along with wheel diameter and reported to the GRiSP board over I2C. They are differenced and integrated over time using the kinematic formula $\dot{\theta} = (v_R - v_L)/b$ with b the wheel base. This is the pose that the path following layer reads. It is computed in `odometry.erl`.

Gyro-integrated yaw (`yaw_gyro`) is read from the LSM9DS1's rate gyro (`out_x_g` in chip frame, ω_z in body frame), corrected for the boot-time bias, projected onto the horizontal plane using the pitch estimate, and Euler-integrated. The estimate is logged alongside wheel odometry but is never fed back into the controller.

Magnetometer compass yaw (`yaw_mag`) reads the LSM9DS1's magnetometer, applies a hard-iron offset captured at boot, projects the body frame field onto the horizontal plane using the accelerometer based gravity vector, and computes a tilt compensated heading via vector projection. The estimate is reported relative to the heading at trajectory start (rather than against magnetic north), so a 0° reading means "facing the direction the robot was facing when the trajectory was started with the Start/Pause command". Like the gyro channel, it is log only.

Every one of the three channels is computed and logged on every tick. The path-follower reads only the first.

11.2 Test paths and physical setting

Two test trajectories were used, both at the same waypoint geometry, run in opposite rotational directions:

- **CCW square loop:** four corners at $(0, 0) \rightarrow (50, 0) \rightarrow (50, 50) \rightarrow (0, 50) \rightarrow (0, 0)$ in cm, traversed counter-clockwise at cruise speed 14 cm/s. Total path length ≈ 200 cm. Total wall-clock run time ≈ 150 s (`traj_logccw.csv`, 1834 ticks logged).
- **CW square loop:** same four corners traversed clockwise, same cruise speed. Run time ≈ 145 s (`traj_log185cw.csv`, 1812 ticks logged).

A separate straight-line trajectory (100 cm forward from origin, no turns) was also run for the position-error analysis in Chapter 12.

The two physical floor-marker rigs on which these trajectories were run — the 50×100 cm rotation rig (with the 10° yaw fan at the origin) and the 250 cm straight-line rig (with the 25 cm distance ticks) are drawn to scale in Appendix C, Figures C.1 and C.2.

The tests were conducted on a clean, dust free flat surface to minimize wheel slippage and balancing disturbances from uneven friction.

11.3 Logging mechanics

Data was broadcast over serial in the TLOG format at every tenth tick (approximately 18 Hz at the measured loop rate) and captured live on a host machine by a Python script.

For the localisation experiments, the per-row schema was extended to carry all three yaw channels and the raw magnetometer components. The recorded columns relevant to this part are summarised in Table 11.1.

Column	Unit	Meaning
<code>T_ms</code>	ms	milliseconds since boot
<code>lifecycle</code>	atom	<code>idle</code> / <code>running</code> / <code>paused</code>
<code>substate</code>	atom	<code>cruise</code> / <code>braking</code> / <code>dwelling</code> (Chapter 8)
<code>x, y</code>	cm	integrated position from wheel odometry
<code>theta_deg</code>	deg	integrated yaw from wheel odometry, the channel actually consumed by the path-follower
<code>yaw_gyro</code>	deg	integrated yaw from the rate gyro, bias-corrected (Chapter 13)
<code>yaw_mag</code>	deg	tilt-compensated compass heading, launch-relative (Chapter 14)
<code>mag_x_raw, mag_y_raw</code>	gauss	raw chip-frame magnetometer components (only in the CCW log)
<code>cal_status</code>	atom	<code>none</code> / <code>running</code> / <code>done</code> / <code>failed</code> of the mag cal FSM

Table 11.1: The columns of the per-tick log used in Part III. The schema is fixed per run. The CCW log carries two extra columns of raw magnetometer data that were added for the soft-iron post-mortem and were not yet present when the CW log was captured.

11.4 Ground truth and measurement uncertainty

The reference for each run was a set of tape marks on the floor: an origin mark and a target mark at the intended endpoint of the path. After each run, the offset between the midpoint of the robot’s base and the target mark was measured with a ruler. Residual yaw error was estimated visually against 10°-spaced marks placed at the path corners. Both measurements had to be taken immediately after the robot stopped; the balancing micro-corrections cause the robot to drift from the position it stopped at within seconds.

Two sources of noise limit how finely these results can be read.

The first is the origin position. Balancing micro corrections keep the base oscillating within roughly ± 1 cm of any nominal position even at zero commanded velocity, so position offsets smaller than this are not meaningful.

The second is the initial yaw. The chassis was aligned visually against the marker line before each run, reliable to only about $\pm 5^\circ$ without an instrument. Any yaw error at the end of a run carries this initial alignment error.

The practical consequence is that any per run error of a few centimetres or a few degrees should be read as “within the floor of what can be measured this way”, not as a meaningful indicator that one estimator is finer than another. The size of the position errors that follow falls largely in this band.

Chapter 12

Wheel Odometry Baseline

Wheel odometry is the cheapest yaw estimate this robot can produce. The two drive wheels turn the chassis, the chassis goes where the wheels make it go, and integrating the wheel velocities and the wheel base back into a pose is, in a sense, just keeping a running tally of what the wheels already did. There is no separate sensor reporting a different physical quantity that has to be reconciled with the motion. This chapter writes out the kinematic model, lists the physical effects that make it imperfect, and reports the measured final-position and final-yaw error on three trajectory runs conducted multiple times against the floor-marker ground truth from Chapter 11.

12.1 Kinematic model

The robot is a two-wheel differential drive with wheel base $b = 18.5$ cm. The pose is the triple (x, y, θ) , with (x, y) in centimetres relative to the trajectory launch point and θ in degrees normalised to $[-180^\circ, 180^\circ]$. The launch heading is $\theta = 0$ by definition. Every `idle` $\rightarrow$ `running` transition (Chapter 9) sets the pose to $(0, 0, 0)$.

Each tick, `odometry.erl` reads the two wheel velocities v_L and v_R in cm/s (computed by the LilyGo from step counts and the known wheel diameter, returned over I²C inside the same 5-byte read that delivers the control byte), forms the linear and angular increments, and Euler-integrates:

$$v = \frac{1}{2}(v_R + v_L), \quad \omega = \frac{v_R - v_L}{b}, \quad (12.1)$$

$$\theta_{k+1} = \theta_k + \omega \cdot \Delta t, \quad x_{k+1} = x_k + v \cos(\theta_k) \Delta t, \quad y_{k+1} = y_k + v \sin(\theta_k) \Delta t, \quad (12.2)$$

with $\Delta t \approx 5.6$ ms at the measured loop rate (Chapter 5). The linear increment uses the heading at the start of the tick; the angular increment uses forward Euler. At this tick rate and turn rates below $30^\circ/\text{s}$, the integration error from this choice is negligible.

12.2 Why it drifts

Equation (12.2) is a kinematic identity. If the wheel velocities v_L, v_R that the integrator sees are exactly the wheel velocities the chassis is experiencing, the integrated pose is exact too. There are two places where that equality breaks down on this platform.

Micro-slip at the contact patch. The load on the wheels is constantly shifted by the balance loop (the robot pitches forward and back by several degrees during cruise. Every time the robot pitches, the normal force on each wheel changes. Whenever the normal force changes faster than the wheel commands change, the wheel briefly slips through a fraction of a millimetre that the encoders see as motion but that the surface does not. The slip is small per tick and roughly unsigned, but the variance does not average out cleanly because the slip on each wheel is correlated with the robot motion, not independent.

Surface unevenness. At the scale of an 18.5 cm wheelbase and 7 cm wheel diameter, small surface variations for example a grain of grit, a hair, a tile edge, even a slight grade can momentarily pitch one wheel relative to the other, which the kinematic formula reads as a small turn. This is the same family of effect as micro slip but originates in the surface rather than in the contact patch. Borenstein’s UMBmark study [63] classifies dead-reckoning error into systematic (geometry mismatches) and non-systematic (slip, bumps). The non-systematic category dominates at this scale.

12.3 Measured final-pose error

Two trajectories were run multiple times against the floor-marker ground truth described in Section 11.4. After each run the offset between the robot’s wheel-axle midpoint and the target floor marker was measured with a ruler, and the residual yaw against the marker-defined direction was estimated visually.

200 cm CCW square loop, return to origin

Four runs of the closed CCW square (corners at (0, 0), (50, 0), (50, 50), (0, 50), returning to (0, 0)) at 14 cm/s cruise. The trajectory layer auto aligns the path back to the origin via the final waypoint. The reported error is the offset between the resting chassis and the origin markings.

Run	Δx (cm)	Δy (cm)	Yaw error
1	3	4	0–10°
2	2	3	0–10°
3	5	3	0–10°
4	4	–2	0–10°

Table 12.1: Final-position and final-yaw error after the 200 cm closed CCW square, measured against the origin floor marker. Position errors are reported as observed offsets to the nearest centimetre. Yaw error is given as a range because the visual alignment against the marker line is not finer than a few degrees (Section 11.4).

200 cm CW square loop, return to origin

Four runs of the closed CW square (corners at $(0, 0)$, $(50, 0)$, $(50, -50)$, $(0, -50)$, returning to $(0, 0)$) at 14 cm/s cruise.

Run	Δx (cm)	Δy (cm)	Yaw error
1	3	2	0–10°
2	4	1	0–10°
3	1	–3	0–10°
4	–1	0	0–10°

Table 12.2: Final-position and final-yaw error after the 200 cm closed CW square, measured against the origin floor marker.

100 cm straight-line run

Five runs of a single 100 cm forward leg from the origin, with no corners. The target is a floor marker 100 cm in front of the launch point along the launch direction.

Run	Δx (cm)	Δy (cm)	Yaw error
1	–1	3	negligible
2	2	–4	negligible
3	1	3	negligible
4	2	4	negligible
5	–1	–3	negligible

Table 12.3: Final-position error after the 100 cm straight-line run, measured against the 100 cm forward marker. No corners are traversed, so the integrated yaw at the end is essentially the same as the integrated yaw at the start (modulo the small heading corrections Pure Pursuit applies to stay on the straight line). The residual Δy values are also consistent with launch alignment error: the robot is placed at the origin visually, and a 1° heading error at launch produces 1.74 cm of lateral deviation over 100 cm.

Chapter 13

Gyro Yaw Fusion Experiment

The rate gyro lives on the same chip as the accelerometer the balance loop already reads (LSM9DS1 inside the Pmod NAV), so adding a yaw estimator that integrates the vertical-axis gyro is essentially free in code and free in tick budget. The interesting question is whether it gives a meaningfully different picture of yaw from wheel odometry. The answer on the test loops in Chapter 11 is that it does not. Over the full 150-second runs, integrated gyro and wheel-odometry yaw stay within a few degrees of each other at all times, and finish under one degree apart. That outcome is what this chapter documents.

A dedicated module `gyro_yaw.erl` was written to mirror the structure of `odometry.erl`: stateless, called once per tick from `main_loop.erl`, returning an updated yaw value. On each tick the chip-frame register `out_x_g` was read, the per-boot bias $\omega_{z,0}$ captured during the existing accelerometer calibration window was subtracted, the corrected rate was projected onto the horizontal plane as $\omega_z^{\text{flat}} = \omega_z \cos(\theta_{\text{pitch}})$ using the tilt-Kalman estimate [68], and the result was Euler-integrated and wrapped into $[-180^\circ, 180^\circ]$. The bias-corrected rate (in deg/s) was also written as column `gx_dps` so that any sign-convention error or bias-drift trend could be read directly off the log. The stability cascade, the tilt Kalman, the odometry, and the trajectory layer were left untouched; `yaw_gyro` was log-only from day one, and the path-follower kept reading the odometry-based `theta_deg`.

13.1 Gyro Drift Theory

A rate gyro measures angular velocity, not angle. To get an angle the signal must be integrated, and integration means any residual bias b left over after calibration grows without bound which means a bias of b deg/s integrated for T seconds gives $b \cdot T$ degrees of drift, regardless of actual motion. The MEMS gyro inside the LSM9DS1 is a commodity part for which manufacturer specs quote a typical zero-rate bias of a few tenths of a deg/s, with a temperature dependence of similar magnitude across the operating range [69]. Selfheating during the first minutes of operation can shift the bias further.

Wheel odometry has the opposite error character. Each wheel step introduces a small slip and quantisation error, and those errors sum over distance driven rather than over

time. On a short run the two estimators accumulate error at comparable rates.

13.2 What the data showed

Tick-by-tick comparison over the full runs. Each direction was run five times on the 200 cm square loop (Section 11.2, ≈ 150 s per run). Across all runs the difference $\theta_{\text{gyro}} - \theta_{\text{odo}}$ had an RMS of about 3° and stayed inside a $\pm 7^\circ$ band. Table 13.1 gives the per-direction averages.

Direction	RMS $ \theta_{\text{gyro}} - \theta_{\text{odo}} $ (deg)	Range (deg)	Terminal diff (deg)
CCW square (5 runs)	2.7	$[-5.3, +2.2]$	+0.7
CW square (5 runs)	2.9	$[-0.5, +7.0]$	+0.2

Table 13.1: Per-tick statistics of the gyro-vs-odometry difference, averaged over five runs per direction on the 200 cm square loop. The two channels agree within a few degrees throughout and finish under one degree apart.

Terminal heading against the floor markers. On both CCW and CW runs the actual observed terminal pose was the same as evaluated before, since the trajectory was still odometry-based. The integrated yaw at that moment was -5.8° on the odometry channel and -5.1° on the gyro channel for the CCW runs, and $+5.7^\circ$ on odometry and $+5.9^\circ$ on gyro for the CW runs. Both channels report essentially the same residual heading, and both fall inside the $\pm 10^\circ$ visual-alignment floor from Section 11.4.

13.3 Outcome

The gyro and the odometry tell the same story about how much the robot has rotated. They share their largest error source (whatever happens during corner negotiation makes both estimators turn by the same amount, because both are responding to actual chassis motion). They differ only in second-order noise that does not accumulate to a meaningful magnitude on a 150-second run.

A complementary filter between two estimators that already agree this closely cannot improve over either alone. There is no independent signal to fuse with. The gyro is not worse than odometry on the test loops, but it does not give the controller any information that odometry was not already carrying. Since odometry is the more physically direct of the two (it is integrating motion that the wheels actually made, rather than integrating a noisy rate from a chip with a temperature-dependent bias), it is the one that the path-follower keeps reading.

Chapter 14

Magnetometer-Based Heading Experiment

The magnetometer is the third sensing element on the LSM9DS1 chip inside the Pmod NAV. It measures the Earth’s magnetic field vector in three body-frame axes. Combined with the accelerometer’s gravity vector, a tilt-compensated heading can be derived: one that is absolute, does not drift with time, and does not accumulate slip error from the wheels. On paper, this makes it the right sensor for correcting odometry over longer paths. In practice, on this platform, extracting a useful signal proved challenging, and the pipeline was eventually retired.

14.1 Hard iron, soft iron, and why calibration is necessary

A magnetometer reading on an ideal robot in an ideal environment would, when rotated through a full 360° , trace a circle centred on the origin with radius equal to the local horizontal field strength. On a real robot in a real environment, the locus is an ellipse offset from the origin. The offset and the distortion come from two distinct physical effects, conventionally called hard-iron and soft-iron [65], [66].

Hard-iron distortion is a constant additive bias in the body frame. It is caused by permanently magnetised materials near the sensor such as the battery casing, the motor magnets at rest, the ferrite chokes on the ESP32 board. Each contributes a fixed offset vector that is added to every reading regardless of orientation. The correction is a single 3-vector subtraction: $\mathbf{m}_{\text{corr}} = \mathbf{m}_{\text{raw}} - \mathbf{b}$, with $\mathbf{b}$ taken from the centre of the observed locus.

Soft-iron distortion is an orientation dependent multiplicative effect caused by ferromagnetic materials that rechannel the field. These could be nearby steel screws, the motor stators or metal parts in the chassis frame. The apparent field strength in a given body frame direction depends on how the body is oriented in the world. The correction is a 3×3 matrix that maps the observed ellipsoid back to a sphere [70].

The harder problem on a self-balancing robot is that the soft-iron distortion is not static. Motor currents change with the power applied, which changes with the balance loop’s

response to whatever the chassis is doing. A calibration captures a static snapshot. A static snapshot cannot model a current dependent effect, in our case.

14.2 The calibration journey

Attempt 1: in-place self-spin

The first approach had the robot balance and spin in place at a constant differential turn rate while `mag_filter.erl` sampled chip-X and chip-Y min/max in the horizontal plane. The hard-iron offset was taken as $(\min + \max)/2$ on each axis. This matches the standard bounding-box calibration in NXP's compass-design application note [65]. The approach worked mechanically but produced compass headings off by 8–10° across the full 360° range. The root cause was that the motors were energised during the entire spin, so motor currents biased the captured extremes, and the balance loop's tilt leaked the vertical-axis field into the horizontal plane.

Attempt 2: 3-axis hand-tumble

To remove the motor current caused bias, the procedure was rewritten. At the same trigger, the motors were commanded idle and the chassis was physically rotated through a varied set of 3-D orientations for $\approx 20\text{--}30\text{ s}$ while all three axes were sampled simultaneously. A sphericity check $\max(\text{range}_i) / \min(\text{range}_i) \leq 1.2$ was added to reject flat loci, and a settling period before motors on was included. Bench tests returned reproducible offsets and the computed heading agreed with an iPhone compass within a few degrees with the motors off.

Attempt 3: tilt-compensated compass

With a clean hard-iron offset on all three axes, heading was still being computed as $\text{atan2}(M_y^{\text{corr}}, M_x^{\text{corr}})$ in the chip's horizontal plane. This formula is correct only when the chip's vertical axis is exactly aligned with gravity. The Brussels magnetic inclination of $\approx 67^\circ$ means even a small chassis tilt projects a meaningful fraction of the vertical field into the horizontal axes and biases the heading by several degrees. The textbook tilt-compensated compass formula was applied [65]. The corrected magnetometer reading was projected onto the plane perpendicular to the accelerometer gravity vector, and heading was computed from that projection. The implementation lives in `magnetometer:read/2`.

14.3 Why it still did not work

The hand-tumble calibration proved more stable than the in-place spin and was carried forward alongside the tilt-compensation formula. On the bench with the motors off, the heading was clean. The moment the motors were switched on, the reading became non-linear. A 30° physical spin in one orientation gave a 40° magnetometer-reported change; a 20° spin in a different orientation gave only 10°. Motor currents distorted the locus differently as a function of heading, because the currents themselves depended on chassis state. No static calibration can model that. The computed heading also jittered

by $\pm 2\text{--}3^\circ$ tick to tick even when the robot was stationary and balancing, which was larger than the per-corner drift the magnetometer was meant to correct.

14.4 Retirement

The magnetometer pipeline was abandoned on 2026-05-23. The compass reading was non-linear under motor load in a way that no static calibration could fix, making it unreliable as an external reference for yaw correction. The per-tick magnetometer read via `pmod_nav:read/3` also dropped the measured loop rate from about 180 Hz to about 129 Hz. That frequency reduction directly affects the balance mechanics: the tilt PID, the velocity ramp, and the pose estimate all run at the slower tick rate. The cost was not worth paying for a signal that could not be trusted.

The modules `magnetometer.erl` and `mag_filter.erl` were removed, along with the `ENABLE_MAG` flag, the yaw fusion code path in `main_loop.erl`, and the three magnetometer registers from the `pmod_nav:read/3` call. The tilt formula in the balance loop was reverted from `atan2( $A_z$ ,  $-A_x$ )` back to `atan( $A_z$  /  $-A_x$ )`.

Chapter 15

Results and Evaluation

15.1 The three estimators on one plot

Figures 15.1 and 15.2 show wheel odometry, gyro integration, and magnetometer compass on the CCW and CW square loops of Chapter 11, each unwrapped through 360° and plotted against time since the trajectory started. Dwell and brake ticks were filtered out so the comparison is only against rows where the robot was actually cruising. Each plot is taken from one of five consecutive runs of its respective direction, and the results were similar across all five runs in each direction, with run-to-run variation on the gyro and odometry channels staying under 2° across the full run.

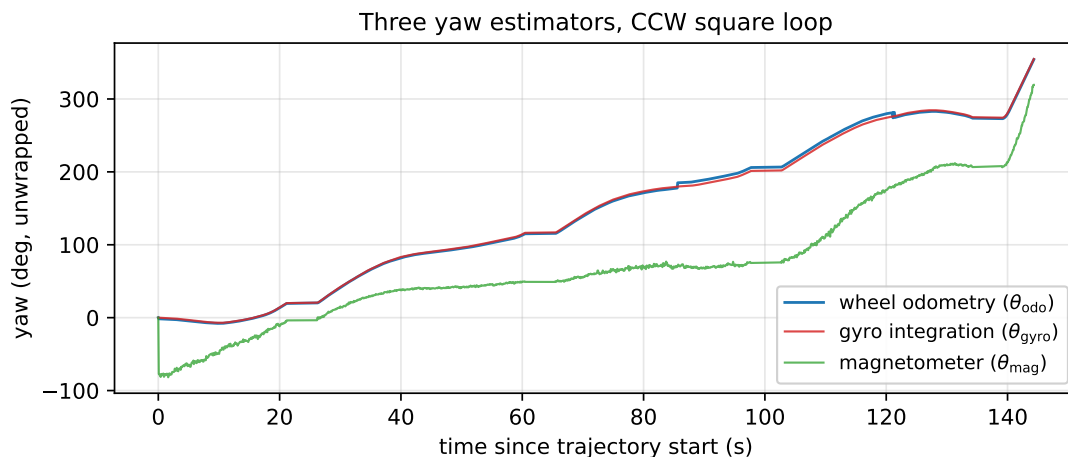


Figure 15.1: Wheel odometry (θ_{odo}), gyro integration (θ_{gyro}), and magnetometer compass (θ_{mag}) over the 150-second CCW square loop, taken from one of five consecutive runs of the same trajectory. The two integration-based channels (odometry and gyro) track each other within a few degrees across the entire run. The magnetometer channel agrees on average direction but its slope is shallower than the actual rotation rate, ends roughly 60° short of the true accumulated heading, and is visibly noisier than either of the other two.

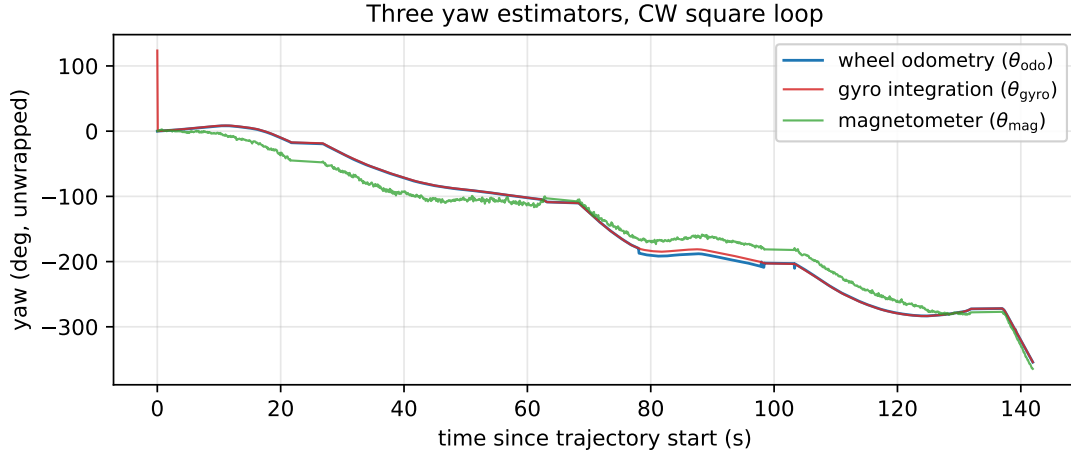


Figure 15.2: Wheel odometry (θ_{odo}), gyro integration (θ_{gyro}), and magnetometer compass (θ_{mag}) over the 142-second CW square loop, taken from one of five consecutive runs of the same trajectory. Odometry and gyro again overlay each other to within a fraction of a degree at the end of the loop. The magnetometer channel tracks the rotation direction but overshoots the accumulated heading by roughly 10° and carries the same tick-to-tick jitter seen in the CCW direction.

Two observations follow directly from the plot.

Odometry and gyro agree throughout. The two curves overlay each other for the entire 150 second runs, separated by only a few degrees at any given tick (RMS difference about 3° , Section 13.2). The gap at the end of the loop is under one degree on both directional runs, so neither channel drifts further than the other on this trajectory. The two estimators are reporting the same physical rotation.

The magnetometer trace does the wrong thing. On the CCW loops it climbs more slowly than the actual rotation rate and ends roughly 60° short of the true accumulated heading. The shape is not random noise. It is the soft-iron-distorted ellipse compressing some heading ranges and stretching others, producing a curve that broadly tracks the rotation direction but with the wrong scale. The visible tick to tick jitter is roughly ± 2 to $\pm 3^\circ$, consistent with the steady state numbers from Section 14.3. On the CW loop the terminal error is much smaller, about 10° of overshoot past the expected -360° , which can be attributed to a more accurate hard and soft iron calibration captured for that direction. Even with the better calibration, however, the trace still drifts off and rejoins the true heading several times mid run, and the gyro and odometry channels remain closer to ground truth at every segment. The magnetometer is therefore not a useful addition for path following on this rig, regardless of how well it happens to be calibrated.

15.2 Why odometry won

The choice of which estimator drives the controller comes down to a simple comparison. Wheel odometry and integrated gyro report the same rotation to within the measurement floor of the experiment, and the magnetometer reports something significantly worse.

Table 15.1 summarises.

Estimator	Terminal yaw error vs. markers	Dominant cause
Wheel odometry	within $\pm 5^\circ$ floor	micro-slip at contact patch, surface unevenness
Integrated gyro	within $\pm 5^\circ$ floor	residual bias after boot calibration
Magnetometer compass	$\geq 30^\circ$	motor soft-iron, current-dependent distortion

Table 15.1: Terminal yaw error for the three estimators on the 200 cm closed square loop at 14 cm/s. The first two rows fall inside the floor of the floor-marker rig’s visual-alignment uncertainty, so neither can be ranked finer than the other on this experiment. The third row is well outside that floor.

Given the table, there is nothing for a complementary filter to do between odometry and gyro because any non trivial weight on one source pulls the fused estimate away from a source it already agrees with. And there is no useful weight on the magnetometer either, because its error is much larger than the disagreement between the other two and changes with stepper motor operation (coil current state, micro step phase, and per tick acceleration commands).

Beyond accuracy, the magnetometer carried a runtime cost that is critical on this platform. When the magnetometer reads were active in the main loop, the achievable loop frequency dropped from approximately 180 Hz down to about 130 Hz. That is a serious regression for a robot whose tilt stabilisation cascade depends on a fast inner loop: every tick lost to magnetometer I/O is a tick the balance PID does not get, and at the lower rate the cart became visibly less stable during straight cruising. A sensor that costs 50 Hz off the balance loop has to be worth that cost in pose accuracy. In my case it was not.

The path follower reads wheel odometry alone for two reasons. The first is that the experiment did not find a second source that was meaningfully more accurate, so adding one would have been added complexity without an accuracy gain. The second is that odometry does not drift while the robot is sitting still, with both wheels stationary, $v_L = v_R = 0$ gives $\omega = 0$ and the integrator does not move. The gyro can drift through any non zero residual bias $\times$ any non zero time, and similarly, the magnetometer can swing tens of degrees if a metal object moves nearby. Of the three candidates, odometry is the one whose error is tied to commanded motion rather than to elapsed time or environment.

That property is the right property for a path follower at the speeds and run times this thesis targets, and for the future developments with sonar or UWB seonsors in play.

Chapter 16

System Evaluation

The bar to clear is the one set in Chapter 4. A waypoint sequence arrives over the air, the robot integrates it into the current path, and the chassis traverses that path on a flat indoor floor, ending at the final waypoint within a margin small enough to be useful. The waypoint sequence is meant to be the live position stream of a second robot in a future multi-robot scenario. The single sender in this thesis is a Python desktop UI that exists as a stub for testing the receiver side.

16.1 What was delivered

Four pieces were specified in Chapter 4 and all four were built end to end on the inherited platform. The planning interface from Chapter 10 was implemented as a one byte wire protocol multiplexed onto the inherited drive-control I2C channel. ADD, CANCEL_LAST, CANCEL_N, and CLEAR_ALL transactions were carried over that surface, and the receiver was made to accept them in any lifecycle state, so a full sequence can be queued while the robot is idle and started with a single START/PAUSE command.

The path representation was implemented as the Catmull-Rom spline of Chapter 6. The polynomial form of Eq. 6.1 was used unchanged; phantom endpoint waypoints were synthesised to give the first and last real waypoints a defined tangent; the curve was rebuilt on every queue mutation so live edits during cruise are picked up on the next tick. Thirty sub-points per segment were emitted, which was found to be the smallest count that the Pure Pursuit lookahead could not visibly skip over at the speeds used in this thesis.

The path tracker was implemented as the Pure Pursuit controller of Chapter 7. A fixed 8 cm lookahead was used together with the curvature formula $\kappa = 2\ell_y/L^2$ from Eq. 7.2. Forward speed was modulated by curvature using the law of Eq. 7.5, with the knee placed at $|\kappa| = 1/b$ and a 1 cm/s floor. The acceleration ramp, the turn-velocity gain, and the brake distance were all tuned against the trajectory sub-state machine of Chapter 8 so the chassis could be pushed close to the limit at which the inherited balance cascade starts to wobble, without being pushed past it. Those ratios were what made the cruise speed usable rather than purely conservative.

The Python desktop UI of Chapter 10 was kept deliberately thin. It existed as a stub

sender so that the receiver state machine could be exercised in isolation; nothing in the receiver was tailored to it. It also turned out to be a useful test instrument on its own. Waypoints could be queued, cancelled, and re-edited while the robot was held in idle, and the resulting spline could be inspected before any F+B tap committed the robot to it. Every test reported in the next section was driven from this UI.

16.2 Further testing

Three end-of-path tests were performed on a flat floor following the methodology of Chapter 11. The visual-measurement floor and the floor-marker rig of Section 11.4 were re-used; the two physical rigs (rotation and straight-line) are drawn to scale in Appendix C, Figures C.1 and C.2. Each test was repeated ten times and the endpoint of the chassis at the end of the run was recorded in centimetres relative to the expected final waypoint. In all of the figures below the x axis is drawn vertical (pointing north / forward) and the y axis horizontal, following the body frame of the chassis at launch.

Test 1: 100 cm straight line

The robot was given a single waypoint at $(100, 0)$ from a starting pose of $(0, 0, 0)$. The path produced by the spline rebuild in this two-point case is a single straight segment. Ten runs were executed back-to-back, with the chassis returned to the origin marker by hand between runs and the lifecycle reset via an FB hold. Endpoint deviations relative to the $(100, 0)$ marker were $(-1, -3)$, $(1, -2)$, $(0, -2)$, $(-1, 2)$, $(-1, 1)$, $(2, -3)$, $(-1, -3)$, $(0, -2)$, $(-1, 1)$, $(0, 2)$ in centimetres.

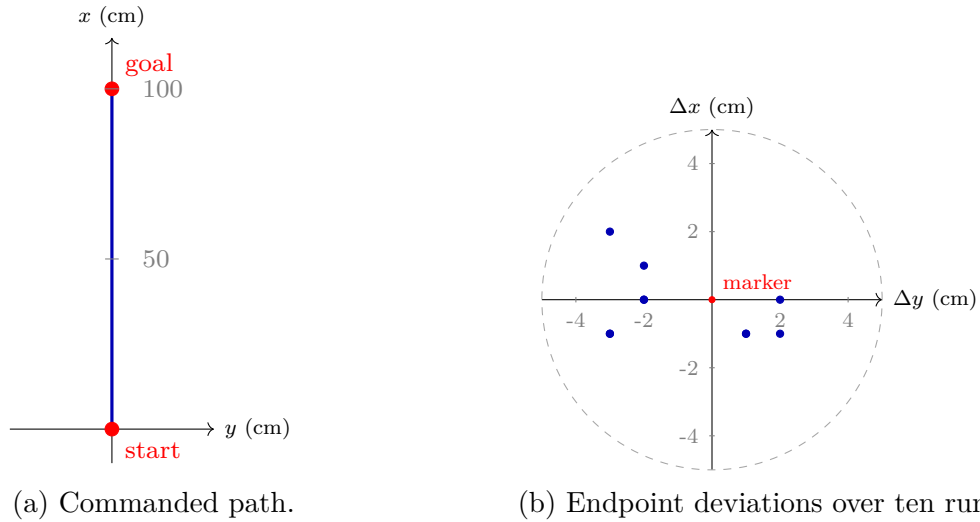
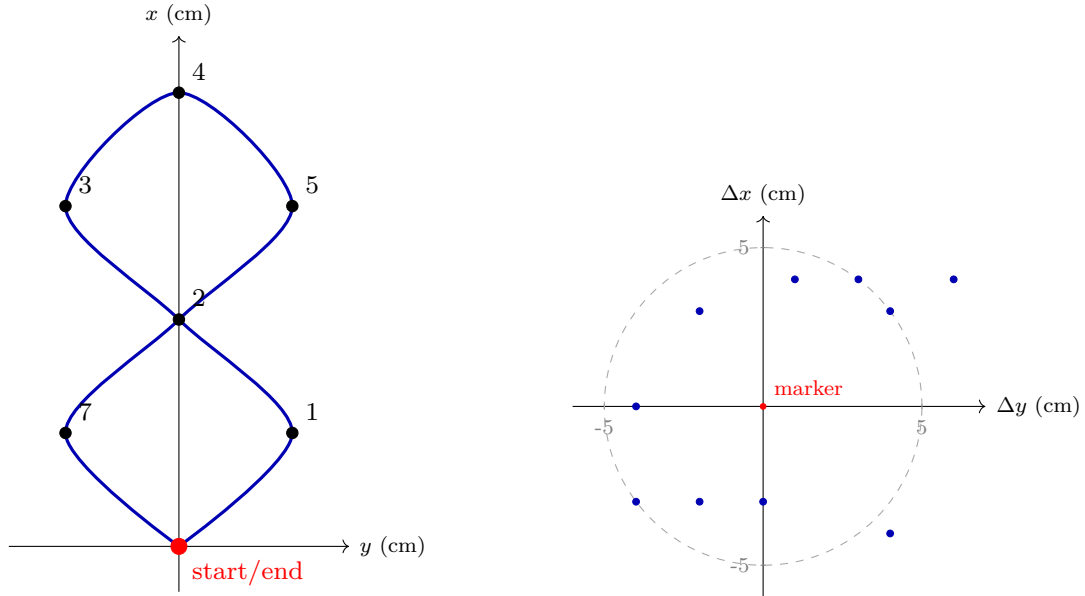


Figure 16.1: 100 cm straight-line test. Commanded path on the left, the ten observed endpoint deviations relative to the $(100, 0)$ marker on the right. The dashed circle is the ± 5 cm visual-measurement floor. x points up (forward / north).

Test 2: figure-eight, 200 cm wide

A figure-eight trajectory was constructed from eight waypoints: $(50, 50)$, $(100, 0)$, $(150, -50)$, $(200, 0)$, $(150, 50)$, $(100, 0)$, $(50, -50)$, $(0, 0)$. Starting and ending pose are

both $(0,0)$. The crossing point at $(100,0)$ is visited twice; the spline rebuild handles the repeated waypoint without special-casing. Ten runs were executed. Endpoint deviations relative to the final $(0,0)$ waypoint were $(-3,0)$, $(4,3)$, $(-3,-2)$, $(4,1)$, $(4,6)$, $(3,4)$, $(3,-2)$, $(-4,4)$, $(-3,-4)$, $(0,-4)$ in centimetres.

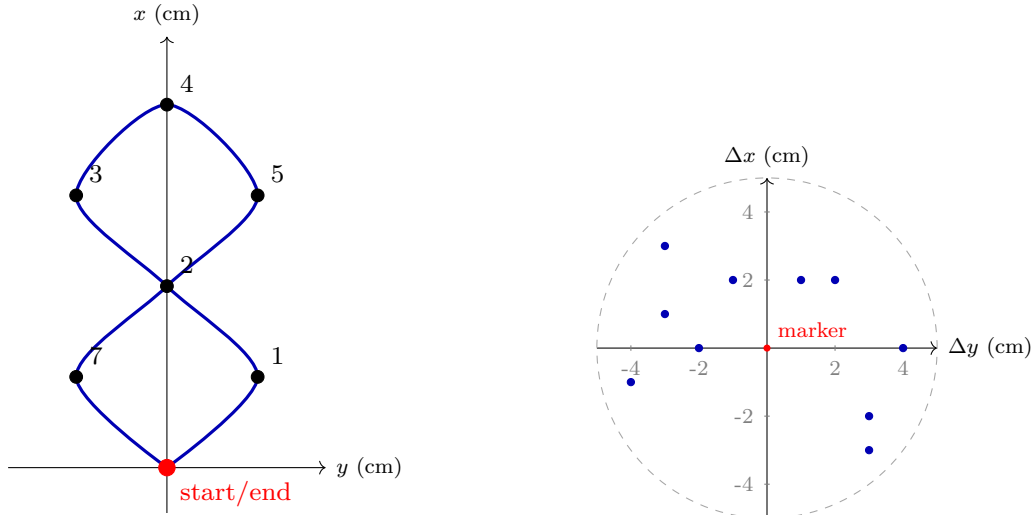


(a) Commanded path (numbered waypoints). (b) Endpoint deviations over ten runs.

Figure 16.2: Figure-eight test at 200 cm full width. Commanded path on the left; the ten observed endpoint deviations relative to the $(0,0)$ return marker on the right. The dashed circle is the ± 5 cm visual-measurement floor. x points up (forward / north).

Test 3: figure-eight, 80 cm wide

The same figure-eight shape was re-issued at 40 % scale, with waypoints $(20,20)$, $(40,0)$, $(60,-20)$, $(80,0)$, $(60,20)$, $(40,0)$, $(20,-20)$, $(0,0)$. The smaller width tightens every curvature on the path, which pushes the Pure Pursuit speed law of Eq. 7.5 further down its curve and increases the share of run time spent near the floor speed. Endpoint deviations relative to the final $(0,0)$ waypoint over ten runs were $(-2,3)$, $(0,-2)$, $(2,2)$, $(2,1)$, $(0,4)$, $(3,-3)$, $(1,-3)$, $(-3,3)$, $(-1,-4)$, $(2,-1)$ in centimetres.



(a) Commanded path (numbered waypoints). (b) Endpoint deviations over ten runs.

Figure 16.3: Figure-eight test at 80 cm full width. Commanded path on the left; the ten observed endpoint deviations relative to the $(0, 0)$ return marker on the right. The dashed circle is the ± 5 cm visual-measurement floor. x points up (forward / north).

Discussion

Across all three sets of tests every observed endpoint falls inside the ± 5 cm visual measurement floor of Section 11.4, with the single exception of one (4, 6) run on the 200 cm figure eight that lands 7.2 cm from the marker. The straight line spread is the tightest, as expected, since there are no curves to accumulate odometry slip over. The two figure eight tests spread slightly wider in both axes; the larger 200 cm shape and the smaller 80 cm shape produce comparable spreads, which is what the curvature-modulated speed law of Chapter 7 is supposed to achieve, since the slower cruise on the tighter curves of the 80 cm shape trades run time for tracking margin.

The accuracy is judged as satisfactory for the path lengths targeted. Considered against the size of the chassis itself, the deviations are smaller than the wheelbase of 18.5 cm and substantially smaller than the chassis footprint. A deviation of a few centimetres at the marker means the origin point is still under the chassis on every run and the robot is sitting over the goal, just not centred on it to the millimetre.

16.3 LED signals

Two onboard RGB LEDs on the GRiSP-2 board are driven directly from the main loop and act as the only on-robot indicator of system state. They were used throughout the testing reported above as a quick visual check of lifecycle and incoming protocol activity. LED 1 (left) carries system status; LED 2 (right) carries the trajectory sub-phase. Both are written through `grisp_led:color/2`; the named colour states live in `led_control.erl`.

Base states

Phase	LED 1	LED 2	Meaning
Booting / accel cal	red	red	IMU accel calibration in progress; not safe to operate.
Idle (post-boot)	green	green	Lifecycle idle ; F+B tap arms a queued path.
Trajectory running	green	cyan	Lifecycle running ; actively following the spline.
Trajectory paused	green	yellow	Lifecycle paused ; tap to resume, hold to reset.
Reset cooldown (1 s)	green	blue	Inside <code>TRAJ_RESET_COOLDOWN_MS</code> ; input gated.

Table 16.1: Base lifecycle LED states. RGB triples as written to `grisp_led:color/2`: red $\{1, 0, 0\}$, green $\{0, 1, 0\}$, cyan $\{0, 1, 1\}$, yellow $\{1, 1, 0\}$, blue $\{0, 0, 1\}$.

Command-RX overrides

When the waypoint injection protocol of Chapter 10 is in session, LED 1 is overridden by the command RX state machine while LED 2 keeps showing the lifecycle. Both pieces of information are therefore visible simultaneously. During a trajectory run with an incoming protocol session, LED 1 shows yellow while LED 2 shows cyan, and a successful frame commit briefly flashes LED 1 white while LED 2 remains cyan.

Phase	LED 1	Meaning
cmd-rx active (sticky, FSM $\neq$ idle)	yellow	Protocol session in progress over I2C.
cmd-rx frame OK (brief pulse)	white	Frame received and parsed cleanly.
cmd-rx committed (≈ 250 ms)	white	Received batch committed to the waypoint queue.
cmd-rx protocol error (≈ 500 ms)	red	Frame rejected (bad header, CRC, or sequence).

Table 16.2: Command-RX overrides on LED 1. LED 2 is not touched by these transitions and continues to carry the lifecycle colour from Table 16.1.

The `frame_ok` and `committed` states share the same white pulse and are not visually distinguishable but they are kept as separate FSM events for logging. The red/red startup state appears only once at boot and is not re entered later, so any subsequent solid red on LED 1 indicates a `cmd_error` pulse rather than a calibration phase.

Part IV

Discussion and Conclusion

Chapter 17

Discussion

17.1 The four-piece decomposition in retrospect

The split in Chapter 4 into planning, spline, tracker, and pose was the right framing for this project. Three of the four pieces had answers that were already obvious from the literature and were only sized for this platform.

Catmull-Rom (Chapter 6) was used because the path is required to pass through every added waypoint without extra tuning. Pure Pursuit (Chapter 7) was used because its lookahead-and-curvature formulation tolerates the slow, balance-constrained turn rates the robot uses. The one-byte wire protocol (Chapter 10) was shaped by the inherited I²C communication line and the constraint to keep the LilyGo firmware unchanged, and did the only job it was asked to do.

The fourth piece, pose, was the one that required real iteration. Three estimators were built in Part III, the simplest was used, and the other two were retired. The division was correct in calling out pose as the open question.

17.2 What the localization journey actually taught

The localization comparison ended in a clean, positive result. Wheel odometry and integrated gyro agreed within about 3° RMS across the full square loop and finished under one degree apart on both directional runs (Chapter 13). Both channels were measured to fall inside the $\pm 5^\circ$ floor of the floor-marker rig (Chapter 15).

The targets were sized so that the marker lay within the robot's own footprint at the end of the path. That target was hit on every run reported in Chapter 16.

The journey itself was useful. Two independent rotation channels were shown to report essentially the same thing at this hardware scale, which is what made the pose decision easy. The magnetometer was tried, calibrated through three iterations (in-place spin, 3-axis hand-tumble, tilt-compensated compass), and judged against the same rig. It was not kept, but the experiment closed the question rather than leaving it open.

The wider point that travels past this thesis is that at the run lengths and footprint sizes this robot is asked to operate at, wheel odometry alone is enough to land the

chassis on the marker.

17.3 Erlang at the measured loop rate, in hindsight

The Erlang runtime on the GRiSP was inherited from the previous thesis [9] and was not modified. The trajectory layer was added on top of the existing OTP supervisor without changing the supervision tree.

The new modules are stateless functions called once per tick from `main_loop`. The only state record that crosses module boundaries is the shared `#pose{}`. Nothing in the hot path uses message passing: state is threaded through the loop as arguments and a recursive tail call closes the loop.

That design is the reason the measured 180 Hz loop holds its budget even when a spline is being rebuilt mid-cruise on top of the per-tick balance and trajectory work. The lesson is that Erlang, used this way, is a workable runtime for soft-real-time control at the tick rates this robot operates at.

17.4 Levers not pulled

No fusion on the kept pose channel. A complementary filter or Kalman fusion of wheel odometry with the integrated gyro was not added. Chapter 13 showed that the two channels already agree within the measurement floor of the experiment, so any non-trivial weight on the gyro pulls the fused estimate away from a source it already matches. There is no second signal worth fusing.

No magnetometer in the final build. The added per-tick SPI cost of the magnetometer read dropped the measured main-loop rate from about 180 Hz to about 130 Hz (Chapter 14). That 50 Hz regression is paid by the inner balance loop, whose tilt PID, velocity ramp, and pose update all run at the tick rate. At the lower rate the cart was visibly less stable during straight cruising. The I/O cost alone, independent of any remaining calibration questions, was the dominant reason the magnetometer was not kept.

Chapter 18

Future Work

18.1 Where this work sits in a longer chain

This thesis is one link in a sequence of master theses on the GRiSP and Erlang platform supervised at UCLouvain. The Hera framework for fault-tolerant sensor fusion was introduced and then matured across several iterations [7], [10], [11], [12], [13], [14]. The chassis and the balance cascade used here were built by Goens and Ponsard [9], and indoor robot localization for domestic automation was investigated in parallel by Daube and Vanbever, whose work on responsibility handover of a mobile robot inside a building runs on the same platform family [71].

The contributions reported in this thesis sit on top of that line. A one-byte wire protocol was added so that waypoints can be uploaded, edited, and cleared over the inherited I²C link (Chapter 10). Three onboard localization channels (wheel odometry, integrated gyro, tilt-compensated magnetometer) were built and compared on the same rig (Part III). On top of that, a smooth and accurate trajectory-following layer was delivered without modifying the inherited balance cascade (Part II). The combined result is a working trajectory stack and a closed answer to the pose question for this class of indoor robot, which is the stage on which the next round of work can stand.

18.2 A second robot and shared UWB localization

The clearest next step is a second physical robot of the same kind. The wire protocol, the lifecycle FSM, and the trajectory stack were all written for a single chassis, but the data structures and the I²C frame layout have no inherent multi-robot assumption. A second chassis would let coordinated paths, follow-the-leader experiments, and shared-waypoint workflows be measured rather than only speculated about.

In parallel with this thesis, other students are finishing work on an Ultra-Wide-Band (UWB) sensor for the same platform family. UWB returns ranging measurements between tagged nodes that are far more accurate than what wheel odometry alone can provide for absolute position. Once that hardware is available, two extensions become possible. First, a UWB anchor in the room can be used as an external reference against which the wheel-odometry pose is corrected, which would close the only open weakness identified in Chapter 13: the unbounded drift of a dead-reckoned channel over long

runs. Second, two robots carrying UWB tags can measure the range between each other, which gives a relative-localization channel that does not depend on any fixed anchor in the room. Combined with the existing waypoint protocol, that range channel would let the two chassis share and update their target queues while keeping their own pose estimates honest.

18.3 Pushing the existing tuning further

The current cruise speed and curvature schedule were chosen conservatively so that the balance loop stays inside its comfortable region while the trajectory layer drives. The same knobs that were used to make the robot safe can be used to make it faster.

The curvature-modulated speed law in Chapter 7 has an explicit knee at $|\kappa| = 1/b$ and a floor of 1 cm/s. Both numbers were picked from observed behaviour, not derived from a saturation limit of the balance cascade. Raising the floor, moving the knee, or raising the cruise speed are all parameter changes rather than design changes. The braking distance constant in Chapter 8 ($d_{\text{brake}} = v^2/(2a) + 2.5 \text{ cm}$) was sized the same way and can be tightened once the balance cascade has been re-characterised at a higher cruise speed. The lookahead of 8 cm and the 30-sub-points-per-segment spline density in Chapter 6 were chosen for the current speed and can be revisited at higher ones. None of these knobs require a new module; they only require a campaign of runs against the floor-marker rig already used in Chapter 16.

18.4 Hardware upkeep

Two physical-side items were noticed during the experimental campaign and are left here for the next person.

The rubber on both wheels is worn and visibly cracked along the contact band, showing the age of the chassis. It still grips well enough on the tested floor for the runs reported in Chapter 16, and no visible slip was detectable in the wheel-odometry traces, so it was left in place for this work. For longer or faster runs the rubber should be replaced before the contact patch starts to glaze or lose grip on the same surface, since any slip event would feed directly into the odometry channel that this thesis kept as the sole pose source.

The current battery pack is sized for short experimental runs. As the cell voltage drops, the balance cascade was observed to become noticeably less stable; the chassis sits less still, and small oscillations grow. Because the kept pose channel is integrated wheel odometry, those extra micro-corrections show up as added heading drift, and in the worst sessions the marker target was approached with a visibly wider final-pose spread than at full charge. A larger pack, or a low-battery cutoff that ends the trajectory cleanly rather than letting it degrade, would extend the usable session length and keep localization accuracy within the bounds reported in Part III.

Chapter 19

Conclusion

A trajectory-following layer was added on top of an inherited two-wheeled balancing robot built on GRiSP-2 and Erlang/OTP. The work was split into four pieces in Chapter 4: a planning interface, a path representation, a closed-loop tracker, and an onboard pose estimator. Three of those pieces drew directly on established methods, sized to this platform. The fourth, pose, was treated as the open question of the project and was answered by building three estimators in Part III and keeping only the simplest one that hit the target.

The path layer uses Catmull-Rom splines that pass through every operator-given waypoint, followed by Pure Pursuit with a fixed 8 cm lookahead and a curvature-modulated speed schedule. A short three-state sub-machine (cruise, braking, dwelling) brings the chassis to rest at the final waypoint. A one-byte wire protocol was added so that waypoints can be queued, edited, and cleared over the inherited I²C link without modifying the LilyGo firmware. Lifecycle, pose, LEDs, and the velocity mux are owned by a single orchestrator, and every reset event is funnelled through one helper so that the idle state has exactly one definition.

On the localization side, wheel odometry and integrated gyro were measured to agree within about 3° RMS across a full square loop, both falling inside the $\pm 5^\circ$ floor of the floor-marker rig (Chapter 15). The magnetometer was built, calibrated through three iterations, and then retired because the added per-tick SPI cost dropped the main-loop rate from about 180 Hz to about 130 Hz and the inner balance loop was visibly less stable at the lower rate. The final pose channel is wheel odometry alone, and the chassis landed on its target marker on every reported run in Chapter 16.

The contribution of this thesis is therefore a readable trajectory stack and a closed evaluation of three onboard localization channels on a balancing robot, both delivered without modifying the inherited balance cascade.

Bibliography

- [1] M. B. Alatise and G. P. Hancke, “A review of sensing technologies for indoor autonomous mobile robots,” *Sensors*, 2024. DOI: 10.3390/s24041172. [Online]. Available: <https://pmc.ncbi.nlm.nih.gov/articles/PMC10893033/>.
- [2] RobotsGuide, *Handle robot*, 2024. [Online]. Available: <https://robotsguide.com/robots/handle>.
- [3] S. Inc., *About us - segway official store*, <https://store.segway.com/about-us>, 2023.
- [4] C. Robotics, *Chronus robotics - kim-e, the self-balancing personal mobility robot*, <https://chronusrobotics.com/>, 2024.
- [5] D. P. P. S. GmbH, *GRiSP*. [Online]. Available: <https://www.grisp.org/>.
- [6] J. Armstrong, “Erlang—a survey of the language and its industrial applications,” in *Proc. INAP*, vol. 96, 1996, pp. 16–18.
- [7] S. Kalbusch, V. Verpoten, and P. Van Roy, “The Hera framework for fault-tolerant sensor fusion with Erlang and GRiSP on an IoT network,” en, in *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, Virtual Republic of Korea: ACM, Aug. 2021, pp. 15–27, ISBN: 9781450386128. DOI: 10.1145/3471871.3472962. [Online]. Available: <https://dl.acm.org/doi/10.1145/3471871.3472962>.
- [8] Various, “Advancing indoor positioning systems: Innovations, challenges, and applications in mobile robotics,” *Robotica*, 2024. [Online]. Available: <https://www.cambridge.org/core/journals/robotica/article/advancing-indoor-positioning-systems-innovations-challenges-and-applications-in-mobile-robotics/179610C4B105839F930B00F8D3EBD7A4>.
- [9] F. Goens and C. Ponsard, “Dynamic balancing in the real world with GRiSP: Two-wheeled self-balancing robot with Hera,” Master’s thesis, Ecole polytechnique de Louvain, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2024.
- [10] N. Guillaume and B. Julien, “Sensor fusion at the extreme edge of an internet of things network,” Ecole polytechnique de Louvain, Université catholique de Louvain, 2020, Van Roy, Peter, Prom.
- [11] K. Sébastien and V. Vincent, “The hera framework for fault-tolerant sensor fusion on an internet of things network with application to inertial navigation and tracking,” Ecole polytechnique de Louvain, Université catholique de Louvain, 2021, Van Roy, Peter, Prom.

- [12] L. Tanguy, “Numerl : Efficient vector and matrix computation for erlang,” Ecole polytechnique de Louvain, Université catholique de Louvain, 2022, Van Roy, Peter, Prom.
- [13] N. Lucas, “Low-cost high-speed sensor fusion with grisp and hera,” Ecole polytechnique de Louvain, Université catholique de Louvain, 2023, Van Roy, Peter, Prom.
- [14] I. Nicolas, “Controlling everyday life objects using the hera platform on GRiSP2.0: The movement_detection application,” Ecole polytechnique de Louvain, Université catholique de Louvain, 2024, Van Roy, Peter, Prom.
- [15] Various, “Robot localization: A comprehensive survey from classical methods to intelligent autonomy,” *International Journal of Intelligent Robotics and Applications*, 2026. DOI: 10.1007/s41315-026-00528-9.
- [16] E. Catmull and R. Rom, “A class of local interpolating splines,” in *Computer Aided Geometric Design*, R. E. Barnhill and R. F. Riesenfeld, Eds., New York: Academic Press, 1974, pp. 317–326.
- [17] R. C. Coulter, “Implementation of the Pure Pursuit path tracking algorithm,” Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-92-01, 1992. [Online]. Available: https://www.ri.cmu.edu/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf.
- [18] D. P. P. S. GmbH, *Stritzinger.com — Home page*. [Online]. Available: <https://www.stritzinger.com/>.
- [19] Espressif, *Esp32 pico series datasheet*, 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-pico_series_datasheet_en.pdf.
- [20] Digilent Inc., *Pmod nav*, 2024. [Online]. Available: <https://digilent.com/reference/pmod/pmodnav/start>.
- [21] “Ieee standard for floating-point arithmetic,” *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.
- [22] M. Salwa and I. Krzysztófik, “Application of filters to improve flight stability of rotary unmanned aerial objects,” *Sensors*, vol. 22, p. 1677, Feb. 2022. DOI: 10.3390/s22041677.
- [23] M. N. Öz, S. Budak, E. Kurnaz, and A. Durdu, “Orientation Determination in IMU Sensor with Complementary Filter,” *Turkish Journal of Forecasting*, vol. 06, no. 1, pp. 34–39, Aug. 2022, ISSN: 2618-6594. DOI: 10.34110/forecasting.1126184. [Online]. Available: <http://dergipark.org.tr/en/doi/10.34110/forecasting.1126184>.
- [24] RTEMS Project, *RTEMS Real-Time Operating System*, <https://www.rtems.org/>, 2024.
- [25] Digilent Inc., *Digilent Pmod™ Interface Specification*, Oct. 2020. [Online]. Available: <https://digilent.com/reference/pmod/start>.
- [26] R. project, <https://github.com/rosie-project>, 2022.
- [27] M. Quigley et al., “Ros: An open-source robot operating system,” in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3, IEEE, 2009, pp. 5–10. [Online]. Available: <https://www.ros.org/>.

- [28] Orocos Project, *Orocos: Open robot control software*, 2024. [Online]. Available: <https://orocos.org/>.
- [29] MoveIt Development Team, *Moveit: The motion planning framework for ros*, 2024. [Online]. Available: <http://moveit.ros.org/>.
- [30] Robot Framework Foundation, *Robot framework*, 2024. [Online]. Available: <https://robotframework.org/>.
- [31] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, “Rt-middleware: Distributed component middleware for rt (robot technology),” in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 3933–3938. DOI: 10.1109/IROS.2005.1545521.
- [32] Thymio Team, *Thymio robot official website*, 2024. [Online]. Available: <https://www.thymio.org/>.
- [33] B. Benavides. “‘erlang behaviors’ erlang battleground. ”[Online]. Available: <https://medium.com/erlang-battleground/erlang-behaviors-4348e89351ff>.
- [34] MeetiBOT, *Meet ibot - your personal mobility device*, <https://meetibot.com/>, 2023.
- [35] F. I. for Material Flow and L. (IML), *Evobot - the evolution of autonomous mobile robotic systems*, https://www.inkl.fraunhofer.de/en/fields_of_activity/material-flow-systems/iot-and-embedded-systems/evobot.html, 2024.
- [36] P. Corporation, *Balboa 32u4 balancing robot*, <https://www.pololu.com/product/3575>, 2024.
- [37] O. M. Mohamed Gad, S. Z. M. Saleh, M. A. Bulbul, and S. Khadraoui, “Design and control of two wheeled self balancing robot (twabr),” in *2022 Advances in Science and Engineering Technology International Conferences (ASET)*, 2022, pp. 1–6. DOI: 10.1109/ASET53988.2022.9735004.
- [38] W. Vega et al., “Conceptual mechatronics design and prototyping of autonomous inverted pendulum-system applied on two-wheeled mobile robot,” in *2023 Third International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, 2023, pp. 1–6. DOI: 10.1109/ICAECT57570.2023.10118071.
- [39] S. Kim and K. Yeom, “Development of a hand-fan-shaped arm and a model predictive controller for leg crossing, walking, and one-legged balancing of a wheeled-bipedal jumping robot,” *Machines*, vol. 12, no. 5, 2024, ISSN: 2075-1702. DOI: 10.3390/machines12050284. [Online]. Available: <https://www.mdpi.com/2075-1702/12/5/284>.
- [40] C.-C. Tsai, W.-T. Hsu, F.-C. Tai, and S.-C. Chen, “Adaptive motion control of a terrain-adaptive self-balancing leg-wheeled mobile robot over rough terrain,” in *2022 International Automatic Control Conference (CACs)*, 2022, pp. 1–6. DOI: 10.1109/CACS55319.2022.9969857.
- [41] N. Hasanah et al., “Adaptive motion control of two-wheeled robot based on complementarity filter and fuzzy logic controller,” *International Journal of Robotics and Automation*, vol. 9, no. 1, pp. 53–60, 2020. DOI: 10.11591/ijres.v9.i1.pp53-60. [Online]. Available: <https://ijres.iaescore.com/index.php/IJRES/article/view/20676/pdf>.

- [42] M. Han, K. Kim, D. Y. Kim, and J. Lee, “Implementation of unicycle segway using unscented kalman filter in lqr control,” in *2013 10th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, 2013, pp. 695–698. DOI: 10.1109/URAI.2013.6677427.
- [43] A. Petrovsky, I. Kalinov, P. Karpyshev, D. Tsetserukou, A. Ivanov, and A. Golkar, “The two-wheeled robotic swarm concept for mars exploration,” *Acta Astronautica*, vol. 194, pp. 1–8, 2022, ISSN: 0094-5765. DOI: <https://doi.org/10.1016/j.actaastro.2022.01.025>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0094576522000340>.
- [44] LEGO Education, *Ev3 program description: Gyroboy*, 2015. [Online]. Available: <https://assets.education.lego.com/v3/assets/blt293eea581807678a/blt6442d49d724f3570/5f8803d01c5db60f7d0ae38e/ev3-program-description-gyroboy.pdf?locale=en-us>.
- [45] T. H. Hughes, G. H. Willetts, and J. A. Kryczka, “Lqg controller for the lego mindstorms ev3 gyroboy segway robot,” *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 17 282–17 287, 2020, 21st IFAC World Congress, ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2020.12.1811>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2405896320324216>.
- [46] STMicroelectronics, *Hardware developer kit*, 2016. [Online]. Available: https://www.mikrocontroller.net/attachment/338591/hardware_developer_kit.pdf.
- [47] G. A. Muñoz-Hernandez, J. Díaz-Téllez, J. Estevez-Carreón, and R. S. García-Ramírez, “Adrc attitude controller based on ros for a two-wheeled self-balancing mobile robot,” *IEEE Access*, vol. 11, pp. 94 636–94 646, 2023. DOI: 10.1109/ACCESS.2023.3308948.
- [48] R. Renaud, *Lecture notes in robot modelling and control lelme2732*, 2022.
- [49] R. Weber, *Optimization and control*, 2010.
- [50] L. Piegl and W. Tiller, *The NURBS Book*, 2nd. Springer, 1997. DOI: 10.1007/978-3-642-59223-2.
- [51] C. de Boor, *A Practical Guide to Splines*. Springer-Verlag, 1978.
- [52] E. Bertolazzi and M. Frego, “G1 fitting with clothoids,” *Mathematical Methods in the Applied Sciences*, vol. 38, no. 5, pp. 881–897, 2015. DOI: 10.1002/mma.3114.
- [53] L. E. Dubins, “On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents,” *American Journal of Mathematics*, vol. 79, no. 3, pp. 497–516, 1957. DOI: 10.2307/2372560.
- [54] J. A. Reeds and L. A. Shepp, “Optimal paths for a car that goes both forwards and backwards,” *Pacific Journal of Mathematics*, vol. 145, no. 2, pp. 367–393, 1990. DOI: 10.2140/pjm.1990.145.367.
- [55] R. L. Burden and J. D. Faires, *Numerical Analysis*, 9th. Brooks/Cole, 2010.
- [56] S. Thrun et al., “Stanley: The robot that won the DARPA grand challenge,” *Journal of Field Robotics*, vol. 23, no. 9, pp. 661–692, 2006. DOI: 10.1002/rob.20147.

- [57] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli, “Kinematic and dynamic vehicle models for autonomous driving control design,” in *2015 IEEE Intelligent Vehicles Symposium (IV)*, 2015, pp. 1094–1099. DOI: 10.1109/IVS.2015.7225830.
- [58] P. Falcone, F. Borrelli, J. Asgari, H. E. Tseng, and D. Hrovat, “Predictive active steering control for autonomous vehicle systems,” *IEEE Transactions on Control Systems Technology*, vol. 15, no. 3, pp. 566–580, 2007. DOI: 10.1109/TCST.2007.894653.
- [59] B. D. O. Anderson and J. B. Moore, *Optimal Control: Linear Quadratic Methods*. Dover Publications, 2007.
- [60] J. Wit, C. D. Crane, and D. Armstrong, “Autonomous ground vehicle path tracking,” in *Journal of Robotic Systems*, vol. 21, 2004, pp. 439–449. DOI: 10.1002/rob.20031.
- [61] Various, “AI-based approaches for improving autonomous mobile robot localization in indoor environments: A comprehensive review,” *Robotics and Computer-Integrated Manufacturing*, 2025. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2215098625000321>.
- [62] J. Borenstein, H. R. Everett, and L. Feng, *Where am I? Sensors and Methods for Mobile Robot Positioning*. Ann Arbor, MI: The University of Michigan, 1996. [Online]. Available: https://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/integrated2/borens_pos_book.pdf.
- [63] J. Borenstein and L. Feng, “Measurement and correction of systematic odometry errors in mobile robots,” *IEEE Transactions on Robotics and Automation*, vol. 12, no. 6, pp. 869–880, 1996. DOI: 10.1109/70.544770.
- [64] D. H. Titterton and J. L. Weston, *Strapdown Inertial Navigation Technology*, 2nd. IET, 2004.
- [65] NXP Semiconductors, *Calibrating an eCompass in the presence of hard- and soft-iron interference*, Application Note AN4246, Rev. 4, 2015. [Online]. Available: <https://www.nxp.com/docs/en/application-note/AN4246.pdf>.
- [66] VectorNav, *Magnetometer hard- and soft-iron calibration*, Inertial Navigation Primer, 2024. [Online]. Available: <https://www.vectornav.com/resources/inertial-navigation-primer/specifications--and--error-budgets/specs-hsicalibration>.
- [67] J. M. Snider, “Automatic steering methods for autonomous automobile path tracking,” Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-09-08, 2009.
- [68] O. J. Woodman, “An introduction to inertial navigation,” in *Technical Report UCAM-CL-TR-696*, University of Cambridge Computer Laboratory, 2007. [Online]. Available: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-696.pdf>.
- [69] X. Niu, Y. Li, H. Zhang, Q. Wang, and Y. Ban, “Fast thermal calibration of low-grade inertial sensors and inertial measurement units,” *Sensors*, vol. 13, no. 9, pp. 12 192–12 217, 2013. DOI: 10.3390/s130912192.
- [70] V. Renaudin, M. H. Afzal, and G. Lachapelle, “Complete triaxis magnetometer calibration in the magnetic domain,” *Journal of Sensors*, vol. 2010, p. 967 245, 2010. DOI: 10.1155/2010/967245.

- [71] N. Daube and T. Vanbever, “Practical domotics using sensor fusion: Responsibility handover of a mobile robot in a building,” Master’s thesis, Ecole polytechnique de Louvain, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2025.

Part V

Appendices

Appendix A

Code Listings

The full source for the system is reproduced in this appendix. It is grouped into five sections: the Erlang code that runs on the GRiSP 2 board, the Python logger that captures the serial telemetry stream, the Python remote-control and waypoint-planner UI, and the two Arduino sketches that run on the robot-side and operator-side LilyGo LoRa32 boards. The wire protocol between the GRiSP and the operator-side LilyGo is documented in Appendix B.

A.1 GRiSP code (Erlang)

A.1.1 balancing_robot.erl

```
1 -module(balancing_robot).
2
3 -behavior(application).
4
5 -export([start_robot/0]).
6 -export([start/2, stop/1]).
7
8 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
9 %% API
10 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
11
12 start_robot() ->
13     timer:sleep(5000), %Waiting for setup of GRiSP application before launching the robot
14     hera:start_measure(hera_interface, []),
15     ok.
16
17 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
18 %% Callbacks
19 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20
21 start(_Type, _Args) ->
22     {ok, Supervisor} = balancing_robot_sup:start_link(),
23     grisp_led:color(1, {1, 0, 1}),
24     grisp_led:color(2, {1, 0, 1}),
25     _ = grisp:add_device(spi2, pmod_nav),
26     pmod_nav:config(acc, #{odr_g => {hz,238}}),
27     numerl:init(),
28     Pid = spawn(balancing_robot, start_robot, []),
29     {ok, Supervisor}.
30
31 stop(_State) -> ok.
```

A.1.2 balancing_robot_sup.erl

```

1 % @private
2 % @doc balancing_robot top level supervisor.
3 -module(balancing_robot_sup).
4
5 -behavior(supervisor).
6
7 % API
8 -export([start_link/0]).
9
10 % Callbacks
11 -export([init/1]).
12
13 %--- API -----
14
15 start_link() -> supervisor:start_link({local, ?MODULE}, ?MODULE, []).
16
17 %--- Callbacks -----
18
19 init([]) -> {ok, { {one_for_all, 0, 1}, [] } }.

```

A.1.3 hera_interface.erl

```

1 -module(hera_interface).
2
3 -behavior(hera_measure).
4
5 -export([init/1, measure/1]).
6
7 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
8 %% Callbacks
9 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
10
11 init(_Args) ->
12   Spec = #{name=>kaltest, iter=>infinity, sync=>false},
13   T0 = erlang:system_time()/1.0e6,
14   Robot_Pid = spawn(main_loop, robot_init, [self()]),
15   %
16   Names = "T,Freq,Gy,Acc,CtrlByte,AngleAccelerometer,AngleKalman,AngleComplem,Vref,Select,Adv_Ref,
17           Turn_Ref,Speed",
18   Len = 13,
19
20   io:format("[Hera] Pid of the Hera interface: ~p.~n", [self()]),
21   grisp_led:color(1, {1, 1, 0}),
22   grisp_led:color(2, {1, 1, 0}),
23
24   {ok, {Robot_Pid, {Names, Len, 1}, T0, 0, 0, 200, T0}, Spec}.
25
26 measure({Robot_Pid, {Names, Len, File_Number}, StartTime, N, Freq, Mean_Freq, T0}) ->
27
28   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29   %% Interface with the robot Features %%%
30   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
31
32   Robot_Pid ! {self(), get_all_data},
33   receive {_, data, Robot_Data} -> ok end,
34
35   T1 = erlang:system_time()/1.0e6,
36   Dt = (T1- T0)/1000.0,
37   % if
38   %   N == 10000 ->
39   %     % io:format("[Hera] freq: ~.2f~n", [Freq]),
40   %     N_New = 0,
41   %     Freq_New = 0,
42   %     Mean_Freq_New = Freq;
43   %   true ->
44   %     N_New = N+1,
45   %     Freq_New = ((Freq*N)+(1/Dt))/(N+1),
46   %     Mean_Freq_New = Mean_Freq
47   % end,

```

```

47 {N_New, Freq_New, Mean_Freq_New} = {0.0, 0.0, 0.0},
48
49 %%%%%%%%%%%
50 %% Logging %%
51 %%%%%%%%%%%
52
53 File_Name = "measures"++integer_to_list(File_Number)+"_txt",
54
55 %Retrieve start and end of logging sequence
56 receive
57   {_, data, Robot_Data} ->
58     Start_Log = false,
59     Stop_Log = false;
60   {_, start_log} ->
61     Start_Log = true,
62     Stop_Log = false;
63   {_, stop_log} ->
64     Start_Log = false,
65     Stop_Log = true
66 after 0 ->
67   Start_Log = false,
68   Stop_Log = false
69 end,
70
71 %Start of logging sequence: create new file to write data in.
72 if
73   Start_Log ->
74     io:format("[Hera] Start logging~n"),
75     log_init(Names, File_Name);
76   true ->
77     ok
78 end,
79
80 %End of logging sequence: retrieve data and write to file.
81 if
82   Stop_Log ->
83     File_Number_New = File_Number+1,
84     io:format("[Hera] New log file with name: ~s~n",[File_Name]),
85     log_init(Names, File_Name),
86     grisp_led:color(1, {1, 0, 0}),
87     Robot_Pid ! {self(), log_values},
88     receive {_, log, Log_Values} -> ok end,
89     log(Log_Values, File_Name, Len),
90     grisp_led:color(1, {1, 1, 0}),
91     io:format("[Hera] Done writing to file!~n");
92   true ->
93     File_Number_New = File_Number
94 end,
95
96 {ok, [0,1], {Robot_Pid, {Names, Len, File_Number_New}, StartTime, N_New, Freq_New, Mean_Freq_New, T1}}
97
98
99 %%%%%%%%%%%
100 %% API
101 %%%%%%%%%%%
102
103 %
104 %Initialise the log file by writing the header
105 %
106 log_init(Names, File) ->
107   file:write_file(File, io_lib:fwrite("~s~n", [Names])).
108
109 %
110 %Append the values to the log file
111 %
112 log([], _, _) -> ok;
113 log(Val_List, File, Len) ->
114   [Sub_Vals | Rest] = Val_List,
115   % Sub_Vals = lists:sublist(Val_List,1,Len),
116   Vals = lists:map(fun(V) -> lists:flatten(io_lib:format("~p", [V])) end, Sub_Vals),
117   S = string:join(Vals, ","),
118   Bytes = io_lib:format("~s~n", [S]),

```

```

119 % io:format("writing file-n"),
120 ok = file:write_file(File, Bytes, [append]),
121 log(Rest,File,Len).

```

A.1.4 main_loop.erl

```

1 -module(main_loop).
2
3 %%% =====
4 %%% Main robot control loop -- orchestrator.
5 %%%
6 %%% Owns:
7 %%%   • Pose (#pose{}) -- updated every tick via odometry:integrate/3.
8 %%%   • Trajectory lifecycle state machine (idle/running/paused).
9 %%%   • Button edge/hold detection and cooldown timers.
10 %%%   • LED display.
11 %%%   • Velocity mux (trajectory vs manual).
12 %%%
13 %%% Lifecycle state machine (owns here):
14 %%%
15 %%% FB_Tap = F+B released after a *short* press (held < 500 ms).
16 %%% FB_Hold ≥ 500 ms always wins on the same gesture -- the tap is suppressed
17 %%% when the press crosses the hold threshold, so a single long press cannot
18 %%% chain reset → tap action.
19 %%%
20 %%% Every → idle transition (hold, fall, traj-done, cancel-drains, CLEAR_ALL)
21 %%% wipes pose to {0,0,0}, clears wps_remaining AND wps_traversed, and stamps
22 %%% the 1 s reset cooldown. Idle is the single terminal state -- there is no
23 %%% sticky "finished" state. Pre-flight queueing (ADD while idle) is allowed
24 %%% and survives a fall in idle.
25 %%%
26 %%% idle      --(FB_Tap, reset_cooldown_ok)--> running
27 %%% idle      --(FB_Hold ≥ 500 ms)--> idle (safety wipe + cooldown)
28 %%% running   --(FB_Tap, 500 ms cooldown)--> paused
29 %%% running   --(FB_Hold ≥ 500 ms)--> idle (wipe + cooldown)
30 %%% running   --(Robot_Up false)--> idle (wipe + cooldown, fall reset)
31 %%% running   --(traj_planner done)--> idle (wipe + cooldown)
32 %%% running   --(cancel drains wps_remaining)--> idle (wipe + cooldown)
33 %%% paused    --(FB_Tap, 500 ms cooldown)--> running
34 %%% paused    --(FB_Hold ≥ 500 ms)--> idle (wipe + cooldown)
35 %%% paused    --(Robot_Up false)--> idle (wipe + cooldown, fall reset)
36 %%% paused    --(cancel drains wps_remaining)--> idle (wipe + cooldown)
37 %%% *         --(CLEAR_ALL)--> idle (wipe + cooldown)
38 %%%
39 %%% Non-draining cancel (wps_remaining still non-empty after cancel_last /
40 %%% cancel_n) does NOT transition -- rebuild_path/1 runs, robot keeps cruising.
41 %%% =====
42
43 -export([robot_init/1, modify_frequency/1]).
44 -include("robot_types.hrl").
45
46 %% --- Feature flags -----
47 -define(ENABLE_TRAJECTORY, true).    %% trajectory layer
48 -define(USE_KALMAN, true).          %% true=Kalman tilt filter, false=complementary
49
50 %% --- Angles & velocity limits -----
51 -define(RAD_TO_DEG, 180.0/math:pi()).
52 -define(DEG_TO_RAD, math:pi()/180.0).
53 -define(ADV_V_MAX, 30.0).            %% cm/s -- manual forward cap
54 -define(TURN_V_MAX, 80.0).           %% cm/s -- manual turn cap
55 -define(COEF_FILTER, 0.667).
56
57 %% --- Button timing (ms) -----
58 -define(TRAJ_PAUSE_COOLDOWN_MS, 500). %% between pause/resume taps
59 -define(TRAJ_HOLD_MS, 500).           %% FB held this long → reset
60 -define(TRAJ_RESET_COOLDOWN_MS, 1000). %% after reset, FB ignored for this long
61
62 %% --- Legacy logging -----
63 -define(LOG_DURATION, 15000).
64
65 %% wp injection fsm - phase 2 just prints + leds, no traj stuff yet

```

```

66 -record(cmd_rx, {
67     state                = idle :: idle
68                         | want_x_hi | want_x_hi_to_lo
69                         | want_x_lo | want_x_to_y
70                         | want_y_hi | want_y_hi_to_lo
71                         | want_y_lo | want_commit
72                         | want_n    | want_n_commit,
73     x_acc                = 0      :: 0..1023,
74     y_acc                = 0      :: 0..1023,
75     n_acc                = 0      :: 0..31,
76     last_byte            = -1     :: integer(), %% raw byt last seen
77     stable_count         = 0      :: non_neg_integer(),
78     last_committed_byte = -1     :: integer(), %% byt we already accepted, ignore til it changes
79     led_pulse_kind       = none   :: none | cmd_frame_ok | cmd_committed | cmd_error,
80     led_pulse_left       = 0      :: non_neg_integer()
81 }).
82
83 %% --- Main state record -----
84 -record(rstate, {
85     pose                = #pose{} :: #pose{},
86     lifecycle           = idle     :: idle | running | paused,
87     lifecycle_entered_ms = 0       :: non_neg_integer(),
88
89     fb_held_ms          = 0        :: non_neg_integer(),
90     lr_held_ms          = 0        :: non_neg_integer(),
91
92     prev_fb_combo       = false    :: boolean(),
93     prev_lr_combo       = false    :: boolean(),
94
95     last_traj_action_ms = 0        :: non_neg_integer(),
96     last_reset_ms       = 0        :: non_neg_integer(),
97
98     traj_wps_left       = 0        :: non_neg_integer(),
99     traj_substate       = idle     :: atom(),
100    traj_counter         = 0        :: non_neg_integer(),
101
102    traj                 = undefined :: term(),
103
104    robot_state          = rest     :: atom(),
105    robot_up             = false    :: boolean(),
106
107    %% Latched on every PROTO=0 (drive) byte, replayed during PROTO=1 bursts so
108    %% protocol payloads cannot ghost-press Get_Up / F / B / L / R.
109    last_drive_byte      = 0        :: 0..255,
110
111    cmd_rx               = #cmd_rx{} :: #cmd_rx{}
112 }).
113
114 %%% =====
115 %%% Initialization
116 %%% =====
117
118 robot_init(Hera_pid) ->
119     process_flag(priority, max),
120     T0 = erlang:system_time() / 1.0e6,
121
122     ets:new(variables, [set, public, named_table]),
123     ets:insert(variables, {"Freq_Goal", 300.0}),
124
125     io:format("[Robot] Calibrating... Do not move the pmod_nav!~n"),
126     led_control:accel_calibrating(),
127     [Gx0, Gy0, _Gz0] = calibrate(),
128     io:format("[Robot] Gyro bias: Gx0=~.4f Gy0=~.4f dps~n", [Gx0, Gy0]),
129     io:format("[Robot] Done calibrating~n"),
130     led_control:idle(),
131
132     X0 = mat:matrix([[0], [0]]),
133     P0 = mat:matrix([[0.1, 0], [0, 0.1]]),
134     I2Cbus = grisp_i2c:open(i2c1),
135
136     Pid_Speed = spawn(pid_controller, pid_init, [-0.12, -0.07, 0.0, -1, 60.0, 0.0]),
137     Pid_Stability = spawn(pid_controller, pid_init, [17.0, 0.0, 4.0, -1, -1, 0.0]),
138     io:format("[Robot] Pid speed=~p stability=~p~n", [Pid_Speed, Pid_Stability]),

```

```

139 io:format("[Robot] Starting robot.\n"),
140
141 %% init traj at boot so the wire-protocol decoder always has a live
142 %% #traj_state{} to mutate, laptop can queue waypoints even while idle.
143 %% lifecycle_step no longer wipes this on transitions, only Clear_All over
144 %% the wire empties it (FB_Hold reset preserves the array now)
145 RS = #rstate{traj = traj_planner:init()},
146
147 robot_main(T0, Hera_pid, {T0, X0, P0}, I2Cbus,
148     {0, T0, []},
149     {Gy0, Gx0, 0.0, 0.0},
150     {Pid_Speed, Pid_Stability},
151     {0.0, 0.0},
152     {0, 0, 200.0, T0},
153     RS).
154
155 %% =====
156 %% Main loop
157 %% =====
158
159 robot_main(Start_Time, Hera_pid,
160     {T0, X0, P0}, I2Cbus,
161     {Logging, Log_End, Log_List},
162     {Gy0, Gx0, Angle_Complem, Angle_Rate},
163     {Pid_Speed, Pid_Stability},
164     {Adv_V_Ref, Turn_V_Ref},
165     {N, Freq, Mean_Freq, T_End},
166     RS) ->
167
168 %%--- 1. Time -----
169 T1 = erlang:system_time() / 1.0e6,
170 Dt = (T1 - T0) / 1000.0,
171 Dt_ms = max(1, round(T1 - T0)),
172 Now = erlang:system_time(millisecond),
173
174 %%--- 2. IMU -----
175 [Gy, Ax, Az] = pmod_nav:read(acc,
176     [out_y_g, out_x_xl, out_z_xl], #{g_unit => dps}),
177
178 %%--- 3. ESP32 -----
179 [<<SL1,SL2,SR1,SR2,CtrlByte>>] = grisp_i2c:transfer(I2Cbus, [{read, 16#40, 1, 5}]),
180 [Speed_L, Speed_R] = hera_com:decode_half_float([<<SL1, SL2>>, <<SR1, SR2>>]),
181 Speed = (Speed_L + Speed_R) / 2.0,
182
183 %% Drive-bit gating during proto bursts:
184 %% bit 7 (Arm_Ready) - always live (ESP owns it)
185 %% bit 4 (Get_Up) - latched from last drive byte (preserve stand)
186 %% bits 3..0 (F/B/L/R) - forced zero (Python suppresses these on tx)
187 %% bits 6,5 (Switch/Test) - decoded but ignored (filter is ?USE_KALMAN,
188 %% Hera log trigger removed)
189 Proto_Bit = (CtrlByte band ?CMD_PROTO_MASK) /= 0,
190 Drive_Byte = if Proto_Bit ->
191     (CtrlByte band 16#80) %% live Arm_Ready
192     bor (RS#rstate.last_drive_byte band 16#10); %% latched Get_Up
193     true ->
194         CtrlByte
195     end,
196 [Arm_Ready, _Switch_unused, _Test_unused, Get_Up, Forward, Backward, Left, Right] =
197     hera_com:get_bits(Drive_Byte),
198 Last_Drive_Byte_New = if Proto_Bit -> RS#rstate.last_drive_byte;
199     true -> CtrlByte
200     end,
201
202 %%--- 3b. wp inject fsm + traj-array mutation (phases 2+5) -----
203 %% b6=1 -> protocol byt, else its just normal drive
204 %% runs regardless of lifecycle so op can q waypoints anytime
205 %% phase 5, dispatch produces a CmdAction, we apply it to the live traj
206 %% below. phase 6 (deferred), spline rebuild on each mutation
207 HadRemainBefore = traj_planner:has_remaining(RS#rstate.traj),
208 {CmdAction, CmdRx1} = cmd_rx_step(CtrlByte, RS#rstate.cmd_rx),
209 {Traj_Raw, Pose_Raw, LC_FromCmd} =
210     apply_cmd_action(CmdAction, RS#rstate.traj,
211         RS#rstate.pose, RS#rstate.lifecycle),

```

```

212 HasRemainingAfter = traj_planner:has_remaining(Traj_Raw),
213
214 %% Wire-driven lifecycle overrides -- every reset destination is idle now:
215 %% CLEAR_ALL (any state) → idle, full wipe + cooldown
216 %% running/paused + cancel drained list → idle, full wipe + cooldown
217 %% Anything else passes through untouched. Non-draining cancel and ADD
218 %% rebuild the spline inside apply_cmd_action but don't transition.
219 {LC_Override, Traj0, Pose0, ResetStamp} =
220     case CmdAction of
221     clear_all ->
222         traj_planner:print_transition(RS#rstate.lifecycle, idle, Traj_Raw),
223         {idle, Traj_Raw, #pose{}, Now};
224     - when HadRemainingBefore, not HasRemainingAfter,
225         (LC_FromCmd == running orelse LC_FromCmd == paused) ->
226         Traj_Wiped = traj_planner:clear_remaining(Traj_Raw),
227         traj_planner:print_transition(LC_FromCmd, idle, Traj_Wiped),
228         {idle, Traj_Wiped, #pose{}, Now};
229     - ->
230         {LC_FromCmd, Traj_Raw, Pose_Raw, keep}
231     end,
232 Last_Reset_Override =
233     case ResetStamp of
234     keep -> RS#rstate.last_reset_ms;
235     T -> T
236     end,
237
238 %%--- 4. Tilt -- only the filter selected by ?USE_KALMAN runs -----
239 %% Selection is now a compile-time flag, not a remote-toggled bit. The
240 %% original 'Switch' wire-bit doubled as PROTO and made the filter flip
241 %% mid-waypoint upload; that's gone.
242 Angle_Acc_Val = math:atan(Az / (-Ax)) * ?RAD_TO_DEG,
243 {X1, P1, Angle_Kalman, Angle_Complem_New, Angle_Rate_New, Angle} =
244     case ?USE_KALMAN of
245     true ->
246         {Xk, Pk} = kalman_angle(Dt, Ax, Az, Gy, Gy0, X0, P0),
247         [Th_K, _W] = mat:to_array(Xk),
248         Ak = Th_K * ?RAD_TO_DEG,
249         {Xk, Pk, Ak, Angle_Complem, Angle_Rate, Ak};
250     false ->
251         K = 1.25 / (1.25 + (1.0 / Mean_Freq)),
252         {Acn, Arn} =
253             complem_angle({Dt, Ax, Az, Gy, Gy0, K, Angle_Complem, Angle_Rate}),
254         {X0, P0, 0.0, Acn, Arn, Angle_Complem}
255     end,
256
257 %%--- 5. Robot_Up -----
258 Robot_Up = RS#rstate.robot_up,
259 Robot_Up_New =
260     if Robot_Up andalso (abs(Angle) > 20) -> false;
261     not Robot_Up andalso (abs(Angle) < 18) -> true;
262     true -> Robot_Up
263     end,
264
265 %%--- 6. Button combos: edges + hold accumulation -----
266 FB_Combo = Forward andalso Backward,
267 LR_Combo = Left andalso Right,
268 Prev_FB = RS#rstate.prev_fb_combo,
269 Prev_LR = RS#rstate.prev_lr_combo,
270
271 FB_Falling = input_combo:falling_edge(FB_Combo, Prev_FB),
272
273 FB_Base = input_combo:hold_check(FB_Combo, Prev_FB, RS#rstate.fb_held_ms),
274 FB_Held = if FB_Combo -> FB_Base + Dt_ms; true -> 0 end,
275 LR_Base = input_combo:hold_check(LR_Combo, Prev_LR, RS#rstate.lr_held_ms),
276 LR_Held = if LR_Combo -> LR_Base + Dt_ms; true -> 0 end,
277
278 FB_Hold_500 = FB_Held >= ?TRAJ_HOLD_MS,
279 %% Tap = release within the hold threshold. fb_held_ms holds the duration
280 %% RIGHT BEFORE the falling-edge tick (hold_check returns 0 on release, so
281 %% FB_Held is already 0 here -- we need the stored value). A release after
282 %% FB_Hold_500 has fired will have fb_held_ms >= ?TRAJ_HOLD_MS and is
283 %% suppressed, so the long-hold reset cannot chain into a tap.
284 FB_Tap = FB_Falling andalso (RS#rstate.fb_held_ms < ?TRAJ_HOLD_MS),

```

```

285
286 %%--- 7. Odometry (running only) -----
287 %% pose output is only consumed by traj_planner and csv_logger, both gated
288 %% on lifecycle=running. lifecycle_step resets pose to zero on idle→running,
289 %% so skipping integration outside running is safe.
290 %% Pose0 came from apply_cmd_action, normally same as RS.pose but Clear_All
291 %% zeroes it (doesn't drive home, just clears the values)
292 Pose1 = case RS#rstate.lifecycle of
293     running -> odometry:integrate({Speed_L, Speed_R}, Dt, Pose0);
294     _       -> Pose0
295 end,
296
297 %%--- 8. Lifecycle state machine -----
298 RS1 = RS#rstate{
299     pose           = Pose1,
300     robot_up       = Robot_Up_New,
301     fb_held_ms     = FB_Held,
302     lr_held_ms     = LR_Held,
303     prev_fb_combo  = FB_Combo,
304     prev_lr_combo  = LR_Combo,
305     cmd_rx         = CmdRx1,
306     traj           = Traj0,
307     lifecycle      = LC_Override,
308     last_reset_ms  = Last_Reset_Override
309 },
310 {LC_New, TrajState1, Pose2, RS2} =
311     lifecycle_step(RS1, FB_Tap, FB_Hold_500, Now),
312
313 %%--- 10. Trajectory controller (only when running) -----
314 {Adv_V_Traj, Turn_V_Traj, TrajFinished, WpsLeft, TrajSS, TrajCnt, TrajDistWP, TrajState2} =
315     case LC_New of
316         running when ?ENABLE_TRAJECTORY, TrajState1 /= undefined ->
317             Sensors = #{speed => Speed, dt => Dt},
318             {TOut, TS2} = traj_planner:step(Sensors, Pose2, TrajState1),
319             #{adv_v    := AV,
320              turn_v    := TV,
321              finished  := Fin,
322              wps_left  := WL,
323              substate  := SS,
324              debug     := #{counter := Cnt, dist_wp := DWP}} = TOut,
325             {AV, TV, Fin, WL, SS, Cnt, DWP, TS2};
326         _ ->
327             {0.0, 0.0, false, 0, idle, 0, 0.0, TrajState1}
328     end,
329
330 %% Traj completion lands directly in idle (no sticky finished state).
331 %% Full wipe + cooldown keeps idle's invariant: pose=0, traj cleared.
332 {LC_Final, RS2_Final, TrajState_Final} =
333     if TrajFinished andalso LC_New == running ->
334         io:format("===LOG_END_TRAJ===~n"),
335         traj_planner:print_transition(running, idle, TrajState2),
336         RS_Idle = go_idle(RS2, TrajState2, Now),
337         {idle, RS_Idle, RS_Idle#rstate.traj};
338     true ->
339         {LC_New, RS2, TrajState2}
340     end,
341
342 %%--- 11. Velocity mux -----
343 Man_Adv = speed_ref(Forward, Backward),
344 Man_Turn = turn_ref(Left, Right),
345 {Adv_V_Goal, Turn_V_Goal} =
346     case LC_Final of
347         running -> {Adv_V_Traj + Man_Adv, Turn_V_Traj + Man_Turn};
348         _       -> {Man_Adv, Man_Turn}
349     end,
350
351 %%--- 12. Stability controller -----
352 {Acc, Adv_V_Ref_New, Turn_V_Ref_New} =
353     stability_engine:controller(
354         {Dt, Angle, Speed},
355         {Pid_Speed, Pid_Stability},
356         {Adv_V_Goal, Adv_V_Ref},
357         {Turn_V_Goal, Turn_V_Ref}),

```

```

358
359 %%--- 13. Robot_State machine -----
360 Next_Robot_State =
361     next_robot_state(RS2_Final#rstate.robot_state, Robot_Up_New, Get_Up,
362                     Angle, Arm_Ready),
363     {Power, Freeze, Extend, Robot_Up_Bit} = state_outputs(Next_Robot_State),
364
365 %%--- 14. ESP32 output -----
366 F_B = if Angle > 0.0 -> 1; true -> 0 end,
367 Output_Byte = get_byte([Power, Freeze, Extend, Robot_Up_Bit, F_B, 0, 0, 0]),
368 [HF1, HF2] = hera_com:encode_half_float([Acc, Turn_V_Ref_New]),
369 grisp_i2c:transfer(I2Cbus, [{write, 16#40, 1, [HF1, HF2, <<Output_Byte>>]}]),
370
371 %%--- 15. LED -----
372 update_leds(LC_Final, RS2_Final#rstate.last_reset_ms, Now, RS2_Final#rstate.cmd_rx),
373
374 %%--- 16. CSV logging -----
375 if LC_Final :=: running ->
376     case csv_logger:append() of
377         logged ->
378             #pose{x = PX, y = PY, theta = PT} = Pose2,
379             csv_logger:emit_serial({t_ms => Now,
380                                     lifecycle => LC_Final,
381                                     substate => TrajSS,
382                                     x => PX,
383                                     y => PY,
384                                     speed => Speed,
385                                     adv_v => Adv_V_Goal,
386                                     turn_v => Turn_V_Goal,
387                                     dist_wp => TrajDistWP,
388                                     wps_left => WpsLeft,
389                                     yaw_odo => PT});
390             skipped -> ok
391         end;
392     true -> ok
393 end,
394
395 %%--- 17. Frequency + shell-query plumbing -----
396 %% The legacy Hera log path (Test bit -> 15s buffered log) is removed.
397 %% csv_logger covers run logging. The Log_* triple is kept as a no-op so
398 %% the recursive signature is unchanged.
399 {N_New, Freq_New, Mean_Freq_New} = frequency_computation(Dt, N, Freq, Mean_Freq),
400 Log_End_New = Log_End,
401 Logging_New = Logging,
402 Log_List_New = Log_List,
403 handle_messages(T1, Start_Time, Dt, Gy, Acc, CtrlByte,
404                 Angle_Acc_Val, Angle_Kalman, Angle_Complem,
405                 Adv_V_Ref, Adv_V_Ref_New, Turn_V_Ref_New, Speed,
406                 Log_List),
407
408 %%--- 18. Rate limiter -----
409 T2 = erlang:system_time() / 1.0e6,
410 [{_, Freq_Goal}] = ets:lookup(variables, "Freq_Goal"),
411 Delay_Goal = 1.0 / Freq_Goal * 1000.0,
412 if T2 - T_End < Delay_Goal -> wait(Delay_Goal - (T2 - T1)); true -> ok end,
413 T_End_New = erlang:system_time() / 1.0e6,
414
415 %%--- 19. Recurse -----
416 %% RS2_Final carries the go_idle wipe (pose=0, traj cleared, cooldown
417 %% stamped) when traj-done fired this tick, otherwise it equals RS2.
418 RS_Next = RS2_Final#rstate{
419     lifecycle      = LC_Final,
420     traj           = TrajState_Final,
421     robot_state    = Next_Robot_State,
422     robot_up       = Robot_Up_New,
423     traj_wps_left  = WpsLeft,
424     traj_substate  = TrajSS,
425     traj_counter   = TrajCnt,
426     last_drive_byte = Last_Drive_Byte_New
427 },
428 robot_main(Start_Time, Hera_pid,
429             {T1, X1, P1}, I2Cbus,
430             {Logging_New, Log_End_New, Log_List_New},

```

```

431         {Gy0, Gx0, Angle_Complem_New, Angle_Rate_New},
432         {Pid_Speed, Pid_Stability},
433         {Adv_V_Ref_New, Turn_V_Ref_New},
434         {N_New, Freq_New, Mean_Freq_New, T_End_New},
435         RS_Next).
436
437     %%% =====
438     %%% Lifecycle state machine
439     %%% =====
440
441     lifecycle_step(RS, FB_Tap, FB_Hold_500, Now) ->
442         LC = RS#rstate.lifecycle,
443         RobUp = RS#rstate.robot_up,
444         TS = RS#rstate.traj,
445
446         Traj_OK = (Now - RS#rstate.last_traj_action_ms) >= ?TRAJ_PAUSE_COOLDOWN_MS,
447         Reset_OK = (Now - RS#rstate.last_reset_ms) >= ?TRAJ_RESET_COOLDOWN_MS,
448
449         case LC of
450             %% Note: idle intentionally has no fall (~RobUp) handler. Pre-flight
451             %% queued WPs (ADD before first FB_Tap) survive being knocked over.
452             idle ->
453                 if FB_Tap andalso Reset_OK ->
454                     %% Idle is started by tap only -- no instant-on-press. Waypoints
455                     %% are preserved (only Clear_All / hold-reset wipes them).
456                     csv_logger:reset(),
457                     traj_planner:print_transition(idle, running, TS),
458                     RS2 = RS#rstate{lifecycle = running,
459                                     lifecycle_entered_ms = Now,
460                                     last_traj_action_ms = Now,
461                                     pose = #pose{}}},
462                     {running, TS, RS2#rstate.pose, RS2};
463                 FB_Hold_500 andalso Reset_OK ->
464                     %% Safety wipe -- Reset_OK gates against repeat-firing while
465                     %% the operator keeps holding past 500 ms.
466                     traj_planner:print_transition(idle, idle, TS),
467                     RS2 = go_idle(RS, TS, Now),
468                     {idle, RS2#rstate.traj, RS2#rstate.pose, RS2};
469                 true ->
470                     {idle, TS, RS#rstate.pose, RS}
471             end;
472
473             running ->
474                 if not RobUp ->
475                     traj_planner:print_transition(running, idle, TS),
476                     RS2 = go_idle(RS, TS, Now),
477                     {idle, RS2#rstate.traj, RS2#rstate.pose, RS2};
478                 FB_Hold_500 ->
479                     traj_planner:print_transition(running, idle, TS),
480                     RS2 = go_idle(RS, TS, Now),
481                     {idle, RS2#rstate.traj, RS2#rstate.pose, RS2};
482                 FB_Tap andalso Traj_OK ->
483                     traj_planner:print_transition(running, paused, TS),
484                     RS2 = RS#rstate{lifecycle = paused,
485                                     lifecycle_entered_ms = Now,
486                                     last_traj_action_ms = Now},
487                     {paused, TS, RS2#rstate.pose, RS2};
488                 true ->
489                     {running, TS, RS#rstate.pose, RS}
490             end;
491
492             paused ->
493                 if not RobUp ->
494                     traj_planner:print_transition(paused, idle, TS),
495                     RS2 = go_idle(RS, TS, Now),
496                     {idle, RS2#rstate.traj, RS2#rstate.pose, RS2};
497                 FB_Hold_500 ->
498                     traj_planner:print_transition(paused, idle, TS),
499                     RS2 = go_idle(RS, TS, Now),
500                     {idle, RS2#rstate.traj, RS2#rstate.pose, RS2};
501                 FB_Tap andalso Traj_OK ->
502                     traj_planner:print_transition(paused, running, TS),
503                     RS2 = RS#rstate{lifecycle = running,

```

```

504         lifecycle_entered_ms = Now,
505         last_traj_action_ms = Now},
506         {running, TS, RS2#rstate.pose, RS2};
507     true ->
508         {paused, TS, RS#rstate.pose, RS}
509     end
510 end.
511
512 %% Single-source-of-truth -> idle helper. Every reset destination (hold, fall,
513 %% traj-done, cancel-drain, CLEAR_ALL) routes through here: pose to origin,
514 %% traj fully wiped (including last_pose -- see traj_planner:clear_remaining/1),
515 %% 1 s reset cooldown stamped so the LED + Reset_OK gate stay consistent.
516 go_idle(RS, TS, Now) ->
517     TS2 = traj_planner:clear_remaining(TS),
518     RS#rstate{lifecycle = idle,
519             last_reset_ms = Now,
520             pose = #pose{},
521             traj = TS2}.
522
523 %%% =====
524 %%% LED display
525 %%% =====
526
527 update_leds(Lifecycle, Last_Reset_Ms, Now, CmdRx) ->
528     In_Cooldown = (Now - Last_Reset_Ms) < ?TRAJ_RESET_COOLDOWN_MS,
529     case Lifecycle of
530         idle when In_Cooldown -> led_control:traj_reset_cooldown();
531         idle -> led_control:idle();
532         running -> led_control:traj_running();
533         paused -> led_control:traj_paused()
534     end,
535     %% override led1 if cmd-rx fsm has something to show
536     %% led2 stays w/ lifecycle so we can see both at the same time
537     case cmd_led_override(CmdRx) of
538         none -> ok;
539         cmd_active -> led_control:cmd_active();
540         cmd_frame_ok -> led_control:cmd_frame_ok();
541         cmd_committed -> led_control:cmd_committed();
542         cmd_error -> led_control:cmd_error()
543     end.
544
545 cmd_led_override(#cmd_rx{led_pulse_left = N, led_pulse_kind = K}) when N > 0, K /= none -> K;
546 cmd_led_override(#cmd_rx{state = idle}) -> none;
547 cmd_led_override(#cmd_rx{}) -> cmd_active.
548
549 %%% =====
550 %%% Robot state machine (unchanged from original)
551 %%% =====
552
553 next_robot_state(Robot_State, Robot_Up, Get_Up, Angle, Arm_Ready) ->
554     case Robot_State of
555         rest ->
556             if Get_Up -> raising; true -> rest end;
557         raising ->
558             if Robot_Up -> stand_up;
559             not Get_Up -> soft_fall;
560             true -> raising
561         end;
562         stand_up ->
563             if not Get_Up -> wait_for_extend;
564             not Robot_Up -> rest;
565             true -> stand_up
566         end;
567         wait_for_extend ->
568             prepare_arms;
569         prepare_arms ->
570             if Arm_Ready -> free_fall;
571             Get_Up -> stand_up;
572             not Robot_Up -> rest;
573             true -> prepare_arms
574         end;
575         free_fall ->
576             if abs(Angle) > 10 -> wait_for_retract; true -> free_fall end;

```

```

577     wait_for_retract ->
578         soft_fall;
579     soft_fall ->
580         if Arm_Ready -> rest;
581         Get_Up -> raising;
582         true -> soft_fall
583     end
584 end.
585
586 state_outputs(State) ->
587     case State of
588         rest -> {0, 0, 0, 0};
589         raising -> {1, 0, 1, 0};
590         stand_up -> {1, 0, 0, 1};
591         wait_for_extend -> {1, 0, 1, 1};
592         prepare_arms -> {1, 0, 1, 1};
593         free_fall -> {1, 1, 1, 1};
594         wait_for_retract -> {1, 0, 0, 0};
595         soft_fall -> {1, 0, 0, 0}
596     end.
597
598 %%%
599 %%% wp ota inject fsm (rx + print + led only)
600 %%%
601 %%%
602 %%% each tick feeds CtrlByte to cmd_rx_step/2:
603 %%% 1. tick down any in-flight led pulse counter
604 %%% 2. count consecutive same-byt reads (debounce)
605 %%% 3. if fresh stable PROTO=1 byt -> run thru fsm (advance/commit/err)
606 %%% 4. if stable PROTO=0 byt -> drop commit gate so repeated commands
607 %%% (eg cancel_last twice) get re-accepted insted of dedup'd
608 %%%
609 %%% effects rn = io:format prints + led pulse state on #cmd_rx{}
610 %%% no traj mutation yet, comes phases 4-5
611
612 %%% Returns {Action, NewCR}. Action is consumed by apply_cmd_action/3 below.
613 %%% Action ::= none
614 %%% | {append, DX, DY}
615 %%% | cancel_last
616 %%% | {cancel_n, N}
617 %%% | clear_all
618 cmd_rx_step(CtrlByte, CR = #cmd_rx{}) ->
619     CR1 = tick_led_pulse(CR),
620     Stable_New =
621         if CtrlByte == CR1#cmd_rx.last_byte -> CR1#cmd_rx.stable_count + 1;
622         true -> 1
623     end,
624     CR2 = CR1#cmd_rx{last_byte = CtrlByte, stable_count = Stable_New},
625     Proto = (CtrlByte band ?CMD_PROTO_MASK) /= 0,
626     Stable = Stable_New >= 2,
627     Changed = CtrlByte /= CR2#cmd_rx.last_committed_byte,
628     case {Proto, Stable, Changed} of
629         {true, true, true} ->
630             CR3 = CR2#cmd_rx{last_committed_byte = CtrlByte},
631             Kind = (CtrlByte band ?CMD_KIND_MASK) /= 0,
632             Payload = CtrlByte band ?CMD_PAYLOAD_MASK,
633             dispatch_frame(Kind, Payload, CR3);
634         {false, true, _} ->
635             %% stable drive byt - drop the gate so next proto byt counts fresh
636             {none, CR2#cmd_rx{last_committed_byte = -1}};
637         _ ->
638             {none, CR2}
639     end.
640
641 %%% ABORT in idle = silent noop, else reset w/ err blink
642 dispatch_frame(false, ?CMD_CTRL_ABORT, CR = #cmd_rx{state = idle}) ->
643     {none, CR};
644 dispatch_frame(false, ?CMD_CTRL_ABORT, CR) ->
645     {none, pulse(reset_cmd_rx(CR), cmd_error, ?CMD_LED_PULSE_ERROR)};
646
647 %%% from idle a ctrl frame kicks off a transaction
648 dispatch_frame(false, ?CMD_CTRL_ADD_HEADER, CR = #cmd_rx{state = idle}) ->
649     {none, pulse(CR#cmd_rx{state = want_x_hi}, cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};

```

```

650 dispatch_frame(false, ?CMD_CTRL_CANCEL_LAST, CR = #cmd_rx{state = idle}) ->
651   io:format("RX_CANCEL_LAST~n"),
652   {cancel_last,
653     pulse(reset_cmd_rx(CR), cmd_committed, ?CMD_LED_PULSE_COMMITTED)};
654 dispatch_frame(false, ?CMD_CTRL_CLEAR_ALL, CR = #cmd_rx{state = idle}) ->
655   io:format("RX_CLEAR_ALL~n"),
656   {clear_all,
657     pulse(reset_cmd_rx(CR), cmd_committed, ?CMD_LED_PULSE_COMMITTED)};
658 dispatch_frame(false, ?CMD_CTRL_CANCEL_N_HDR, CR = #cmd_rx{state = idle}) ->
659   {none, pulse(CR#cmd_rx{state = want_n}, cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
660
661 %% ADD seq: data HI_TO_LO data X_TO_Y data HI_TO_LO data ADD_COMMIT
662 dispatch_frame(true, Payload, CR = #cmd_rx{state = want_x_hi}) ->
663   {none, pulse(CR#cmd_rx{state = want_x_hi_to_lo, x_acc = Payload bs1 5},
664     cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
665 dispatch_frame(false, ?CMD_CTRL_HI_TO_LO, CR = #cmd_rx{state = want_x_hi_to_lo}) ->
666   {none, pulse(CR#cmd_rx{state = want_x_lo}, cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
667 dispatch_frame(true, Payload, CR = #cmd_rx{state = want_x_lo}) ->
668   {none, pulse(CR#cmd_rx{state = want_x_to_y, x_acc = CR#cmd_rx.x_acc bor Payload},
669     cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
670 dispatch_frame(false, ?CMD_CTRL_X_TO_Y, CR = #cmd_rx{state = want_x_to_y}) ->
671   {none, pulse(CR#cmd_rx{state = want_y_hi}, cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
672 dispatch_frame(true, Payload, CR = #cmd_rx{state = want_y_hi}) ->
673   {none, pulse(CR#cmd_rx{state = want_y_hi_to_lo, y_acc = Payload bs1 5},
674     cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
675 dispatch_frame(false, ?CMD_CTRL_HI_TO_LO, CR = #cmd_rx{state = want_y_hi_to_lo}) ->
676   {none, pulse(CR#cmd_rx{state = want_y_lo}, cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
677 dispatch_frame(true, Payload, CR = #cmd_rx{state = want_y_lo}) ->
678   {none, pulse(CR#cmd_rx{state = want_commit, y_acc = CR#cmd_rx.y_acc bor Payload},
679     cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
680 dispatch_frame(false, ?CMD_CTRL_ADD_COMMIT, CR = #cmd_rx{state = want_commit}) ->
681   DX = CR#cmd_rx.x_acc - ?CMD_SIGN_OFFSET,
682   DY = CR#cmd_rx.y_acc - ?CMD_SIGN_OFFSET,
683   case in_offset_range(DX) andalso in_offset_range(DY) of
684     true ->
685       io:format("RX_WP: dx=~p dy=~p~n", [DX, DY]),
686       {{append, DX, DY},
687        pulse(reset_cmd_rx(CR), cmd_committed, ?CMD_LED_PULSE_COMMITTED)};
688     false ->
689       io:format("RX_WP_ERR: dx=~p dy=~p (out of range)~n", [DX, DY]),
690       {none, pulse(reset_cmd_rx(CR), cmd_error, ?CMD_LED_PULSE_ERROR)};
691   end;
692
693 %% CANCEL_N seq: data, CANCEL_N_COMMIT
694 dispatch_frame(true, Payload, CR = #cmd_rx{state = want_n}) ->
695   {none, pulse(CR#cmd_rx{state = want_n_commit, n_acc = Payload},
696     cmd_frame_ok, ?CMD_LED_PULSE_FRAME_OK)};
697 dispatch_frame(false, ?CMD_CTRL_CANCEL_N_COMMIT, CR = #cmd_rx{state = want_n_commit}) ->
698   N = CR#cmd_rx.n_acc,
699   io:format("RX_CANCEL_N: ~p~n", [N]),
700   {{cancel_n, N},
701    pulse(reset_cmd_rx(CR), cmd_committed, ?CMD_LED_PULSE_COMMITTED)};
702
703 %% anything else in any state = proto err, reset + err blink
704 dispatch_frame(_Kind, _Payload, CR) ->
705   {none, pulse(reset_cmd_rx(CR), cmd_error, ?CMD_LED_PULSE_ERROR)}.
706
707 %% Apply a decoded action to the live #traj_state{} and print the resulting
708 %% array. Returns {NewTS, NewPose, NewLC} so the caller can also pick up a
709 %% pose reset (Clear_All zeroes pose values). Spline rebuild now happens
710 %% inside traj_planner's mutators (phase 6), and the finished→running
711 %% reactivation is owned by the lifecycle FSM in lifecycle_step/4 -- so no
712 %% lifecycle hints are emitted from here anymore.
713 %%
714 %% Reference point for ADD is the last queued waypoint, or current pose if
715 %% the array is empty.
716 apply_cmd_action(none, TS, Pose, LC) -> {TS, Pose, LC};
717
718 apply_cmd_action({append, DX, DY}, TS, Pose, LC) ->
719   {LastX, LastY, Has_Prev} =
720     case traj_planner:last_waypoint_abs(TS) of
721       empty -> {Pose#pose.x, Pose#pose.y, false};
722       {LX, LY} -> {LX, LY, true}

```

```

723     end,
724     AbsX = LastX + float(DX),
725     AbsY = LastY + float(DY),
726     %% Reject relative-zero (or near-zero) appends when a previous WP exists --
727     %% duplicate consecutive control points poison the Catmull-Rom tangents and
728     %% feed a zero-length final segment to pure_pursuit. First-ever WP is
729     %% allowed regardless (operator may intentionally anchor at current pose).
730     Is_Dup = Has_Prev andalso (float(DX)*float(DX) + float(DY)*float(DY)) < 1.0,
731     case Is_Dup of
732     true ->
733         io:format("===WP_DUPLICATE_SKIPPED=== rel=(~p,~p)~n", [DX, DY]),
734         {TS, Pose, LC};
735     false ->
736         TS2 = traj_planner:append_waypoint(AbsX, AbsY, TS),
737         traj_planner:print_waypoints(TS2),
738         {TS2, Pose, LC}
739     end;
740
741 apply_cmd_action(cancel_last, TS, Pose, LC) ->
742     TS2 = traj_planner:cancel_last(TS),
743     traj_planner:print_waypoints(TS2),
744     {TS2, Pose, LC};
745
746 apply_cmd_action({cancel_n, N}, TS, Pose, LC) ->
747     TS2 = traj_planner:cancel_last_n(N, TS),
748     traj_planner:print_waypoints(TS2),
749     {TS2, Pose, LC};
750
751 apply_cmd_action(clear_all, TS, _Pose, LC) ->
752     TS2 = traj_planner:clear_remaining(TS),
753     traj_planner:print_waypoints(TS2),
754     %% Clear_All also zeroes the pose values (no driving back to 0,0,0,
755     %% just clears the numbers so future ADD references start fresh)
756     {TS2, #pose{}, LC}.
757
758 reset_cmd_rx(CR) ->
759     %% keep debounce fields, just wipe fsm state + accs
760     CR#cmd_rx{state = idle, x_acc = 0, y_acc = 0, n_acc = 0}.
761
762 pulse(CR, Kind, Ticks) ->
763     CR#cmd_rx{led_pulse_kind = Kind, led_pulse_left = Ticks}.
764
765 tick_led_pulse(CR = #cmd_rx{led_pulse_left = 0}) -> CR;
766 tick_led_pulse(CR = #cmd_rx{led_pulse_left = 1}) ->
767     CR#cmd_rx{led_pulse_left = 0, led_pulse_kind = none};
768 tick_led_pulse(CR = #cmd_rx{led_pulse_left = N}) ->
769     CR#cmd_rx{led_pulse_left = N - 1}.
770
771 in_offset_range(V) ->
772     V >= -?CMD_MAX_OFFSET andalso V <= ?CMD_MAX_OFFSET.
773
774 %%% =====
775 %%% Internal helpers
776 %%% =====
777
778 calibrate() ->
779     N = 500,
780     Data = [list_to_tuple(pmod_nav:read(acc, [out_x_g, out_y_g, out_z_g]))
781             || _ <- lists:seq(1, N)],
782     {X, Y, Z} = lists:unzip3(Data),
783     [lists:sum(X)/N, lists:sum(Y)/N, lists:sum(Z)/N].
784
785 kalman_angle(Dt, Ax, Az, Gy, Gy0, X0, P0) ->
786     R = mat:matrix([[3.0, 0.0], [0, 3.0e-6]]),
787     Q = mat:matrix([[3.0e-5, 0.0], [0.0, 10.0]]),
788     F = fun(X) -> [Th,W] = mat:to_array(X), mat:matrix([[Th+Dt*W],[W]]) end,
789     Jf = fun(X) -> [_,_] = mat:to_array(X), mat:matrix([[1,Dt],[0,1]]) end,
790     H = fun(X) -> [Th,W] = mat:to_array(X), mat:matrix([[Th],[W]]) end,
791     Jh = fun(X) -> [_,_] = mat:to_array(X), mat:matrix([[1,0],[0,1]]) end,
792     Z = mat:matrix([[math:atan(Az / (-Ax))], [(Gy - Gy0) * ?DEG_TO_RAD]]),
793     kalman:ekf({X0, P0}, {F, Jf}, {H, Jh}, Q, R, Z).
794
795 complem_angle({Dt, Ax, Az, Gy, Gy0, K, Angle_Complem, Angle_Rate}) ->

```

```

796 Rate_New = (Gy - Gy0) * ?COEF_FILTER + Angle_Rate * (1 - ?COEF_FILTER),
797 Delta_Gyr = Rate_New * Dt,
798 Angle_Acc = math:atan(Az / (-Ax)) * 180 / math:pi(),
799 {(Angle_Complem + Delta_Gyr) * K + Angle_Acc * (1 - K), Rate_New}.
800
801 speed_ref(Forward, Backward) ->
802     if Forward andalso Backward -> 0.0;    %% combo gesture -- not a direction
803         Forward -> ?ADV_V_MAX;
804         Backward -> -?ADV_V_MAX;
805         true -> 0.0
806     end.
807
808 turn_ref(Left, Right) ->
809     if Left andalso Right -> 0.0;          %% combo gesture -- not a direction
810         Right -> ?TURN_V_MAX;
811         Left -> -?TURN_V_MAX;
812         true -> 0.0
813     end.
814
815 frequency_computation(Dt, N, Freq, Mean_Freq) ->
816     if N := 100 -> {0, 0, Freq};
817     true -> {N+1, ((Freq*N) + (1/Dt)) / (N+1), Mean_Freq}
818 end.
819
820 handle_messages(T1, Start_Time, Dt, Gy, Acc, CtrlByte,
821     Angle_Acc, Angle_Kalman, Angle_Complem,
822     Adv_V_Ref, Adv_V_Ref_New, Turn_V_Ref_New, Speed,
823     Log_List) ->
824     receive
825         {From, log_values} -> From ! {self(), log, Log_List};
826         {From1, get_all_data} ->
827             From1 ! {self(), data,
828                 [T1-Start_Time, 1/Dt, Gy, Acc, CtrlByte,
829                 -Angle_Acc, -Angle_Kalman, -Angle_Complem,
830                 Adv_V_Ref, Adv_V_Ref_New, Turn_V_Ref_New, Speed]};
831         {From1, freq} -> From1 ! {self(), 1/Dt};
832         {From2, acc} -> From2 ! {self(), Acc};
833         {_, Msg} -> io:format("[Robot] Unknown msg: ~p~n", [Msg])
834     after 0 -> ok
835 end.
836
837 wait(T) ->
838     Tnow = erlang:system_time() / 1.0e6,
839     wait_loop(Tnow, Tnow + T).
840 wait_loop(Tnow, Tend) when Tnow >= Tend -> ok;
841 wait_loop(_, Tend) -> wait_loop(erlang:system_time() / 1.0e6, Tend).
842
843 get_byte([A, B, C, D, E, F, G, H]) ->
844     A*128 + B*64 + C*32 + D*16 + E*8 + F*4 + G*2 + H.
845
846 modify_frequency(Freq) ->
847     ets:insert(variables, {"Freq_Goal", Freq}),
848     ok.

```

A.1.5 traj_planner.erl

```

1 -module(traj_planner).
2
3 %%% =====
4 %%% Trajectory path-following controller -- no lifecycle state.
5 %%%
6 %%% This module handles the math of following a Catmull-Rom spline through
7 %%% the operator-injected waypoint list: pure-pursuit steering, speed scaling
8 %%% on curvature, asymmetric acceleration/braking ramps, and waypoint dwell.
9 %%% It knows nothing about idle/paused/finished -- those states live in
10 %%% main_loop, which calls step/3 only when lifecycle == running.
11 %%%
12 %%% Sub-state machine (only active during running lifecycle):
13 %%%
14 %%%   cruise --(dist < BRAKE_DIST, last WP)--> braking
15 %%%   cruise --(dist < WAYPOINT_REACHED, intermediate WP)--> pop, stay cruise

```

```

16  %%% braking --(near WP & stopped)--► dwelling
17  %%% dwelling --(counter = 0)--► [finished = true returned]
18  %%%
19  %%% Waypoint bookkeeping: two lists are kept at all times.
20  %%% wps_remaining -- still to visit (head = next target)
21  %%% wps_traversed -- already visited, in chronological order
22  %%% Their concatenation wps_traversed ++ wps_remaining is the full
23  %%% intended trajectory. When a WP is reached, it migrates from the head of
24  %%% wps_remaining to the tail of wps_traversed; nothing is ever silently
25  %%% dropped. Every pop emits a WP_REACHED 3-block print.
26  %%% =====
27
28  -export([init/0, step/3, finished/1, has_remaining/1, config_string/0,
29          append_waypoint/3, cancel_last/1, cancel_last_n/2,
30          clear_remaining/1, last_waypoint_abs/1, waypoint_count/1,
31          print_waypoints/1, print_transition/3,
32          wps_remaining/1, wps_traversed/1, wps_full/1]).
33  -export_type([traj_state/0]).
34
35  -include("robot_types.hrl").
36
37  %% --- Physical -----
38  -define(WHEEL_BASE,      18.5).    %% cm
39
40  %% --- Speed -----
41  -define(MAX_SPEED,      14.0).    %% cm/s -- straight-line cruise speed
42  -define(MIN_CORNER_SPEED, 2.5).    %% cm/s -- floor when curvature is high
43  -define(MAX_TURN_V,     18.0).    %% cm/s -- cap on differential turn velocity
44
45  %% --- Ramps (asymmetric: slow accel, crisp brake) -----
46  -define(ACCEL_RATE,     12.0).    %% cm/s2
47  -define(BRAKE_RATE,     15.0).    %% cm/s2
48  -define(TURN_ACCEL_RATE, 15.0).    %% cm/s2
49  -define(TURN_BRAKE_RATE, 25.0).    %% cm/s2
50
51  %% --- Pure Pursuit -----
52  %% Must match pure_pursuit.erl's ?LOOKAHEAD -- both modules read the constant
53  %% independently (no shared header), so update them together.
54  -define(LOOKAHEAD,      12.0).    %% cm
55
56  %% --- Waypoint stop logic -----
57  %% Braking trigger is speed-adaptive: Brake_Dist = v2 / (2*BRAKE_RATE) + BRAKE_MARGIN.
58  %% Margin keeps the kinematic stop point inside the WAYPOINT_REACHED ring across
59  %% the full speed range (a fixed trigger would stop short at low approach speeds).
60  -define(BRAKE_MARGIN,   2.5).    %% cm -- added to kinematic stop distance
61  -define(WAYPOINT_REACHED, 4.0).    %% cm -- close-enough radius to declare arrival
62  -define(SETTLE_VEL,     1.0).    %% cm/s -- speed below this = considered stopped
63  -define(DWELL_TICKS,    600).    %% ~2 s at 300 Hz -- only applied at the final WP
64
65  %% --- Waypoints -----
66  %% Default is empty: the operator injects waypoints over the wire protocol
67  %% (PROTO frames decoded in main_loop). When this list is empty the controller
68  %% starts in a benign "nothing to do" state -- step/3 returns zero velocities
69  %% and done=true so main_loop's lifecycle FSM drops to finished.
70  -define(WAYPOINTS, []).
71
72  -record(traj_state, {
73      path                = []      :: [{float(), float()}],
74      wps_remaining       = []      :: [{float(), float()}],
75      wps_traversed       = []      :: [{float(), float()}],
76      substate            = cruise   :: cruise | braking | dwelling,
77      counter             = 0        :: non_neg_integer(),    %% dwell countdown
78      prev_adv_v          = 0.0      :: float(),
79      prev_turn_v         = 0.0      :: float(),
80      done                = false    :: boolean(),
81      %% Tangent-preservation seed for mid-cruise rebuilds. second_last_visited
82      %% lags last_visited by one waypoint so rebuild_path/1 can splice the new
83      %% spline through a point the robot has already passed through.
84      last_visited_wp     = none     :: {float(), float()} | none,
85      second_last_visited_wp = none  :: {float(), float()} | none,
86      %% Latched on every step/3 tick. Used as the rebuild seed before any
87      %% waypoint has been visited, so a first ADD from idle/finished anchors
88      %% the new spline to the robot's current world position.

```

```

89     last_pose                = #pose{} :: #pose{}
90 }).
91
92 -type traj_state() :: #traj_state{}.
93
94 %% Build the spline path and seed the waypoint list. Called by main_loop on
95 %% idle→running. traj_planner starts clean -- no memory of a previous run.
96 %% Empty WAYPOINTS produces an empty state; step/3 will immediately report done.
97 -spec init() -> traj_state().
98 init() ->
99     init_from(?WAYPOINTS).
100
101 init_from([]) ->
102     #traj_state{};
103 init_from([_Start | Rest_WPs] = All) ->
104     Path = spline:build_path(All, 30),
105     #traj_state{path = Path, wps_remaining = Rest_WPs}.
106
107 %% Called by main_loop every tick while lifecycle == running.
108 %% Sensors = #{speed => float(), dt => float()}
109 %% Pose = #pose{} (owned by main_loop / odometry)
110 %% Returns {Output, NewState} where Output = #{adv_v, turn_v, finished, substate, debug}
111 -spec step(map(), #pose{}, traj_state()) -> {map(), traj_state()}.
112 step(Sensors, Pose, State) ->
113     #{speed := Speed, dt := Dt} = Sensors,
114     #pose{x = X, y = Y, theta = Theta_Deg} = Pose,
115     #traj_state{path = Path, wps_remaining = WPs_Rem, substate = SubState,
116                 counter = Counter, prev_adv_v = Prev_Adv, prev_turn_v = Prev_Turn,
117                 wps_traversed = WPs_Trav} = State,
118
119     Theta_Rad = Theta_Deg * math:pi() / 180.0,
120
121     %% Controller substep
122     {Adv_Tgt, Turn_Tgt, Path2, WPs2, SS2, Cnt2, Done} =
123         substep(X, Y, Theta_Rad, Speed, Path, WPs_Rem, SubState, Counter),
124
125     %% Ramps: slow accel / crisp brake
126     DV = Adv_Tgt - Prev_Adv,
127     Adv_Out = if DV > 0.0 -> Prev_Adv + min(DV, ?ACCEL_RATE * Dt);
128               DV < 0.0 -> Prev_Adv + max(DV, -?BRAKE_RATE * Dt);
129               true -> Prev_Adv
130     end,
131
132     DTurn = Turn_Tgt - Prev_Turn,
133     Turn_Out = if DTurn > 0.0 -> Prev_Turn + min(DTurn, ?TURN_ACCEL_RATE * Dt);
134               DTurn < 0.0 -> Prev_Turn + max(DTurn, -?TURN_BRAKE_RATE * Dt);
135               true -> Prev_Turn
136     end,
137
138     %% Detect WP pop(s) this tick. substep may pop more than one in the
139     %% intermediate-WP pass-through recursion, so we compare lengths and slice
140     %% the prefix of WPs_Rem that was consumed.
141     Popped_Count = length(WPs_Rem) - length(WPs2),
142     Popped = lists:sublist(WPs_Rem, Popped_Count),
143     WPs_Trav2 = WPs_Trav ++ Popped,
144     {LastVis, SecondLastVis} =
145         update_visited(Popped, State#traj_state.last_visited_wp,
146                       State#traj_state.second_last_visited_wp),
147
148     NewState = State#traj_state{
149         path = Path2,
150         wps_remaining = WPs2,
151         wps_traversed = WPs_Trav2,
152         substate = SS2,
153         counter = Cnt2,
154         prev_adv_v = Adv_Out,
155         prev_turn_v = Turn_Out,
156         done = Done,
157         last_visited_wp = LastVis,
158         second_last_visited_wp = SecondLastVis,
159         last_pose = Pose
160     },
161

```

```

162 %% Emit WP_REACHED whenever any waypoint migrated to traversed this tick.
163 if Popped_Count > 0 ->
164     io:format("===WP_REACHED===\n"),
165     print_waypoints(NewState);
166     true -> ok
167 end,
168
169 Output = #{adv_v    => Adv_Out,
170            turn_v    => -Turn_Out,    %% negated to match robot's differential convention
171            finished => Done,
172            substate  => SS2,
173            wps_left  => length(WPs2),
174            debug     => #{dist_wp => dist_to_head(X, Y, WPs_Rem),
175                        counter => Cnt2}},
176 {Output, NewState}.
177
178 %% Shift-register update: handles 0/1/N popped WPs in one tick.
179 update_visited([], LV, SLV) -> {LV, SLV};
180 update_visited([P], LV, _SLV) -> {P, LV};
181 update_visited(Ps, _LV, _SLV) ->
182     [Last, SecondLast | _] = lists:reverse(Ps),
183     {Last, SecondLast}.
184
185 %% True after the last waypoint completes. main_loop uses this to transition
186 %% lifecycle to finished.
187 -spec finished(traj_state()) -> boolean().
188 finished(#traj_state{done = D}) -> D.
189
190 %% True when there are still waypoints to visit. main_loop reads this after
191 %% every apply_cmd_action to decide the wire-driven transitions
192 %% (finished+ADD -> paused, running/paused + cancel-drains -> paused/idle).
193 -spec has_remaining(traj_state()) -> boolean().
194 has_remaining(#traj_state{wps_remaining = WR}) -> WR /= [].
195
196 -spec config_string() -> string().
197 config_string() ->
198     io_lib:format(
199         "traj_planner: MAX_SPEED=~p MIN_CORNER=~p MAX_TURN_V=~p "
200         "LOOKAHEAD=~p BRAKE_MARGIN=~p BRAKE_RATE=~p DWELL_TICKS=~p",
201         [?MAX_SPEED, ?MIN_CORNER_SPEED, ?MAX_TURN_V,
202          ?LOOKAHEAD, ?BRAKE_MARGIN, ?BRAKE_RATE, ?DWELL_TICKS]).
203
204 %%% =====
205 %%% Public waypoint-array API
206 %%%
207 %%% Pure functions over #traj_state{}. main_loop calls these from the wire-
208 %%% protocol decoder. wps_traversed is preserved by every mutator unless the
209 %%% operator explicitly Clear_All's (or main_loop wipes it on a -> idle
210 %%% transition).
211 %%% =====
212
213 %% New WP is positioned relative to the last appended WP (which may already be
214 %% traversed). Falls back to current pose only when no WPs have ever been
215 %% appended. Always appended to wps_remaining, never to wps_traversed.
216 -spec append_waypoint(float(), float(), traj_state()) -> traj_state().
217 append_waypoint(AbsX, AbsY, S = #traj_state{wps_remaining = WR}) ->
218     NewWP = {AbsX, AbsY},
219     %% Coming out of an empty/finished state we also clear the done flag so the
220     %% controller is willing to accept a fresh run once a path is built.
221     S1 = S#traj_state{
222         wps_remaining = WR ++ [NewWP],
223         done          = false,
224         substate      = cruise,
225         counter       = 0
226     },
227     rebuild_path(S1).
228
229 %% Cancel operates on wps_remaining only. Traversed WPs are real history and
230 %% are never erased by cancel. No-op when remaining is empty.
231 -spec cancel_last(traj_state()) -> traj_state().
232 cancel_last(S = #traj_state{wps_remaining = []}) -> S;
233 cancel_last(S = #traj_state{wps_remaining = WR}) ->
234     rebuild_path(reset_if_no_remaining(S#traj_state{

```

```

235     wps_remaining = lists:droplast(WR)
236   })).
237
238 -spec cancel_last_n(non_neg_integer(), traj_state()) -> traj_state().
239 cancel_last_n(0, S) -> S;
240 cancel_last_n(_, S = #traj_state{wps_remaining = []}) -> S;
241 cancel_last_n(N, S = #traj_state{wps_remaining = WR}) ->
242   Keep = max(0, length(WR) - N),
243   rebuild_path(reset_if_no_remaining(S#traj_state{
244     wps_remaining = lists:sublist(WR, Keep)
245   }))).
246
247 %% Clear_All is a hard wipe: both lists empty, visited-shift register cleared,
248 %% substate machine rewound. main_loop's post-apply_cmd_action logic uses
249 %% has_remaining/1 == false to decide the lifecycle consequence.
250 %% last_pose is also reset -- every →idle path funnels through here, and a
251 %% stale last_pose would otherwise seed rebuild_path/1 from the previous
252 %% run's end position when the next ADD lands in idle.
253 -spec clear_remaining(traj_state()) -> traj_state().
254 clear_remaining(S) ->
255   S#traj_state{wps_remaining = [], wps_traversed = [], path = [],
256     substate = cruise, counter = 0, done = false,
257     last_visited_wp = none, second_last_visited_wp = none,
258     last_pose = #pose{}}.
259
260 %% Stop the substate machine when a cancel drains wps_remaining. Traversed
261 %% history is preserved here (only Clear_All / main_loop wipes it).
262 reset_if_no_remaining(S = #traj_state{wps_remaining = []}) ->
263   S#traj_state{path = [], substate = cruise, counter = 0, done = false};
264 reset_if_no_remaining(S) -> S.
265
266 %%% --- Spline rebuild -----
267 %% Seed precedence: second_last_visited → last_visited → last_pose. The first
268 %% two preserve tangent continuity with the path the robot already followed;
269 %% the pose fallback anchors a fresh-start spline to the robot's current world
270 %% position so pure_pursuit's lookahead begins where the robot actually is.
271 rebuild_path(S = #traj_state{wps_remaining = WR}) ->
272   Seed = case S#traj_state.second_last_visited_wp of
273     none ->
274       case S#traj_state.last_visited_wp of
275         none ->
276           P = S#traj_state.last_pose,
277           {P#pose.x, P#pose.y};
278         LW -> LW
279       end;
280     SLW -> SLW
281   end,
282   NewPath =
283     case WR of
284       [] -> [];
285       [Only] -> straight_line(Seed, Only, 30);
286       _ -> spline:build_path_continuing(Seed, WR, 30)
287     end,
288   S#traj_state{path = NewPath}.
289
290 %% Linear interpolation fallback for the 1-waypoint case. Catmull-Rom needs
291 %% at least two real control points; with a single target we just stretch a
292 %% straight segment from Seed to it so pure_pursuit has something to chase.
293 straight_line({Ax, Ay}, {Bx, By}, N) ->
294   [{Ax + (Bx - Ax) * float(I) / float(N),
295     Ay + (By - Ay) * float(I) / float(N)} || I <- lists:seq(0, N - 1)].
296
297 %% Last appended WP -- anchor for the next relative ADD. Looks at remaining
298 %% first (newest appends live there), falls back to traversed (everything
299 %% appended has been visited), then empty.
300 -spec last_waypoint_abs(traj_state()) -> {float(), float()} | empty.
301 last_waypoint_abs(#traj_state{wps_remaining = WR, wps_traversed = WT}) ->
302   case WR of
303     [] -> case WT of [] -> empty; _ -> lists:last(WT) end;
304     _ -> lists:last(WR)
305   end.
306
307 %% Total WPs ever appended (and not cancelled): traversed + remaining.

```

```

308 -spec waypoint_count(traj_state()) -> non_neg_integer().
309 waypoint_count(#traj_state{wps_remaining = WR, wps_traversed = WT}) ->
310     length(WR) + length(WT).
311
312 -spec wps_remaining(traj_state()) -> [{float(), float()}].
313 wps_remaining(#traj_state{wps_remaining = WR}) -> WR.
314
315 -spec wps_traversed(traj_state()) -> [{float(), float()}].
316 wps_traversed(#traj_state{wps_traversed = WT}) -> WT.
317
318 -spec wps_full(traj_state()) -> [{float(), float()}].
319 wps_full(#traj_state{wps_remaining = WR, wps_traversed = WT}) -> WT ++ WR.
320
321 %% Three-block dump: full, traversed, remaining. Called on every state
322 %% transition by main_loop and on every WP pop by step/3.
323 -spec print_waypoints(traj_state()) -> ok.
324 print_waypoints(S = #traj_state{}) ->
325     print_block("WPS_FULL", wps_full(S)),
326     print_block("WPS_TRAVERSED", wps_traversed(S)),
327     print_block("WPS_REMAINING", wps_remaining(S)),
328     ok.
329
330 print_block(Label, []) ->
331     io:format("~s: (empty)~n", [Label]);
332 print_block(Label, WPs) ->
333     io:format("~s: ~p waypoint(s)~n", [Label, length(WPs)]),
334     lists:foldl(
335         fun({X, Y}, I) ->
336             io:format(" #~2..0B x=~7.2f y=~7.2f~n", [I, X, Y]),
337             I + 1
338         end, 1, WPs),
339     ok.
340
341 %% Emit a one-line transition tag followed by the 3-block WP dump. Called by
342 %% main_loop on every lifecycle transition the operator should see.
343 -spec print_transition(atom(), atom(), traj_state()) -> ok.
344 print_transition(From, To, S) ->
345     io:format("===STATE: ~p -> ~p===~n", [From, To]),
346     print_waypoints(S).
347
348 %%% =====
349 %%% Sub-state controller
350 %%% =====
351
352 %% Empty waypoint list -- declare done. Reached only on the first running tick
353 %% when no WPs were ever queued; mid-run cancel-drain is handled by main_loop's
354 %% override which transitions out of running before step/3 runs.
355 substep(_X, _Y, _Theta_Rad, _Speed, Path, [], SubState, Counter) ->
356     {0.0, 0.0, Path, [], SubState, Counter, true};
357
358 substep(X, Y, Theta_Rad, Speed, Path, [{WPx, WPy} | Rest_WPs] = WPs_Rem, SubState, Counter) ->
359     Dx_WP = WPx - X,
360     Dy_WP = WPy - Y,
361     Dist_To_WP = math:sqrt(Dx_WP*Dx_WP + Dy_WP*Dy_WP),
362
363     Is_Last = (Rest_WPs == []),
364     %% v2/(2a) + margin -- collapses to BRAKE_MARGIN at standstill, expands with
365     %% approach speed so the stop point always lands inside WAYPOINT_REACHED.
366     Brake_Dist = (Speed * Speed) / (2.0 * ?BRAKE_RATE) + ?BRAKE_MARGIN,
367
368     case SubState of
369     cruise ->
370         if (not Is_Last) andalso (Dist_To_WP < ?WAYPOINT_REACHED) ->
371             %% Intermediate WP: pass through without braking or dwelling.
372             %% The spline is smooth at WPs, so stopping here is wasteful.
373             %% Recurse to evaluate the next WP this same tick.
374             substep(X, Y, Theta_Rad, Speed, Path, Rest_WPs, cruise, 0);
375             Is_Last andalso (Dist_To_WP < Brake_Dist) ->
376                 {0.0, 0.0, Path, WPs_Rem, braking, 0, false};
377             Is_Last andalso (Dist_To_WP < ?LOOKAHEAD) ->
378                 %% Pure-pursuit lookahead collapses onto the last WP inside
379                 %% this radius (no path point is LOOKAHEAD-far anymore), which
380                 %% saturates Turn_V and makes the robot spin around the WP.

```

```

381         %% Coast straight, tapering advance toward MIN_CORNER_SPEED so
382         %% the existing Brake_Dist trigger still takes over cleanly.
383         Adv_Tgt = ?MAX_SPEED * clamp(Dist_To_WP / ?LOOKAHEAD,
384                                     ?MIN_CORNER_SPEED / ?MAX_SPEED,
385                                     1.0),
386         {Adv_Tgt, 0.0, Path, WPs_Rem, cruise, 0, false};
387     Path =:= [] ->
388         %% Defensive: mutators rebuild the spline so a non-empty
389         %% wps_remaining normally implies a non-empty path. This
390         %% branch only fires across the boundary where Clear_All
391         %% has just wiped path before the running→finished lands.
392         {0.0, 0.0, Path, WPs_Rem, cruise, 0, false};
393     true ->
394         Path_New = pure_pursuit:advance(Path, X, Y, Theta_Rad),
395         {Lx, Ly} = pure_pursuit:find_lookahead(Path_New, X, Y, ?LOOKAHEAD),
396         Kappa = pure_pursuit:curvature({Lx, Ly}, X, Y, Theta_Rad),
397         Turn_Tgt = clamp(Kappa * ?MAX_SPEED * ?WHEEL_BASE / 2.0,
398                         -?MAX_TURN_V, ?MAX_TURN_V),
399         Spd_Factor = clamp(1.0 - abs(Kappa) * ?WHEEL_BASE,
400                           ?MIN_CORNER_SPEED / ?MAX_SPEED, 1.0),
401         {?MAX_SPEED * Spd_Factor, Turn_Tgt, Path_New, WPs_Rem, cruise, 0, false}
402     end;
403
404     braking ->
405         if (Dist_To_WP < ?WAYPOINT_REACHED) andalso (abs(Speed) < ?SETTLE_VEL) ->
406             {0.0, 0.0, Path, WPs_Rem, dwelling, ?DWELL_TICKS, false};
407         true ->
408             {0.0, 0.0, Path, WPs_Rem, braking, 0, false}
409         end;
410
411     dwelling ->
412         if Counter =< 0 ->
413             %% With the new "no intermediate dwell" rule (only the last
414             %% WP enters braking→dwelling), Rest_WPs is always [] here.
415             %% Keep the case for defensive completeness in case future
416             %% changes restore dwell at intermediates.
417             case Rest_WPs of
418                 [] -> {0.0, 0.0, Path, [], dwelling, 0, true};
419                 _ -> {0.0, 0.0, Path, Rest_WPs, cruise, 0, false}
420             end;
421         true ->
422             {0.0, 0.0, Path, WPs_Rem, dwelling, Counter - 1, false}
423         end
424     end.
425
426     %%% --- Utilities -----
427
428     dist_to_head(X, _Y, []) -> 0.0;
429     dist_to_head(X, Y, [{Wx, Wy} | _]) ->
430         Dx = Wx - X, Dy = Wy - Y,
431         math:sqrt(Dx*Dx + Dy*Dy).
432
433     clamp(V, Lo, Hi) ->
434         if V < Lo -> Lo; V > Hi -> Hi; true -> V end.

```

A.1.6 spline.erl

```

1  -module(spline).
2
3  %%% =====
4  %%% Catmull-Rom spline generation.
5  %%%
6  %%% build_path/2 takes an ordered list of {X,Y} waypoints and returns a dense
7  %%% list of {X,Y} subpoints suitable for pure_pursuit lookahead.
8  %%%
9  %%% The first and last waypoints are reflected outward to create phantom
10 %%% control points so every segment starts and ends with the correct tangent.
11 %%%
12 %%% build_path_continuing/3 is the mid-cruise rebuild variant. Instead of
13 %%% reflecting the second waypoint to seed the initial tangent, it takes an
14 %%% explicit Seed control point -- typically the second-last visited waypoint,

```

```

15  %% a point the robot has already passed through. That preserves the tangent
16  %% at the splice with the previous path so pure_pursuit doesn't see a step.
17  %% =====
18
19  -export([build_path/2, build_path_continuing/3, config_string/0]).
20
21  -define(SPLINE_RES, 30).    %% subpoints per segment
22
23  %% Number of subpoints depends on waypoint count and SPLINE_RES.
24  -spec build_path([float(), float()], pos_integer()) -> [float(), float()].
25  build_path(WPs, _Res) ->
26      [P1, P2 | _] = WPs,
27      P_Last = lists:last(WPs),
28      P_Prev = lists:nth(length(WPs) - 1, WPs),
29      Extended = [reflect(P1, P2) | WPs] ++ [reflect(P_Last, P_Prev)],
30      sample_all_segments(Extended).
31
32  %% Mid-cruise / fresh-start rebuild. Seed is a real endpoint of the spline:
33  %% a leading phantom reflected from WP1 around Seed makes the first sampled
34  %% segment span Seed → WP1, so the path actually starts at the robot's pose
35  %% (or at second_last_visited_wp on mid-run rebuilds). Tangent at WP1 stays
36  %% (WP2 - Seed)/2, identical to the old build, so downstream segments are
37  %% unchanged. Matches the operator UI's spline preview, which already
38  %% includes the robot pose as the first anchor.
39  -spec build_path_continuing(float(), float()),
40      [float(), float()],
41      pos_integer() ->
42      [float(), float()].
43  build_path_continuing(_Seed, [], _Res) -> [];
44  build_path_continuing(_Seed, [_Single], _Res) -> [];
45  build_path_continuing(Seed, WPs = [WP1 | _], _Res) ->
46      P_Last = lists:last(WPs),
47      P_Prev = lists:nth(length(WPs) - 1, WPs),
48      Leading = reflect(Seed, WP1),
49      Trailing = reflect(P_Last, P_Prev),
50      Extended = [Leading, Seed | WPs] ++ [Trailing],
51      sample_all_segments(Extended).
52
53  -spec config_string() -> string().
54  config_string() ->
55      io_lib:format("spline: SPLINE_RES=~p", [?SPLINE_RES]).
56
57  %%% --- Internal -----
58
59  reflect({X1, Y1}, {X2, Y2}) ->
60      {2.0 * X1 - X2, 2.0 * Y1 - Y2}.
61
62  sample_all_segments([P0, P1, P2, P3 | Rest]) ->
63      sample_segment(P0, P1, P2, P3) ++ sample_all_segments([P1, P2, P3 | Rest]);
64  sample_all_segments(_) ->
65      [].
66
67  sample_segment(P0, P1, P2, P3) ->
68      N = ?SPLINE_RES,
69      [catmull_rom(P0, P1, P2, P3, float(I) / float(N)) || I <- lists:seq(0, N - 1)].
70
71  catmull_rom({X0,Y0}, {X1,Y1}, {X2,Y2}, {X3,Y3}, T) ->
72      T2 = T*T, T3 = T2*T,
73      X = 0.5 * ((2.0*X1)
74          + (-X0 + X2) * T
75          + ( 2.0*X0 - 5.0*X1 + 4.0*X2 - X3) * T2
76          + (-X0 + 3.0*X1 - 3.0*X2 + X3) * T3),
77      Y = 0.5 * ((2.0*Y1)
78          + (-Y0 + Y2) * T
79          + ( 2.0*Y0 - 5.0*Y1 + 4.0*Y2 - Y3) * T2
80          + (-Y0 + 3.0*Y1 - 3.0*Y2 + Y3) * T3),
81      {X, Y}.

```

A.1.7 pure_pursuit.erl

```

1 -module(pure_pursuit).
2
3 %%% =====
4 %%% Pure Pursuit path-following helpers.
5 %%%
6 %%% advance/4 -- drop path points that are already behind the robot.
7 %%% find_lookahead/4 -- pick the first path point at least LOOKAHEAD cm away.
8 %%% curvature/4 -- signed curvature  $\kappa$  from robot pose to a lookahead point.
9 %%%
10 %%% All functions are stateless pure math; caller owns the path list.
11 %%% =====
12
13 -export([advance/4, find_lookahead/4, curvature/4, config_string/0]).
14
15 -define(LOOKAHEAD, 12.0).    %% cm -- must match traj_planner.erl's ?LOOKAHEAD
16
17 %% Drop path points already behind the robot (local_x < 0 in robot frame).
18 -spec advance([float(), float()], float(), float(), float()) -> [float(), float()].
19 advance([], _, _, _) -> [];
20 advance([_] = Path, _, _, _) -> Path;
21 advance([Px, Py | Rest] = Path, X, Y, Theta_Rad) ->
22     Dx = Px - X, Dy = Py - Y,
23     Local_X = Dx * math:cos(Theta_Rad) + Dy * math:sin(Theta_Rad),
24     if Local_X < 0.0 -> advance(Rest, X, Y, Theta_Rad);
25     true -> Path
26 end.
27
28 %% Return the first path point at least LOOKAHEAD cm from (X,Y).
29 %% Falls back to the last point if none is far enough.
30 -spec find_lookahead([float(), float()], float(), float(), float()) -> {float(), float()}.
31 find_lookahead([Last], _, _, _) -> Last;
32 find_lookahead([Px, Py] = P | Rest, X, Y, Lookahead) ->
33     Dx = Px - X, Dy = Py - Y,
34     if Dx*Dx + Dy*Dy >= Lookahead * Lookahead -> P;
35     true -> find_lookahead(Rest, X, Y, Lookahead)
36 end.
37
38 %% Signed curvature  $\kappa$  from robot position+heading to lookahead point.
39 %% Positive = turn left, negative = turn right (robot-frame convention).
40 %% Returns 0 if the lookahead coincides with the robot (degenerate case).
41 -spec curvature({float(), float()}, float(), float(), float()) -> float().
42 curvature({Lx, Ly}, X, Y, Theta_Rad) ->
43     Dx = Lx - X, Dy = Ly - Y,
44     L2 = Dx*Dx + Dy*Dy,
45     if L2 > 0.001 ->
46         Local_Y = -Dx * math:sin(Theta_Rad) + Dy * math:cos(Theta_Rad),
47         2.0 * Local_Y / L2;
48     true -> 0.0
49 end.
50
51 -spec config_string() -> string().
52 config_string() ->
53     io_lib:format("pure_pursuit: LOOKAHEAD=~p cm", [?LOOKAHEAD]).

```

A.1.8 odometry.erl

```

1 -module(odometry).
2
3 %%% =====
4 %%% Differential-drive pose integration from wheel speeds.
5 %%%
6 %%% integrate/3 updates a #pose{} record every tick using the midpoint-angle
7 %%% approximation (Euler with heading averaged over the tick).
8 %%%
9 %%% Called every tick regardless of trajectory lifecycle so the pose
10 %%% continues to track reality during paused/idle/finished states.
11 %%% =====
12
13 -export([integrate/3, config_string/0]).
14 -include("robot_types.hrl").
15

```

```

16 -define(WHEEL_BASE, 18.5).    %% cm, left-to-right distance between contact patches
17
18 %% Speed_L, Speed_R in cm/s; Dt in s.
19 -spec integrate({float(), float()}, float(), #pose{}) -> #pose{}.
20 integrate({Speed_L, Speed_R}, Dt, Pose) ->
21   #pose{x = X, y = Y, theta = Theta} = Pose,
22   Speed = (Speed_L + Speed_R) / 2.0,
23   D_Dist = Speed * Dt,
24
25   D_Theta_Rad = ((Speed_R - Speed_L) / ?WHEEL_BASE) * Dt,
26   D_Theta_Deg = D_Theta_Rad * 180.0 / math:pi(),
27   Theta_New = norm_angle(Theta + D_Theta_Deg),
28
29   %% Midpoint heading for position integration reduces error vs. using start heading.
30   Theta_Mid_Rad = ((Theta + Theta_New) / 2.0) * math:pi() / 180.0,
31   X_New = X + D_Dist * math:cos(Theta_Mid_Rad),
32   Y_New = Y + D_Dist * math:sin(Theta_Mid_Rad),
33   #pose{x = X_New, y = Y_New, theta = Theta_New}.
34
35 -spec config_string() -> string().
36 config_string() ->
37   io_lib:format("odometry: WHEEL_BASE=~p cm", [?WHEEL_BASE]).
38
39 %%% --- Internal -----
40
41 norm_angle(Angle) ->
42   A = math:fmod(Angle, 360.0),
43   if A > 180.0 -> A - 360.0;
44   A < -180.0 -> A + 360.0;
45   true -> A
46 end.

```

A.1.9 stability_engine.erl

```

1 -module(stability_engine).
2
3 -export([controller/4]).
4
5 -define(ADV_V_MAX, 15.0). %originally 30.0
6 -define(ADV_ACCEL, 40.0). %original 75.0
7
8 -define(TURN_V_MAX, 40.0). %original 80.0
9 -define(TURN_ACCEL, 200.0). %original 400.0
10
11
12 %V_ref_new must be looped to V_ref
13 controller({Dt, Angle, Speed}, {Pid_Speed, Pid_Stability}, {Adv_V_Goal, Adv_V_Ref}, {Turn_V_Goal,
14   Turn_V_Ref}) ->
15
16   %% Advance velocity ramp -- track the GOAL value (not just its sign).
17   %% Adv_V_Ref moves toward Adv_V_Goal at rate ADV_ACCEL, clamped to ±ADV_V_MAX.
18   %% Downstream Speed PI / Stability PD are unchanged -- they still consume
19   %% Adv_V_Ref exactly as before; only what value gets fed there has changed.
20   Adv_Diff = Adv_V_Goal - Adv_V_Ref,
21   Adv_V_Ref_Pre =
22     if Adv_Diff > 0.0 -> Adv_V_Ref + min(Adv_Diff, ?ADV_ACCEL*Dt);
23     Adv_Diff < 0.0 -> Adv_V_Ref + max(Adv_Diff, -?ADV_ACCEL*Dt);
24     true -> Adv_V_Ref
25   end,
26   %% Snap to exact 0 inside a small deadband when Goal is 0, to avoid float drift.
27   Adv_V_Ref_New = pid_controller:saturation(
28     if (Adv_V_Goal == 0.0) andalso (abs(Adv_V_Ref_Pre) < 0.5) -> 0.0;
29     true -> Adv_V_Ref_Pre
30   end, ?ADV_V_MAX),
31
32   %% Same magnitude-tracking ramp for turning.
33   Turn_Diff = Turn_V_Goal - Turn_V_Ref,
34   Turn_V_Ref_Pre =
35     if Turn_Diff > 0.0 -> Turn_V_Ref + min(Turn_Diff, ?TURN_ACCEL*Dt);
36     Turn_Diff < 0.0 -> Turn_V_Ref + max(Turn_Diff, -?TURN_ACCEL*Dt);
37     true -> Turn_V_Ref

```

```

37     end,
38     Turn_V_Ref_New = pid_controller:saturation(
39         if (Turn_V_Goal == 0.0) andalso (abs(Turn_V_Ref_Pre) < 0.5) -> 0.0;
40         true -> Turn_V_Ref_Pre
41     end, ?TURN_V_MAX),
42
43     %Speed PI
44     Pid_Speed ! {self(), {set_point, Adv_V_Ref_New}},
45     Pid_Speed ! {self(), {input, Speed}},
46     receive {_, {control, Target_angle}} -> ok end,
47
48     %TODO: send Target_angle to log
49
50     % io:format("~p~n",[Target_angle]),
51
52     %Stability PD
53     Pid_Stability ! {self(), {set_point, Target_angle}},
54     Pid_Stability ! {self(), {input, Angle}},
55     receive {_, {control, Acc}} -> ok end,
56
57     {Acc, Adv_V_Ref_New, Turn_V_Ref_New}.

```

A.1.10 robot.erl

```

1  -module(robot).
2
3  -export([robot_init/1,modify_frequency/1]).
4
5  -define(RAD_TO_DEG, 180.0/math:pi()).
6  -define(DEG_TO_RAD, math:pi()/180.0).
7
8  %Advance constant
9  -define(ADV_V_MAX, 30.0).
10
11 %Turning constant
12 -define(TURN_V_MAX, 80.0).
13
14 %Angle at which robot is considered "down"
15 -define(MAX_ANGLE, 25.0).
16
17 %Coefficient for the complementary filter
18 -define(COEF_FILTER, 0.667).
19
20 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
21 %% Robot
22 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
23
24 robot_init(Hera_pid) ->
25
26     process_flag(priority, max),
27
28     %Starting timestamp
29     T0 = erlang:system_time()/1.0e6,
30
31     %Table for global variables
32     ets:new(variables, [set, public, named_table]),
33     ets:insert(variables, {"Freq_Goal", 300.0}),
34
35     %Calibration
36     io:format("Calibrating... Do not move the pmod_nav!~n"),
37     grisp_led:color(1, {1, 0, 0}),
38     grisp_led:color(2, {1, 0, 0}),
39     [_Gx0,Gy0,_Gz0] = calibrate(),
40     io:format("Done calibrating~n"),
41
42     %Kalman matrices
43     X0 = mat:matrix([[0], [0]]),
44     P0 = mat:matrix([[0.1, 0], [0, 0.1]]),
45
46     %I2C bus
47     I2Cbus = grisp_i2c:open(i2c1),

```

```

48
49 %PIDs initialisation
50 Pid_Speed = spawn(pid_controller, pid_init, [-0.12, -0.07, 0.0, -1, 60.0, 0.0]),
51 Pid_Stability = spawn(pid_controller, pid_init, [17.0, 0.0, 4.0, -1, -1, 0.0]),
52 io:format("Pid of the speed controller: ~p.~n", [Pid_Speed]),
53 io:format("Pid of the stability controller: ~p.~n", [Pid_Stability]),
54 io:format("Starting movement of the robot.~n"),
55
56 %Call main loop
57 robot_main(T0, Hera_pid, {rest, false}, {T0, X0, P0}, I2Cbus, {0, T0, []}, {Gy0, 0.0, 0.0}, {
    Pid_Speed, Pid_Stability}, {0.0, 0.0}, {0, 0, 200.0, T0}).
58
59 robot_main(Start_Time, Hera_pid, {Robot_State, Robot_Up}, {T0, X0, P0}, I2Cbus, {Logging, Log_End,
    Log_List}, {Gy0, Angle_Complem, Angle_Rate}, {Pid_Speed, Pid_Stability}, {Adv_V_Ref, Turn_V_Ref}, {
    N, Freq, Mean_Freq, T_End}) ->
60
61 %Delta time of loop
62 T1 = erlang:system_time()/1.0e6, %[ms]
63 Dt = (T1- T0)/1000.0,          %[s]
64
65 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
66 %%% Input from Sensor %%%
67 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
68
69 %Read data
70 [Gy,Ax,Az] = pmod_nav:read(acc, [out_y_g, out_x_xl, out_z_xl], #{g_unit => dps}),
71
72 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
73 %%% Input from ESP32 %%%
74 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
75
76 %Receive I2C and conversion
77 [<<SL1,SL2,SR1,SR2,CtrlByte>>] = grisp_i2c:transfer(I2Cbus, [{read, 16#40, 1, 5}]),
78 [Speed_L,Speed_R] = hera_com:decode_half_float([<<SL1, SL2>>, <<SR1, SR2>>]),
79 Speed = (Speed_L + Speed_R)/2,
80
81 %Retrieve flags from ESP32
82 [Arm_Ready, Switch, Test, Get_Up, Forward, Backward, Left, Right] = hera_com:get_bits(CtrlByte),
83
84 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
85 %%% Command Logic %%%
86 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
87
88 %Set advance speed from flags
89 Adv_V_Goal = speed_ref(Forward, Backward),
90
91 %Set turning speed from flags
92 Turn_V_Goal = turn_ref(Left, Right),
93
94 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
95 %%% Angle Computations %%%
96 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
97
98 %Angle based directly on the sensors
99 Angle_Accelerometer = math:atan(Az / (-Ax))*RAD_TO_DEG,
100
101 %Kalman filter computation
102 {X1, P1} = kalman_angle(Dt, Ax, Az, Gy, Gy0, X0, P0),
103 [Th_Kalman, _W_Kalman] = mat:to_array(X1),
104 Angle_Kalman = Th_Kalman*RAD_TO_DEG,
105
106 %%%
107 T_after_kalman = erlang:system_time()/1.0e6, %[ms]
108 %%%
109
110 %Complementary angle computation
111 K = 1.25/(1.25+(1.0/Mean_Freq)),
112 {Angle_Complem_New, Angle_Rate_New} = complem_angle({Dt, Ax, Az, Gy, Gy0, K, Angle_Complem,
    Angle_Rate}),
113
114 %Select angle between kalman or complementary
115 Angle = select_angle(Switch, Angle_Kalman, Angle_Complem),
116

```

```

117
118 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
119 %%% Controller %%%
120 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
121
122 %Takes as input:
123 % -Measures
124 % -Process IDs of the two PID controllers
125 % -Advance and turning velocity to reach and loopback velocity of trapezoidal profile
126
127 {Acc, Adv_V_Ref_New, Turn_V_Ref_New} = stability_engine:controller({Dt, Angle, Speed}, {Pid_Speed,
Pid_Stability}, {Adv_V_Goal, Adv_V_Ref}, {Turn_V_Goal, Turn_V_Ref}),
128 %Gives as output:
129 % -Acceleration of the motors -> sent to ESP32
130 % -Loopback V_Ref for advance
131 % -Loopback V_Ref for turning -> sent to ESP32
132
133 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
134 %%% Output to ESP32 %%%
135 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
136
137 %State of the robot
138 if
139     Robot_Up and (abs(Angle) > 20.0) ->
140         Robot_Up_New = false;
141     not Robot_Up and (abs(Angle) < 18.0) ->
142         Robot_Up_New = true;
143     true ->
144         Robot_Up_New = Robot_Up
145 end,
146 if
147     Angle > 0.0 ->
148         F_B = 1;
149     true ->
150         F_B = 0
151 end,
152
153 %State change
154 case Robot_State of
155     rest ->
156         if
157             Get_Up -> Next_Robot_State = raising;
158             true -> Next_Robot_State = rest
159         end;
160     raising ->
161         if
162             Robot_Up -> Next_Robot_State = stand_up;
163             not Get_Up -> Next_Robot_State = soft_fall;
164             true -> Next_Robot_State = raising
165         end;
166     stand_up ->
167         if
168             not Get_Up -> Next_Robot_State = wait_for_extend;
169             not Robot_Up -> Next_Robot_State = rest;
170             true -> Next_Robot_State = stand_up
171         end;
172     wait_for_extend -> %Buffer state while Next_Robot_State switches state
173         Next_Robot_State = prepare_arms;
174     prepare_arms ->
175         if
176             Arm_Ready -> Next_Robot_State = free_fall;
177             Get_Up -> Next_Robot_State = stand_up;
178             not Robot_Up -> Next_Robot_State = rest;
179             true -> Next_Robot_State = prepare_arms
180         end;
181     free_fall ->
182         if
183             abs(Angle) > 10 -> Next_Robot_State = wait_for_retract;
184             true -> Next_Robot_State = free_fall
185         end;
186     wait_for_retract -> %Buffer state while Next_Robot_State switches state
187         Next_Robot_State = soft_fall;
188     soft_fall ->

```

```

189         if
190             Arm_Ready -> Next_Robot_State = rest;
191             Get_Up -> Next_Robot_State = raising;
192             true -> Next_Robot_State = soft_fall
193         end
194     end,
195
196     %State output
197     case Next_Robot_State of
198         rest ->
199             Power      = 0,
200             Freeze     = 0,
201             Extend     = 0,
202             Robot_Up_Bit = 0;
203         raising ->
204             Power      = 1,
205             Freeze     = 0,
206             Extend     = 1,
207             Robot_Up_Bit = 0;
208         stand_up ->
209             Power      = 1,
210             Freeze     = 0,
211             Extend     = 0,
212             Robot_Up_Bit = 1;
213         wait_for_extend ->
214             Power      = 1,
215             Freeze     = 0,
216             Extend     = 1,
217             Robot_Up_Bit = 1;
218         prepare_arms ->
219             Power      = 1,
220             Freeze     = 0,
221             Extend     = 1,
222             Robot_Up_Bit = 1;
223         free_fall ->
224             Power      = 1,
225             Freeze     = 1,
226             Extend     = 1,
227             Robot_Up_Bit = 1;
228         wait_for_retract ->
229             Power      = 1,
230             Freeze     = 0,
231             Extend     = 0,
232             Robot_Up_Bit = 0;
233         soft_fall ->
234             Power      = 1,
235             Freeze     = 0,
236             Extend     = 0,
237             Robot_Up_Bit = 0
238     end,
239
240     %Send output to ESP32
241     Output_Byte = get_byte([Power, Freeze, Extend, Robot_Up_Bit, F_B, 0, 0, 0]),
242     [HF1, HF2] = hera_com:encode_half_float([Acc, Turn_V_Ref_New]),
243     grisp_i2c:transfer(I2Cbus, [{write, 16#40, 1, [HF1, HF2, <<Output_Byte>>]}]),
244
245     %%%
246     %%% Testing Features %%%
247     %%%
248
249     %Frequency computation
250     {N_New, Freq_New, Mean_Freq_New} = frequency_computation(Dt, N, Freq, Mean_Freq),
251
252     %Logging
253     if
254         Test ->
255             Log_End_New = erlang:system_time()/1.0e6 + 6000.0;
256         true ->
257             Log_End_New = Log_End
258     end,
259     Logging_New = erlang:system_time()/1.0e6 < Log_End_New,
260
261     %LED flickering while logging

```

```

262 if
263     Logging_New ->
264         if
265             N rem 9 < 4 ->
266                 grisp_led:color(1, {1, 1, 0}),
267                 grisp_led:color(2, {1, 1, 0});
268             true->
269                 grisp_led:color(1, {0, 0, 0}),
270                 grisp_led:color(2, {0, 0, 0})
271         end;
272     true ->
273         ok
274 end,
275
276 %Check for start or end of logging sequence
277 if
278     not Logging and Logging_New ->
279         io:format("send start log-n"),
280         Hera_pid ! {self(), start_log};
281     Logging and not Logging_New ->
282         io:format("send stop log-n"),
283         grisp_led:color(1, {1, 1, 0}),
284         grisp_led:color(2, {1, 1, 0}),
285         Hera_pid ! {self(), stop_log};
286     true ->
287         ok
288 end,
289
290 %Send values to ESP32 if asked, else stack up values in list
291 receive
292     {From, log_values} ->
293         io:format("send log vals-n"),
294         From ! {self(), log, Log_List},
295         Log_List_New = []
296 after 0 ->
297     if
298         Logging_New ->
299             Log_List_New = lists:append(Log_List, [T1-Start_Time, 1/Dt, Gy, Acc, CtrlByte,
300             Angle_Accelerometer, Angle_Kalman, Angle_Complem, Adv_V_Ref, Switch]);
301         true ->
302             Log_List_New = Log_List
303     end
304 end,
305
306 %Communication with Hera (more messages can be implemented by the user)
307 receive
308     {From1, get_all_data} -> From1 ! {self(), data, [T1-Start_Time, 1/Dt, Gy, Acc, CtrlByte,
309     Angle_Accelerometer, Angle_Kalman, Angle_Complem, Adv_V_Ref, Switch]};
310     {From1, freq} -> From1 ! {self(), 1/Dt};
311     {From2, acc} -> From2 ! {self(), Acc};
312     {_, Msg} -> io:format("Message [-p] not recognized. ~nPossible querries are: [get_all_data,
313     freq, acc]. ~nMore querries can be added.", [Msg])
314 after 0 ->
315     ok
316 end,
317
318 %Imposed maximum frequency
319 T2 = erlang:system_time()/1.0e6,
320 [{_,Freq_Goal}] = ets:lookup(variables, "Freq_Goal"),
321 Delay_Goal = 1.0/Freq_Goal * 1000.0,
322 if
323     T2-T_End < Delay_Goal ->
324         wait(Delay_Goal-(T2-T1));
325     true ->
326         ok
327 end,
328 T_End_New = erlang:system_time()/1.0e6,
329
330 %Loop back with updated state
331 robot_main(Start_Time, Hera_pid, {Next_Robot_State, Robot_Up_New}, {T1, X1, P1}, I2Cbus, {
332     Logging_New, Log_End_New, Log_List_New}, {Gy0, Angle_Complem_New, Angle_Rate_New}, {Pid_Speed,
333     Pid_Stability}, {Adv_V_Ref_New, Turn_V_Ref_New}, {N_New, Freq_New, Mean_Freq_New, T_End_New}).

```

```

330
331
332 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
333 %% Internal functions
334 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
335
336 calibrate() ->
337     N = 500,
338     Data = [list_to_tuple(pmod_nav:read(acc, [out_x_g, out_y_g, out_z_g])) || _ <- lists:seq(1,N)],
339     {X, Y, Z} = lists:unzip3(Data),
340     [lists:sum(X)/N, lists:sum(Y)/N, lists:sum(Z)/N]. % [Gx0, Gy0, Gz0]
341
342 kalman_angle(Dt, Ax, Az, Gy, Gy0, X0, P0) ->
343     R = mat:matrix([[3.0, 0.0], [0, 3.0e-6]]),
344     Q = mat:matrix([[3.0e-5, 0.0], [0.0, 10.0]]),
345     F = fun (X) -> [Th, W] = mat:to_array(X),
346         mat:matrix([ [Th+Dt*W],
347                     [W      ] ]),
348     end,
349     Jf = fun (X) -> [_Th, _W] = mat:to_array(X),
350         mat:matrix([ [1, Dt],
351                     [0, 1 ] ]),
352     end,
353     H = fun (X) -> [Th, W] = mat:to_array(X),
354         mat:matrix([ [Th],
355                     [W ] ]),
356     end,
357     Jh = fun (X) -> [_Th, _W] = mat:to_array(X),
358         mat:matrix([ [1, 0],
359                     [0, 1 ] ]),
360     end,
361     Z = mat:matrix([[math:atan(Az / (-Ax))], [(Gy-Gy0)*?DEG_TO_RAD]]),
362     kalman:ekf({X0, P0}, {F, Jf}, {H, Jh}, Q, R, Z).
363
364 complem_angle({Dt, Ax, Az, Gy, Gy0, K, Angle_Complem, Angle_Rate}) ->
365
366     Angle_Rate_New = (Gy - Gy0) * ?COEF_FILTER + Angle_Rate * (1 - ?COEF_FILTER),
367
368     %Angle increment computed from gyroscope
369     Delta_Gyr = Angle_Rate_New * Dt,
370
371     %Absolute angle computed form accelerometer
372     Angle_Acc = math:atan(Az / (-Ax)) * 180 / math:pi(),
373
374     %Complementary filter combining gyroscope and accelerometer
375     Angle_Complem_New = (Angle_Complem + Delta_Gyr) * K + Angle_Acc * (1 - K),
376
377     {Angle_Complem_New, Angle_Rate_New}.
378
379 select_angle(Switch, Angle_Kalman, Angle_Complem) ->
380     if
381         Switch ->
382             Angle = Angle_Kalman;
383         true ->
384             Angle = Angle_Complem
385     end,
386     Angle.
387
388 speed_ref(Forward, Backward) ->
389     if
390         Forward ->
391             Adv_V_Goal = ?ADV_V_MAX;
392         Backward ->
393             Adv_V_Goal = - ?ADV_V_MAX;
394         true ->
395             Adv_V_Goal = 0.0
396     end,
397     Adv_V_Goal.
398
399 turn_ref(Left, Right) ->
400     if
401         Right ->
402             Turn_V_Goal = ?TURN_V_MAX;

```

```

403     Left ->
404         Turn_V_Goal = - ?TURN_V_MAX;
405     true ->
406         Turn_V_Goal = 0.0
407 end,
408 Turn_V_Goal.
409
410
411 frequency_computation(Dt, N, Freq, Mean_Freq) ->
412 if
413     N == 100 ->
414         io:format("robot freq:      ~.2f~n", [Freq]),
415         N_New = 0,
416         Freq_New = 0,
417         Mean_Freq_New = Freq,
418
419         %Reset LED color after some time
420         grisp_led:color(1, {1, 1, 0}),
421         grisp_led:color(2, {1, 1, 0});
422     true ->
423         N_New = N+1,
424         Freq_New = ((Freq*N)+(1/Dt))/(N+1),
425         Mean_Freq_New = Mean_Freq
426 end,
427 {N_New, Freq_New, Mean_Freq_New}.
428
429 wait(T) ->
430     Tnow = erlang:system_time()/1.0e6,
431     wait_help(Tnow, Tnow+T).
432 wait_help(Tnow, Tend) when Tnow >= Tend -> ok;
433 wait_help(_, Tend) ->
434     Tnow = erlang:system_time()/1.0e6,
435     wait_help(Tnow, Tend).
436
437 get_byte(List) ->
438     [A, B, C, D, E, F, G, H] = List,
439     A*128 + B*64 + C*32 + D*16 + E*8 + F*4 + G*2 + H.
440
441
442 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
443 %% Global variables
444 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
445
446 modify_frequency(Freq) ->
447     ets:insert(variables, {"Freq_Goal", Freq}),
448     ok.

```

A.1.11 input_combo.erl

```

1 -module(input_combo).
2
3 %%% =====
4 %%% Button combo edge and hold detection -- stateless helpers.
5 %%%
6 %%% The caller owns all timing state; these functions are pure transforms.
7 %%%
8 %%% rising_edge/2 -- true on the single tick where a combo goes false→true.
9 %%% hold_check/3 -- accumulates held-time in ms; resets to 0 on release.
10 %%% Caller compares the returned value against its threshold.
11 %%%
12 %%% Short-press vs long-hold pattern (caller logic):
13 %%% - Track hold_check each tick.
14 %%% - Long hold fires when held_ms exceeds threshold, even without release.
15 %%% - Short press fires on the rising_edge tick that begins the press
16 %%% (or on release if the press was brief -- caller decides convention).
17 %%% - A cooldown timer after each action prevents double-firing.
18 %%% =====
19
20 -export([rising_edge/2, falling_edge/2, hold_check/3, config_string/0]).
21
22 %% True only on the single tick where Curr becomes true.

```

```

23 -spec rising_edge(boolean(), boolean()) -> boolean().
24 rising_edge(true, false) -> true;
25 rising_edge(_Curr, _Prev) -> false.
26
27 %% True only on the single tick where Curr becomes false.
28 %% Pair with the caller's stored held_ms to distinguish a tap (short held)
29 %% from the trailing release of a long-hold action.
30 -spec falling_edge(boolean(), boolean()) -> boolean().
31 falling_edge(false, true) -> true;
32 falling_edge(_Curr, _Prev) -> false.
33
34 %% Returns updated hold duration in ms.
35 %% Curr false      -> 0 (released, reset)
36 %% Curr true, was off -> Dt_ms (start counting)
37 %% Curr true, was on  -> Held_ms + Dt_ms (accumulate)
38 -spec hold_check(boolean(), boolean(), non_neg_integer()) -> non_neg_integer().
39 hold_check(false, _Prev, _Held_ms) -> 0;
40 hold_check(true, _Prev, Held_ms) -> Held_ms. %% caller adds Dt_ms before comparing
41
42 -spec config_string() -> string().
43 config_string() ->
44     "input_combo: (stateless, no tuning parameters)".

```

A.1.12 led_control.erl

```

1 -module(led_control).
2
3 %%% =====
4 %%% Named LED states -- maps system phase to LED colors.
5 %%%
6 %%% LED 1 = left indicator (system status).
7 %%% LED 2 = right indicator (trajectory sub-phase).
8 %%%
9 %%% LED table:
10 %%% Phase          LED1      LED2
11 %%% Booting / accel cal    red      red
12 %%% Idle (post-boot)      green    green
13 %%% Trajectory running    green    cyan
14 %%% Trajectory paused     green    yellow
15 %%% Traj reset cooldown (1 s) green    blue
16 %%%
17 %%% cmd-rx overrides only on LED1 (LED2 stays w/ lifecycle so we still see it):
18 %%% cmd-rx active        yellow  (sticky while fsm not idle)
19 %%% cmd-rx frame ok      white   (brief pulse per frame)
20 %%% cmd-rx committed     white   (~250ms)
21 %%% cmd-rx proto err     red     (~500ms)
22 %%% =====
23
24 -export([accel_calibrating/0, idle/0,
25         traj_running/0, traj_paused/0, traj_reset_cooldown/0,
26         cmd_active/0, cmd_frame_ok/0, cmd_committed/0, cmd_error/0]).
27
28 accel_calibrating() -> set({1,0,0}, {1,0,0}).
29 idle() -> set({0,1,0}, {0,1,0}).
30
31 traj_running() -> set({0,1,0}, {0,1,1}).
32 traj_paused() -> set({0,1,0}, {1,1,0}).
33 traj_reset_cooldown() -> set({0,1,0}, {0,0,1}).
34
35 cmd_active() -> set1({1,1,0}).
36 cmd_frame_ok() -> set1({1,1,1}).
37 cmd_committed() -> set1({1,1,1}).
38 cmd_error() -> set1({1,0,0}).
39
40 %%% --- Internal -----
41
42 set(C1, C2) ->
43     grisp_led:color(1, C1),
44     grisp_led:color(2, C2).
45
46 set1(C1) ->

```

```
47 | grisp_led:color(1, C1).
```

A.1.13 csv_logger.erl

```
1 | -module(csv_logger).
2 |
3 | %%% =====
4 | %%% Serial trajectory logger.
5 | %%%
6 | %%% Rows are rate-limited (every LOG_RATE_DIV ticks) and emitted immediately
7 | %%% to the serial port for real-time capture by the Python script.
8 | %%% No SD card buffering.
9 | %%%
10 | %%% CSV column order:
11 | %%%   T_ms, lifecycle, substate, x, y, speed,
12 | %%%   adv_v, turn_v, dist_wp, wps_left, yaw_odo
13 | %%% =====
14 |
15 | -export([reset/0, append/0, emit_serial/1]).
16 |
17 | -define(LOG_RATE_DIV, 10).    %% emit every Nth tick; at 300 Hz → 30 Hz serial rate
18 |
19 | -define(CSV_HEADER,
20 |   "T_ms,lifecycle,substate,x,y,speed,"
21 |   "adv_v,turn_v,dist_wp,wps_left,yaw_odo\n").
22 |
23 | %%% --- Public API -----
24 |
25 | %% Call on lifecycle idle→running to reset the tick counter and emit the header.
26 | -spec reset() -> ok.
27 | reset() ->
28 |   put(csv_t0, erlang:system_time(millisecond)),
29 |   put(csv_tick, 0),
30 |   io:format("===LOG_START_TRAJ===~n"),
31 |   io:format("TLOG_HEADER," ?CSV_HEADER),
32 |   ok.
33 |
34 | %% Rate-limit check: returns 'logged' every LOG_RATE_DIV ticks, 'skipped' otherwise.
35 | %% Caller should gate emit_serial/1 on the 'logged' return.
36 | -spec append() -> logged | skipped.
37 | append() ->
38 |   Tick = case get(csv_tick) of undefined -> 0; T -> T end,
39 |   put(csv_tick, Tick + 1),
40 |   if Tick rem ?LOG_RATE_DIV == 0 -> logged;
41 |     true -> skipped
42 |   end.
43 |
44 | %% Write one CSV row to serial immediately (for the Python capture script).
45 | -spec emit_serial(map()) -> ok.
46 | emit_serial(Row) ->
47 |   T0 = case get(csv_t0) of undefined -> erlang:system_time(millisecond); V -> V end,
48 |   T_ms = erlang:system_time(millisecond) - T0,
49 |   #{lifecycle := LC,
50 |     substate := SS,
51 |     x := X,
52 |     y := Y,
53 |     speed := Speed,
54 |     adv_v := Adv,
55 |     turn_v := Turn,
56 |     dist_wp := Dist,
57 |     wps_left := WpsL,
58 |     yaw_odo := Y0} = Row,
59 |   io:format("TLOG,~p,~p,~p,~.2f,~.2f,~.2f,~.2f,~.2f,~p,~.2f~n",
60 |     [T_ms, LC, SS, X, Y, Speed, Adv, Turn, Dist, WpsL, Y0]).
```

A.1.14 numerl.erl

```
1 | -module(numerl).
```

```

2 %-on_load(init/0).
3 -export([ eval/1, eye/1, zeros/2, equals/2, add/2, sub/2,mult/2, divide/2, matrix/1, rnd_matrix/1, get
   /3, at/2, mtfli/1, mtfl/1, row/2, col/2, transpose/1, inv/1, nrm2/1, vec_dot/2, dot/2, init/0]).
4
5 %Matrices are represented as such:
6 %-record(matrix, {n_rows, n_cols, bin}).
7
8 init()->
9     ok = erlang:load_nif(atom_to_list(?MODULE), 0).
10
11 %Creates a random matrix.
12 rnd_matrix(N)->
13     L = [[rand:uniform(20) || _ <- lists:seq(1,N) ] || _ <- lists:seq(1,N)],
14     matrix(L).
15
16 %Combine multiple functions.
17 eval([L,O,R|T])->
18     F = fun numer1:0/2,
19     eval([F(L,R) |T]);
20 eval([Res])->
21     Res.
22
23 %%Creates a matrix.
24 %List: List of doubles, of length N.
25 %Return: a matrix of dimension MxN, containing the data.
26 matrix(_) ->
27     nif_not_loaded.
28
29 %%Returns the Nth value contained within Matrix.
30 at(_Matrix,_Nth)->
31     nif_not_loaded.
32
33 %%Returns the matrix as a flattened list of ints.
34 mtfli(_mtrix)->
35     nif_not_loaded.
36
37 %%Returns the matrix as a flattened list of doubles.
38 mtfl(_mtrix)->
39     nif_not_loaded.
40
41 %%Returns a value from a matrix.
42 get(_,_,_) ->
43     nif_not_loaded.
44
45 %%Returns requested row.
46 row(_,_) ->
47     nif_not_loaded.
48
49
50 %%Returns requested col.
51 col(_,_) ->
52     nif_not_loaded.
53
54
55 %%Equality test between matrixes.
56 equals(_, _) ->
57     nif_not_loaded.
58
59
60 %%Addition of matrix.
61 add(_, _) ->
62     nif_not_loaded.
63
64
65 %%Subtraction of matrix.
66 sub(_, _) ->
67     nif_not_loaded.
68
69
70 %% Matrix multiplication.
71 mult(A,B) when is_number(B) -> '*_num'(A,B);
72 mult(A,B) -> '*_matrix'(A,B).
73

```

```

74 '*_num'(_,_) ->
75     nif_not_loaded.
76
77 '*_matrix'(_,_) ->
78     nif_not_loaded.
79
80 %Matrix division by a number
81 divide(_,_) ->
82     nif_not_loaded.
83
84
85 %% build a null matrix of size NxM
86 zeros(_,_) ->
87     nif_not_loaded.
88
89 %%Returns an Identity matrix NxN.
90 eye(_) ->
91     nif_not_loaded.
92
93 %Returns the transpose of the given square matrix.
94 transpose(_) ->
95     nif_not_loaded.
96
97 %Returns the inverse of asked square matrix.
98 inv(_) ->
99     nif_not_loaded.
100
101
102 %-----CBLAS-----
103
104 %nrm2
105 %Calculates the squared root of the sum of the squared contents.
106 nrm2(_) ->
107     nif_not_loaded.
108
109 % : dot product of two vectors
110 % Arguments: vector x, vector y.
111 %   x and y are matrices
112 % Returns the dot product of all the coordinates of X,Y.
113 vec_dot(_,_) ->
114     nif_not_loaded.
115
116 % dgemm: A dot B
117 % Arguments: Matrix A, Matrix B.
118 %   alpha, beta: numbers (float or ints) used as doubles.
119 %   A,B,C: matrices.
120 % Returns the matrice resulting of the operations alpha * A * B + beta * C.
121 dot(_,_) ->
122     nif_not_loaded.

```

A.2 Trajectory logger (Python)

The script below subscribes to the GRiSP serial port, decodes the TLOG, stream, and writes a per-run CSV mirror of the on-board SD-card log.

A.2.1 traj_log.py

```

1 #!/usr/bin/env python3
2 """
3 Trajectory log capture -- listens on the GRiSP serial port and records
4 trajectory log rows to a CSV file.
5
6 Protocol (emitted by csv_logger.erl in main_loop):
7     ===LOG_START_TRAJ===          start sentinel -- emitted on F+B press (idle→running)
8     TLOG_HEADER,col1,col2,...    CSV header line
9     TLOG,val1,val2,...           CSV data rows (~30 Hz, rate-limited in csv_logger)
10    ===LOG_END_TRAJ===            end sentinel -- emitted when trajectory finishes

```

```

11
12 Lines without the TLOG prefix between the sentinels (other shell chatter,
13 [Robot] messages, etc.) are ignored. Column headers are read dynamically
14 from the TLOG_HEADER line, so no changes are needed here when columns change.
15
16 Note: the CONFIG footer is written to the SD card (LOGS/traj_log.csv) by the
17 robot, not captured over serial. Pull it from the SD card after the run.
18
19 Usage:
20     pip install pyserial
21     python3 traj_log.py --port /dev/tty.usbserial-XXXX
22     python3 traj_log.py --port COM5 --out my_run.csv
23
24 Press Ctrl+C at any time to abort and save what's been captured so far.
25 """
26
27 import argparse
28 import csv
29 import re
30 import sys
31 import time
32
33 # Strips the Erlang shell prompt like "(balancing_robot@my_grisp_board)1> "
34 # that gets prepended to io:format output when running inside the shell.
35 PROMPT_RE = re.compile(r'^\s*([^\s]+\s+)\d+>\s*')
36
37 try:
38     import serial
39     import serial.tools.list_ports
40 except ImportError:
41     print("ERROR: pyserial not installed. Run: pip install pyserial", file=sys.stderr)
42     sys.exit(2)
43
44
45 def pick_port():
46     """List available serial ports and prompt user for a selection."""
47     ports = list(serial.tools.list_ports.comports())
48     print("\nAvailable Serial Ports:")
49     for i, p in enumerate(ports):
50         desc = f" ({p.description})" if p.description and p.description != "n/a" else ""
51         print(f"  [{i}] {p.device}{desc}")
52
53     if not ports:
54         print("No devices found. Check your connection.")
55         sys.exit(1)
56
57     while True:
58         choice = input("\nSelect port index: ").strip()
59         try:
60             idx = int(choice)
61             if 0 <= idx < len(ports):
62                 return ports[idx].device
63         except ValueError:
64             pass
65         print(f"Enter a number between 0 and {len(ports)-1}.")
66
67
68 # --- Sentinels & prefixes (must match the Erlang side) -----
69 START_SENT = "===LOG_START_TRAJ==="
70 END_SENT = "===LOG_END_TRAJ==="
71 HEADER_PFX = "TLOG_HEADER,"
72 ROW_PFX = "TLOG,"
73
74
75
76 def main():
77     ap = argparse.ArgumentParser()
78     ap.add_argument("--port", default=None,
79                     help="Serial port (skip to choose interactively). "
80                          "e.g. /dev/tty.usbserial-XXXX (macOS), "
81                          "/dev/ttyUSB0 (Linux), COM5 (Windows)")
82     ap.add_argument("--baud", type=int, default=115200)
83     ap.add_argument("--out", default="traj_log.csv")

```

```

84     args = ap.parse_args()
85
86     port = args.port if args.port else pick_port()
87
88     print(f"[traj_log] Opening {port} @ {args.baud} baud...", flush=True)
89     ser = serial.Serial(port, args.baud, timeout=1.0)
90
91     print(f'[traj_log] Waiting for "{START_SENT}". Trigger the trajectory on the robot.', flush=True)
92     print("[traj_log] (Ctrl+C to abort.)", flush=True)
93
94     capturing = False
95     header    = None
96     rows      = []
97     t_open    = time.time()
98
99     try:
100         while True:
101             raw = ser.readline()
102             if not raw:
103                 # Show occasional heartbeat while waiting
104                 if not capturing and time.time() - t_open > 10 and int(time.time()) % 10 == 0:
105                     print(f"[traj_log] ...still waiting (no serial data yet?)", flush=True)
106                     t_open = time.time()
107                     continue
108
109                 try:
110                     line = raw.decode("utf-8", errors="replace").strip()
111                 except Exception:
112                     continue
113
114                 # Strip Erlang shell prompt prefix if present.
115                 line = PROMPT_RE.sub("", line)
116
117                 if not capturing:
118                     if line == START_SENT:
119                         capturing = True
120                         print("[traj_log] >>> START detected. Capturing...", flush=True)
121                     # else: ignore pre-start chatter
122                     continue
123
124                 # Now in capturing mode
125                 if line == END_SENT:
126                     print("[traj_log] >>> END detected. Closing capture.", flush=True)
127                     break
128                 elif line.startswith(HEADER_PFX):
129                     header = line[len(HEADER_PFX):].split(",")
130                     print(f"[traj_log] Header: {header}", flush=True)
131                 elif line.startswith(ROW_PFX):
132                     fields = line[len(ROW_PFX):].split(",")
133                     rows.append(fields)
134                     # Live print: pair with header if we have one, else raw fields.
135                     if header is not None and len(fields) == len(header):
136                         pairs = " ".join(f"{h}={v}" for h, v in zip(header, fields))
137                         print(f"[{len(rows):04d}] {pairs}", flush=True)
138                     else:
139                         print(f"[{len(rows):04d}] {'','.join(fields)}", flush=True)
140                     # else: filter out non-TLOG lines (other Erlang prints)
141                 except KeyboardInterrupt:
142                     print("\n[traj_log] Interrupted -- saving partial capture.", flush=True)
143             finally:
144                 ser.close()
145
146         if header is None and not rows:
147             print("[traj_log] No data captured. Nothing written.", flush=True)
148             sys.exit(1)
149
150         # If we captured rows but no header (unlikely), fabricate one with positional names.
151         if header is None:
152             n = max(len(r) for r in rows)
153             header = [f"col{i}" for i in range(n)]
154
155         with open(args.out, "w", newline="") as f:
156             w = csv.writer(f)

```

```

157         w.writerow(header)
158         for r in rows:
159             w.writerow(r)
160
161         print(f"[traj_log] Wrote {len(rows)} rows to {args.out}", flush=True)
162
163
164 if __name__ == "__main__":
165     main()

```

A.3 Waypoint planner and RC UI (Python)

The remote-control and waypoint-planner UI is shown in Figure A.1. Waypoints are entered on the top-down grid, packaged into the wire-protocol format from Appendix B, and sent over the same serial port that carries the telemetry stream.

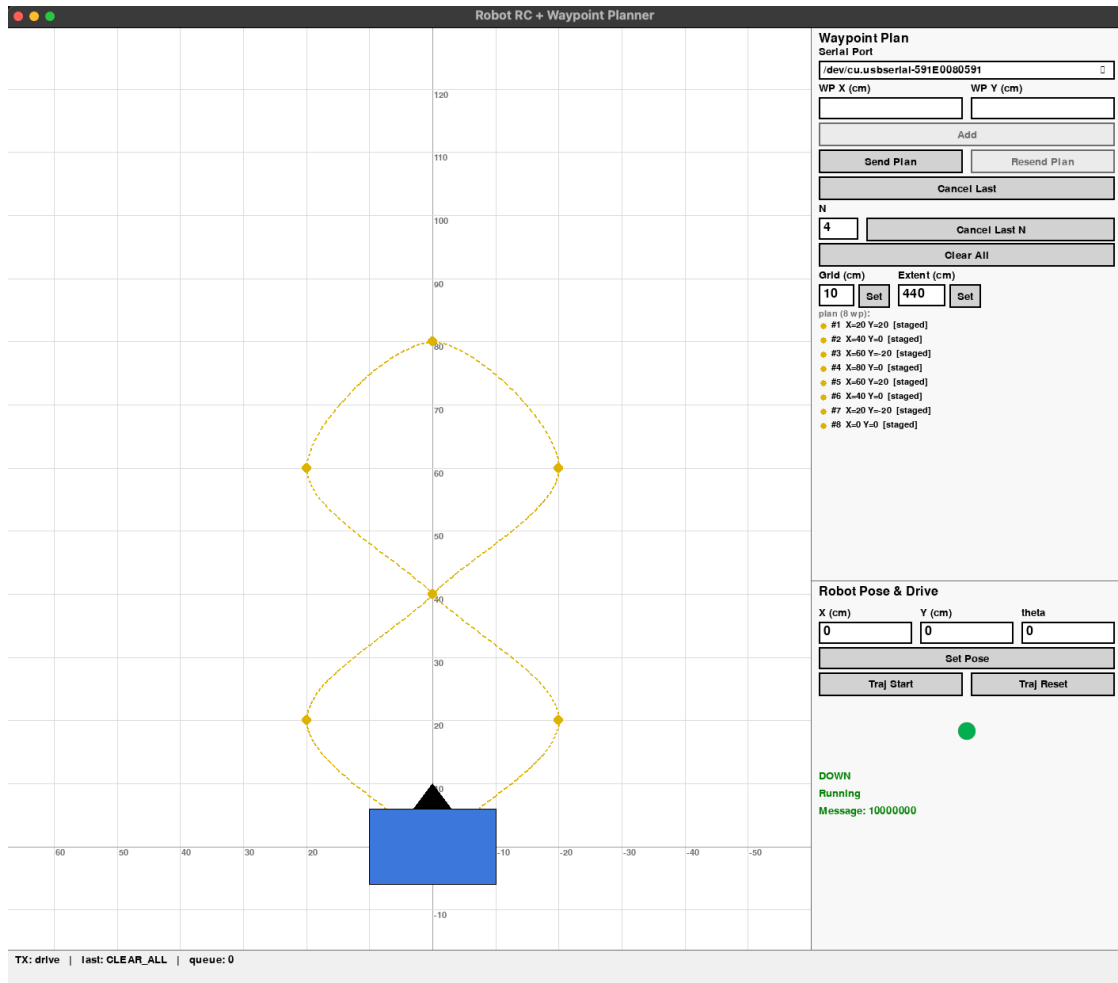


Figure A.1: Operator-side waypoint planner and remote-control UI. The figure-eight plan visible on the canvas (eight waypoints) is the same plan referenced in the experiments of Part III.

A.3.1 RC_robot_UI.py

```

1 import pygame
2 import sys

```

```

3 import math
4 import re
5 import numpy as np
6 import serial
7 import serial.tools.list_ports
8
9
10 # wp inject wire protocol - keep in sync w/ balancing_robot/include/robot_types.hrl
11 RUN_BIT          = 0x80    # bit 7 - E-stop arm (ESP eats this)
12 PROTO_BIT        = 0x40    # bit 6
13 KIND_BIT         = 0x20    # bit 5
14
15 CTRL_ABORT       = 0b00000
16 CTRL_CANCEL_LAST = 0b10000
17 CTRL_CLEAR_ALL   = 0b10001
18 CTRL_CANCEL_N_HDR = 0b10010
19 CTRL_CANCEL_N_COMMIT = 0b10011
20 CTRL_ADD_HEADER   = 0b11000
21 CTRL_HI_TO_LO     = 0b11010
22 CTRL_X_TO_Y       = 0b11100
23 CTRL_ADD_COMMIT   = 0b11110
24
25 SIGN_OFFSET      = 50
26 MAX_OFFSET_CM    = 50
27 FRAME_HOLD_TICKS = 6      # ui ticks per proto frame (~30ms at 200hz)
28 INTER_TX_GAP_TICKS = 1    # PROTO=0 bytes after tx so robot debounce sees a clean exit
29 MAX_CANCEL_N     = 15
30
31 ADD_COMMIT_BYTE   = PROTO_BIT | CTRL_ADD_COMMIT    # detect "wp sent" edge in tx queue
32
33
34 # grid + viz stuff
35 grid_visible_extent_cm = 400.0    # smaller canvas dim covers this many cm
36 BOUNDARY_CLEARANCE_CM = 20.0      # spline cant go closer than this to edges
37 SPLINE_RES          = 30          # = spline.erl ?SPLINE_RES
38
39 # robot top-view rect dims, 20cm wheelbase (lateral) x 12cm front-back
40 ROBOT_LATERAL_CM     = 20.0
41 ROBOT_LONGITUDINAL_CM = 12.0
42
43
44 # frame generators (from phase 3, unchanged)
45 def _clamp_offset(v):
46     return max(-MAX_OFFSET_CM, min(MAX_OFFSET_CM, v))
47
48
49 def build_add_frames(dx_cm, dy_cm):
50     """9 byte ADD seq, each byt already w/ PROTO=1.
51     HI_TO_LO sep between hi/lo data bytes so adjacent bytes always differ
52     even when the 5-bit payloads collide (eg dx=16 -> x_hi=x_lo=2)."""
53     dx = _clamp_offset(int(dx_cm))
54     dy = _clamp_offset(int(dy_cm))
55     xu = dx + SIGN_OFFSET    # 0..100 after offset
56     yu = dy + SIGN_OFFSET
57     x_hi, x_lo = (xu >> 5) & 0x1F, xu & 0x1F
58     y_hi, y_lo = (yu >> 5) & 0x1F, yu & 0x1F
59     return [
60         PROTO_BIT | CTRL_ADD_HEADER,
61         PROTO_BIT | KIND_BIT | x_hi,
62         PROTO_BIT | CTRL_HI_TO_LO,
63         PROTO_BIT | KIND_BIT | x_lo,
64         PROTO_BIT | CTRL_X_TO_Y,
65         PROTO_BIT | KIND_BIT | y_hi,
66         PROTO_BIT | CTRL_HI_TO_LO,
67         PROTO_BIT | KIND_BIT | y_lo,
68         PROTO_BIT | CTRL_ADD_COMMIT,
69     ]
70
71
72 def build_cancel_last_frames():
73     return [PROTO_BIT | CTRL_CANCEL_LAST]
74
75

```

```

76 def build_cancel_n_frames(n):
77     n = max(1, min(MAX_CANCEL_N, int(n)))
78     return [
79         PROTO_BIT | CTRL_CANCEL_N_HDR,
80         PROTO_BIT | KIND_BIT | n,
81         PROTO_BIT | CTRL_CANCEL_N_COMMIT,
82     ]
83
84
85 def build_clear_all_frames():
86     return [PROTO_BIT | CTRL_CLEAR_ALL]
87
88
89 def describe_tx_byte(b):
90     """lil label for the status strip"""
91     if b is None or not (b & PROTO_BIT):
92         return "drive"
93     payload = b & 0x1F
94     if b & KIND_BIT:
95         return f"DATA({payload})"
96     return {
97         CTRL_ADD_HEADER: "ADD_HEADER",
98         CTRL_HI_TO_LO: "HI_TO_LO",
99         CTRL_X_TO_Y: "X_TO_Y",
100         CTRL_ADD_COMMIT: "ADD_COMMIT",
101         CTRL_CANCEL_LAST: "CANCEL_LAST",
102         CTRL_CLEAR_ALL: "CLEAR_ALL",
103         CTRL_CANCEL_N_HDR: "CANCEL_N_HDR",
104         CTRL_CANCEL_N_COMMIT: "CANCEL_N_COMMIT",
105         CTRL_ABORT: "ABORT",
106     }.get(payload, f"CTRL({payload})")
107
108
109 # catmull-rom port of spline.erl build_path/2
110 # uniform CR, reflected phantom ctrl pts on both ends, SPLINE_RES per segment
111 def _reflect(a, b):
112     """reflect a across b (used for the phantom endpoints)"""
113     return (2 * b[0] - a[0], 2 * b[1] - a[1])
114
115
116 def _catmull_rom(p0, p1, p2, p3, t):
117     t2, t3 = t * t, t * t * t
118     x = 0.5 * (
119         (2 * p1[0])
120         + (-p0[0] + p2[0]) * t
121         + (2*p0[0] - 5*p1[0] + 4*p2[0] - p3[0]) * t2
122         + (-p0[0] + 3*p1[0] - 3*p2[0] + p3[0]) * t3)
123     y = 0.5 * (
124         (2 * p1[1])
125         + (-p0[1] + p2[1]) * t
126         + (2*p0[1] - 5*p1[1] + 4*p2[1] - p3[1]) * t2
127         + (-p0[1] + 3*p1[1] - 3*p2[1] + p3[1]) * t3)
128     return (x, y)
129
130
131 def build_path(waypoints, res=SPLINE_RES):
132     """list of (x,y) samples. matches spline.erl exactly:
133     (n-1)*res samples for n wps, t in [0,1) per seg, NO final endpoint."""
134     n = len(waypoints)
135     if n == 0:
136         return []
137     if n == 1:
138         return [waypoints[0]]
139     # stick reflected phantom ctrl pts on both ends (same direction as erlang)
140     ctrl = [_reflect(waypoints[1], waypoints[0])] \
141         + list(waypoints) \
142         + [_reflect(waypoints[-2], waypoints[-1])]
143     pts = []
144     for i in range(len(ctrl) - 3):
145         p0, p1, p2, p3 = ctrl[i], ctrl[i+1], ctrl[i+2], ctrl[i+3]
146         for k in range(res):
147             pts.append(_catmull_rom(p0, p1, p2, p3, k / res))
148     return pts

```

```

149
150
151 # serial line parser - drops TLOG / TLOG_HEADER / sentinels / shell prompt
152 # returns ('heartbeat', int) | ('tlog', str) | ('sentinel', str) | None
153 # (old code did int() on everything which crashed when robot streamed tlog)
154 TLOG_PREFIX = "TLOG,"
155 TLOG_HEADER_PREFIX = "TLOG_HEADER,"
156 START_SENT = "===LOG_START_TRAJ==="
157 END_SENT = "===LOG_END_TRAJ==="
158 _ERLANG_PROMPT_RE = re.compile(r'^\s*\[^\s]+\s+\d+>\s*')
159
160
161 def parse_serial_line(raw_bytes):
162     if not raw_bytes:
163         return None
164     try:
165         line = raw_bytes.decode().replace("\r\n", "").strip()
166     except (UnicodeDecodeError, AttributeError):
167         return None
168     line = _ERLANG_PROMPT_RE.sub("", line)
169     if not line:
170         return None
171     if line == START_SENT or line == END_SENT:
172         return ('sentinel', line)
173     if line.startswith(TLOG_HEADER_PREFIX) or line.startswith(TLOG_PREFIX):
174         return ('tlog', line)
175     try:
176         return ('heartbeat', int(line))
177     except ValueError:
178         return None
179
180
181 # Pygame + serial init
182 pygame.init()
183
184 ser = None
185 ports = list(serial.tools.list_ports.comports())
186
187 def on_port_select(index):
188     global ser, ports
189     if not ports or index >= len(ports):
190         return
191     selected_port = ports[index].device
192     if ser is not None:
193         try:
194             ser.close()
195         except:
196             pass
197     try:
198         ser = serial.Serial(port=selected_port, baudrate=115200, timeout=0.1)
199     except Exception as e:
200         print(f"Failed to connect to {selected_port}: {e}")
201         ser = None
202
203 INIT_W, INIT_H = 1000, 700
204 screen = pygame.display.set_mode((INIT_W, INIT_H), pygame.RESIZABLE)
205 pygame.display.set_caption("Robot RC + Waypoint Planner")
206
207 # colours
208 white = (255, 255, 255)
209 black = (0, 0, 0)
210 gray = (210, 210, 210)
211 gray_dark = (110, 110, 110)
212 gray_disabled = (235, 235, 235)
213 gray_grid_lt = (220, 220, 220)
214 gray_grid_axis = (160, 160, 160)
215 gray_dotted = (140, 140, 140)
216 green_dark = (0, 128, 0)
217 green_solid = (10, 160, 60)
218 yellow_dot = (220, 180, 0)
219 red_warn = (210, 40, 40)
220 blue_focus = (200, 220, 255)
221 blue_robot = (60, 120, 220)

```

```

222 strip_bg      = (240, 240, 240)
223 panel_bg      = (248, 248, 248)
224
225 # load imgs from cwd -- keep originals; scaled lazily in draw_drive_panel
226 arrow_orig    = pygame.image.load('arrow.png').convert_alpha()
227 circle_orig   = pygame.image.load('point.png').convert_alpha()
228 stop_orig     = pygame.image.load('Stop_sign.png').convert_alpha()
229 arrow_img     = circle_img = stop_img = None
230 _last_icon_size = 0
231
232 font_big      = pygame.font.Font(None, 30)
233 font_med      = pygame.font.Font(None, 24)
234 font_small    = pygame.font.Font(None, 20)
235 font_tiny     = pygame.font.Font(None, 16)
236
237
238 # ui widgets - small + dependency free
239 class TextField:
240     def __init__(self, label, max_len=4, allow_minus=True):
241         self.rect = pygame.Rect(0, 0, 0, 0)
242         self.label = label
243         self.buf = ""
244         self.max_len = max_len
245         self.allow_minus = allow_minus
246         self.focused = False
247
248     def set_rect(self, rect):
249         self.rect = pygame.Rect(rect)
250
251     def handle_event(self, ev, mouse_pos):
252         """returns True if we ate an ENTER (caller uses this to skip stand-toggle)"""
253         if ev.type == pygame.MOUSEBUTTONDOWN and ev.button == 1:
254             self.focused = self.rect.collidepoint(mouse_pos)
255             return False
256         if ev.type == pygame.TEXTINPUT and self.focused:
257             for ch in ev.text:
258                 if ch.isdigit() and len(self.buf) < self.max_len:
259                     self.buf += ch
260                 elif ch == "-" and self.allow_minus and not self.buf:
261                     self.buf += ch
262             return False
263         if ev.type == pygame.KEYDOWN and self.focused:
264             if ev.key == pygame.K_BACKSPACE:
265                 self.buf = self.buf[:-1]
266                 return False
267             if ev.key in (pygame.K_RETURN, pygame.K_KP_ENTER):
268                 self.focused = False
269                 return True
270             if ev.key == pygame.K_TAB:
271                 self.focused = False
272                 return False
273             if not hasattr(pygame, 'TEXTINPUT'):
274                 ch = getattr(ev, 'unicode', '')
275                 if ch.isdigit() and len(self.buf) < self.max_len:
276                     self.buf += ch
277                 elif ch == "-" and self.allow_minus and not self.buf:
278                     self.buf += ch
279             return False
280
281     def value(self, default=0):
282         if not self.buf or self.buf == "-":
283             return default
284         try:
285             return int(self.buf)
286         except ValueError:
287             return default
288
289     def has_value(self):
290         return bool(self.buf) and self.buf != "-"
291
292     def clear(self):
293         self.buf = ""
294

```

```

295     def draw(self, surf):
296         if self.label:
297             label_surf = font_small.render(self.label, True, black)
298             surf.blit(label_surf, (self.rect.x, self.rect.y - 18))
299             bg = blue_focus if self.focused else white
300             pygame.draw.rect(surf, bg, self.rect)
301             pygame.draw.rect(surf, black, self.rect, 2)
302             text_surf = font_med.render(self.buf, True, black)
303             surf.blit(text_surf, (self.rect.x + 6, self.rect.y + 4))
304
305
306     class Button:
307     def __init__(self, label, callback, enabled_fn=lambda: True):
308         self.rect = pygame.Rect(0, 0, 0, 0)
309         self.label = label
310         self.callback = callback
311         self.enabled_fn = enabled_fn
312
313     def set_rect(self, rect):
314         self.rect = pygame.Rect(rect)
315
316     def handle_event(self, ev, mouse_pos):
317         if (ev.type == pygame.MOUSEBUTTONDOWN and ev.button == 1
318             and self.rect.collidepoint(mouse_pos) and self.enabled_fn()):
319             self.callback()
320             return False
321
322     def draw(self, surf):
323         enabled = self.enabled_fn()
324         bg = gray if enabled else gray_disabled
325         fg = black if enabled else gray_dark
326         pygame.draw.rect(surf, bg, self.rect)
327         pygame.draw.rect(surf, fg, self.rect, 2)
328         text_surf = font_small.render(self.label, True, fg)
329         surf.blit(text_surf, text_surf.get_rect(center=self.rect.center))
330
331
332     class Dropdown:
333     def __init__(self, label, options, initial_index=0, on_select=None):
334         self.rect = pygame.Rect(0, 0, 0, 0)
335         self.label = label
336         self.options = options
337         self.selected_index = initial_index if options and len(options) > 0 and options[0] != "No Ports
Found" else None
338         self.on_select = on_select
339         self.expanded = False
340
341     def set_rect(self, rect):
342         self.rect = pygame.Rect(rect)
343
344     def handle_event(self, ev, mouse_pos):
345         if ev.type == pygame.MOUSEBUTTONDOWN and ev.button == 1:
346             if self.rect.collidepoint(mouse_pos):
347                 self.expanded = not self.expanded
348                 return True
349             elif self.expanded:
350                 for i in range(len(self.options)):
351                     option_rect = pygame.Rect(self.rect.x, self.rect.y + self.rect.height * (i + 1),
self.rect.width, self.rect.height)
352                     if option_rect.collidepoint(mouse_pos):
353                         self.selected_index = i
354                         self.expanded = False
355                         if self.on_select:
356                             self.on_select(i)
357                         return True
358                 self.expanded = False
359             return False
360
361     def draw(self, surf):
362         if self.label:
363             label_surf = font_small.render(self.label, True, black)
364             surf.blit(label_surf, (self.rect.x, self.rect.y - 18))
365

```

```

366         bg = white
367         pygame.draw.rect(surf, bg, self.rect)
368         pygame.draw.rect(surf, black, self.rect, 2)
369
370         selected_text = ""
371         if self.selected_index is not None and self.selected_index < len(self.options):
372             selected_text = self.options[self.selected_index]
373         elif self.options:
374             selected_text = self.options[0]
375
376         text_surf = font_small.render(selected_text, True, black)
377         surf.blit(text_surf, (self.rect.x + 6, self.rect.y + (self.rect.height - text_surf.get_height()
378 ) // 2))
379
380         arrow_char = "▲" if self.expanded else "▼"
381         arrow_surf = font_small.render(arrow_char, True, black)
382         surf.blit(arrow_surf, (self.rect.right - 18, self.rect.y + (self.rect.height - arrow_surf.
383 get_height()) // 2))
384
385         if self.expanded:
386             for i, opt in enumerate(self.options):
387                 opt_rect = pygame.Rect(self.rect.x, self.rect.y + self.rect.height * (i + 1), self.rect.
388 width, self.rect.height)
389                 mouse_pos = pygame.mouse.get_pos()
390                 opt_bg = blue_focus if opt_rect.collidepoint(mouse_pos) else white
391                 pygame.draw.rect(surf, opt_bg, opt_rect)
392                 pygame.draw.rect(surf, black, opt_rect, 1)
393                 opt_surf = font_small.render(opt, True, black)
394                 surf.blit(opt_surf, (opt_rect.x + 6, opt_rect.y + (opt_rect.height - opt_surf.
395 get_height()) // 2))
396
397 # state
398 running = True
399 message = 0
400 run = True
401 stand = False
402 release_space = True
403 release_enter = True
404
405 # low level tx queue (raw bytes)
406 tx_queue = []
407 tx_hold_remaining = 0
408 current_tx_byte = None
409 last_tx_label = "idle"
410 prev_tx_byte = None # used for ADD_COMMIT edge detection
411
412 # robot pose (op types it, viz only - robot doesnt actually know this)
413 robot_x_cm = 0.0
414 robot_y_cm = 0.0
415 robot_th_deg = 0.0
416
417 # the plan - each item = {'dx': int, 'dy': int, 'state': 'staged' | 'sent'}
418 plan = []
419
420 # tracking which plan idxs are currently being tx'd
421 send_batch = [] # list[int] of plan indices, in send order
422 send_batch_idx = 0 # cursor - next ADD_COMMIT to land mks plan[send_batch[idx]]=sent
423
424 # grid spacing (op-settable)
425 grid_spacing_cm = 10
426
427 # does the preview pt break the 20cm boundary? set each frame
428 preview_violates = False
429
430 # state for panning & zooming
431 pan_offset_x = 0.0
432 pan_offset_y = 0.0
433 zoom_scale = 1.0
434 dragging = False
435 drag_start_pos = (0, 0)
436 drag_start_offset = (0.0, 0.0)
437 wp_list_scroll_offset = 0

```

```

435
436
437
438 def tx_in_progress():
439     return bool(tx_queue) or tx_hold_remaining > 0
440
441
442 def waypoints_world():
443     """abs (x,y) per wp - cumsum of deltas from robot pose"""
444     pts = []
445     x, y = robot_x_cm, robot_y_cm
446     for wp in plan:
447         x += wp['dx']
448         y += wp['dy']
449         pts.append((x, y))
450     return pts
451
452
453 def preview_point():
454     """world pt of the preview, if user typed sth in either field. None otherwise"""
455     if not (dx_field.has_value() or dy_field.has_value()):
456         return None
457     pts = waypoints_world()
458     if pts:
459         bx, by = pts[-1]
460     else:
461         bx, by = robot_x_cm, robot_y_cm
462     x = dx_field.value(int(bx)) if dx_field.has_value() else bx
463     y = dy_field.value(int(by)) if dy_field.has_value() else by
464     return (float(x), float(y))
465
466
467 def full_path_points():
468     """anchors going thru the spline: [robot, ...plan, preview if any]"""
469     pts = [(robot_x_cm, robot_y_cm)] + waypoints_world()
470     pv = preview_point()
471     if pv is not None:
472         pts.append(pv)
473     return pts
474
475
476 def viewport_bounds_for(canvas_rect, px_per_cm):
477     """world bounds for the visible viewport screen (x_min, x_max, y_min, y_max).
478     With CCW 90 rotation: +X (forward) maps to screen UP, +Y (left) to screen LEFT."""
479     cx = canvas_rect.x + canvas_rect.width / 2 + pan_offset_x
480     cy = canvas_rect.y + canvas_rect.height / 2 + pan_offset_y
481     x_min = (cy - canvas_rect.bottom) / px_per_cm
482     x_max = (cy - canvas_rect.top) / px_per_cm
483     y_min = (cx - canvas_rect.right) / px_per_cm
484     y_max = (cx - canvas_rect.left) / px_per_cm
485     return (x_min, x_max, y_min, y_max)
486
487
488 def world_bounds_for(canvas_rect, px_per_cm):
489     """fixed physical world bounds for the grid centered at (0,0), independent of screen borders/zoom/
490     pan"""
491     limit = grid_visible_extent_cm / 2.0
492     return (-limit, limit, -limit, limit)
493
494
495 def spline_within_bounds(sample_pts, bounds):
496     x_min, x_max, y_min, y_max = bounds
497     for (x, y) in sample_pts:
498         if (x - x_min) < BOUNDARY_CLEARANCE_CM: return False
499         if (x_max - x) < BOUNDARY_CLEARANCE_CM: return False
500         if (y - y_min) < BOUNDARY_CLEARANCE_CM: return False
501         if (y_max - y) < BOUNDARY_CLEARANCE_CM: return False
502     return True
503
504
505 # button callbacks
506 def can_send_plan():

```

```

507     return any(wp['state'] == 'staged' for wp in plan) and not any_staged_waypoint_violates() and not
508         tx_in_progress()
509
510 def can_resend_plan():
511     return bool(plan) and not tx_in_progress() and any(wp['state'] == 'sent' for wp in plan)
512
513
514 def on_resend_plan():
515     global send_batch, send_batch_idx, last_tx_label
516     if not can_resend_plan(): return
517     if tx_in_progress(): return
518
519     # 1. Enqueue clear all frames to clear robot's plan first
520     _enqueue_simple(build_clear_all_frames(), "CLEAR_ALL")
521
522     # 2. Revert all local waypoints' state to staged
523     for wp in plan:
524         wp['state'] = 'staged'
525
526     # 3. Queue them for transmission just like send plan
527     staged = [i for i, wp in enumerate(plan)]
528     send_batch = staged[:]
529     send_batch_idx = 0
530     for i in staged:
531         wp = plan[i]
532         tx_queue.extend(build_add_frames(wp['dx'], wp['dy']))
533         tx_queue.extend([0] * INTER_TX_GAP_TICKS)
534
535     last_tx_label = f"Resend Plan ({len(staged)} wp)"
536
537
538 def has_any_waypoints():
539     return bool(plan) and not tx_in_progress()
540
541
542 def on_add():
543     global wp_list_scroll_offset, error_message, error_timer
544     if preview_violates or tx_in_progress(): return
545     if not (dx_field.has_value() or dy_field.has_value()): return
546     pts = waypoints_world()
547     if pts:
548         bx, by = pts[-1]
549     else:
550         bx, by = robot_x_cm, robot_y_cm
551     target_x = dx_field.value(int(bx)) if dx_field.has_value() else bx
552     target_y = dy_field.value(int(by)) if dy_field.has_value() else by
553     dx_total = int(target_x - bx)
554     dy_total = int(target_y - by)
555     if abs(dx_total) > 50 or abs(dy_total) > 50:
556         error_message = "Error: New point cannot be more than 50,50 from the last!"
557         error_timer = 600
558         return
559     dx = _clamp_offset(dx_total)
560     dy = _clamp_offset(dy_total)
561     plan.append({'dx': dx, 'dy': dy, 'state': 'staged'})
562     dx_field.clear()
563     dy_field.clear()
564     wp_list_scroll_offset = len(plan)
565
566 def on_send_plan():
567     global send_batch, send_batch_idx, last_tx_label
568     if not can_send_plan(): return
569     if tx_in_progress(): return
570     staged = [i for i, wp in enumerate(plan) if wp['state'] == 'staged']
571     if not staged: return
572     send_batch = staged[:]
573     send_batch_idx = 0
574     for i in staged:
575         wp = plan[i]
576         tx_queue.extend(build_add_frames(wp['dx'], wp['dy']))
577         tx_queue.extend([0] * INTER_TX_GAP_TICKS)
578     last_tx_label = f"Send Plan ({len(staged)} wp)"

```

```

579
580
581 def _enqueue_simple(frames, label):
582     global last_tx_label
583     last_tx_label = label
584     tx_queue.extend(frames)
585     tx_queue.extend([0] * INTER_TX_GAP_TICKS)
586
587
588 def on_cancel_last():
589     if not plan or tx_in_progress(): return
590     last = plan[-1]
591     if last['state'] == 'staged':
592         plan.pop() # local-only, no traffic
593     else:
594         _enqueue_simple(build_cancel_last_frames(), "CANCEL_LAST")
595         plan.pop() # mirror robot, dont wait for reply
596
597
598 def on_cancel_n():
599     if not plan or tx_in_progress(): return
600     n_req = max(1, min(MAX_CANCEL_N, n_field.value(1)))
601     n = min(n_req, len(plan))
602     # walk tail backwards - drop staged ones locally first, then tx CANCEL_N for sent ones
603     staged_in_tail = 0
604     for wp in reversed(plan[-n:]):
605         if wp['state'] == 'staged':
606             staged_in_tail += 1
607         else:
608             break
609     sent_to_remove = n - staged_in_tail
610     # pop staged tail locally
611     for _ in range(staged_in_tail):
612         plan.pop()
613     if sent_to_remove > 0:
614         _enqueue_simple(build_cancel_n_frames(sent_to_remove),
615                         f"CANCEL_N ({sent_to_remove})")
616     # mirror robot side - drop the sent ones from local plan too
617     for _ in range(sent_to_remove):
618         plan.pop()
619
620
621 def on_clear_all():
622     if not plan or tx_in_progress(): return
623     has_sent = any(wp['state'] == 'sent' for wp in plan)
624     if has_sent:
625         _enqueue_simple(build_clear_all_frames(), "CLEAR_ALL")
626     plan.clear()
627
628
629 # pose handlers - re-read field bufs into pose state when 'Set Pose' clicked
630 def commit_pose():
631     global robot_x_cm, robot_y_cm, robot_th_deg
632     robot_x_cm = float(pose_x_field.value(int(robot_x_cm)))
633     robot_y_cm = float(pose_y_field.value(int(robot_y_cm)))
634     robot_th_deg = float(pose_th_field.value(int(robot_th_deg)))
635
636
637 def commit_grid_spacing():
638     global grid_spacing_cm
639     v = grid_field.value(10)
640     grid_spacing_cm = max(1, min(100, v))
641
642
643 def check_and_cancel_out_of_bounds():
644     global plan, wp_list_scroll_offset
645     if not plan or 'grid_rect' not in globals():
646         return
647     # Get absolute coordinates of all waypoints
648     wps = waypoints_world()
649
650     # Get the current bounds
651     px_per_cm = (min(grid_rect.width, grid_rect.height) / grid_visible_extent_cm) * zoom_scale

```

```

652     bounds = world_bounds_for(grid_rect, px_per_cm)
653     x_min, x_max, y_min, y_max = bounds
654
655     # Find the first waypoint index that is out of bounds
656     first_out_idx = None
657     for i, (wx, wy) in enumerate(wps):
658         if wx < x_min or wx > x_max or wy < y_min or wy > y_max:
659             first_out_idx = i
660             break
661
662     if first_out_idx is not None:
663         # Count sent waypoints from first_out_idx to the end
664         sent_in_tail = 0
665         for i in range(first_out_idx, len(plan)):
666             if plan[i]['state'] == 'sent':
667                 sent_in_tail += 1
668
669         # Send cancel command if there are sent waypoints to remove from the robot
670         if sent_in_tail > 0 and not tx_in_progress():
671             _enqueue_simple(build_cancel_n_frames(sent_in_tail), f"CANCEL_N ({sent_in_tail})")
672
673         # Revert all waypoints from first_out_idx to staged
674         for i in range(first_out_idx, len(plan)):
675             plan[i]['state'] = 'staged'
676
677         # Scroll the list to show the reverted staged waypoints
678         wp_list_scroll_offset = first_out_idx
679
680
681     def commit_extent():
682         global grid_visible_extent_cm
683         v = extent_field.value(400)
684         grid_visible_extent_cm = float(max(10, min(10000, v)))
685         check_and_cancel_out_of_bounds()
686
687
688     error_message = ""
689     error_timer = 0
690     selected_wp_idx = None
691     traj_trigger_ticks_remaining = 0
692
693     def on_edit_save():
694         global plan, selected_wp_idx, error_message, error_timer
695         if selected_wp_idx is None or selected_wp_idx >= len(plan):
696             return
697         wps = waypoints_world()
698         old_x, old_y = wps[selected_wp_idx]
699         new_x = float(edit_x_field.value(int(old_x))) if edit_x_field.has_value() else old_x
700         new_y = float(edit_y_field.value(int(old_y))) if edit_y_field.has_value() else old_y
701         bx, by = (robot_x_cm, robot_y_cm) if selected_wp_idx == 0 else wps[selected_wp_idx - 1]
702
703         dx = int(new_x - bx)
704         dy = int(new_y - by)
705         if abs(dx) > 50 or abs(dy) > 50:
706             error_message = "Error: Edited point cannot be more than 50,50 from the last!"
707             error_timer = 600
708             selected_wp_idx = None
709             return
710
711         if selected_wp_idx + 1 < len(plan):
712             next_abs_x, next_abs_y = wps[selected_wp_idx + 1]
713             next_dx = int(next_abs_x - new_x)
714             next_dy = int(next_abs_y - new_y)
715             if abs(next_dx) > 50 or abs(next_dy) > 50:
716                 error_message = "Error: Next point would be more than 50,50 from this one!"
717                 error_timer = 600
718                 selected_wp_idx = None
719                 return
720
721         plan[selected_wp_idx]['dx'] = dx
722         plan[selected_wp_idx]['dy'] = dy
723         if selected_wp_idx + 1 < len(plan):
724             plan[selected_wp_idx + 1]['dx'] = next_dx

```

```

725     plan[selected_wp_idx + 1]['dy'] = next_dy
726     selected_wp_idx = None
727     check_and_cancel_out_of_bounds()
728
729
730 def delete_waypoint(idx):
731     global plan, error_message, error_timer
732     if idx >= len(plan): return
733     wps = waypoints_world()
734     bx, by = (robot_x_cm, robot_y_cm) if idx == 0 else wps[idx - 1]
735     if idx + 1 < len(wps):
736         next_abs_x, next_abs_y = wps[idx + 1]
737         dx_next = next_abs_x - bx
738         dy_next = next_abs_y - by
739         if abs(dx_next) > 50 or abs(dy_next) > 50:
740             sent_count = sum(1 for wp in plan[idx:] if wp['state'] == 'sent')
741             if sent_count > 0:
742                 _enqueue_simple(build_cancel_n_frames(sent_count), f"CANCEL_N ({sent_count})")
743             plan = plan[:idx]
744             error_message = "Warning: Removed coordinate chain to prevent >50cm distance violation!"
745             error_timer = 600
746             check_and_cancel_out_of_bounds()
747             return
748     sent_count = sum(1 for wp in plan[idx:] if wp['state'] == 'sent')
749     if sent_count > 0:
750         _enqueue_simple(build_cancel_n_frames(sent_count), f"CANCEL_N ({sent_count})")
751     plan.pop(idx)
752     if idx < len(plan):
753         bx, by = (robot_x_cm, robot_y_cm) if idx == 0 else wps[idx - 1]
754         next_abs_x, next_abs_y = wps[idx + 1]
755         plan[idx]['dx'] = _clamp_offset(int(next_abs_x - bx))
756         plan[idx]['dy'] = _clamp_offset(int(next_abs_y - by))
757     check_and_cancel_out_of_bounds()
758
759
760 def on_traj_start():
761     global traj_trigger_ticks_remaining
762     traj_trigger_ticks_remaining = 10
763
764
765 def on_traj_reset():
766     global traj_trigger_ticks_remaining
767     traj_trigger_ticks_remaining = 120
768
769
770 # widgets - built once, positions set per-frame so resize Just Works
771 dx_field = TextField("WP X (cm)")
772 dy_field = TextField("WP Y (cm)")
773 n_field = TextField("N", max_len=2, allow_minus=False)
774 pose_x_field = TextField("X (cm)", allow_minus=True)
775 pose_y_field = TextField("Y (cm)", allow_minus=True)
776 pose_th_field = TextField("theta", allow_minus=True)
777 grid_field = TextField("Grid (cm)", max_len=3, allow_minus=False)
778 extent_field = TextField("Extent (cm)", max_len=4, allow_minus=False)
779
780 btn_add = Button("Add", on_add,
781                 lambda: (dx_field.has_value() or dy_field.has_value())
782                        and not preview_violates
783                        and not tx_in_progress())
784 btn_send_plan = Button("Send Plan", on_send_plan, can_send_plan)
785 btn_resend_plan = Button("Resend Plan", on_resend_plan, can_resend_plan)
786 btn_cancel_last = Button("Cancel Last", on_cancel_last, has_any_waypoints)
787 btn_cancel_n = Button("Cancel Last N", on_cancel_n, has_any_waypoints)
788 btn_clear_all = Button("Clear All", on_clear_all, has_any_waypoints)
789 btn_set_pose = Button("Set Pose", commit_pose)
790 btn_set_grid = Button("Set", commit_grid_spacing)
791 btn_set_extent = Button("Set", commit_extent)
792
793 btn_traj_start = Button("Traj Start", on_traj_start)
794 btn_traj_reset = Button("Traj Reset", on_traj_reset)
795
796 edit_x_field = TextField("WP X (cm)", allow_minus=True)
797 edit_y_field = TextField("WP Y (cm)", allow_minus=True)

```

```

798 btn_edit_save = Button("Save", on_edit_save)
799
800 serial_dropdown = Dropdown("Serial Port", [p.device for p in ports] if ports else ["No Ports Found"],
801                             on_select=on_port_select)
802 if ports:
803     on_port_select(0)
804
805 widgets = [dx_field, dy_field, n_field,
806            pose_x_field, pose_y_field, pose_th_field, grid_field, extent_field,
807            btn_add, btn_send_plan, btn_resend_plan, btn_cancel_last, btn_cancel_n, btn_clear_all,
808            btn_set_pose, btn_set_grid, btn_set_extent,
809            btn_traj_start, btn_traj_reset,
810            edit_x_field, edit_y_field, btn_edit_save]
811
812 # layout - redone every frame so resize doesnt break anything
813 def layout(w, h):
814     panel_w = max(220, w * 28 // 100)
815     strip_h = max(22, h * 4 // 100)
816     wp_h = (h - strip_h) * 6 // 10
817     grid_rect = pygame.Rect(0, 0, w - panel_w, h - strip_h)
818     wp_panel = pygame.Rect(w - panel_w, 0, panel_w, wp_h)
819     drive_panel = pygame.Rect(w - panel_w, wp_h, panel_w, h - strip_h - wp_h)
820     strip = pygame.Rect(0, h - strip_h, w, strip_h)
821     return grid_rect, wp_panel, drive_panel, strip
822
823
824 def place_widgets(wp_panel, drive_panel):
825     # wp ctrls top-right
826     x0 = wp_panel.x + 10
827     y = wp_panel.y + 42 # shift starting y to accommodate serial_dropdown
828     w = wp_panel.width - 20
829
830     serial_dropdown.set_rect((x0, y, w, 24))
831     y += 46
832
833     half = (w - 10) // 2
834
835     dx_field.set_rect((x0, y, half, 28)); y += 32
836     dy_field.set_rect((x0 + half + 10, y - 32, half, 28))
837     btn_add.set_rect((x0, y, w, 30)); y += 34
838     btn_send_plan.set_rect((x0, y, half, 30))
839     btn_resend_plan.set_rect((x0 + half + 10, y, half, 30)); y += 34
840     btn_cancel_last.set_rect((x0, y, w, 30)); y += 34
841     n_field.set_rect((x0, y + 18, 50, 28))
842     btn_cancel_n.set_rect((x0 + 60, y + 18, w - 60, 30)); y += 50
843     btn_clear_all.set_rect((x0, y, w, 30)); y += 34
844     grid_field.set_rect((x0, y + 18, 45, 28))
845     btn_set_grid.set_rect((x0 + 50, y + 18, 40, 30))
846     extent_field.set_rect((x0 + 100, y + 18, 60, 28))
847     btn_set_extent.set_rect((x0 + 165, y + 18, 40, 30)); y += 50
848
849     # wp list anchor - draw_wp_panel uses this
850     place_widgets.wp_list_top = y
851
852     # robot pose + drive indicator (bottom-right)
853     x0 = drive_panel.x + 10
854     y = drive_panel.y + 52
855     third = (w - 20) // 3
856     pose_x_field.set_rect((x0, y, third, 28))
857     pose_y_field.set_rect((x0 + third + 10, y, third, 28))
858     pose_th_field.set_rect((x0 + 2*(third + 10), y, third, 28))
859     y += 32
860     btn_set_pose.set_rect((x0, y, w, 28)); y += 32
861     half = (w - 10) // 2
862     btn_traj_start.set_rect((x0, y, half, 30))
863     btn_traj_reset.set_rect((x0 + half + 10, y, half, 30))
864     y += 34
865     place_widgets.drive_indicator_top = y
866
867
868 # rendering helpers
869 def w2s(p, grid_rect, px_per_cm):

```

```

870     """world (x_cm, y_cm) -> screen (px, py). origin = canvas center + pan offset.
871     With CCW 90 rotation: +X (forward) is screen UP, +Y (left) is screen LEFT."""
872     cx = grid_rect.x + grid_rect.width / 2 + pan_offset_x
873     cy = grid_rect.y + grid_rect.height / 2 + pan_offset_y
874     return (int(cx - p[1] * px_per_cm),
875            int(cy - p[0] * px_per_cm))
876
877
878 def s2w(pos, grid_rect, px_per_cm):
879     """screen (px, py) -> world (x_cm, y_cm). Inverts the w2s transformation."""
880     cx = grid_rect.x + grid_rect.width / 2 + pan_offset_x
881     cy = grid_rect.y + grid_rect.height / 2 + pan_offset_y
882     sx, sy = pos
883     wy = (cx - sx) / px_per_cm
884     wx = (cy - sy) / px_per_cm
885     return (wx, wy)
886
887
888 def draw_grid(surf, grid_rect, px_per_cm):
889     # bg
890     pygame.draw.rect(surf, white, grid_rect)
891     # Dynamic grid spacing based on zoom scale:
892     # Zoomed in >= 2x from default: 10s cm markings
893     # Zoomed out < 2x: 50s cm markings
894     spacing = 10 if zoom_scale >= 2.0 else 50
895     label_step = 10 if zoom_scale >= 2.0 else 50
896
897     bounds = viewport_bounds_for(grid_rect, px_per_cm)
898     x_min, x_max, y_min, y_max = bounds
899     cx = grid_rect.x + grid_rect.width / 2 + pan_offset_x
900     cy = grid_rect.y + grid_rect.height / 2 + pan_offset_y
901     step_px = spacing * px_per_cm
902     if step_px >= 4: # skip drawing if lines would just be a blob
903         # vertical lines correspond to constant y_cm
904         y_cm = math.floor(y_min / spacing) * spacing
905         while y_cm <= y_max:
906             sx = int(cx - y_cm * px_per_cm)
907             color = gray_grid_axis if abs(y_cm) < 0.001 else gray_grid_lt
908             pygame.draw.line(surf, color, (sx, grid_rect.top), (sx, grid_rect.bottom), 1)
909             y_cm += spacing
910         # horizontal lines correspond to constant x_cm
911         x_cm = math.floor(x_min / spacing) * spacing
912         while x_cm <= x_max:
913             sy = int(cy - x_cm * px_per_cm)
914             color = gray_grid_axis if abs(x_cm) < 0.001 else gray_grid_lt
915             pygame.draw.line(surf, color, (grid_rect.left, sy), (grid_rect.right, sy), 1)
916             x_cm += spacing
917     # axis number labels every label_step
918     x_cm = math.floor(x_min / label_step) * label_step
919     while x_cm <= x_max:
920         if abs(x_cm) > 0.001:
921             sy = int(cy - x_cm * px_per_cm)
922             txt = font_tiny.render(f"{int(x_cm)}", True, gray_dark)
923             surf.blit(txt, (int(cx) + 2, sy + 2))
924             x_cm += label_step
925     y_cm = math.floor(y_min / label_step) * label_step
926     while y_cm <= y_max:
927         if abs(y_cm) > 0.001:
928             sx = int(cx - y_cm * px_per_cm)
929             txt = font_tiny.render(f"{int(y_cm)}", True, gray_dark)
930             surf.blit(txt, (sx + 2, int(cy) + 2))
931             y_cm += label_step
932     # pink square showing 20cm boundary clearance inside physical playground
933     limit = grid_visible_extent_cm / 2.0
934     allowed_limit = limit - BOUNDARY_CLEARANCE_CM
935     if allowed_limit > 0:
936         p1 = w2s((allowed_limit, allowed_limit), grid_rect, px_per_cm)
937         p2 = w2s((-allowed_limit, -allowed_limit), grid_rect, px_per_cm)
938         rect_width = p2[0] - p1[0]
939         rect_height = p2[1] - p1[1]
940         pygame.draw.rect(surf, (255, 230, 230), pygame.Rect(p1[0], p1[1], rect_width, rect_height), 1)
941
942

```

```

943 def draw_dotted_line(surf, color, p1, p2, dot=3, gap=4):
944     """simple dotted segment in screen coords"""
945     x1, y1 = p1
946     x2, y2 = p2
947     dx, dy = x2 - x1, y2 - y1
948     length = max(1.0, math.hypot(dx, dy))
949     nx, ny = dx / length, dy / length
950     pos = 0.0
951     while pos < length:
952         sx, sy = x1 + nx * pos, y1 + ny * pos
953         ex_pos = min(pos + dot, length)
954         ex, ey = x1 + nx * ex_pos, y1 + ny * ex_pos
955         pygame.draw.line(surf, color, (sx, sy), (ex, ey), 2)
956         pos += dot + gap
957
958
959 def draw_spline_polyline(surf, pts_screen, color, dotted=False, width=2):
960     if len(pts_screen) < 2:
961         return
962     if dotted:
963         for i in range(len(pts_screen) - 1):
964             draw_dotted_line(surf, color, pts_screen[i], pts_screen[i+1])
965     else:
966         pygame.draw.lines(surf, color, False, pts_screen, width)
967
968
969 def waypoint_violates(idx):
970     if 'grid_rect' not in globals() or not plan:
971         return False
972     wps = waypoints_world()
973     if idx >= len(wps):
974         return False
975     limit = grid_visible_extent_cm / 2.0
976     bounds = (-limit, limit, -limit, limit)
977     anchors = [(robot_x_cm, robot_y_cm)] + wps
978     segment_anchors = anchors[idx:idx+2]
979     if len(segment_anchors) >= 2:
980         pts = build_path(segment_anchors, SPLINE_RES) + segment_anchors
981     else:
982         pts = segment_anchors
983     return not spline_within_bounds(pts, bounds)
984
985
986 def any_staged_waypoint_violates():
987     for i, wp in enumerate(plan):
988         if wp['state'] == 'staged' and waypoint_violates(i):
989             return True
990     return False
991
992
993 def draw_path_and_waypoints(surf, grid_rect, px_per_cm):
994     """draws full spline + wp dots. each seg coloured by its destination wp state"""
995     # anchors = [robot, wp1, wp2, ..., wpN, ?preview]
996     anchors = [(robot_x_cm, robot_y_cm)]
997     anchor_state = ['robot'] # parallel list of dest states
998     for wp in plan:
999         x_cm, y_cm = anchors[-1]
1000         anchors.append((x_cm + wp['dx'], y_cm + wp['dy']))
1001         anchor_state.append(wp['state']) # 'staged' or 'sent'
1002     pv = preview_point()
1003     if pv is not None:
1004         anchors.append(pv)
1005         anchor_state.append('preview')
1006
1007     # one big spline thru all anchors (= what robot will recompute eventually).
1008     # build_path gives SPLINE_RES samples per seg, ends at t=29/30 of last seg
1009     # so it does NOT touch the last anchor - matches spline.erl
1010     if len(anchors) >= 2:
1011         all_pts = build_path(anchors, SPLINE_RES)
1012     else:
1013         all_pts = []
1014
1015     n_seg = max(0, len(anchors) - 1)

```

```

1016 for seg_i in range(n_seg):
1017     dest_state = anchor_state[seg_i + 1]
1018     start = seg_i * SPLINE_RES
1019     end = start + SPLINE_RES # exclusive
1020     seg_world = all_pts[start:end]
1021     # tack the destination anchor on the end so segs join up seamlessly
1022     # AND the final seg actually reaches its endpoint
1023     seg_world = list(seg_world) + [anchors[seg_i + 1]]
1024     seg_screen = [w2s(p, grid_rect, px_per_cm) for p in seg_world]
1025     violates = waypoint_violates(seg_i) if dest_state != 'preview' else preview_violates
1026     if dest_state == 'sent':
1027         color = red_warn if violates else green_solid
1028         draw_spline_polyline(surf, seg_screen, color, dotted=False, width=2)
1029     elif dest_state == 'staged':
1030         color = red_warn if violates else yellow_dot
1031         draw_spline_polyline(surf, seg_screen, color, dotted=True)
1032     else: # preview
1033         color = red_warn if preview_violates else gray_dotted
1034         draw_spline_polyline(surf, seg_screen, color, dotted=True)
1035
1036 # waypoint dots (skip robot anchor at idx 0)
1037 for i in range(1, len(anchors)):
1038     x_cm, y_cm = anchors[i]
1039     sx, sy = w2s((x_cm, y_cm), grid_rect, px_per_cm)
1040     dest_state = anchor_state[i]
1041     violates = waypoint_violates(i - 1) if dest_state != 'preview' else preview_violates
1042     if dest_state == 'sent':
1043         color = red_warn if violates else green_solid
1044         pygame.draw.circle(surf, color, (sx, sy), 6)
1045     elif dest_state == 'staged':
1046         color = red_warn if violates else yellow_dot
1047         pygame.draw.circle(surf, color, (sx, sy), 6)
1048     else:
1049         color = red_warn if preview_violates else gray_dotted
1050         pygame.draw.circle(surf, color, (sx, sy), 5, 2)
1051
1052
1053 def draw_robot(surf, grid_rect, px_per_cm):
1054     """blue rect 20x12cm. 20=wheelbase (lateral), 12=front-to-back"""
1055     half_lon = ROBOT_LONGITUDINAL_CM / 2
1056     half_lat = ROBOT_LATERAL_CM / 2
1057     # local frame: +X = fwd, +Y = left
1058     local = [(+half_lon, +half_lat), (+half_lon, -half_lat),
1059             (-half_lon, -half_lat), (-half_lon, +half_lat)]
1060     th = math.radians(robot_th_deg)
1061     c, s = math.cos(th), math.sin(th)
1062     world_corners = [(robot_x_cm + c*lx - s*ly,
1063                     robot_y_cm + s*lx + c*ly) for (lx, ly) in local]
1064     screen_corners = [w2s(p, grid_rect, px_per_cm) for p in world_corners]
1065     pygame.draw.polygon(surf, blue_robot, screen_corners)
1066     pygame.draw.polygon(surf, black, screen_corners, 1)
1067     # heading notch - small black triangle off the front edge
1068     nose_world = (robot_x_cm + c * (half_lon + 4),
1069                 robot_y_cm + s * (half_lon + 4))
1070     front_mid_world = (robot_x_cm + c * half_lon, robot_y_cm + s * half_lon)
1071     side_a = (front_mid_world[0] - s * 3, front_mid_world[1] + c * 3)
1072     side_b = (front_mid_world[0] + s * 3, front_mid_world[1] - c * 3)
1073     pygame.draw.polygon(surf, black,
1074                       [w2s(nose_world, grid_rect, px_per_cm),
1075                       w2s(side_a, grid_rect, px_per_cm),
1076                       w2s(side_b, grid_rect, px_per_cm)])
1077
1078
1079 def draw_wp_panel(surf, wp_panel):
1080     pygame.draw.rect(surf, panel_bg, wp_panel)
1081     pygame.draw.line(surf, gray_dark, (wp_panel.x, wp_panel.y),
1082                     (wp_panel.x, wp_panel.bottom), 1)
1083     # heading
1084     title = font_med.render("Waypoint Plan", True, black)
1085     surf.blit(title, (wp_panel.x + 10, wp_panel.y + 6))
1086     # widgets do their own drawing
1087     dx_field.draw(surf)
1088     dy_field.draw(surf)

```

```

1089 btn_add.draw(surf)
1090 btn_send_plan.draw(surf)
1091 btn_resend_plan.draw(surf)
1092 btn_cancel_last.draw(surf)
1093 n_field.draw(surf)
1094 btn_cancel_n.draw(surf)
1095 btn_clear_all.draw(surf)
1096 grid_field.draw(surf)
1097 btn_set_grid.draw(surf)
1098 extent_field.draw(surf)
1099 btn_set_extent.draw(surf)
1100 # wp list (scrollable)
1101 list_x = wp_panel.x + 10
1102 list_y = place_widgets.wp_list_top
1103 list_h = wp_panel.bottom - list_y - 6
1104 if list_h > 16:
1105     max_rows = max(1, list_h // 18 - 1)
1106     global wp_list_scroll_offset
1107     wp_list_scroll_offset = max(0, min(len(plan) - max_rows, wp_list_scroll_offset))
1108     header = font_tiny.render(f"plan ({len(plan)} wp):", True, gray_dark)
1109     surf.blit(header, (list_x, list_y))
1110     # Draw scroll indicators if applicable
1111     if wp_list_scroll_offset > 0:
1112         up_indicator = font_tiny.render("^ (scrolled up)", True, gray_dark)
1113         surf.blit(up_indicator, (list_x + 100, list_y))
1114     if len(plan) - wp_list_scroll_offset > max_rows:
1115         down_indicator = font_tiny.render("v (scrolled down)", True, gray_dark)
1116         surf.blit(down_indicator, (list_x + 100, list_y + list_h - 12))
1117     row_y = list_y + 14
1118     wps = waypoints_world()
1119     for i in range(wp_list_scroll_offset, len(plan)):
1120         wp = plan[i]
1121         violates = waypoint_violates(i)
1122         if violates:
1123             color = red_warn
1124         else:
1125             color = green_solid if wp['state'] == 'sent' else yellow_dot
1126         pygame.draw.circle(surf, color, (list_x + 6, row_y + 7), 4)
1127         status_text = "INVALID" if violates else wp['state']
1128         txt = font_tiny.render(
1129             f"#{i+1} X={int(wps[i][0])} Y={int(wps[i][1])} [{status_text}]",
1130             True, red_warn if violates else black)
1131         surf.blit(txt, (list_x + 16, row_y))
1132         row_y += 18
1133         if row_y > wp_panel.bottom - 16:
1134             break
1135     serial_dropdown.draw(surf)
1136
1137
1138 def draw_drive_panel(surf, drive_panel):
1139     global arrow_img, circle_img, stop_img, _last_icon_size
1140     pygame.draw.rect(surf, panel_bg, drive_panel)
1141     pygame.draw.line(surf, gray_dark, (drive_panel.x, drive_panel.y),
1142                     (drive_panel.x, drive_panel.bottom), 1)
1143     pygame.draw.line(surf, gray_dark, (drive_panel.x, drive_panel.y),
1144                     (drive_panel.right, drive_panel.y), 1)
1145     title = font_med.render("Robot Pose & Drive", True, black)
1146     surf.blit(title, (drive_panel.x + 10, drive_panel.y + 6))
1147     pose_x_field.draw(surf)
1148     pose_y_field.draw(surf)
1149     pose_th_field.draw(surf)
1150     btn_set_pose.draw(surf)
1151     btn_traj_start.draw(surf)
1152     btn_traj_reset.draw(surf)
1153
1154     # scale all icons to the same size derived from panel dimensions
1155     icon_size = max(32, min(80, min(drive_panel.width, drive_panel.height) // 3))
1156     if icon_size != _last_icon_size:
1157         arrow_img = pygame.transform.smoothscale(arrow_orig, (icon_size, icon_size))
1158         circle_img = pygame.transform.smoothscale(circle_orig, (icon_size, icon_size))
1159         stop_img = pygame.transform.smoothscale(stop_orig, (icon_size, icon_size))
1160         _last_icon_size = icon_size
1161

```

```

1162 # drive indicator (the arrow/circle/stop)
1163 cx = drive_panel.x + drive_panel.width // 2
1164 cy = place_widgets.drive_indicator_top + 40
1165 if cy + 60 > drive_panel.bottom:
1166     return
1167 # arrow direction = current keyb drive (suppressed during tx)
1168 if message < 10000000:
1169     img = stop_img
1170     rect = img.get_rect(center=(cx, cy))
1171     surf.blit(img, rect)
1172 elif abs(_drive_x) == 0:
1173     img = circle_img
1174     rect = img.get_rect(center=(cx, cy))
1175     surf.blit(img, rect)
1176 else:
1177     angle = np.angle(_drive_x, deg=True)
1178     rotated = pygame.transform.rotate(arrow_img, angle)
1179     rect = rotated.get_rect(center=(cx, cy))
1180     surf.blit(rotated, rect.topleft)
1181
1182 # text lines below the indicator
1183 sy = cy + 50
1184 lines = [
1185     ("UP" if stand else "DOWN"),
1186     ("Running" if run else "Stopped"),
1187     f"Message: {message:08d}",
1188 ]
1189 for line in lines:
1190     if sy + 18 > drive_panel.bottom: break
1191     text_surf = font_small.render(line, True, green_dark)
1192     surf.blit(text_surf, (drive_panel.x + 10, sy))
1193     sy += 22
1194
1195 # init helper - prefill pose / grid fields w/ current state values so they
1196 # arent blank on first frame. op-typing updates the buf, Set Pose commits it
1197 def populate_pose_fields_if_empty():
1198     for fld, val in [(pose_x_field, int(robot_x_cm)),
1199                     (pose_y_field, int(robot_y_cm)),
1200                     (pose_th_field, int(robot_th_deg)),
1201                     (grid_field, grid_spacing_cm),
1202                     (extent_field, int(grid_visible_extent_cm))]:
1203         if not fld.buf and not fld.focused:
1204             fld.buf = str(val)
1205
1206 populate_pose_fields_if_empty()
1207
1208 drag_has_moved = False
1209
1210 def handle_grid_click(pos, grid_rect, px_per_cm):
1211     global selected_wp_idx, plan, error_message, error_timer
1212     wps = waypoints_world()
1213     clicked_wp = None
1214     for i, wp_pos in enumerate(wps):
1215         ws_x, ws_y = w2s(wp_pos, grid_rect, px_per_cm)
1216         dist = math.hypot(pos[0] - ws_x, pos[1] - ws_y)
1217         if dist <= 12:
1218             clicked_wp = i
1219             break
1220
1221 if clicked_wp is not None:
1222     if selected_wp_idx == clicked_wp:
1223         delete_waypoint(clicked_wp)
1224         selected_wp_idx = None
1225     else:
1226         selected_wp_idx = clicked_wp
1227         edit_x_field.buf = str(int(wps[clicked_wp][0]))
1228         edit_y_field.buf = str(int(wps[clicked_wp][1]))
1229         edit_x_field.focused = False
1230         edit_y_field.focused = False
1231
1232 else:

```

```

1235     if selected_wp_idx is not None:
1236         selected_wp_idx = None
1237     else:
1238         wx, wy = s2w(pos, grid_rect, px_per_cm)
1239         target_x = int(round(wx))
1240         target_y = int(round(wy))
1241         bx, by = (robot_x_cm, robot_y_cm) if not plan else wps[-1]
1242         dx_total = target_x - bx
1243         dy_total = target_y - by
1244         if abs(dx_total) > 50 or abs(dy_total) > 50:
1245             error_message = "Error: New point cannot be more than 50,50 from the last!"
1246             error_timer = 600
1247             return
1248         dx = _clamp_offset(dx_total)
1249         dy = _clamp_offset(dy_total)
1250         plan.append({'dx': dx, 'dy': dy, 'state': 'staged'})
1251         check_and_cancel_out_of_bounds()
1252
1253
1254 # main loop
1255 _drive_x = 0 # rotating arrow value, module-level so draw fn can grab it
1256
1257 while running:
1258     mouse_pos = pygame.mouse.get_pos()
1259
1260     # layout before events so widget rects are current when clicks arrive
1261     w_, h_ = screen.get_size()
1262     grid_rect, wp_panel, drive_panel, strip_rect = layout(w_, h_)
1263     place_widgets(wp_panel, drive_panel)
1264
1265     # -- events -----
1266     enter_consumed_by_field = False
1267     for event in pygame.event.get():
1268         if serial_dropdown.handle_event(event, mouse_pos):
1269             continue
1270         if event.type == pygame.QUIT:
1271             running = False
1272             continue
1273         if event.type == pygame.VIDEORESIZE:
1274             # need to call set_mode again or surface stays old size
1275             screen = pygame.display.set_mode((event.w, event.h), pygame.RESIZABLE)
1276             continue
1277
1278         # Dragging and zooming logic for the grid / list scrolling
1279         if event.type == pygame.MOUSEBUTTONDOWN:
1280             list_rect = pygame.Rect(wp_panel.x + 10, place_widgets.wp_list_top, wp_panel.width - 20,
wp_panel.bottom - place_widgets.wp_list_top - 6)
1281             if list_rect.collidepoint(event.pos):
1282                 if event.button == 4: # scroll up
1283                     wp_list_scroll_offset = max(0, wp_list_scroll_offset - 1)
1284                 elif event.button == 5: # scroll down
1285                     max_rows = max(1, list_rect.height // 18 - 1)
1286                     wp_list_scroll_offset = max(0, min(len(plan) - max_rows, wp_list_scroll_offset + 1)
)
1287             elif grid_rect.collidepoint(event.pos):
1288                 if event.button == 1: # Left click to drag
1289                     # Check if we clicked on any active text field or button to avoid dragging when
interacting with UI
1290                     clicked_widget = False
1291                     for w in widgets:
1292                         if w in [edit_x_field, edit_y_field, btn_edit_save] and selected_wp_idx is None:
1293                             continue
1294                         if hasattr(w, 'rect') and w.rect.collidepoint(event.pos):
1295                             clicked_widget = True
1296                             break
1297                     # Also check if we clicked inside the active edit box background
1298                     clicked_box = False
1299                     if selected_wp_idx is not None and selected_wp_idx < len(plan):
1300                         wps = waypoints_world()
1301                         sx, sy = w2s(wps[selected_wp_idx], grid_rect, px_per_cm)
1302                         box_w, box_h = 160, 110
1303                         box_x = sx + 15

```

```

1304         box_y = sy - 55
1305         box_x = max(grid_rect.left + 5, min(grid_rect.right - box_w - 5, box_x))
1306         box_y = max(grid_rect.top + 5, min(grid_rect.bottom - box_h - 5, box_y))
1307         box_rect = pygame.Rect(box_x, box_y, box_w, box_h)
1308         if box_rect.collidepoint(event.pos):
1309             clicked_box = True
1310         if not clicked_widget and not clicked_box:
1311             dragging = True
1312             drag_has_moved = False
1313             drag_start_pos = event.pos
1314             drag_start_offset = (pan_offset_x, pan_offset_y)
1315         elif event.button == 4: # Scroll wheel up to zoom in
1316             zoom_scale = min(10.0, zoom_scale * 1.1)
1317         elif event.button == 5: # Scroll wheel down to zoom out
1318             zoom_scale = max(0.1, zoom_scale / 1.1)
1319
1320     elif event.type == pygame.MOUSEBUTTONUP:
1321         if event.button == 1:
1322             if dragging:
1323                 dragging = False
1324                 if not drag_has_moved:
1325                     handle_grid_click(event.pos, grid_rect, px_per_cm)
1326
1327     elif event.type == pygame.MOUSEMOTION:
1328         if dragging:
1329             dx = event.pos[0] - drag_start_pos[0]
1330             dy = event.pos[1] - drag_start_pos[1]
1331             if math.hypot(dx, dy) > 5:
1332                 drag_has_moved = True
1333                 pan_offset_x = drag_start_offset[0] + dx
1334                 pan_offset_y = drag_start_offset[1] + dy
1335
1336     elif event.type == pygame.KEYDOWN:
1337         any_focused = any(getattr(w, 'focused', False) for w in widgets)
1338         if not any_focused:
1339             if event.key in (pygame.K_PLUS, pygame.K_KP_PLUS, pygame.K_EQUALS):
1340                 zoom_scale = min(10.0, zoom_scale * 1.1)
1341             elif event.key in (pygame.K_MINUS, pygame.K_KP_MINUS):
1342                 zoom_scale = max(0.1, zoom_scale / 1.1)
1343
1344         for w in widgets:
1345             if w in [edit_x_field, edit_y_field, btn_edit_save] and selected_wp_idx is None:
1346                 continue
1347             if w.handle_event(event, mouse_pos):
1348                 enter_consumed_by_field = True
1349
1350 # if a field ate an ENTER, hold off stand-toggle til ENTER is released
1351 if enter_consumed_by_field:
1352     release_enter = False
1353
1354 any_field_focused = any(getattr(w, 'focused', False) for w in widgets)
1355
1356 keys = pygame.key.get_pressed()
1357
1358 # drive keys suppressed during tx (bits encode payload) and while typing in a field
1359 if tx_in_progress() or any_field_focused:
1360     forward = backward = left = right = False
1361 else:
1362     forward = bool(keys[pygame.K_z] or keys[pygame.K_UP])
1363     backward = bool(keys[pygame.K_s] or keys[pygame.K_DOWN])
1364     left = bool(keys[pygame.K_q] or keys[pygame.K_LEFT])
1365     right = bool(keys[pygame.K_d] or keys[pygame.K_RIGHT])
1366
1367 # used only for the arrow display
1368 _drive_x = ((1 if forward else 0) + (-1 if backward else 0)
1369             + (1j if left else 0) + (-1j if right else 0))
1370
1371 if keys[pygame.K_ESCAPE]:
1372     running = False
1373
1374 # SPACE - arm/disarm (legacy)
1375 if keys[pygame.K_SPACE] and not any_field_focused:
1376     if release_space:

```

```

1377         release_space = False
1378         run = True if message < 10000000 else False
1379     else:
1380         release_space = True
1381
1382     # ENTER - toggle stand (Get_Up)
1383     if keys[pygame.K_RETURN] and not any_field_focused:
1384         if release_enter:
1385             release_enter = False
1386             stand = not stand
1387         else:
1388             release_enter = True
1389
1390     # -- 20cm boundary check (decides preview colour + Add enable) -----
1391     px_per_cm = (min(grid_rect.width, grid_rect.height) / grid_visible_extent_cm) * zoom_scale
1392     bounds = world_bounds_for(grid_rect, px_per_cm)
1393     pv = preview_point()
1394     if pv is None:
1395         preview_violates = False
1396     else:
1397         # build_path doesnt include the final anchor (matches spline.erl) so we
1398         # check both dense samples AND every anchor pt against the boundary
1399         predicted_anchors = [(robot_x_cm, robot_y_cm)] + waypoints_world() + [pv]
1400         if len(predicted_anchors) >= 2:
1401             predicted_pts = build_path(predicted_anchors, SPLINE_RES) + predicted_anchors
1402         else:
1403             predicted_pts = predicted_anchors
1404         preview_violates = not spline_within_bounds(predicted_pts, bounds)
1405
1406     # -- compose outgoing byte -----
1407     prev_tx_byte = current_tx_byte
1408     if tx_in_progress():
1409         if tx_hold_remaining == 0:
1410             current_tx_byte = tx_queue.pop(0)
1411             tx_hold_remaining = FRAME_HOLD_TICKS
1412             data = current_tx_byte | (RUN_BIT if run else 0)
1413             # Gap bytes between proto frames (queued as 0 for cmd_rx clean-exit)
1414             # must still carry the operator's live Get_Up state, or the robot reads
1415             # b4=0 and falls out of stand_up. Proto frames keep b4 as payload.
1416             if (current_tx_byte & PROTO_BIT) == 0:
1417                 data |= (1 if stand else 0) << 4
1418             tx_hold_remaining -= 1
1419             status_label = describe_tx_byte(current_tx_byte)
1420             # mark wp as sent the tick its ADD_COMMIT finishes its hold
1421             if tx_hold_remaining == 0 and current_tx_byte == ADD_COMMIT_BYTE:
1422                 if send_batch_idx < len(send_batch):
1423                     plan_i = send_batch[send_batch_idx]
1424                     if 0 <= plan_i < len(plan):
1425                         plan[plan_i]['state'] = 'sent'
1426                     send_batch_idx += 1
1427             else:
1428                 # drive mode: PROTO=0 KIND=0, bits 4..0 from keys
1429                 if traj_trigger_ticks_remaining > 0:
1430                     forward = True
1431                     backward = True
1432                     left = False
1433                     right = False
1434                     traj_trigger_ticks_remaining -= 1
1435                     status_label = f"traj_cmd ({traj_trigger_ticks_remaining})"
1436                 else:
1437                     status_label = "drive"
1438
1439             data = (
1440                 (RUN_BIT if run else 0)
1441                 | 0 # bit 6: PROTO
1442                 | 0 # bit 5: KIND
1443                 | ((1 if stand else 0) << 4)
1444                 | (int(forward) << 3)
1445                 | (int(backward) << 2)
1446                 | (int(left) << 1)
1447                 | (int(right) << 0)
1448             )
1449             # queue just drained - reset send batch tracker

```

```

1450         if send_batch:
1451             send_batch = []
1452             send_batch_idx = 0
1453
1454     # -- render -----
1455     screen.fill(white)
1456     draw_grid(screen, grid_rect, px_per_cm)
1457     draw_path_and_waypoints(screen, grid_rect, px_per_cm)
1458     draw_robot(screen, grid_rect, px_per_cm)
1459
1460     # Draw floating waypoint edit box if active
1461     if selected_wp_idx is not None and selected_wp_idx < len(plan):
1462         wps = waypoints_world()
1463         sx, sy = wps[selected_wp_idx], grid_rect, px_per_cm)
1464         box_w, box_h = 160, 110
1465         box_x = sx + 15
1466         box_y = sy - 55
1467         box_x = max(grid_rect.left + 5, min(grid_rect.right - box_w - 5, box_x))
1468         box_y = max(grid_rect.top + 5, min(grid_rect.bottom - box_h - 5, box_y))
1469
1470         edit_x_field.set_rect((box_x + 10, box_y + 10, 65, 26))
1471         edit_y_field.set_rect((box_x + 85, box_y + 10, 65, 26))
1472         btn_edit_save.set_rect((box_x + 10, box_y + 45, 140, 26))
1473
1474         pygame.draw.rect(screen, (40, 40, 40), pygame.Rect(box_x, box_y, box_w, box_h), border_radius
1475 =8)
1476         pygame.draw.rect(screen, (200, 200, 200), pygame.Rect(box_x, box_y, box_w, box_h), 2,
1477 border_radius=8)
1478
1479         lbl = font_tiny.render("Click pt again to delete", True, (220, 220, 220))
1480         screen.blit(lbl, (box_x + 10, box_y + 80))
1481
1482         edit_x_field.draw(screen)
1483         edit_y_field.draw(screen)
1484         btn_edit_save.draw(screen)
1485
1486     draw_wp_panel(screen, wp_panel)
1487     draw_drive_panel(screen, drive_panel)
1488
1489     # bottom status strip
1490     pygame.draw.rect(screen, strip_bg, strip_rect)
1491     pygame.draw.line(screen, gray_dark, (strip_rect.x, strip_rect.y),
1492 (strip_rect.right, strip_rect.y), 1)
1493     bnd_note = " | BOUNDARY!" if preview_violates else ""
1494     if error_timer > 0:
1495         error_timer -= 1
1496         surf = font_small.render(error_message, True, red_warn)
1497     else:
1498         info = (f"TX: {status_label} | last: {last_tx_label} | "
1499 f"queue: {len(tx_queue)}{bnd_note}")
1500         surf = font_small.render(info, True, red_warn if preview_violates else black)
1501     screen.blit(surf, (10, strip_rect.y + 6))
1502
1503     pygame.display.flip()
1504     pygame.time.Clock().tick(200)
1505
1506     # -- send + read -----
1507     if ser is not None:
1508         try:
1509             ser.write(bytes([data]))
1510             Content = ser.readline()
1511             parsed = parse_serial_line(Content)
1512             if parsed is not None:
1513                 kind, val = parsed
1514                 if kind == 'heartbeat':
1515                     message = val
1516                     print(f"{val:08d}")
1517                 # tlog + sentinel lines are dropped silently in this phase
1518             except Exception as e:
1519                 print(f"Serial communication error: {e}")
1520                 ser = None
1521
1522     pygame.quit()

```

```
1521 sys.exit()
```

A.4 LilyGo robot-side firmware (Arduino/C++)

A.4.1 lilygo_bot.ino

```

1  #include <SPI.h>
2  #include <LoRa.h>
3  #include <Arduino.h>
4  #include <Wire.h>
5
6  #include "motor_engine.h"
7
8  #define BAND 868E6
9  #define CONFIG_MOSI 27
10 #define CONFIG_MISO 19
11 #define CONFIG_CLK 5
12 #define CONFIG_NSS 18
13 #define CONFIG_RST 23
14 #define CONFIG_DIO0 26
15
16 #define SDCARD_MOSI 15
17 #define SDCARD_MISO 2
18 #define SDCARD_SCLK 14
19 #define SDCARD_CS 13
20
21 #define I2C_SLAVE_ADDR 0x40
22
23 // trapezoidal speed command parametter for turning, migration on GRiSP for this part
24 #define max_turn_speed 80
25 #define turn_acc 400
26
27
28 float I2C_command[2] = {0.0, 0.0}; // value received from GRiSP : {wheels acceleration , turn speed}
29
30 float freq_lim [13] = {300,200,175,150,125,100,90,80,70,60,50,40,30};
31 int size_test_freq = sizeof(freq_lim)/sizeof(freq_lim[0]);
32 int index_lim = 0;
33
34 // time mesure variable
35 unsigned long t_GRiSP;
36 unsigned long t_LORA;
37 unsigned long t_test;
38 unsigned long t_ESP;
39
40 // freq and period variable
41 float dt_GRiSP = 10;
42 float freq_GRiSP = 200;
43 float dt_ESP = 0;
44
45 // control byte received
46 byte cmd = 0; // received from LoRa communication and transfered to GRiSP
47 byte GRiSP_flags = 0; // Received from GRiSP
48
49 //control flag
50 bool new_cmd =false;
51 bool test = false;
52 bool disturb = false;
53 bool ext_end = true;
54
55
56
57 void setup() {
58   Serial.begin(115200);
59   // SPI - LoRa init
60   SPI.begin(CONFIG_CLK, CONFIG_MISO, CONFIG_MOSI, CONFIG_NSS);
61   LoRa.setPins(CONFIG_NSS, CONFIG_RST, CONFIG_DIO0);
62   if (!LoRa.begin(BAND)) {
63     Serial.println("Starting LoRa failed!");

```

```

64     while (1);
65 }
66
67 // I2C Slave init, work with IRQ so no need to incorporate into the main loop
68 Wire.begin(I2C_SLAVE_ADDR);
69 Wire.onReceive(GRiSP_receiver);
70 Wire.onRequest(GRiSP_sender);
71
72 // motor init, works on core n°2
73 engine_init();
74 delay(1000);
75 set_speed(0, 0);
76 set_acceleration(0, 0);
77
78 // time init
79 t_GRiSP = millis();
80 t_LORA = t_GRiSP;
81 t_ESP = t_GRiSP;
82 }
83
84
85 void loop() {
86     unsigned long new_t_ESP = millis();
87     dt_ESP = (new_t_ESP - t_ESP) / 1000.0;
88     t_ESP = new_t_ESP;
89     LoRa_receiver();
90     Event_handle();
91     delay(1);
92 }
93
94
95
96 void GRiSP_receiver(int howMany) {
97     if(howMany == 5){ // check if the packet match the expected lenght
98         unsigned long new_t_GRiSP = millis();
99         dt_GRiSP = (new_t_GRiSP - t_GRiSP) / 1000.0;
100         freq_GRiSP = freq_GRiSP * 0.99 + 1.0 / dt_GRiSP * 0.01;
101         t_GRiSP = new_t_GRiSP;
102
103         byte A;
104         byte B;
105         if (Wire.available()) {
106
107             // read and decode the wheel acceleration
108             A = Wire.read();
109             B = Wire.read();
110             I2C_command[0] = decoder(A, B);
111
112             // read and decode the differential turn speed
113             A = Wire.read();
114             B = Wire.read();
115             I2C_command[1] = decoder(A, B);
116
117             // read the flags
118             GRiSP_flags = Wire.read();
119         }
120
121         //set acceleration
122         if(!disturb && bitRead(GRiSP_flags, 4) && !bitRead(GRiSP_flags, 6)){
123             set_acceleration(I2C_command[0], I2C_command[0]);
124         }
125
126         // Free fall, null wheel speed
127         if(bitRead(GRiSP_flags, 6)){
128             set_acceleration(0, 0);
129             set_speed(0, 0);
130         }
131
132         // extension/retraction of the rising system
133         if(bitRead(GRiSP_flags, 5)){
134             ext_end = stand(-48,30.0);
135         } else {
136             ext_end = stand(0,30.0);

```

```

137     }
138
139     // wheel counter rotation activation and direction selection during rise
140     int stand_speed_dir = 0;
141     if(!bitRead(GRiSP_flags, 4)){ // if down
142         stand_speed_dir = (bitRead(GRiSP_flags, 3)) ? -1 : 1;
143     }
144     raise_dir(stand_speed_dir); // -1 back, 0 null, 1 front
145 }
146
147 // Empty the stack
148 while(Wire.available()){
149     Wire.read();
150 }
151 }
152
153 void GRiSP_sender()
154 {
155     byte v[5];
156     float* speeds = get_speed();
157     encoder(v, speeds[0]);
158     encoder(v+2, speeds[2]);
159
160     // control byte send to GRiSP with : finish extension/retraction flag and the command inputs
161     v[4] = (cmd & 127) | (is_ready() * 128);
162
163     //send
164     Wire.write((byte*) v, sizeof(v));
165 }
166
167 double decoder(byte X, byte Y) {
168     // decode half float to double
169
170     byte A = (X & 192);
171     if ((A & 64) == 0) A = A | 63; // fill the missing exponent bytes with the right value
172     byte B = ((X << 2) & 252) | ((Y >> 6) & 3);
173     byte C = ((Y << 2) & 252);
174
175     byte vals[] = { 0x00, 0x00, 0x00, 0x00, 0x00, C, B, A };
176     double d = 0;
177     memcpy(&d, vals, 8);
178
179     return d;
180 }
181
182 void encoder(byte* res, double X){
183     // encode double to half float
184     byte vals[8];
185     memcpy(vals, &X, 8);
186     byte A = vals[7];
187     byte B = vals[6];
188     byte C = vals[5];
189
190     res[0] = (A&192)|((B>>2)&63);
191     res[1] = ((B<<6)&192)|((C>>2)&63);
192
193     return ;
194 }
195
196 void LoRa_receiver(){
197     // reception of LoRa packets
198     if (LoRa.parsePacket()) {
199         if(LoRa.available()>=2){
200             byte cmd1 = LoRa.read();
201             byte cmd2 = LoRa.read();
202             if(cmd1 == cmd2){
203                 cmd = cmd1;
204                 new_cmd = true;
205             }
206         }
207         while (LoRa.available()){
208             LoRa.read();
209         }

```

```

210 }
211 }
212
213 void Event_handle(){
214
215     //emergency stop
216     emergency(!bitRead(cmd, 7) || !bitRead(GRiSP_flags, 7));
217     if (!bitRead(cmd, 7) || !bitRead(GRiSP_flags, 7)){
218         set_acceleration(0, 0);
219         set_speed(0, 0);
220     }
221
222
223     // start test procedure
224     if(bitRead(cmd, 5)){
225         test = true;
226         t_test = millis();
227     }
228     if(test){
229         //start the disturbance 500ms to let the record start
230         if(millis() > t_test + 500){
231             //disturb = true;
232             //set_acceleration(40, 40);
233         }
234         // the disturbance is only applied between t=500 and t=800
235         if(millis() > t_test + 800){
236             disturb = false;
237             test = false;
238         }
239     }
240
241     set_turn(I2C_command[1]);
242
243     new_cmd = false;
244 }

```

A.4.2 motor_engine.h

```

1  #pragma once
2
3  // #include <stdint.h>
4  #include <Arduino.h>
5
6  #define sgn(x) ((x) < 0 ? -1 : ((x) > 0 ? 1 : 0))
7
8
9
10 #define MOTOR_AC_EN_PIN 14
11 #define MOTOR_A_STEP_PIN 12
12 #define MOTOR_A_DIR_PIN 13
13 #define MOTOR_B_EN_PIN 15
14 #define MOTOR_B_STEP_PIN 2
15 #define MOTOR_B_DIR_PIN 0
16 #define MOTOR_C_STEP_PIN 4
17 #define MOTOR_C_DIR_PIN 25
18
19 #define v_max 100 // speed max, in cm/s
20 #define microsteps 16
21 #define steps 400
22 #define diameter 7 // cm
23 #define lever_ratio 0.09090909090909090909090909090909 // 12/132
24 #define lenght_btw_wheels 18.5 // cm
25
26
27
28 void engine_init();
29
30 void Motor_engine(void *);
31
32 void engine_update(unsigned long dt_loop);
33

```

```

34 void set_speed(float,float);
35
36 void set_angular_speed(float,float);
37
38 void set_acceleration(float,float);
39
40 void set_angular_acceleration(float,float);
41
42 float* get_speed();
43
44 float* get_dist();
45
46 void reset_dist();
47
48 void set_turn(float);
49
50 void emergency(bool);
51
52 bool stand(float,float);
53
54 void raise_dir(int);
55
56 bool is_ready();

```

A.4.3 motor_engine.cpp

```

1  #include "motor_engine.h"
2  TaskHandle_t Motor_engine_task;
3
4  int long total_stepA = 0;
5  int long total_stepB = 0;
6  int long total_stepC = 0;
7
8  //motor dir
9  int dirA = 0;
10 int dirB = 0;
11 int dirC = 0;
12
13 //motor dt
14 int dt_MA = 0;
15 int dt_MB = 0;
16 int dt_MC = 0;
17
18 //motor avance speed, rot/s
19 float v_MA = 0;
20 float v_MB = 0;
21 float v_MC = 0;
22
23 // wheel counter rotation speed during rise, rot/sec
24 float v_MA_rise = 0;
25 float v_MC_rise= 0;
26 int wheel_raise_dir = 0;
27
28 // Total wheel speed
29 float v_tot_A = 0;
30 float v_tot_B = 0;
31 float v_tot_C = 0;
32
33 float v_diff =0.0;
34
35 //motor accelerations, only applied on motor avance speed, rot/s2
36 float a_MA = 0;
37 float a_MB = 0;
38 float a_MC = 0;
39
40 //motor dist
41 float total_dist_A = 0;
42 float total_dist_B = 0;
43 float total_dist_C = 0;
44
45 //motor freq

```

```

46 double f_MA = 0;
47 double f_MB = 0;
48 double f_MC = 0;
49
50 //Rise system parameters
51 int target_step = 0;
52 float Speed_stand = 0;
53
54 float v_max_ang = v_max /(PI * diameter);
55
56 void engine_init() {
57
58     pinMode(MOTOR_AC_EN_PIN, OUTPUT);
59     pinMode(MOTOR_B_EN_PIN, OUTPUT);
60     digitalWrite(MOTOR_AC_EN_PIN, LOW);
61     digitalWrite(MOTOR_B_EN_PIN, LOW);
62
63     pinMode(MOTOR_A_STEP_PIN, OUTPUT);
64     pinMode(MOTOR_B_STEP_PIN, OUTPUT);
65     pinMode(MOTOR_C_STEP_PIN, OUTPUT);
66     digitalWrite(MOTOR_A_STEP_PIN, LOW);
67     digitalWrite(MOTOR_B_STEP_PIN, LOW);
68     digitalWrite(MOTOR_C_STEP_PIN, LOW);
69
70     pinMode(MOTOR_A_DIR_PIN, OUTPUT);
71     pinMode(MOTOR_B_DIR_PIN, OUTPUT);
72     pinMode(MOTOR_C_DIR_PIN, OUTPUT);
73     digitalWrite(MOTOR_A_DIR_PIN, LOW);
74     digitalWrite(MOTOR_B_DIR_PIN, LOW);
75     digitalWrite(MOTOR_C_DIR_PIN, HIGH);
76
77
78     xTaskCreatePinnedToCore(
79         Motor_engine,      /* Task function. */
80         "Motor_engine",    /* name of task. */
81         10000,             /* Stack size of task */
82         (void*)NULL,       /* parameter of the task */
83         1,                 /* priority of the task */
84         &Motor_engine_task, /* Task handle to keep track of created task */
85         1);                /* pin task to core 1 */
86 }
87
88
89 void Motor_engine(void* Parameters) {
90     //setup
91     Serial.print("Motor Engine running on core ");
92     Serial.println(xPortGetCoreID());
93
94
95     unsigned long t_MA = 0;
96     unsigned long t_MB = 0;
97     unsigned long t_MC = 0;
98
99     unsigned long t_motor = micros();
100     unsigned long dt_motor;
101     unsigned long new_t_motor;
102
103     //loop
104     while (true) {
105
106         //time calculation
107         new_t_motor = micros();
108         dt_motor = new_t_motor - t_motor;
109         t_motor = new_t_motor;
110         t_MA += dt_motor;
111         t_MB += dt_motor;
112         t_MC += dt_motor;
113
114
115         // incrementation of the stepper motors
116         if (dirA != 0 && t_MA > dt_MA) {
117             digitalWrite(MOTOR_A_STEP_PIN, HIGH);
118             total_stepA += dirA;

```

```

119     t_MA = 0;
120 } else {
121     digitalWrite(MOTOR_A_STEP_PIN, LOW);
122 }
123
124 if (dirB != 0 && t_MB > dt_MB) {
125     digitalWrite(MOTOR_B_STEP_PIN, HIGH);
126     total_stepB += dirB;
127     t_MB = 0;
128 } else {
129     digitalWrite(MOTOR_B_STEP_PIN, LOW);
130 }
131
132 if (dirC != 0 && t_MC > dt_MC) {
133     digitalWrite(MOTOR_C_STEP_PIN, HIGH);
134     total_stepC += dirC;
135     t_MC = 0;
136 } else {
137     digitalWrite(MOTOR_C_STEP_PIN, LOW);
138 }
139
140 engine_update(dt_motor); // update each motor step period
141 }
142 }
143
144
145
146 void set_speed(float vA, float vC) {
147     v_MA = vA/(PI * diameter);
148     v_MC = vC/(PI * diameter);
149 }
150
151 void set_angular_speed(float vA, float vC) {
152     v_MA = vA;
153     v_MC = vC;
154 }
155
156 void set_acceleration(float aA, float aC) {
157     a_MA = aA/(PI * diameter);
158     a_MC = aC/(PI * diameter);
159 }
160
161 void set_angular_acceleration(float aA, float aC) {
162     a_MA = aA;
163     a_MC = aC;
164 }
165
166 float list_speed[3] = { 0.0, 0.0, 0.0 };
167 float* get_speed() {
168     list_speed[0] = v_tot_A*(PI * diameter);
169     list_speed[1] = v_tot_B * lever_ratio;
170     list_speed[2] = v_tot_C*(PI * diameter);
171     return list_speed;
172 }
173
174 float list_dist[3] = { 0.0, 0.0, 0.0 };
175 float* get_dist() {
176     list_dist[0] = total_stepA * 1.0 / (steps * microsteps) * diameter * PI;
177     list_dist[1] = total_stepB * 1.0 / (steps * microsteps) * lever_ratio;
178     list_dist[2] = total_stepC * 1.0 / (steps * microsteps) * diameter * PI;
179     return list_dist;
180 }
181
182 //low speed loop
183 void engine_update(unsigned long dt_loop) {
184
185     //acceleration calculation of motor A with limit to +/- vmax
186     v_MA += a_MA * dt_loop * 1e-6 ;
187     if (v_MA > v_max_ang) {
188         v_MA = v_max_ang;
189     } else if (v_MA < -v_max_ang) {
190         v_MA = -v_max_ang;
191     }

```

```

192
193 //displacement calculation of motor B to set the extension/retraction speed
194 //and the wheel counter rotation
195 if (total_stepB > target_step){
196     v_MB = Speed_stand / lever_ratio;
197     v_MA_rise = Speed_stand * wheel_raise_dir;
198     v_MC_rise = Speed_stand * wheel_raise_dir;
199 } else if (total_stepB < target_step){
200     v_MB = -Speed_stand / lever_ratio;
201     v_MA_rise = -Speed_stand * wheel_raise_dir;
202     v_MC_rise = -Speed_stand * wheel_raise_dir;
203 } else {
204     v_MB = 0;
205     v_MA_rise = 0;
206     v_MC_rise = 0;
207 }
208
209 //acceleration calculation of motor A with limit to +/- vmax
210 v_MC += a_MC * dt_loop * 1e-6;
211 if (v_MC > v_max_ang) {
212     v_MC = v_max_ang;
213 } else if (v_MC < -v_max_ang) {
214     v_MC = -v_max_ang;
215 }
216
217 //total speeds
218 v_tot_A = v_MA + v_diff + v_MA_rise;
219 v_tot_B = v_MB;
220 v_tot_C = v_MC - v_diff + v_MC_rise;
221
222
223 //compute freq from speed
224 f_MA = v_tot_A * steps * microsteps;
225 f_MB = v_tot_B * steps * microsteps;
226 f_MC = v_tot_C * steps * microsteps;
227
228 //compute the increment period from freq and set the direction
229 if (f_MA < 0) {
230     dt_MA = 1e6 / abs(f_MA);
231     digitalWrite(MOTOR_A_DIR_PIN, LOW);
232     dirA = 1;
233 } else if (f_MA > 0) {
234     dt_MA = 1e6 / abs(f_MA);
235     digitalWrite(MOTOR_A_DIR_PIN, HIGH);
236     dirA = -1;
237 } else {
238     dirA = 0;
239 }
240
241
242 if (f_MB > 0) {
243     dt_MB = 1e6 / abs(f_MB);
244     digitalWrite(MOTOR_B_DIR_PIN, LOW);
245     dirB = -1;
246 } else if (f_MB < 0) {
247     dt_MB = 1e6 / abs(f_MB);
248     digitalWrite(MOTOR_B_DIR_PIN, HIGH);
249     dirB = 1;
250 } else {
251     dirB = 0;
252 }
253
254 if (f_MC > 0) {
255     dt_MC = 1e6 / abs(f_MC);
256     digitalWrite(MOTOR_C_DIR_PIN, LOW);
257     dirC = -1;
258 } else if (f_MC < 0) {
259     dt_MC = 1e6 / abs(f_MC);
260     digitalWrite(MOTOR_C_DIR_PIN, HIGH);
261     dirC = 1;
262 } else {
263     dirC = 0;
264 }

```

```

265 }
266
267 void reset_dist() { //reset function
268     total_stepA = 0;
269     total_stepB = 0;
270     total_stepC = 0;
271 }
272
273 void set_turn(float angular_speed){ // apply a differential speed on the wheel to turn
274     v_diff = (lenght_btwh_wheels*PI*angular_speed / (180.0*2))/(PI * diameter) ;
275 }
276
277 void emergency(bool stop){ //shut off the power
278     if(stop){
279         digitalWrite(MOTOR_AC_EN_PIN, HIGH);
280         digitalWrite(MOTOR_B_EN_PIN, HIGH);
281     }else{
282         digitalWrite(MOTOR_AC_EN_PIN, LOW);
283         digitalWrite(MOTOR_B_EN_PIN, LOW);
284     }
285 }
286
287 bool stand(float angle, float speed){
288     target_step = angle/ (360 * lever_ratio) * steps * microsteps;
289     Speed_stand = speed*1.0/360.0 ; // go from deg/s to rot/s
290     return total_stepB == target_step;
291 }
292
293 void raise_dir(int dir){ // Use to specify the wheel counter rotation direction during raise/fall
294     wheel_raise_dir = dir;
295 }
296
297 bool is_ready(){
298     return total_stepB == target_step;
299 }

```

A.5 LilyGo sender-side firmware (Arduino/C++)

A.5.1 liygo_user.ino

```

1  #include <SPI.h>
2  #include <LoRa.h>
3  #include <Wire.h>
4
5
6  #define Pin 15
7  #define Buzz 13
8
9  #define LORA_PERIOD 868
10 #define BAND 868E6
11
12
13 #define CONFIG_MOSI 27
14 #define CONFIG_MISO 19
15 #define CONFIG_CLK 5
16 #define CONFIG_NSS 18
17 #define CONFIG_RST 23
18 #define CONFIG_DIO0 26
19
20
21
22 unsigned long t ;
23 int state = 0;
24 int prevstate = 0;
25 byte cmd;
26
27 void setup()
28 {
29     // init serial

```

```

30 Serial.begin(115200);
31 while (!Serial);
32
33
34 // init LoRa
35 SPI.begin(CONFIG_CLK, CONFIG_MISO, CONFIG_MOSI, CONFIG_NSS);
36 LoRa.setPins(CONFIG_NSS, CONFIG_RST, CONFIG_DIO0);
37 if (!LoRa.begin(BAND)) {
38   Serial.println("Starting LoRa failed!");
39   while (1);
40 }
41
42 //init Pin
43 pinMode(Pin, OUTPUT);
44 pinMode(Pin, INPUT_PULLUP);
45 pinMode(Buzz, OUTPUT);
46 digitalWrite(Buzz, LOW);
47
48 prevstate = esp_sleep_get_wakeup_cause() != ESP_SLEEP_WAKEUP_TIMER && !digitalRead(Pin);
49 t = millis();
50 }
51
52 int count = 0;
53
54 //main loop
55 void loop(){
56   Keyboard_input();
57   LoRA_sender();
58 }
59
60 void LoRA_sender(){
61
62   state = digitalRead(Pin);
63
64   if(state==HIGH){
65     cmd = cmd & 127;
66   }
67
68   LoRa.beginPacket();
69   // send two packet for redundancy
70   LoRa.write(cmd);
71   LoRa.write(cmd);
72   LoRa.endPacket();
73   Serial.println(cmd, BIN);
74
75
76   //buzzer logic
77   if(state == HIGH && prevstate == LOW){
78     digitalWrite(Buzz, HIGH);
79     delay(100);
80     digitalWrite(Buzz, LOW);
81   }
82
83
84   if(state == LOW && prevstate == HIGH){
85     digitalWrite(Buzz, HIGH);
86     delay(100);
87     digitalWrite(Buzz, LOW);
88   }
89   prevstate = state;
90 }
91
92
93 void Keyboard_input(){
94   if (Serial.available() > 0) {
95     cmd = Serial.read();
96   }
97
98   while(Serial.available()){
99     Serial.read();
100   }
101 }

```

A.5.2 board_def.h

```

1 #include <Arduino.h>
2
3 #define LORA_V1_0_OLED 0
4 #define LORA_V1_2_OLED 0
5 #define LORA_V1_6_OLED 1
6 #define LORA_V2_0_OLED 0
7
8 // Change here whether you define Sender or Receiver.
9 // The rest should be the same
10 // #define LORA_SENDR 0
11 #define LORA_SENDR 1
12
13 #define LORA_PERIOD 868
14 // #define LORA_PERIOD 915
15 // #define LORA_PERIOD 433
16
17 #if LORA_V1_0_OLED
18 #include <Wire.h>
19 #include "SSD1306Wire.h"
20 #define OLED_CLASS_OBJ SSD1306Wire
21 #define OLED_ADDRESS 0x3C
22 #define OLED_SDA 4
23 #define OLED_SCL 15
24 #define OLED_RST 16
25 #define CONFIG_MOSI 27
26 #define CONFIG_MISO 19
27 #define CONFIG_CLK 5
28 #define CONFIG_NSS 18
29 #define CONFIG_RST 14
30 #define CONFIG_DIO0 26
31 // ! There are two versions of TTGO LoRa V1.0,
32 // ! the 868 version uses the 3D WiFi antenna, and the 433 version uses the PCB antenna.
33 // ! You need to change the frequency according to the board.
34
35 #define SDCARD_MOSI -1
36 #define SDCARD_MISO -1
37 #define SDCARD_SCLK -1
38 #define SDCARD_CS -1
39
40 #elif LORA_V1_2_OLED
41 //Lora V1.2 ds3231
42 #include <Wire.h>
43 #include "SSD1306Wire.h"
44 #define OLED_CLASS_OBJ SSD1306Wire
45 #define OLED_ADDRESS 0x3C
46 #define OLED_SDA 21
47 #define OLED_SCL 22
48 #define OLED_RST -1
49 #define CONFIG_MOSI 27
50 #define CONFIG_MISO 19
51 #define CONFIG_CLK 5
52 #define CONFIG_NSS 18
53 #define CONFIG_RST 23
54 #define CONFIG_DIO0 26
55
56 #define SDCARD_MOSI -1
57 #define SDCARD_MISO -1
58 #define SDCARD_SCLK -1
59 #define SDCARD_CS -1
60
61 #define ENABLE_DS3231
62
63 #elif LORA_V1_6_OLED
64 #include <Wire.h>
65 #include "SSD1306Wire.h"
66 #define OLED_CLASS_OBJ SSD1306Wire
67 #define OLED_ADDRESS 0x3C
68 #define OLED_SDA 21
69 #define OLED_SCL 22
70 #define OLED_RST -1
71

```

```
72 #define CONFIG_MOSI 27
73 #define CONFIG_MISO 19
74 #define CONFIG_CLK 5
75 #define CONFIG_NSS 18
76 #define CONFIG_RST 23
77 #define CONFIG_DIO0 26
78
79 #define SDCARD_MOSI 15
80 #define SDCARD_MISO 2
81 #define SDCARD_SCLK 14
82 #define SDCARD_CS 13
83
84 #elif LORA_V2_0_OLED
85 #include <Wire.h>
86 #include "SSD1306Wire.h"
87 #define OLED_CLASS_OBJ SSD1306Wire
88 #define OLED_ADDRESS 0x3C
89 #define OLED_SDA 21
90 #define OLED_SCL 22
91 #define OLED_RST -1
92
93 #define CONFIG_MOSI 27
94 #define CONFIG_MISO 19
95 #define CONFIG_CLK 5
96 #define CONFIG_NSS 18
97 #define CONFIG_RST 23
98 #define CONFIG_DIO0 26
99
100 #define SDCARD_MOSI 15
101 #define SDCARD_MISO 2
102 #define SDCARD_SCLK 14
103 #define SDCARD_CS 13
104
105 #else
106 #error "please select board"
107 #endif
108
109
110 #if LORA_PERIOD == 433
111 #define BAND 433E6
112 #elif LORA_PERIOD == 868
113 #define BAND 868E6
114 #elif LORA_PERIOD == 915
115 #define BAND 915E6
116 #else
117 #error "Please select the correct lora frequency"
118 #endif
```

Appendix B

Wire Protocol Full Specification

This appendix is the normative reference for the one-byte waypoint-injection protocol introduced in Chapter 10. Chapter 10 carries the motivation, the ADD-branch state machine, and the design rationale; what follows here is the exhaustive byte-level description. Every value listed below is taken from `include/robot_types.hrl` and the receiver implementation in `src/main_loop.erl`; the symbol names match the source one-to-one.

B.1 Scope and transport stack

The protocol is transported as the trailing byte of the five-byte I²C frame already used by the inherited platform for wheel-speed feedback. A single control byte (referred to as `CtrlByte` in `CLAUDE.md`) is reused: when bit 6 (`PROTO`) is set, the byte is interpreted as a protocol frame instead of a drive byte. The end-to-end path is shown in Figure B.1.

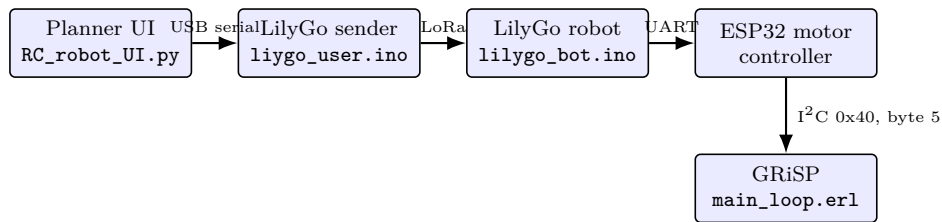


Figure B.1: Transport stack. The same byte travels four hops; only the final hop is on-board the GRiSP. The first four bytes of the I²C frame carry wheel-speed feedback and are unrelated to the protocol.

The protocol is one-way: the GRiSP does not acknowledge frames over the wire. Operator feedback is provided locally by the GRiSP LEDs (Section B.8).

B.2 Control byte format

A single byte is decoded by `hera_com:get_bits/1` into eight bits, indexed b7 (most significant) to b0 (least significant). The layout is fixed; see Figure B.2.

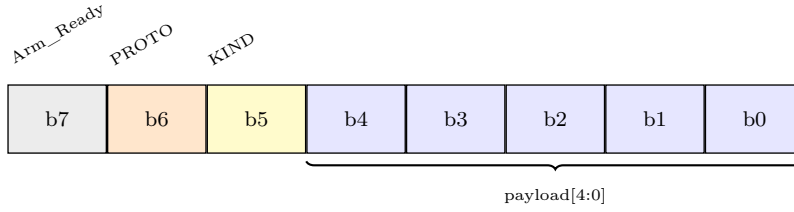


Figure B.2: Control-byte bit field. Bit 7 is set by the ESP32 and is ignored by the protocol decoder. Bits 6 and 5 select the frame class. The bottom five bits carry either a control code or one half of a data axis.

Bit	Name	Set by	Semantics
b7	Arm_Ready	ESP32	motor-arm status; ignored by the protocol decoder
b6	PROTO	sender	1 = protocol frame; 0 = normal drive byte
b5	KIND	sender	valid only when PROTO=1; 1 = data, 0 = control
b4..b0	payload	sender	5-bit control code or 5-bit data half

Table B.1: Bit-by-bit semantics of the control byte.

The masks used by the decoder are listed below.

Symbol	Value	Purpose
CMD_PROTO_MASK	0x40 (0b01000000)	isolate the PROTO bit
CMD_KIND_MASK	0x20 (0b00100000)	isolate the KIND bit
CMD_PAYLOAD_MASK	0x1F (0b00011111)	isolate the 5-bit payload

B.3 Command set

Nine control codes are defined. Each is a 5-bit pattern carried in the payload of a control frame (PROTO=1, KIND=0). The full-byte column lists the byte that is actually transmitted (i.e., `0x40 | code`, with Arm_Ready masked out for clarity).

Pattern	Full byte	Symbol	Effect
00000	0x40	CMD_CTRL_ABORT	receiver is reset to <code>idle</code> ; any in-flight ADD is discarded
10000	0x50	CMD_CTRL_CANCEL_LAST	the most recently staged waypoint is popped
10001	0x51	CMD_CTRL_CLEAR_ALL	the entire staged queue is wiped
10010	0x52	CMD_CTRL_CANCEL_N_HDR	first frame of the two-frame CANCEL_N transaction
10011	0x53	CMD_CTRL_CANCEL_N_COMMIT	second frame; the pending N is consumed and applied
11000	0x58	CMD_CTRL_ADD_HEADER	first frame of the nine-frame ADD transaction
11010	0x5A	CMD_CTRL_HI_TO_LO	separator between the hi and lo halves of one axis
11100	0x5C	CMD_CTRL_X_TO_Y	separator between the X and Y axes
11110	0x5E	CMD_CTRL_ADD_COMMIT	final frame; the two halves of each axis are reassembled and the waypoint is appended

Table B.2: Complete control command set.

B.4 Data-frame encoding

Each waypoint offset is a signed value expressed in centimetres relative to the previous waypoint (or to the origin for the first waypoint). The valid range is $v \in [-50, +50]$ cm per axis.

Each axis is transmitted as a ten-bit unsigned value, split across two data frames. The unsigned value is obtained from the signed value via a bias of 50:

$$\text{enc} = v + \text{CMD_SIGN_OFFSET}, \quad v = ((\text{hi} \ll 5) \mid \text{lo}) - \text{CMD_SIGN_OFFSET}, \quad (\text{B.1})$$

with $\text{CMD_SIGN_OFFSET} = 50$ and $\text{CMD_MAX_OFFSET} = 50$. The encoded value is then split into a 5-bit high half and a 5-bit low half, each of which is sent as one data frame (PROTO=1, KIND=1, payload = the 5 bits).

Worked numeric example. An offset of $v = +20$ cm is encoded as $\text{enc} = 70 = 0b0010\,00110$, giving $\text{hi} = 00010$ and $\text{lo} = 00110$. The two data bytes that appear on the wire are therefore `0x62` and `0x66`.

The receiver rejects any reassembled value that lies outside $[-50, +50]$. Out-of-range values cause the frame to be dropped, the receiver to be reset, and the error LED to be fired.

B.5 ADD transaction

A single waypoint requires nine frames transmitted in fixed order. Table B.3 lists the sequence for the worked offset $(\Delta x, \Delta y) = (+20, -15)$ cm, which encodes to $(\text{enc}_x, \text{enc}_y) = (70, 35)$ and therefore to data halves (2, 6) for X and (1, 3) for Y .

#	Frame name	Class	Byte (hex)	Receiver state after
1	ADD_HEADER	ctrl	0x58	want_x_hi
2	data $X_{hi} = 2$	data	0x62	want_x_hi_to_lo
3	HI_TO_LO	ctrl	0x5A	want_x_lo
4	data $X_{lo} = 6$	data	0x66	want_x_to_y
5	X_TO_Y	ctrl	0x5C	want_y_hi
6	data $Y_{hi} = 1$	data	0x61	want_y_hi_to_lo
7	HI_TO_LO	ctrl	0x5A	want_y_lo
8	data $Y_{lo} = 3$	data	0x63	want_commit
9	ADD_COMMIT	ctrl	0x5E	idle (waypoint appended)

Table B.3: The nine frames of an ADD transaction for the worked offset $(+20, -15)$ cm.

The role of the HI_TO_LO separator is to guarantee that two byte-identical data frames are never sent back to back: without it, an offset such as $\Delta x = 16$ would produce $X_{hi} = X_{lo} = 2$ and the commit-edge gate (Section B.8) would refuse the second one.

The sender’s pacing and the receiver’s stability gate are shown in Figure B.3.

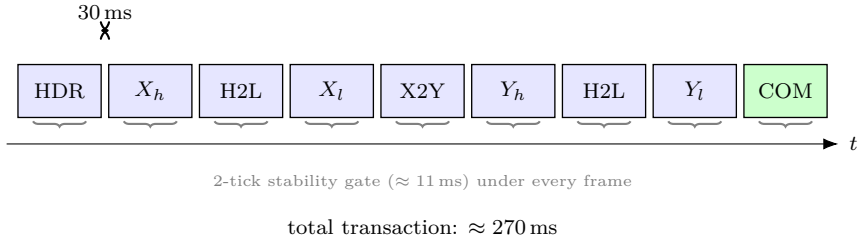


Figure B.3: ADD timing. The sender emits one frame every 30 ms. The receiver requires two consecutive ticks with the same byte before the frame is accepted; at the 300 Hz loop, two ticks is about 11 ms. The full nine-frame transaction lands in roughly 270 ms.

B.6 CANCEL_N transaction

The CANCEL_N transaction is the only other multi-frame transaction in the protocol. Two frames are used.

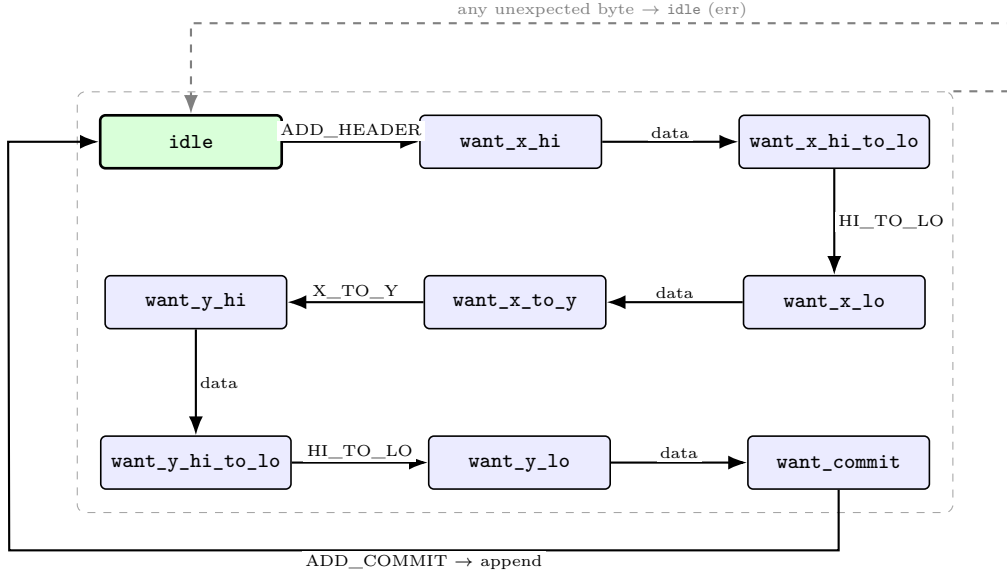
#	Frame name	Class	Byte (hex)	Receiver state after
1	CANCEL_N_HDR	ctrl	0x52	want_cancel_n_data
2	data N (5-bit payload)	data	0x60 N	idle (N waypoints popped)

Table B.4: The two frames of a CANCEL_N transaction. The 5-bit payload allows $N \in [0, 31]$.

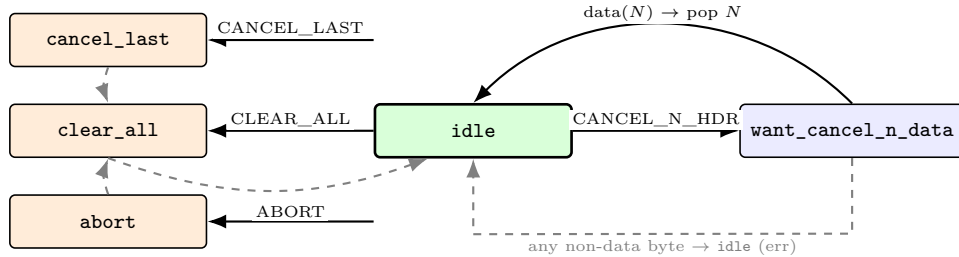
The CANCEL_N_COMMIT code is reserved but not used in current builds; the data frame is itself the commit. The code is kept in the table of Section B.3 so that a future three-frame variant can be added without renumbering.

B.7 Receiver state machine

The complete receiver state machine is shown in Figure B.4. Chapter 10 shows only the ADD branch; the figure below adds the CANCEL_N branch, the three singleton control transitions (CANCEL_LAST, CLEAR_ALL, ABORT), and the universal error edge.



(a) ADD branch with the universal error edge. The solid arrows form the success path. The dashed grey rectangle that wraps the eight chain states is a notational grouping device: it stands for “any state inside this box”. The single dashed grey arrow leaving the rectangle therefore represents one error transition per state inside the group — eight identical edges drawn as one — and reads “any unexpected byte from any state inside the box $\rightarrow$ idle”.



(b) CANCEL_N branch (right) and the three single-frame actions (left). Solid arrows are driven by an input frame; dashed grey arrows show the one-tick returns to `idle`. The CANCEL_N error edge fires when `want_cancel_n_data` receives anything other than a data frame; the singleton actions cannot fail because they fire from `idle` on a recognised byte and complete in the same tick.

Figure B.4: Complete receiver state machine, including every error transition. (a) covers the ADD chain plus its universal error edge; (b) covers the CANCEL_N branch (with its own error edge) and the three single-frame singleton actions. Every transition in Table B.5 is represented in one of the two panels.

The complete transition table is given in Table B.5. The rows enumerate the receiver state; the columns enumerate the frame class actually consumed. An entry of “ $\rightarrow$ idle (err)” means that the receiver returns to idle and the error LED is fired.

State	ADD_HEADER	data	ctrl separator	other / unexpected
idle	want_x_hi	→ idle (err)	per Table B.6	→ idle (err)
want_x_hi	—	want_x_hi_to_lo	—	→ idle (err)
want_x_hi_to_lo	—	—	want_x_lo on HI_TO_LO	→ idle (err)
want_x_lo	—	want_x_to_y	—	→ idle (err)
want_x_to_y	—	—	want_y_hi on X_TO_Y	→ idle (err)
want_y_hi	—	want_y_hi_to_lo	—	→ idle (err)
want_y_hi_to_lo	—	—	want_y_lo on HI_TO_LO	→ idle (err)
want_y_lo	—	want_commit	—	→ idle (err)
want_commit	—	—	idle on ADD_COMMIT (append)	→ idle (err)
want_cancel_n_data	—	idle (pop N)	—	→ idle (err)

Table B.5: State $\times$ input transition table for the receiver FSM. “—” marks an input that cannot legally occur in that state; it is treated as “other” and routed through the error edge.

Control code	Action	Next state
CANCEL_LAST	most recent waypoint popped	idle
CLEAR_ALL	whole staged queue wiped	idle
ABORT	receiver state reset (no queue edit)	idle
CANCEL_N_HDR	N-pop transaction begun	want_cancel_n_data
ADD_HEADER	ADD transaction begun	want_x_hi

Table B.6: Transitions from `idle` on a control frame.

B.8 Guards, error handling, and feedback

Three guards are evaluated in order before any state transition fires. They are listed below in the order in which the byte traverses them.

Stability gate. A frame is accepted only after two consecutive ticks have carried the same byte. At the 300 Hz loop rate, two ticks is approximately 11 ms. The gate exists so that a single noisy tick during a LoRa retransmission cannot fire a stray transition. The sender’s 30 ms inter-frame pacing leaves ample room.

Commit-edge gate. A byte that has already been committed is not committed again until a different byte intervenes. This prevents a held-still wire from re-firing the same frame on every 6.7 ms tick. The gate is released when a stable drive byte ($\text{PROTO} = 0$) is observed, at which point the next protocol frame counts fresh.

Sign-range check. On `ADD_COMMIT`, the two halves of each axis are reassembled per Equation B.1. The result must lie in $[-50, +50]$ on both axes. Out-of-range values cause the waypoint to be dropped, the FSM to be reset, and the error LED to be fired.

Five conditions cause the receiver to be reset to `idle`: (i) a successful `ADD_COMMIT`, (ii) the data frame of a `CANCEL_N` transaction, (iii) any of the singleton control frames, (iv) an unexpected byte in any non-idle state, and (v) an out-of-range commit value. Conditions (i)–(iii) are clean exits; (iv) and (v) fire the error LED.

Operator feedback is provided exclusively by the GRiSP LEDs. The pulse lengths are summarised in Table B.7.

Symbol	Ticks	Approx. ms	Trigger
CMD_LED_PULSE_FRAME_OK	1	≈ 3	each frame that passes the stability and commit-edge gates
CMD_LED_PULSE_COMMITTED	75	≈ 250	a transaction completed cleanly
CMD_LED_PULSE_ERROR	150	≈ 500	receiver reset on an error condition

Table B.7: LED pulse lengths used by the receiver as operator feedback.

B.9 Constants summary

All numeric constants defined by the protocol are collected in Table B.8 so that the reader does not have to chase them through the source.

Symbol	Value	Meaning
CMD_PROTO_MASK	0x40	PROTO bit mask
CMD_KIND_MASK	0x20	KIND bit mask
CMD_PAYLOAD_MASK	0x1F	5-bit payload mask
CMD_SIGN_OFFSET	50	bias added by the encoder and subtracted by the decoder
CMD_MAX_OFFSET	50	magnitude limit per axis, in centimetres
CMD_LED_PULSE_FRAME_OK	1	per-frame LED pulse, in ticks
CMD_LED_PULSE_COMMITTED	75	commit LED pulse, in ticks (≈ 250 ms)
CMD_LED_PULSE_ERROR	150	error LED pulse, in ticks (≈ 500 ms)
stability gate width	2 ticks (≈ 11 ms)	frames are accepted only after this many identical ticks
inter-frame pacing	≈ 30 ms	enforced by the sender between consecutive frames
loop rate	300 Hz	nominal main-loop frequency of the receiver

Table B.8: Every numeric constant exposed by the protocol, with its source symbol and meaning.

Appendix C

Reproducing the Experiments

The localization and trajectory-following experiments reported in Part III were run against two floor-marker rigs taped onto a flat, dust-free indoor floor. The two rigs are reproduced below to scale so that the layout can be re-built by a future operator. In both figures the body frame of the chassis is used: the x axis points forward (north) and the y axis points to the left (west); the robot is drawn as a 12×18.5 cm box (12 cm along the forward axis, 18.5 cm along the wheel axis) with an arrow showing its forward direction.

C.1 Rotation rig

The rotation rig was used for the clockwise and counter-clockwise yaw experiments. Figure C.1 shows the markings. A 50×100 cm rectangle is taped onto the floor with its centre-south edge anchored at the origin and its long side running east–west. A central line bisects the rectangle and runs due north from the origin to $(50, 0)$. At the origin, a fan of 10° -spaced radial lines (12 cm long) is drawn so that residual yaw can be read visually against the closest marker after the robot has stopped. The robot is placed centred at the origin facing north.

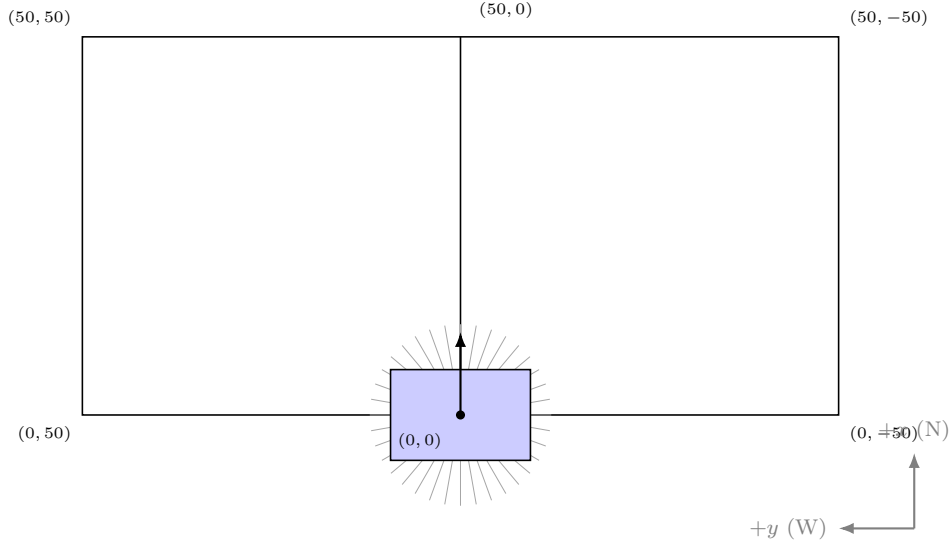


Figure C.1: Rotation rig used for the yaw experiments. The 50×100 cm rectangle defines the floor area; the central line $(0, 0) - (50, 0)$ is the heading reference; the fan of 10° -spaced 12 cm radial lines at the origin lets residual yaw be read after the robot stops. The robot is shown to scale, centred at the origin and facing north.

This rig is referenced in Section 11.2 (where the rotational test paths are defined) and re-used in Section 16.2 (where the end-of-path errors are measured).

C.2 Straight-line rig

The straight-line rig was used for the figure-eight traversal and the 100 cm and 250 cm straight-line experiments. Figure C.2 shows the markings. A single 250 cm line is taped onto the floor running due north from the origin, with cross ticks every 25 cm so that distance travelled can be read off directly. A 20 cm perpendicular start mark crosses the line at the origin, giving a visual reference for the initial pose. The robot is placed centred at the origin facing north.

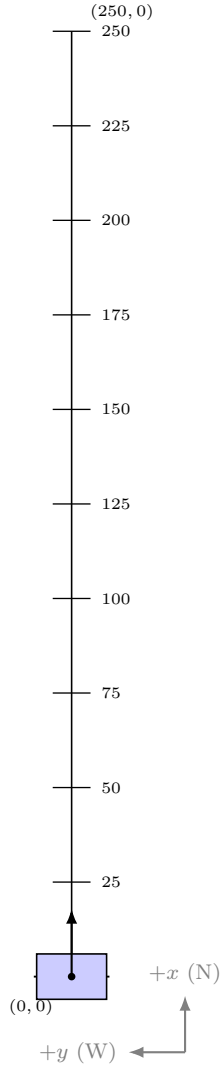


Figure C.2: Straight-line rig used for the straight-line and figure-eight experiments. A 250 cm line runs due north from the origin with ticks every 25 cm; the 20 cm perpendicular bar at the origin gives a visual start-pose reference. The robot is shown to scale, centred at the origin and facing north. Tick labels are in centimetres.

This rig is referenced in Section 11.2 (where the straight-line and figure-eight paths are defined) and re-used in Section 16.2 (where their end-of-path errors are measured).

Appendix D

Code Repositories

The complete source for the system is hosted on GitHub across two repositories. The listings reproduced in Appendix A are snapshots; the up-to-date version of the code can always be found at the URLs below.

- GRiSP Erlang application (`balancing_robot`):
`https://github.com/fozanshahiducl/balancing\_robot`
- Open-source self-balancing robot (operator UI, LilyGo robot-side firmware, and LilyGo sender-side firmware):
`https://github.com/fozanshahiducl/Opensource\_self\_balancing\_robot`

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl