École polytechnique de Louvain

Practical domotics with sensor fusion

A two-wheeled self-balancing butler

Authors: **Arthur Dandoy, Sam Raymakers**
Supervisor: **Peter Van Roy**
Readers: **Peer Stritzinger, Raphael Jungers**
Academic year 2024-2025
Master [120] in Computer Science & Electrical Engineering

Acknowledgements

First, we are deeply grateful to Professor Van Roy for his sustained involvement, insightful ideas, and steady encouragement. His high expectations continually challenged us to strive for the very best.

We would also like to thank the CREDEM team, especially Simon De Jaeger, for their support, invaluable advice, and practical training with all the equipment we used, as well as for granting us access to it.

We gratefully acknowledge François Goens and Cédric Ponsard for their generous advice and hands-on assistance during the robot's construction. Their guidance saved us considerable time, and their prompt, thoughtful responses kept the project moving.

Warm thanks go to Thomas and Nicolas for their helpful suggestions and generous sharing of resources. Facing similar challenges, we were able to support one another and exchange ideas, which helped us advance together.

Finally, our sincere thanks to our families and friends for their patience, and steady support, and for clearing space at home so the robot could run safely. We would also like to apologize for the occasional broken glass along the way; we promise it helped us get through this challenge.

Abstract

Connected systems and the Internet of Things (IoT) are becoming more integrated into daily life, offering greater comfort, efficiency, and safety. However, these technologies often rely on complex, real-time interactions that make the control of unstable systems, such as dynamic self-balancing robots, particularly challenging due to sensor noise, latency, and limited precision. This master's thesis aims to transform a two-wheeled self-balancing robot prototype into a functional domestic butler capable of transporting payloads, like a glass of wine, without spilling and avoiding obstacles in real-world environments.

This work builds upon the Hera framework and employs GRiSP embedded boards to implement a Kalman filter-based sensor fusion, thereby enhancing the reliability of noisy sensor measurements. A two-wheeled self-balancing butler with a tall and robust chassis, retractable support arms, a mounting platform for payloads, and three ultrasonic sonars for obstacle detection, was designed. Stability is achieved through a cascade of PID controllers, while a distributed multi-GRiSP architecture synchronizes sensor data in real time.

Additional features include two operational modes, dynamic self-balancing and static stability mode, and remote control via an ESP module. The resulting system is multi-level: an outer executive loop manages motion and obstacle avoidance, and an inner loop maintains balance. Experimental results show stable operation under different payloads, floor types, and obstacles, demonstrating the feasibility of applying advanced robotics in smart home environments and providing a reusable framework for similar IoT-based systems.

AI disclaimer

The use of AI, such as ChatGPT and DeepL for traduction purpose, grammar correction, and rewriting, and GitHub Copilot for support in code development and structuring, is acknowledged in this work.

Contents

List of Figures

List of Tables

Acronyms

CG center of gravity.

CV Control Variable.

DoF Degree of Freedom.

EKF Extended Kalman Filter.

ESC Electronic Stability Control.

ESP32 Espressif Systems ESP32 Microcontroller.

FCE Final Control Element.

FDM Fused Deposition Modeling.

FFT Fast Fourier Transform.

FSM Finite State Machine.

I^2C Inter-Integrated Circuit.

IMU inertial measurement unit.

IoT Internet of Things.

NIF Native Implemented Function.

NN Neural Network.

OTP Open Telecom Platform.

PCB Printed Circuit Board.

PD Proportional-Derivative.

PI Proportional-Integral.

PID Proportional-Integral-Derivative.

Pmod Peripheral Module.

PV Process Variable.

SP Set Point.

SPI Serial Peripheral Interface.

UART Universal Asynchronous Receiver-Transmitter.

UDP User Datagram Protocol.

USB Universal Serial Bus.

Chapter 1

Introduction

1.1 Context

Nowadays, connected systems are widely used in our daily lives. We all have at least one connected device, such as a smart thermostat, an autonomous vacuum cleaner, or a connected watch. All these technologies are categorized under the term Internet of Things (IoT). IoT is a growing field that combines embedded computing, communication technologies, and automation to enhance comfort, efficiency, and safety across various domains, ranging from home automation to mobile robotics. However, these systems face significant challenges that they must address in order to ensure reliability and safety.

In dynamic environments, stability and responsiveness are critical challenges. The emphasis on stability in robotics and related fields arises from the fact that instability can rapidly result in loss of control and unsafe conditions due to its unpredictable nature. Instability is a phenomenon observed across a wide range of systems. Examples include a car skidding on ice, an aircraft entering a stall, or a nuclear reactor approaching a critical state. Fortunately, such problems occur only very rarely. Indeed, thanks to the application of control theories and sensor technologies, humans have developed effective strategies to avoid and even exploit such instabilities. Systems such as anti-lock braking in cars, stall prevention in aircraft, control rod mechanisms in reactors, and anti-seismic systems in buildings all demonstrate that the capability to restore balance is a fundamental aspect of modern engineering.

Dynamic self-balancing systems are a particularly good example of this principle. Just as humans rely on a combination of senses, such as vision, proprioception and the vestibular system, to maintain balance, artificial systems use sensors and control algorithms. However, the reliability of such systems depends heavily on the quality of the sensor data. In real-time embedded applications, sensor measurements are often affected by noise, latency and limited precision. This makes effective signal processing and sensor fusion crucial. The 'garbage in, garbage out' principle applies: control systems are only as reliable as the data they receive.

Sensor fusion is a widely used mechanism for overcoming these challenges. It involves combining information from multiple sensors to produce a more accurate and robust estimation of a system's state. The Kalman filter is a particularly good implementation of sensor fusion. Its principle is to combine noisy measurements with a predictive model to produce an optimal real-time estimation. This filtering technique is very well suited for embedded systems, where processing power is limited and sensor data is often unreliable. In this context, the Hera framework is a modular architecture that integrates a Kalman-based sensor fusion for real-time embedded and distributed applications. It is an Erlang-based open-source platform that runs on the GRiSP board. It combines Erlang's fault-tolerant concurrency with direct low-level hardware control through Pmod interfaces.

1.2 Objectives

The aim of this master's thesis is to design and build a functional prototype of a robot that acts as a butler. The robot must be able to move around with a full glass of wine on top without spilling any of its contents. The robot must also be equipped with an obstacle avoidance mechanism to prevent collisions with objects and people in its environment. The robot is developed as if intended for commercial production. This approach requires consideration of the technical design as well as factors such as cost, feasibility, scalability, and long-term reliability.

This project builds upon the previous work of François Goens and Cédric Ponsard, who previously developed a 30 cm two-wheeled self-balancing robot using the GRiSP embedded platform and the Hera sensor fusion framework. Their work demonstrated the feasibility of real-time stability control using Kalman filters, PID controllers, and an Erlang-based architecture. The objective here is to develop that prototype into a practical, real-world application: a self-stabilizing butler that is stronger and taller than its predecessor as shown in Figure 1.1. It must be able to carry heavier payloads at a suitable human-accessible height.

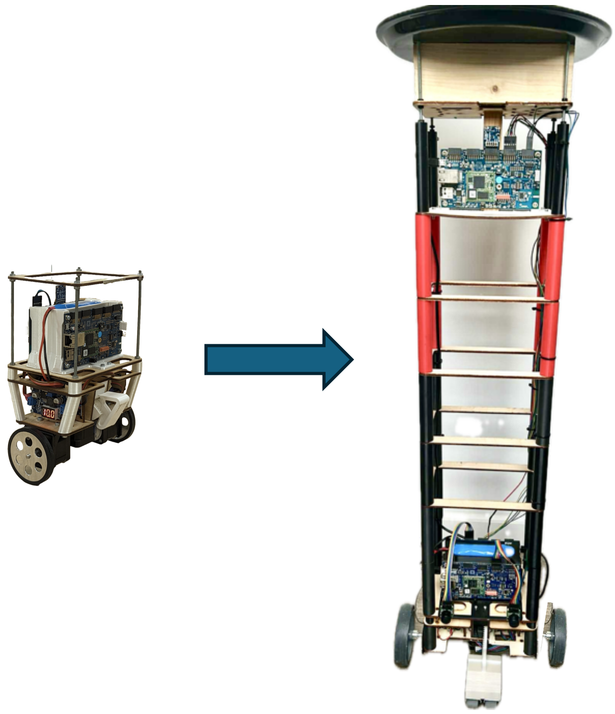


Figure 1.1: Evolution of the robot.

1.3 Contributions

The main contributions of this thesis are the complete development and implementation of a multi-level embedded control system for a self-balancing butler, including:

- A robot designed and built as a two-wheeled, self-balancing robot with a strong, high chassis to facilitate human interaction at an appropriate height.
- A mounting plate for carrying various objects, as well as an improved center of mass distribution, enabling it to effectively carry heavy payloads.
- Retractable support arms that can be inserted into its body, enabling the robot to operate in dynamic balancing mode while moving and to rest in a 'static' mode when stationary.
- A stabilization mechanism using two consecutive PID controllers. The tilt angle is calculated based on IMU data, which, once acquired, is subject to a Kalman filtering process to reduce noise from the sensor before entering the stability loop.

- An obstacle avoidance mechanism featuring three ultrasonic sonars, providing wide coverage across the angles of movement. The robot also exhibits an emergency strategy, where the algorithm calculates if it is getting too close to an obstacle.
- A synchronized multi-GRiSP communication system to exchange sonar data between boards. The distributed architecture employs three GRiSP boards, a server, and an ESP.
- An ESP used for remote control, connected to a user interface that allows users to control the robot from a distance. The robot is tested in domestic scenarios to evaluate its performance and identify its limitations in real-world environments.

This work demonstrates the transformation of an academic prototype into a multi-layered system that integrates multiple technologies. It serves as a proof of concept for future applications in smart home robotics. It integrates technologies not yet widely used in such domains. A demonstration video is available at: `https://youtu.be/wnIwVkGkAvY`. The full source code and schematics are available: `https://github.com/Artal44/Grisp_robot.git`.

1.4 Roadmap

This master's thesis is divided into eight chapters:

- **Chapter 1 - Introduction:** Presents the context, objectives, and contributions of the study.
- **Chapter 2 - Background:** Provides background information and explains the technologies and tools used in the research.
- **Chapter 3 - Overall design:** Introduces the hardware developments along with an explanation of its main components. This chapter also includes a diagram of the system.
- **Chapter 4 - System architecture:** Details the system architecture, explaining its different layers and the interactions between them.
- **Chapter 5 - Hardware implementation:** Focuses on the complete hardware implementation of the robot, providing a comprehensive explanation of each part of the robot, from the bottom to the top.

- **Chapter 6 - Software implementation:** Provides a summary of the software implementation and describes the architecture of the distributed system. It then goes into detail about each piece of software, including the GRiSP architecture, the LilyGO module and the user interface architecture.

- **Chapter 7 - Evaluation and discussion:** Presents an evaluation of the robot and outlines its limitations. It also provides an analysis of materials and costs and a discussion about the evaluation of this work.

- **Chapter 8 - Conclusion and future works**: Concludes the thesis and presents possible future works.

A comprehensive appendix includes multiple documents, tables, photos, and algorithms for further details on specific parts, and the complete source code is also provided.

Chapter 2

Resources and background

2.1 Erlang programming language

Erlang is a programming language used for building massively scalable, soft real-time systems [1]. Those systems require properties such as high availability, fault tolerance, distribution and hot code swapping (i.e., the ability to update parts of the system while it continues to run [2]). Erlang achieves these goals by using a lightweight actor-based concurrency model. The processes communicate via asynchronous message passing and are strongly isolated from one another. The term 'Erlang' is often used with Erlang/OTP, where OTP stands for Open Telecom Platform [3]. This includes the Erlang runtime system, a set of middleware libraries and a suite of ready-to-use design patterns such as `gen_server`, `supervisor` and `application`, which promote best practices for structuring reliable, concurrent systems.

A core strength of Erlang lies in its inherent fault tolerance where individual components may fail without compromising the stability of the overall system [4]. When supervision hierarchies are used, failed processes can be automatically restarted and restored to a defined stable state. This principle is called "let it crash" and is encapsulated in Erlang's philosophy. Additionally, Erlang supports hot code swapping which is essential for applications requiring continuous availability. Erlang is also particularly advantageous for concurrent programming. It offers a simple and efficient mechanism to spawn thousands of lightweight processes. This enables the creation of highly distributed systems with many components interacting concurrently and independently. These design principles contribute to Erlang's natural resilience to faults, while also supporting predictable and deterministic behavior. Erlang has been successfully deployed in large-scale production environments such as WhatsApp, RabbitMQ, and telecoms infrastructure.

To simplify the development of Erlang applications, the ecosystem provides `rebar3`, a modern build tool and package manager designed specifically for Erlang/OTP systems [5]. Rebar3 automates compilation, testing, and dependency management, and facilitates the creation of reproducible releases [6]. Rebar3 enforces standard Erlang/OTP project conventions, which promotes consistency and enhances accessibility, particularly for developers new to Erlang. Its flexible plugin architecture also allows it to be extended and adapted to other languages targeting the BEAM virtual machine.

Despite its strengths, Erlang is not particularly optimized for high-performance numerical computations as highlighted in Lylian Brunet's and Basile Couplet's master thesis [7]. In applications requiring high-frequency control loops or computationally intensive tasks, Erlang may introduce latency or limitations. However, such situations can be addressed by interfacing Erlang with native code through ports or Native Implemented Functions (NIFs). Those are programmed in C to offload performance-critical operations from the system.

2.2 GRiSP$_2$ board and platform

The GRiSP$_2$ board was created by Peer Stritzinger GmbH [8] . It is a modular prototyping platform specifically designed for embedded and distributed systems [9]. It provides support for running Erlang applications directly on microcontrollers, both in terms of hardware and software. As a result, there is no need for a traditional operating system. The device has built-in support for Ethernet and Wi-Fi, as well as a variety of digital and analog interfaces. Those functionalities make it ideal for real-time applications in control systems, robotics, and sensor networks.

Development is done entirely in Erlang using the GRiSP toolchain. Applications are cross-compiled and flashed to the microcontroller, which runs them using a customized version of the BEAM virtual machine. This setup allows developers to take advantage of Erlang/OTP's fault tolerance and message-passing concurrency within the constraints of real-time hardware. In contrast, traditional platforms such as the Raspberry Pi or ESP32 rely on general-purpose operating systems and typically require mixed-language development. In comparison, the GRiSP$_2$ offers a unified, low-latency environment ideal for embedded and distributed robotics. Another key strength of the GRiSP$_2$ lies in its Peripheral Module (Pmod) connectors, which allow direct integration of peripherals such as ultrasonic rangefinders, gyroscopes, and inertial measurement units (IMUs) without the need of microcontrollers. This modularity, combined with Erlang's concurrency model and the deterministic behavior of bare-metal execution, makes it highly adaptable for time-critical embedded systems.

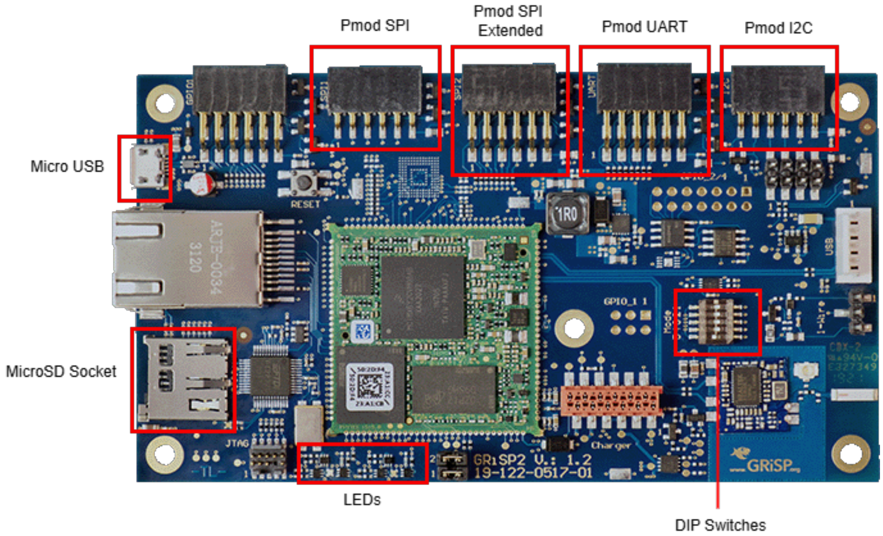


Figure 2.1: Annotated view of the $GRiSP_2$ circuit board, highlighting key specifications and components. Modified from [10].

2.2.1 Technical specifications of the board

Several of the $GRiSP_2$ board's hardware features are particularly useful for designing and implementing embedded robotic systems (see Figure 2.1). First, the microSD card socket supports standard microSD [10]. This socket is commonly used to store application code, log files, and runtime data, which enables autonomous operation of the board without external dependencies. Additionally, the board is equipped with two user-programmable RGB LEDs. These LEDs can be used to indicate connection status, system heartbeat, or errors, which makes them useful for debugging and runtime monitoring. Another important component is the set of five DIP switches that offer a way to configure hardware parameters for the board at startup.

In terms of connectivity, the $GRiSP_2$ board provides several I/O interfaces essential for communicating with external sensors and actuators. These include one Pmod connector compatible with Inter-Integrated Circuit (I^2C), one with Serial Peripheral Interface (SPI), and one with Universal Asynchronous Receiver-Transmitter (UART), among others. This allows Pmod peripherals such as the NAV or MAXSONAR to be connected directly, without additional microcontrollers or converters.

2.2.2 Pmod standards

2.2.2.1 Pmod NAV

The Pmod NAV is a sensor module that combines a 3-axis accelerometer, a 3-axis gyroscope, a 3-axis magnetometer, and a digital barometer, offering a complete 10 Degree of Freedom (DoF) sensor suite (Figure 2.2) [11]. The term "degree of freedom" refers in this context to the number of independent sensor measurements provided [12]. This combination of sensors allows for precise determination of the module's position, orientation, and motion state [11]. It can be used to assess the robot's movement, heading direction, and whether it is tilting, falling, or rotating. The SPI protocol is used by the Pmod NAV to communicate with the GRiSP$_2$ board. Data acquisition is done using the appropriate chip select signal to address each of the four onboard sensors. The accelerometer measures both the gravitational acceleration and any linear acceleration due to movement along the X, Y, and Z axes in 16-bit signed measurements. Since each axis is measured independently, the outputs are not affected by crosstalk or interference between axes. The gyroscope reports the angular velocity of the module.

Figure 2.2: Top view of the Pmod NAV sensor [13].

2.2.2.2 Pmod MAXSONAR

The Pmod MAXSONAR is an ultrasonic range finder that provides accurate measurements of the distance to nearby objects (Figure 2.3) [14]. It has an effective detection range of 15 cm to 648 cm. Range data is obtained by emitting thirteen ultrasonic beams at 42 kHz, each with different widths. The module requires approximately 250 ms to power up. It then performs an internal calibration over 50 ms, waits an additional 100 ms, and begins producing distance measurements at a rate of one reading every 50 ms. The communication with the GRiSP$_2$ board is possible using an UART interface that sends each measurement as a five-character ASCII message.

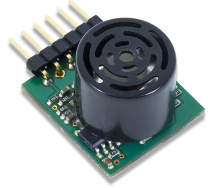


Figure 2.3: Top view of the Pmod MAXSONAR module used for ultrasonic distance measurement [14].

2.3 Communication protocols

A communication protocol is a set of rules and procedures that determine how the exchange of data between two or more devices or systems is done. These rules define how data is formatted, transmitted, received, and interpreted [15]. They play a crucial role in developing microcontroller-based systems by enabling the smooth interaction of various components within an embedded architecture. Among the most commonly used communication protocols in embedded systems are: Inter-Integrated Circuit (I^2C), Serial Peripheral Interface (SPI), Universal Asynchronous Receiver-Transmitter (UART). Each of these protocols has its own characteristics, advantages, and limitations.

2.3.1 I^2C protocol

Inter-Integrated Circuit (I^2C) is a widely used serial communication protocol in embedded systems and IoT devices [16]. I^2C follows a master-slave architecture, where a single master controls the clock and initiates communication with one or more slave devices [16]. It is a synchronous protocol, in which the timing of data exchange is coordinated by the shared clock provided by the master. However, I^2C has its limitations. It has a relatively low bandwidth which makes it unsuitable for high-throughput applications [17]. It is also designed for short-distance communication. Signal integrity may degrade over longer distances due to bus capacitance and electromagnetic noise. Additionally, large networks present challenges in terms of management due to potential address conflicts. Furthermore, I^2C uses single-ended signaling, which is more susceptible to interference than differential signaling. This reduces noise immunity in electrically noisy environments.

2.3.2 SPI protocol

The Serial Peripheral Interface (SPI) is a synchronous serial communication protocol designed for short-distance data exchange of a few centimeters. Longer SPI connections are possible by lowering the data rate, using better wires, and adding shielding [18]. It operates in full-duplex mode and follows a master-slave architecture, where a single master device can communicate with multiple slave devices [19]. One of the key advantages of SPI is its ability to transmit continuous data streams of arbitrary lengths easily. This enables high-speed communication that can reach several megabits per second. However, SPI also has some drawbacks. In fact, each slave device requires its own dedicated communication line with the master. As the number of slaves increases, the wiring becomes more complex, which can limit systems with constrained physical space [20].

2.3.3 UART Protocol

Universal Asynchronous Receiver-Transmitter (UART) is a communication protocol that enables data transmission between a microcontroller and peripheral devices asynchronously. This means that it does not use a clock to synchronize data exchange. UART offers several advantages. Notably, its simplicity of implementation and the fact that it does not require a clock make it well-suited to straightforward or low-bandwidth communication tasks. It also provides flexibility through configurable baud rates and data formats. Additionally, it is widely supported across microcontrollers and peripheral devices. This facilitates the integration into embedded systems. However, UART has notable limitations. Its maximum data rate is generally lower than that of SPI, and it does not support multiple devices on a shared bus like I^2C does. Furthermore, as it is asynchronous, the receiver must remain constantly synchronized, and communication is more susceptible to timing errors, particularly over longer distances where signal degradation can occur [20].

2.4 Hera framework

Internet of Things (IoT) systems are composed of a large number of small, low-power, and wireless devices. They operate at the logical edge of the network, farthest from centralized cloud services [21]. These devices are embedded within the physical environment and are capable of performing computation and coordination directly where the data is produced. The IoT represents a fast-growing segment of the Internet, with constantly evolving computational capabilities and distributed intelligence. In such edge-centric and resource-constrained environments, it becomes difficult to interpret noisy, partial, and asynchronous sensor data. This is where sensor fusion becomes essential. Sensor fusion is the process of combining data from multiple heterogeneous sensors to produce a unified and coherent view of the real world. By integrating redundant or complementary data, sensor fusion improves the accuracy, reliability, and richness of the information. This is especially important for embedded and robotic systems operating under uncertainty.

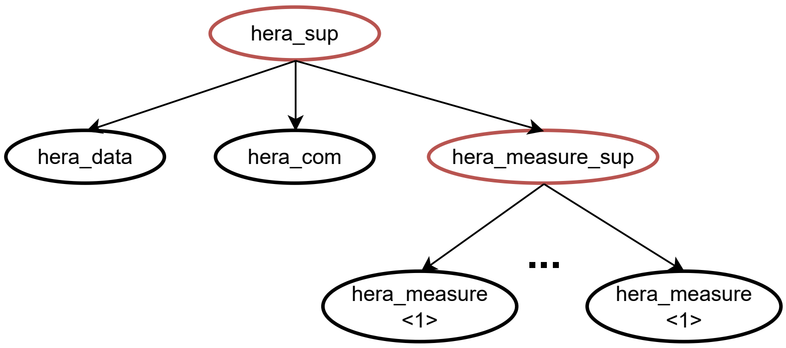


Figure 2.4: Diagram illustrating the Hera framework supervision tree [21].

Hera is a Kalman filter-based sensor fusion framework for Erlang. Lightweight and robust, it was designed for the GRiSP platform to support asynchronous and dynamic sensor measurements. Hera provides high-level abstractions that simplify the development of embedded applications by abstracting away low-level concerns such as network setup, communication, fault tolerance, and hardware access. It follows a supervision tree architecture (Figure 2.4) with two supervisors: `hera_sup` and `hera_measure_sup`. The `hera_sup` supervisor manages three independent processes: `hera_data`, `hera_com`, and `hera_measure_sup`.

`hera_data` is a `gen_server` (i.e., implementation of a client/server model) used for storing measurement data from the sensors. `hera_com` handles communication over the network between different nodes and uses User Datagram Protocol (UDP) for fast but unreliable data sharing. `hera_measure_sup`, the second supervisor, oversees the `hera_measure` processes. These are generic measurement processes that often interface with sensors to interact with the physical world. As a result, they are more likely to fail and thus require supervision. These processes are the only files a user must provide. They are easy to use: the user only needs to implement an `init(Args)` and a `measure(State)` function to perform measurements.

2.4.1 Kalman filter

In applications involving multiple sensors, filters that perform sensor fusion are essential for enhancing measurement accuracy and reducing noise [22]. These filters combine data from different sources to provide a more reliable estimate of the system's state. Several filtering techniques exist for this purpose, including the complementary filter and the Kalman filter. In this work, the focus is on the Kalman filter due to its recursive nature and statistical optimality in linear Gaussian systems. Its effectiveness has already been demonstrated in similar robotics applications, including the master's thesis of Cédric Ponsard and François Goens [22].

The Kalman filter is a powerful algorithm that relies on a mathematical model of the system to predict and correct noisy sensor measurements [23]. It provides accurate and responsive estimations by continuously updating the state of a system using a combination of prediction and observation. One of its key strengths lies in its ability to dynamically adjust the confidence given to each measurement. It makes it an ideal tool for sensor fusion.

Each iteration of the Kalman filter is composed of two main steps:

- **Prediction**: The system's next state is estimated based on the physical model and control inputs.
- **Update**: The prediction is corrected using actual sensor measurements, weighted by their estimated uncertainties.

At each step, the filter manages two key entities:

- x: the vector that represents the system's state,
- P: the covariance matrix representing the uncertainty in the estimated state x. In more intuitive terms, it quantifies the expected error in the state estimation and serves as a measure of confidence in the accuracy of x.

Prediction step:

$$\hat{x}_k = F_k x_{k-1} \tag{2.1}$$

$$\hat{P}_k = F_k P_{k-1} F_k^\top + Q_k \tag{2.2}$$

Update step:

$$K_k = \hat{P}_k H_k^\top (H_k \hat{P}_k H_k^\top + R_k)^{-1} \tag{2.3}$$

$$x_k = \hat{x}_k + K_k(z_k - H_k \hat{x}_k) \tag{2.4}$$

$$P_k = (I - K_k H_k)\hat{P}_k \tag{2.5}$$

Where:

- F: the state transition model, which describes how the state naturally evolves over time.
- Q: the process noise covariance, representing the imperfections of the predictive model.
- H: the observation matrix, mapping states to expected measurements.
- R: the measurement noise covariance, describing the uncertainty in sensor readings.
- z: the vector of the actual sensor measurement.
- K: the Kalman gain, computed to minimize the posterior error covariance.

One of the mathematical foundations of the Kalman filter is the combination of two Gaussian distributions: the predicted state and the measured state [22]. The product of these distributions is a new Gaussian with lower variance. This results in a more accurate estimation with reduced noise. This is ideal for sensor fusion scenarios. As illustrated in Figure 2.5, this fusion process is key to achieving high precision.

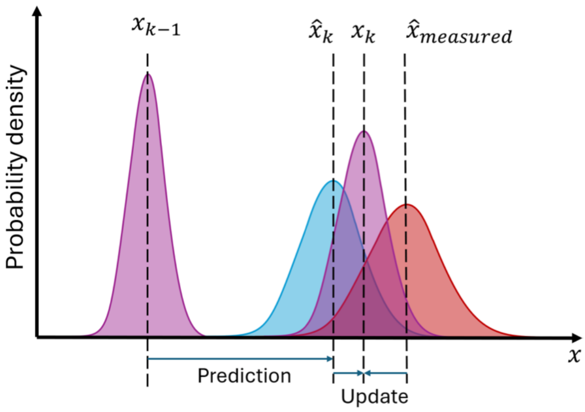


Figure 2.5: The combination of Gaussians in Kalman filtering, showing the prediction and update phases as well as the result of the product of the Gaussians [22].

2.4.2 Extended Kalman Filter (EKF)

The standard Kalman filter is optimal for systems that are linear and can be modeled using linear equations for both the state evolution and measurement processes. However, many real-world systems, including robots, exhibit non-linear dynamics or sensor relationships. In such cases, the **Extended Kalman Filter (EKF)** is used. The EKF extends the original Kalman filter to handle non-linear systems by linearizing the system around the current estimate using first-order Taylor expansions [24]. This allows the application of the Kalman filter logic to non-linear models by approximating them locally with linear ones. The main difference with the standard Kalman filter lies in the fact that the state transition and observation models are no longer required to be linear functions of the state. Instead, they can be any differentiable non-linear functions. Specifically, a function f is used to compute the predicted state from the previous estimate, and a function h is used to compute the expected measurement from the predicted state.

However, unlike linear systems, these non-linear functions f and h cannot be applied directly to the covariance matrix. The Extended Kalman Filter (EKF) relies on linear approximations of these functions, which are obtained through their Jacobians (i.e., matrices of partial derivatives evaluated at the current estimate).

- $F_k = \frac{\partial f}{\partial x}\Big|_{x=\hat{x}_{k-1}}$: Jacobian of the state transition function.
- $H_k = \frac{\partial h}{\partial x}\Big|_{x=\hat{x}_{k|k-1}}$: Jacobian of the observation function.

Consequently, the Jacobian of the state transition function replaces the constant transition matrix, and similarly, the Jacobian of the observation function is used in place of the standard observation matrix, as can be seen in the equations above.

2.5 Control systems

2.5.1 Control loop

The selection and design of the control loop are fundamental elements in the design of a robot. In industrial control, a loop consists of a sensor, a controller, and a Final Control Element (FCE) that work together to drive a process toward a desired Set Point (SP) [25]. The Control Variable (CV) is the physical quantity that is regulated. Sensors produce measurements (i.e., the Process Variable (PV)), which are used by the controller to compute the actuation sent to the FCE. The loop continuously adjusts the actuation so that the CV tracks the SP. Two common classes of control exist:

- **Open-loop control** has no feedback. The control action is computed without using the measured PV. It typically relies on timers, precomputed profiles, or other signals independent of the process measurement.

- **Closed-loop (feedback) control** uses both the desired SP and the measured PV to compute the control action. PV is an estimate of the CV from the sensors' measurements. The controller computes the error $e = SP - PV$ and outputs a command that adjusts the FCE, closing the loop so that the CV tracks the SP. This mechanism is illustrated in Figure 2.6, where the set-point and the PV are compared to compute the error and adapt the actuation accordingly.

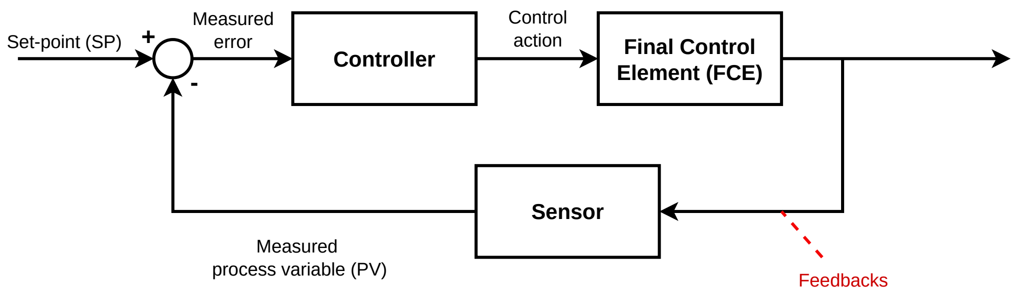


Figure 2.6: Generic schematic representation of a closed-loop control system with a feedback.

2.5.2 PID control strategy

Proportional-Integral-Derivative (PID) controllers are extensively used across various fields, particularly in industry, due to their relative simplicity, versatility, and robustness [26, 27]. A PID controller combines three fundamental control actions, proportional, integral, and derivative, each associated with a gain coefficient. These coefficients can be tuned to achieve the desired response of the control variable. In certain applications, simplified versions using only two of the three terms, such as Proportional-Integral (PI) or Proportional-Derivative (PD) controllers, are also employed. The PID controller operates based on the measured error, e defined as the difference between the desired set-point and the measured output (Figure 2.7). The Control Variable (CV) u (i.e., the system input in Figure 2.6) is then computed by applying the three terms to this error:

$$u = K_p \cdot e + K_i \cdot \int_0^t e\,dt + K_d \cdot \frac{de}{dt} \tag{2.6}$$

Where K_p, K_i, and K_d are the proportional, integral, and derivative gains, respectively. The proportional term adjusts the controller output proportionally to the current error. Increasing the proportional gain K_p increases the speed of the control response. However, if K_p is too large, it may cause overshoot and oscillations around the desired set-point, or even lead to unstable behavior. The integral term reacts to the error accumulated over time; this enables it to correct persistent errors and eliminate the steady-state error. The derivative term responds to the rate of change of the error, effectively providing a predictive action based on the future behaviour of the system. Its contribution to the control output is proportional to how rapidly the error varies; a higher rate of change results in a stronger corrective action. However, the derivative term is highly sensitive to noise, which manifests as rapid fluctuations caused by sensors or external disturbances rather than genuine changes in the measured output. Consequently, an excessively large derivative gain K_d can amplify these noise effects, leading to undesirable oscillations. For this reason, the value of K_d is typically kept small. Additionally, the derivative action helps to dampen the system response, reducing overshoot and slowing the correction as the measured output approaches the set-point.

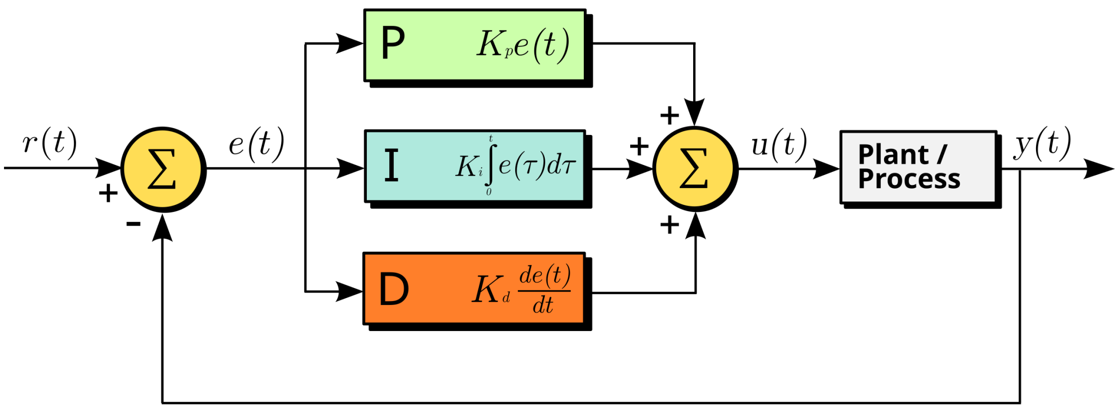


Figure 2.7: Block diagram of a Proportional-Integral-Derivative (PID) controller. The error between the set-point and the measured output is used to compute a control signal composed of three terms: proportional to the error, integral of the error over time, and derivative of the error. This control variable is then applied to the system to minimize the error [26].

2.6 Auto-stabilized two-wheeled robot

Cédric Ponsard and François Goens's master's thesis focuses on building a two-wheeled self-balancing robot (Figure 2.8) [22]. The robot measures approximately 32cm in height, 20cm in length, and 12cm in width. It weighs approximately 1.5kg. It is compact, with a low center of gravity, yet remains large enough to integrate all the necessary sensors and electronics. A short video of their robot can be found at: `https://youtu.be/-GYXGzXmlVE`. The main objective of their master's thesis was to design a stability engine capable of controlling unstable systems and to make it easily deployable across different devices. They implemented this engine specifically on the GRiSP board and then tested and validated it by building a two-wheeled inverted pendulum robot. To further increase the autonomy of the robot, they also designed a lifting mechanism, which allows the robot to recover from a fall and return to its self-balancing state without human intervention. This mechanism, coupled with a robust control strategy, enables the robot to withstand external disturbances and remain functional even in the event of a complete crash.

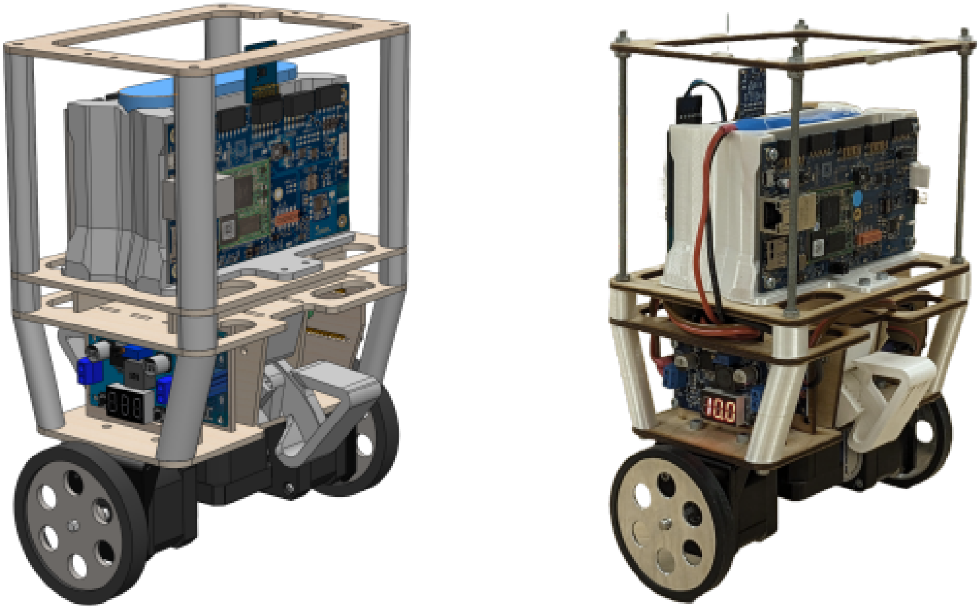


Figure 2.8: Two-wheeled self-balancing robot designed by Cédric Ponsard and François Goens. The SolidWorks 3D model is represented on the left, while the final design is on the right [22].

The system uses the Hera framework as the interface for asynchronous sensor fusion and relies heavily on sensor data provided by the Pmod NAV module. The IMU is the primary source for real-time estimation of the robot's tilt and angular velocity. Two filtering methods are implemented to preprocess this data: a complementary filter and a Kalman filter. The complementary filter combines accelerometer and gyroscope data with fixed gains to provide a lightweight solution, whereas the Kalman filter uses a dynamic weighting approach that takes sensor noise and system uncertainty into account. These two filters are implemented to enable a performance comparison. Finally, it turns out that the Kalman filter is more robust and accurate. This is particularly the case when dealing with sensor noise and transient disturbances. This results in better state estimation and smoother control. However, the Kalman filter has slightly higher computational complexity.

Cédric and François implement a Kalman filter based on a simple 2D state model that focuses on the robot's tilt angle and angular velocity. This basic Kalman filter dynamically adjusts the weighting of the gyroscope and accelerometer readings with each iteration. Its simplicity means that it can run efficiently on the GRiSP board. The only issue is that it cannot handle situations involving strong linear accelerations or gyroscope saturation, as it does not take into account the full dynamics of the robot or its control inputs. However, this is sufficient for a small robot like theirs.

The control strategy is implemented as a cascade control system, with two dedicated controllers:

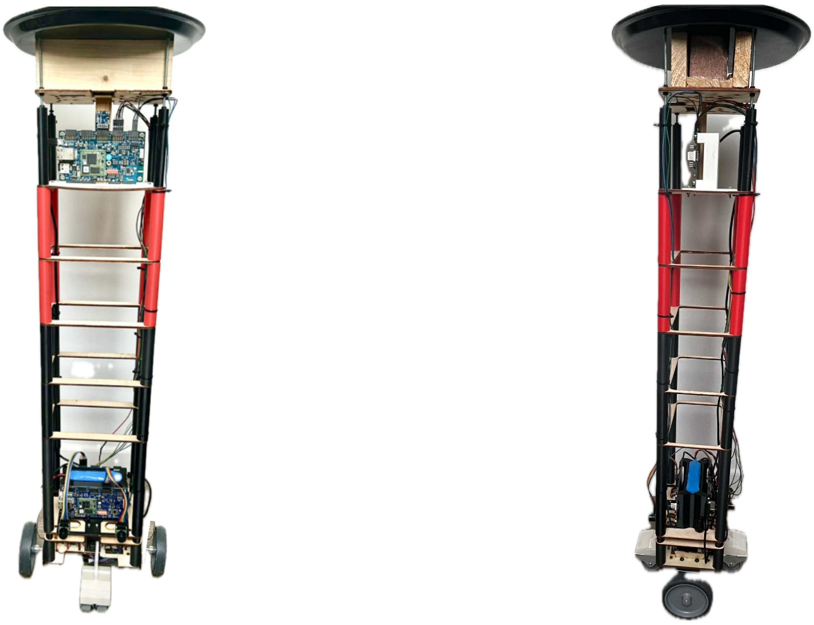
- A Proportional-Integral (PI) controller is responsible for computing the target tilt angle based on the difference between the reference speed and the measured wheel speed. This controller ensures that the robot reaches and maintains its desired speed.
- A Proportional-Derivative (PD) controller then compares the reference tilt angle (i.e., output of the PI) with the measured tilt angle (i.e., output of the Kalman filter). Its output directly sets the wheel acceleration command.

Despite its robustness, the system has a few limitations. For example, the Kalman filter requires careful tuning of the process and measurement noise parameters. This can affect its performance if the characteristics of the sensors change. The IMU of the Pmod NAV is noisy and less accurate under certain conditions. In fact, it is sensitive to vibration and magnetic disturbances, which can cause drift in heading estimation. Additionally, the lifting mechanism increases the robot's weight and mechanical complexity.

Chapter 3

Overall design

This chapter introduces the self-balancing butler robot (Figure 3.1) as a complete system integrating mechanical design, embedded electronics, and complex software architecture. Its purpose is to outline the overall structure and the key design choices that guided its development. It also provides the foundation for the next chapter, which details the system architecture, hardware, and software.



(a) Front view of the butler robot. (b) Side view of the butler robot.

Figure 3.1: Front and side views of the butler robot.

3.1 Hardware overview

The self-balancing butler is designed to operate in indoor environments with flat, even floors and without significant surface irregularities. The robot measures 95 cm in height and 18.5 cm in width. The robot features a tall, narrow form with two wheels for balancing, retractable arms for different operating modes, a central frame housing the electronics, and a flat top surface to carry objects. The robot can carry different types of objects, such as phones, bottles, or glasses, on its upper platform. The design ensures that the payload remains stable during transport, even when the robot is in motion.

The self-balancing butler is able to move forward, backward, and turn in both directions. These movements are controlled through simple user commands provided via a user interface. An emergency stop button is also implemented for safety, allowing the power to be cut immediately if needed. The user can switch between two operating modes: static and dynamic. The static mode is used when the robot does not move; the arms are deployed so that the robot rests on four wheels and no longer needs to actively maintain balance. The dynamic mode is used for movement and maneuverability; the robot operates in balancing mode on two wheels and can move in all directions.

To prevent collisions, the robot is equipped with an obstacle-avoidance system. It uses sonars to detect objects in its path and automatically brakes or turn to avoid them. This feature ensures safe navigation in indoor environments and helps protect both the robot and surrounding objects.

3.2 Objectives and specifications

The capabilities described in the previous section can be translated into specific functional requirements and design constraints. These have been formalized into measurable objectives and technical specifications, which are presented in Table 3.1.

Table 3.1: Specifications of the self-balancing butler. F: Function, FR: Functional Requirement, C: Constraint, CR: Constraint Requirement. Adapted from [22].

Functions	
F1	High table on two wheels using a dynamic balancing algorithm implemented with Hera on GRiSP.
F2	Multi-GRiSP communication system enabling data sharing between multiple GRiSP boards for obstacle avoidance.
Functional Requirements	
FR1.1	Mechanically unstable device.
FR1.2	Designed for domestic use.
FR1.3	Use of actuation to stabilize the device.
FR1.4	Dynamic and static stabilization.
FR1.5	Autonomous: no need for an external power supply.
FR1.6	Must support a minimum payload of 500 grams.
Constraints	
C1	Use of the GRiSP 2 board.
C2	Easy to prototype and replicate.
C3	Safe to use.
Constraint Requirements	
CR1.1	5 V power supply for GRiSP.
CR2.1	Use of off-the-shelf components, 3D printing, laser cutting and a custom PCB.
CR2.2	Keep costs below a reasonable threshold.
CR3.1	Include an emergency stop.
CR3.2	Operate with voltages lower than 20 V.
CR3.3	Use non-flammable batteries.
CR3.4	Total weight less than 8 kg.
CR3.5	Height between 70 and 100 cm.
CR3.6	Width small enough to pass through typical home doors.

3.3 Overall system diagram

Figure 3.2 presents a high-level representation of the overall architecture, which is composed of several interconnected levels, each fulfilling a specific role within the system. It is built as a multi-level stack where a low-level block handles devices and data acquisition; intermediate layers perform data filtering, state estimation, and control; and an upper layer orchestrates decision-making, coordination, and communication.

The system architecture of the robot is made of five parts: the executive loop, the stability loop, the obstacle-avoidance block, the sensor-fusion block, and the sensors block. The executive loop supervises higher-level behaviors and manages transitions between the robot's operational modes. Specifically, it monitors flags and authorization signals, handles the switch between dynamic and static modes, and manages entry into emergency mode when necessary. The stability loop is a low-level control loop responsible for dynamically stabilizing the robot. It computes the appropriate actuation commands through the stability controllers and drives the actuators in real time. The obstacle-avoidance block handles communication between the GRiSP boards in order to retrieve distance measurements from the sonar sensors. It then processes these measurements through its obstacle-avoidance mechanism to avoid nearby obstacles. The sensor-fusion block is a critical part of the architecture. It collects data from the IMU and combines it with motion data from the robot to estimate its precise state. This fusion process filters out noise and improves the accuracy of orientation and velocity measurements, which are then given as inputs to the control loops. Finally, the sensors block manages the raw data acquisition from the various sensors. These sensors are either triggered by the executive loop or activated at fixed intervals based on a predefined timeout, ensuring regular and consistent measurements.

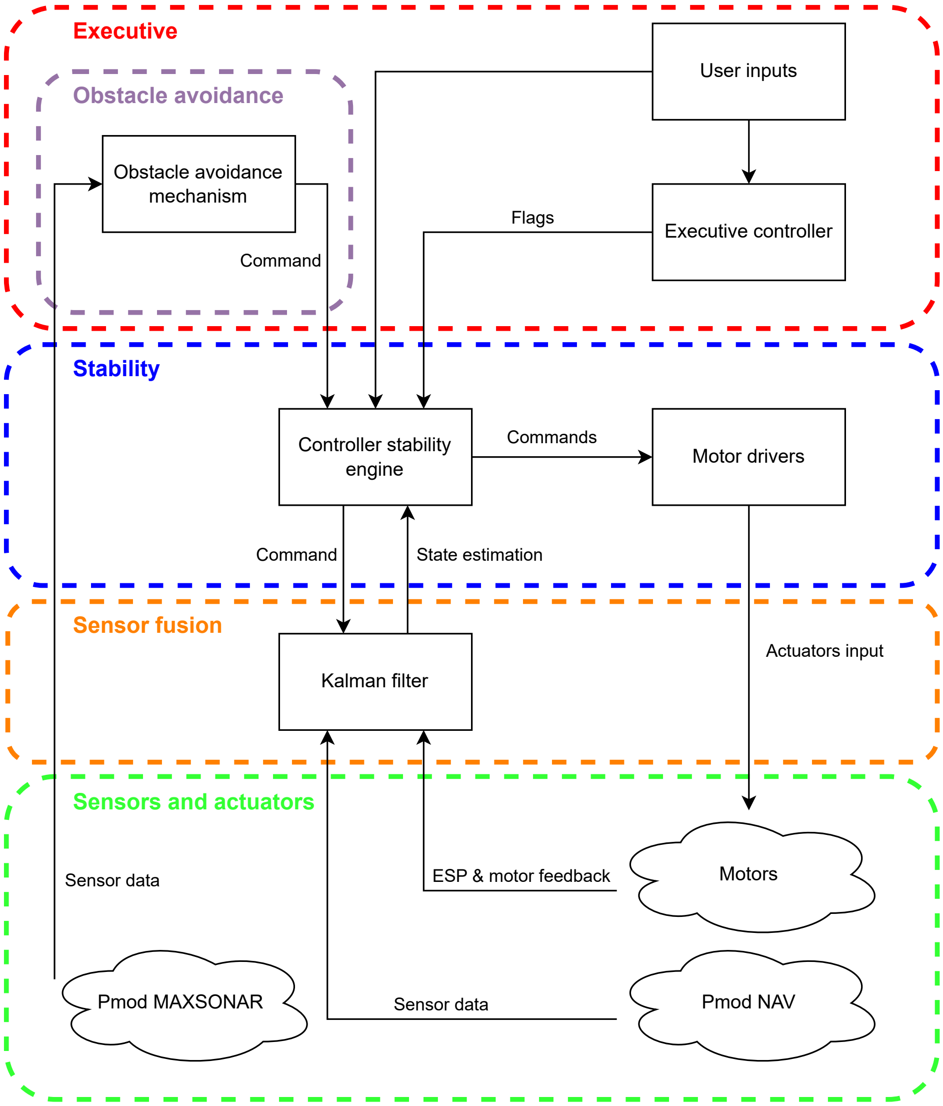


Figure 3.2: High-level architecture of the self-balancing table.

Chapter 4

Multi-level system architecture

This chapter details the multi-level system architecture of the butler robot by describing how its different functional blocks work and interact with each other (Figure 4.1). It starts with the sensor fusion block, which receives raw measurements from the sensors and then filters and transforms them. Then it sends the measured pitch angle to the controller. The controller then adjusts the pitch angle to follow the desired reference input to maintain stability during a dynamic phase. In parallel, the obstacle avoidance block monitors the environment using sonar sensors to detect potential collisions and adjusts commands if necessary. The output of the controller is then sent to the motor drivers, which convert these commands into low-level signals for the motors, allowing the robot to move, stay balanced, and avoid obstacles. Above all these blocks, the executive loop serves as the high-level control layer. It manages transitions between different operating modes using a Finite State Machine (FSM), based on user inputs and motor feedback.

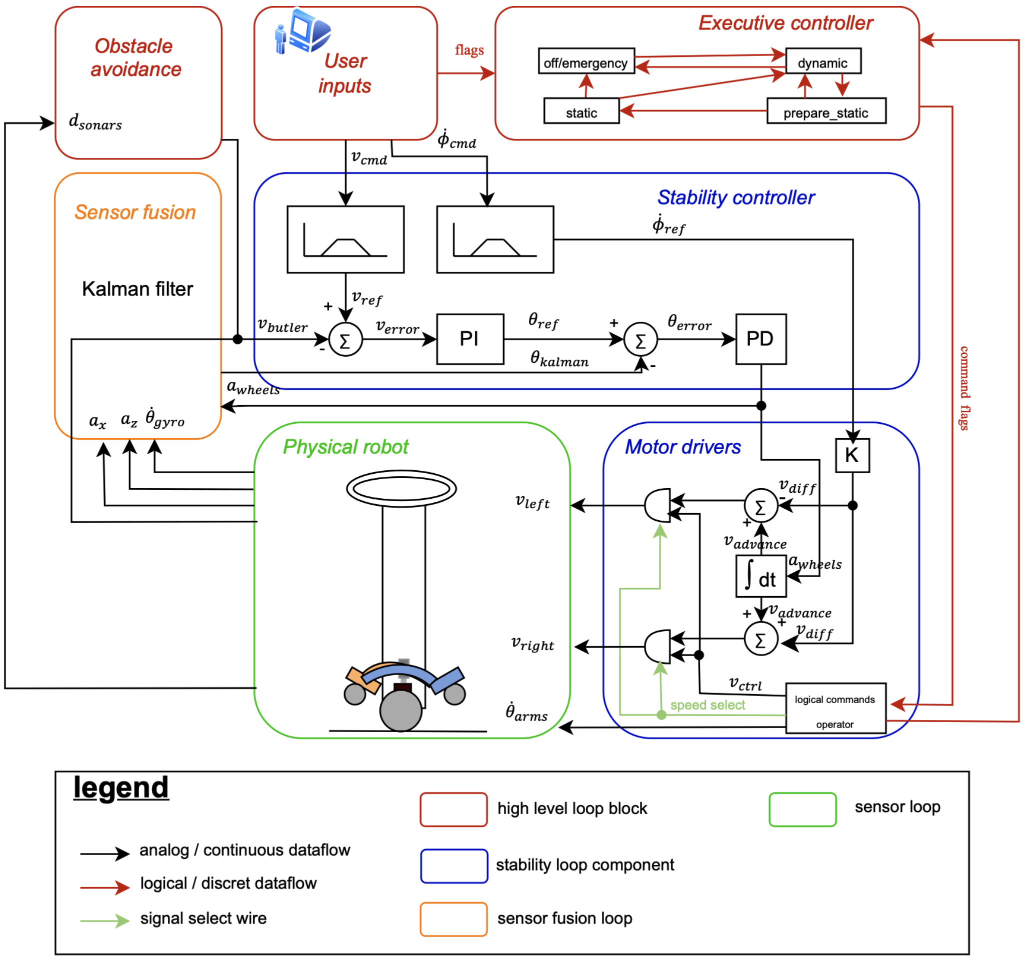


Figure 4.1: More detailed system diagram of the architecture, showing the internal workings of the blocks, their inputs and their outputs. Based on the design from the previous master's thesis [22].

4.1 Sensor fusion : Kalman filter

The sensor fusion unit is responsible for filtering and interpreting the raw data collected from the sensors before it can be used by the control system. In this master's thesis, the sensor fusion focuses on data from the inertial measurement unit (IMU), which include the accelerations from the accelerometer and the angular velocity from the gyroscope. These data are used to estimate the pitch angle (θ) of the robot. Sensor fusion is necessary because raw sensor data cannot be used directly for two main reasons. First, sensor readings are often affected by noise and interference, which can vary depending on the sensor type and operating environment. Therefore, filtering is necessary to improve data quality. Then, the physical quantities measured by the sensors, such as acceleration, are not always directly representative of the variable of interest (i.e., the pitch angle). A transformation is thus required to derive the desired information. To accomplish both of these tasks, the sensor fusion step uses an Extended Kalman Filter (EKF).

4.1.1 Physical modeling

The Kalman filter performs predictions based on a mathematical model. Therefore, it is essential to first develop a physical model of the robot based on a set of hypotheses. This allows us to derive a mathematical representation of the robot's behavior. The schematic representation of the physical model, along with the associated modeling assumptions, is presented in Appendix F and serves as the foundation for the state-space formulation used in the Kalman filter.

4.1.1.1 Movement equations

From this physical model, the Newton-Euler equations were derived. The full development is provided in Appendix G. The resulting equation of motion, which will be used in the EKF, is given below:

$$\ddot{x}\cos\theta + \left(h + \frac{J}{mh}\right)\ddot{\theta} = g\sin\theta \tag{4.1}$$

Table 4.1: Description of variables and constants in the equation of motion.

Symbol	**Description**
$\ddot{x}$	Linear acceleration of the robot along the x-axis [m/s^2]
θ	Tilt angle of the robot relative to the vertical axis [rad]
$\ddot{\theta}$	Angular acceleration of the robot [rad/s^2]
h	Distance between wheel axis and center of gravity [m]
J	Moment of inertia of the robot [kg · m^2]
m	Mass of the robot [kg]
g	Gravitational acceleration [m/s^2]

4.1.2 Advanced "digital twin" model for the Extended Kalman Filter

In François Goens and Cédric Ponsard's master's thesis, the Kalman filter relied on a simplified model where most of the physics of the robot is ignored [22]. While this simple model was sufficiently accurate for their robot, it proved inadequate for the butler. To improve estimation accuracy, an advanced digital twin model was used in this thesis based on the physical equation of motion of the robot (Equation 4.1). A digital twin is a model of a system's physics running in real time. The resulting nonlinear state model allows for more accurate predictions by integrating the real dynamics of the robot into the Extended Kalman Filter (EKF) framework. To construct this EKF, the following sections define the vectors and matrices necessary for the prediction and update steps.

4.1.2.1 State and input definition

The robot's state vector is defined as:

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$$

where θ is the pitch angle of the robot with respect to the vertical axis, and $\dot{\theta}$ is the angular velocity.

4.1.2.2 Non-linear dynamic model

The nonlinear motion model is derived from the equation 4.1 of motion:

$$\ddot{x}\cos\theta + \left(h + \frac{J}{mh}\right)\ddot{\theta} = g\sin\theta$$

Solving for $\ddot{\theta}$:

$$\ddot{\theta} = \frac{g\sin\theta - u\cos\theta}{h + \frac{J}{mh}} = \frac{g\sin\theta - u\cos\theta}{c} \quad \text{with } c = h + \frac{J}{mh}$$

The linear acceleration of the robot is simplified along the x-axis by assuming that the acceleration of the wheel command input u directly corresponds to this acceleration :

$$u = \ddot{x}$$

Using Euler integration with timestep Δt, the state update equations are:

$$\theta_k = \theta_{k-1} + \dot{\theta}_{k-1} \cdot \Delta t \tag{4.2}$$

$$\dot{\theta}_k = \dot{\theta}_{k-1} + \ddot{\theta}_{k-1} \cdot \Delta t \tag{4.3}$$

The nonlinear state transition function becomes:

$$f(x,u) = \begin{bmatrix} \theta + \dot{\theta} \cdot \Delta t \\ \dot{\theta} + \left(\frac{g \sin\theta - u \cos\theta}{c}\right) \cdot \Delta t \end{bmatrix}$$

4.1.2.3 State transition Jacobian

To use the EKF, a linearization $f(x,u)$ is performed by computing its Jacobian with respect to the state x:

$$J_f = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial \theta} & \frac{\partial f_1}{\partial \dot{\theta}} \\ \frac{\partial f_2}{\partial \theta} & \frac{\partial f_2}{\partial \dot{\theta}} \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ \left(\frac{g \cos\theta + u \sin\theta}{c}\right) \Delta t & 1 \end{bmatrix}$$

This Jacobian is used as the state transition matrix F_k in the EKF equations.

4.1.2.4 Measurement model

The measurement vector consists of:

$$z = \begin{bmatrix} a_x \\ a_z \\ \dot{\theta} \end{bmatrix}$$

where a_x and a_z are the accelerations measured along the robot's vertical axis and $\dot{\theta}$ is the angular velocity measured by the gyroscope.

4.1.2.5 The observation model

To use EKF, the observation model should map the accelerometer inputs. For that a nonlinear observation function $h(x,u)$ is defined that predicts what the IMU would measure given the current state and input:

$$h(x,u) = \begin{bmatrix} \hat{a}_x \\ \hat{a}_z \\ \hat{\omega}_y \end{bmatrix}$$

The acceleration of any point on the robot's vertical axis $\hat{X}_3$, is a combination of four different phenomena (as derived from Appendix G and Figure F.1):

- Lateral acceleration: $\ddot{x}\hat{I}_1 = u\cos(\theta)\hat{X}_1 + u\sin(\theta)\hat{X}_3$
- Angular acceleration effect: $\ddot{\theta}r\hat{X}_1 = \left(\frac{g}{c}\sin(\theta) - \frac{u}{c}\cos(\theta)\right)r\hat{X}_1$
- Centripetal acceleration effect: $-\dot{\theta}^2 r\hat{X}_3$
- Gravitational acceleration: $-g\hat{I}_3 = g\sin(\theta)\hat{X}_1 - g\cos(\theta)\hat{X}_3$

Where r is the distance between the wheel axis and the sensors mounted on the top of the robot. The total acceleration perceived by the sensor axis is the sum of the preceding phenomena:

$$\hat{a}_x = u\cos\theta + \left(\frac{g\sin\theta - u\cos\theta}{c}\right)r + g\sin\theta$$

$$\hat{a}_z = u\sin\theta - \dot{\theta}^2 r - g\cos\theta$$

$$\hat{\dot{\theta}} = \dot{\theta}$$

The observation function becomes:

$$h(x,u) = \begin{bmatrix} u\cos\theta + \left(\frac{g\sin\theta - u\cos\theta}{c}\right)r + g\sin\theta \\ u\sin\theta - \dot{\theta}^2 r - g\cos\theta \\ \dot{\theta} \end{bmatrix}$$

4.1.2.6 Observation Jacobian

Finally, the Jacobian of $h(x,u)$ with respect to the state x:

$$J_h = \frac{\partial h}{\partial x} = \begin{bmatrix} \left(-u\sin\theta + \frac{g\cos\theta + u\sin\theta}{c}r + g\cos\theta\right) & 0 \\ u\cos\theta + g\sin\theta & -2\dot{\theta}r \\ 0 & 1 \end{bmatrix}$$

This matrix is used as the observation model Jacobian H_k in the EKF update step.

4.1.2.7 R : The measurement noise covariance

The covariance matrices for the measurement (R) and process noise (Q) are based on sensor datasheets, general rules of thumb, and have been further adjusted through empirical tuning.

$$R = \begin{bmatrix} 3.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 3.0 \times 10^{-6} \end{bmatrix}$$

The measurement noise covariance matrix R is defined as a 3×3 diagonal matrix, corresponding to the three components of the measurement vector: the horizontal and vertical accelerations (a_x, a_z), and the angular velocity $\dot{\theta}$. The first two diagonal elements, associated with the accelerometer, have relatively large values, reflecting the high noise levels typically observed in acceleration measurements. In contrast, the much smaller value of $R_{3,3}$ indicates greater confidence in the gyroscope readings, as gyroscopic data is generally more stable and accurate [11].

4.1.2.8 Q : The process noise covariance

$$Q = \begin{bmatrix} 1.0 \times 10^{-6} & 0.0 \\ 0.0 & 2.5 \end{bmatrix}$$

The magnitude of the entries in the process noise matrix Q reflects the relative confidence in different components of the prediction model. The low value of $Q_{1,1}$ indicates strong trust in the Euler integration of angular velocity over time. On the other hand, a higher variance in $Q_{2,2}$ acknowledges the uncertainty introduced by using a physical model to estimate angular acceleration. Since this model may not perfectly capture all real-world effects such as unmodeled forces, parameter inaccuracies, it is reasonable to assign a larger uncertainty to this component.

4.2 Stability loop

The stability loop is composed of two main components: the controller block and the motor driver block. Its purpose is to stabilize the robot whether it is stationary, moving forward or backward. The overall structure of the stability loop was originally designed in a previous master's thesis [22]. However the real challenge in this work was to retune this control to fit it to the robot, which is higher and inherently more unstable.

4.2.1 Controllers

Once the pitch angle has been estimated from the sensor fusion, the controller block is responsible for computing the corrective commands that maintain the robot in its stable position. The controller is designed to make the robot follow a given angle reference by producing appropriate accelerations at the wheels. A PD controller is sufficient to do that (Figure 4.2). However, this approach alone is problematic when the reference angle does not correspond to the actual equilibrium angle of the system. In such cases, a persistent force imbalance arises, which would require a constant acceleration to maintain position. This is not feasible for the motors as they would saturate in speed.

To address this, the control strategy imposes the reference angle as the equilibrium angle, the angle at which all forces acting on the robot cancel each other out. Since this angle depends on dynamic factors such as speed and external forces. Thus, it cannot be predefined. Instead, it is continuously estimated and updated by a dedicated controller. This controller dynamically computes the equilibrium angle based on the robot's motion, allowing the system to adapt as the robot moves away from or returns to its balance point.

For this dedicated controller, the robot's speed command is used as input to estimate the equilibrium angle. Speed is a reliable indicator of the robot's equilibrium position. When the robot accelerates and reaches a high speed in a given direction, the equilibrium angle shifts in the same direction. This shift occurs because the robot must lean forward (or backward) to counteract the inertial forces generated by the acceleration and maintain stability. Speed control is also good for making the robot move with the desired speed and preventing excessive motion that could exceed hardware capabilities. To generate this equilibrium angle, a PI controller is used (Figure 4.2). The proportional term acts directly on the speed error, providing a fast correction that tilts the robot appropriately to match the desired speed. The integral term is linked to the integral of the speed and therefore linked to a distance. In this way, stabilization takes place over the distance. This results in a stabilizing behavior similar to a marble settling at the bottom of a bowl.

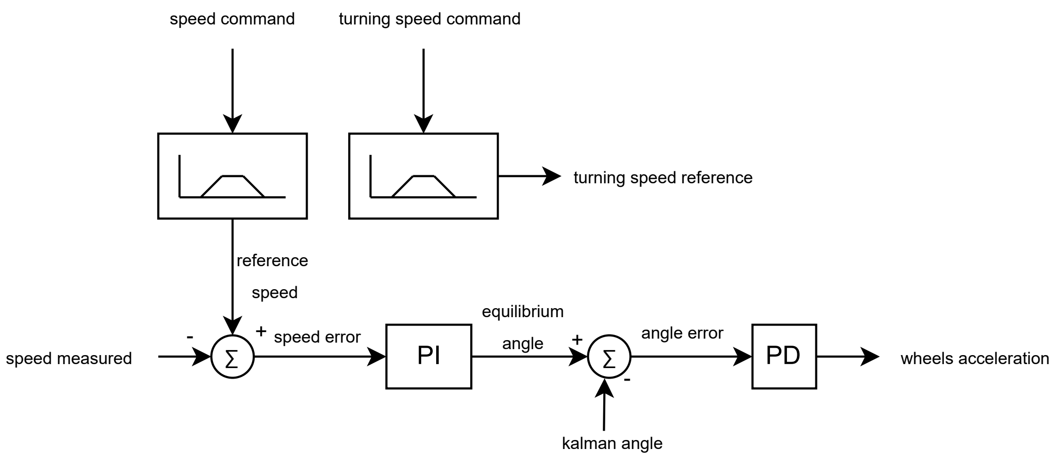


Figure 4.2: The whole schematic representation of the controller block.

4.2.1.1 Trapezoidal speed profiles

To ensure smooth movement, the speed command is not directly sent to the PI controller. Instead, a trapezoidal speed profile is used, which introduces gradual acceleration and deceleration phases (Figure 4.3). This approach limits the rate of change in speed, reducing the risk of instability and abrupt transitions that could destabilize the robot.

The same trapezoidal speed profile is applied to the turning speed command. Although the turning speed is not directly linked to the stability, applying a similar profile to rotational speed control helps reduce sudden, sharp movements. This in turn lowers the risk of wheel slippage.

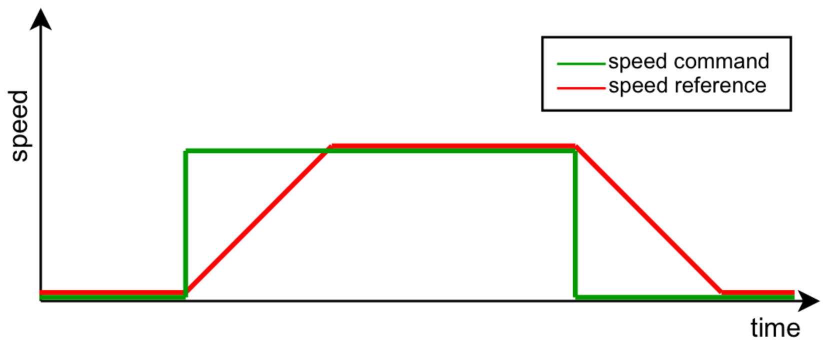


Figure 4.3: Trapezoidal profile for speed reference transitions [22].

4.2.1.2 Controller parameters tuning

One of the most time-consuming yet important tasks was tuning the parameters. To tune the parameters of the PD controller initially, the Ziegler-Nichols tuning method [28] was applied, a classical heuristic approach for tuning the parameters of a PID controller. The method involves disabling the integral and derivative components, then gradually increasing the proportional gain K_p until the system begins to exhibit consistent oscillations. The gain at this point is known as the ultimate gain K_u, and the period of oscillation is referred to as T_u. The two values were obtained experimentally: $K_u = 22, T_u = 4.5s$. Then, the Ziegler-Nichols formulas were used to estimate the initial parameters of the PD controller:

$$K_p = 0.8 \cdot K_u = 17.6, \quad K_d = 0.1 \cdot K_p \cdot T_u = 9.9$$

While these values provided a solid starting point, The behavior obtained showed instability during testing. To improve the system's behavior, a manual fine-tuning through iterative experimentation was done.

For the PI controller, the parameters used in the previous thesis were retained [22], with only slight adjustments through manual tuning while keeping them within the same order of magnitude. The final gains were selected as follows:

- **PI controller:**
  $K_p = -0.063, \quad K_i = -0.053, \quad K_d = 0.0$
- **PD controller:**
  $K_p = 16.3, \quad K_i = 0.0, \quad K_d = 9.4$

The PI controller gains are kept small to ensure a smooth response and to prevent the speed loop from dominating the stability (PD) loop. The negative signs of the PI gains come from the robot's sign convention: forward movement corresponds to a negative pitch angle. Therefore, a positive velocity error requires a negative pitch reference to accelerate in the desired direction. This, in turn, requires negative gains in the PI controller.

4.2.2 Motor drivers

The motor drivers block is responsible for generating low-level commands to control both the wheel motors and the arm motor, based on inputs received from the controller block and the executive loop.

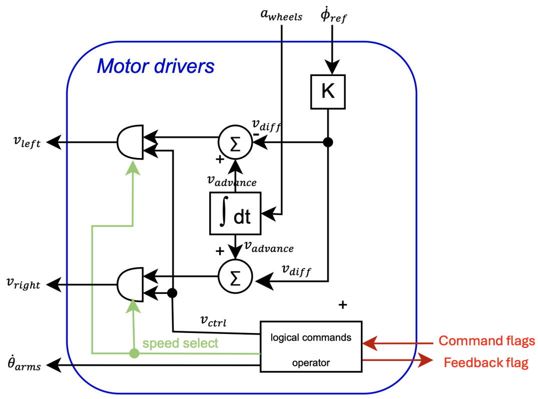


Figure 4.4: Schematic representation of the motor driver block with its inputs and outputs.

First, the motor drivers receive 2 types of commands which are used to control the robot's wheels (Figure 4.4):

- a_{wheels}: wheels acceleration.
- $\dot{\phi}$: turning velocity of the robot (°/s)

However, wheel motors require other types of inputs:

- v_{left}: left wheel velocity.
- v_{right}: right wheel velocity.

It is therefore necessary to convert these two commands into individual wheel velocities. Additionally, a logical control layer must be added to take into account high-level commands.

4.2.2.1 Wheel motors control

To obtain two individual wheel velocities, the system first integrates the acceleration command over time to obtain the translational (forward/backward) velocity:

$$v_{\text{advance}} = \int a_{\text{wheels}} \, dt \tag{4.4}$$

Then, to enable rotation, a velocity difference is required between the wheels (Figure 4.5). To compute the wheel velocity difference v_{diff} from the turning velocity command $\dot{\phi}$, the following relation is used:

$$\dot{\phi} = \frac{v_{\text{diff}}}{L} \cdot \frac{180}{\pi} \quad \Rightarrow \quad v_{\text{diff}} = \dot{\phi} \cdot \frac{L \cdot \pi}{180} \tag{4.5}$$

Where L is the distance between the left and right wheels.

The final wheel velocities are obtained by combining these two components:

$$v_{\text{left}} = v_{\text{advance}} - v_{\text{diff}} \tag{4.6}$$

$$v_{\text{right}} = v_{\text{advance}} + v_{\text{diff}} \tag{4.7}$$

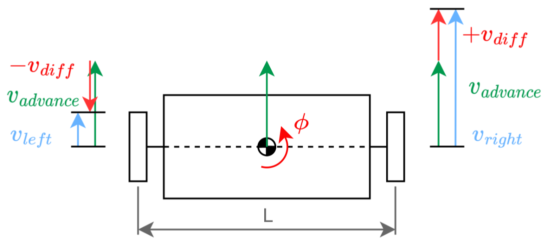


Figure 4.5: Top view of the robot performing a left turn. A higher speed on the right wheel compared to the left wheel generates a counterclockwise rotation around the robot's center [22].

4.2.2.2 Arm motor control

The motor drivers block also controls the arms mechanism (see section 5.1.1.1). This mechanism is driven by a separate motor, which receives a command in the form of an angular velocity $\dot{\theta}_{\text{arms}}$. This command is generated by the executive loop, through the logical control block, to manage transitions between modes.

4.2.2.3 Logical command operator

A logical operator manages the three motors' behavior based on state flags from the executive loop. It serves several purposes:

- To control the velocity of the arms.
- To disable all motors in off/emergency mode.
- To override wheel velocities (v_{ctrl}) when in static mode.
- To provide feedback on the status of the arm deployment (arm_ready) to the executive loop, facilitating state transitions.

4.3 Obstacle avoidance

The obstacle avoidance block is responsible for ensuring that the robot does not bump into an object by detecting and reacting to nearby obstacles. It acts as a safety layer within the high-level control system, continuously monitoring the environment and taking corrective actions when necessary. This mechanism relies on three sonars mounted on the lower part of the robot, which measure the distance to the closest obstacle in their respective field of view. These measurements are then compared to two thresholds:

- A safety threshold at 65 cm, which triggers a smooth deceleration and stop.
- An emergency threshold at 30 cm, which triggers a more immediate reaction by initiating an evasive turn.

These two thresholds were selected by considering both the robot's dynamic braking capabilities and its physical footprint. Assuming a maximum speed of 24 cm/s and a deceleration of 9 cm/s^2, the theoretical stopping distance is:

$$d_{\text{brake}} = \frac{v^2}{2a} = \frac{(0.24)^2}{2 \cdot 0.09} = \frac{0.0576}{0.018} = 0.32 \text{ m} = 32 \text{ cm}$$

The system latency was neglected as it is very small (in microseconds). Thus, the total minimum required distance to safely come to a stop is approximately 32 cm. The chosen safety threshold of 65 cm integrates this value, while also accounting for residual inertial effects. Those may cause the robot to overshoot slightly beyond the theoretical value. The emergency threshold of 30 cm may appear far away from the object, but it is intentionally set to ensure that the robot has enough space for evasive maneuvers. Given a body width of 29 cm, a clearance of 30 cm guarantees that the robot can perform a full-body turn without risk of immediate collision with nearby obstacles.

The obstacle avoidance logic is executed in the executive loop in parallel to the stability loop. It does not require explicit commands from the user. Instead, it acts autonomously, overriding or adapting motor commands when a threat is detected. When no obstacle is present, the system remains passive and lets the robot follow the intended path.

4.4 Executive loop

The executive loop is the high-level control structure responsible for supervising the robot's global behavior and managing transitions between different operational modes. Unlike the inner stability loop, which maintains the robot's pitch in real time, the executive loop determines the current operating mode: off, dynamic balancing, transitioning to static support or resting on the static support. There are several ways to implement the executive loop, but a Finite State Machine (FSM) was chosen because it is simple to visualize, maintain and a FSM framework had already been implemented in previous work, making it easier to build upon an existing structure rather than starting from scratch.

4.4.1 FSM overview

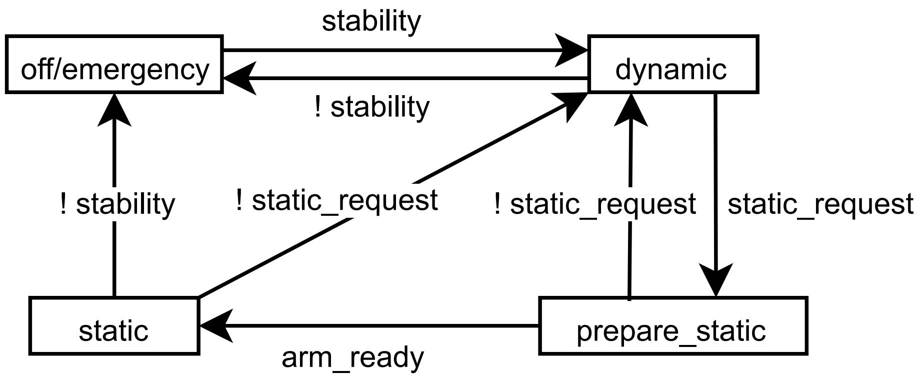


Figure 4.6: Overview of the FSM representing the states of the butler with each transition.

An FSM is composed of a finite set of states and transitions (Figure 4.6). In this system, each state corresponds to a well-defined behavior of the robot, while transitions are triggered by events such as user inputs or actuator feedback.

At every iteration, the FSM evaluates the current state and the conditions for transition and generates command flags for the motor drivers operator.

4.4.2 State descriptions

The FSM implemented includes the following states:

- **off/emergency:** Motors are disabled and the robot is off. This mode is also used in case of emergency stop.
- **dynamic:** The robot has fully transitioned into dynamic balancing mode. The support arms are retracted, and the control loop maintains stability.

- **prepare_static:** The robot prepares to enter a resting position. While still dynamically balancing, it extends its arms in anticipation of resting.

- **static:** The robot rests fully on the extended support arms. The motors enter freeze mode (motors in holding state) to minimize power consumption.

4.4.3 Transition triggers

Transitions between states are determined by user commands and actuator feedback. On the user side, two flags are used: the stability flag indicates whether the robot should actively maintain balance and the *static_request* flag specifies if the robot must rest on its support arms. On the actuator side, a feedback flag *arm_ready* confirms that the arms are either fully extended or fully retracted. This flag is used to switch from prepare_static to static mode.

Chapter 5

Hardware implementation

To meet the defined objectives and constraints and to assemble the interconnected blocks in one physical robot, the robot illustrated in Figure 5.1 was designed and constructed. An important point to consider during the design process is that the weight distribution between the right and left wheels must be balanced to ensure similar grip on both wheels. The system is divided into four distinct parts, each fulfilling a specific function. The following sections provide a detailed explanation of the role and characteristics of each part.

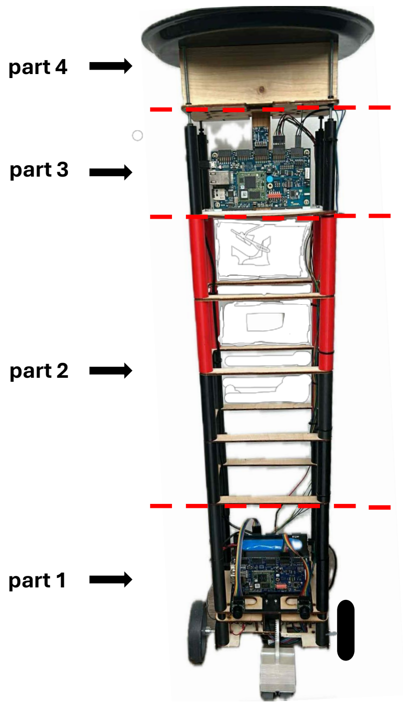


Figure 5.1: Complete front view of the two-wheeled self-balancing table.

5.1 Part 1 : The base of the robot

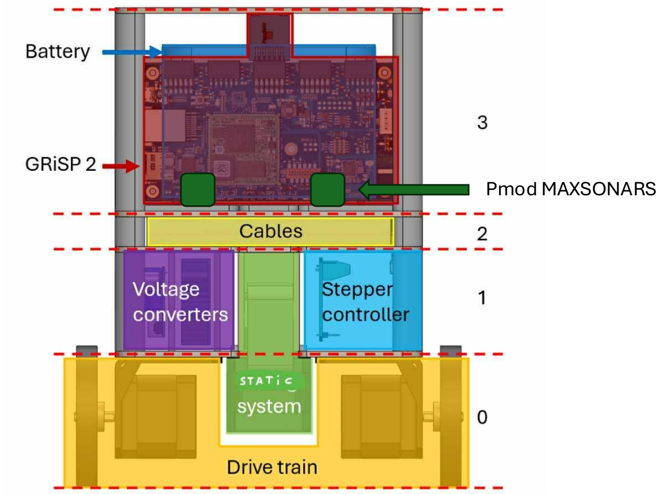


Figure 5.2: Front view of the base of the two-wheeled self-balancing butler. Modified from [22].

As shown in Figure 5.2, the base of the robot is divided into four distinct levels:

- **Floor 0**: The robot's drivetrain. This includes the entire wheel drive system as well as the actuator for the static support mechanism.
- **Floor 1**: Contains the stepper motor drivers, the Lilygo LoRa32, components of the static system and voltage converters.
- **Floor 2**: Dedicated space for electrical cables.
- **Floor 3**: Hosts two GRiSP 2 boards, two front-mounted Pmod MAXSONARs, one rear Pmod MAXSONAR and a $LiFePO_4$ battery with a 3D-printed support structure.

Laser-cut wooden boards are used to separate the different levels of the structure. 6 mm thick laser-cut wooden boards were used because they provided sufficient mechanical strength to support the weight of the various components while offering a good balance between quality and cost. To support and secure the various components, vertical wooden panels are inserted between the horizontal layers using a mortise and tenon joint system. This technique allows for quick assembly but relies on compression to maintain structural integrity. To ensure the assembly remains secure, vertical screws working in tension apply compressive force on the joints, thereby holding the entire structure firmly together.

5.1.1 Floor 0: The robot's drivetrain

One of the main constraints of the project was that the robot had to balance on two wheels. The robot requires actuators to convert electrical signals into physical motion and drive the wheels. For this purpose, stepper motors were selected, more specifically NEMA 17 series stepper motors. Compared to other types of electric motors (such as DC motors, servo motors, or BLDC motors), the stepper motor was the most suitable technology for this application because it has good torque/speed ratio, easy to control and relatively cheap for actuators ([22]). The NEMA 17 stepper motor is a good choice as stepper motor because it is commonly used in the 3D printing industry where the torque and speed requirements are comparable to those of this robot [29].

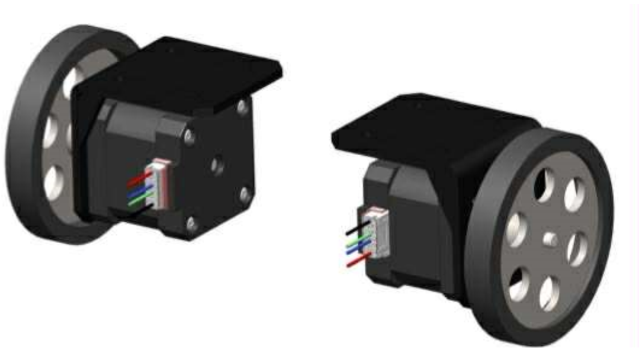


Figure 5.3: The drive train with the 2 wheels connected to 2 stepper motors with their brackets [22].

As illustrated in the Figure 5.3, the two stepper motors share the same horizontal axis, each directly driving one wheel mounted on its shaft. The wheels have a diameter of 100mm and consist of a solid polyurethane. Polyurethane was selected for its excellent grip, vibration-damping capabilities, and non-marking properties, making it ideal for indoor use. The wheel is secured to the motor shaft via a steel hub with a set screw ensuring a reliable mechanical connection and good torque transmission. Each stepper motor is mounted on its own right-angle bracket, which is then attached to the chassis.

5.1.1.1 The static support system

The static support system is used when the robot no longer needs to move for a time and switch to a stationary mode. In this configuration, two support arms, illustrated in Figure 5.5, can be deployed to stabilize the structure.

These arms are equipped with small wheels to allow the robot to be manually displaced when necessary. To guarantee stability and smooth manual movement, the wheels had to be tangent to the ground once the arms were fully deployed. To meet these criteria, wheels with a diameter of 50 mm were selected, which corresponds to the radius of the robot's wheels. This made it possible to align the contact surface of the arm with the center of the wheel, as shown in Figure 5.6, ensuring both stability and ease of manual handling.

The design of the support arms draws inspiration from the lifting mechanism developed in the master's thesis by Goens and Ponsard [22]. The same core idea of maintaining a compact system that integrates seamlessly with the robot's structure was retained. Due to the complex geometry of the arms and the need for them to support the robot's weight, resin 3D printing was chosen. This fabrication method was selected for its simplicity and speed. The choice of resin as material was made for its price/quality balance.

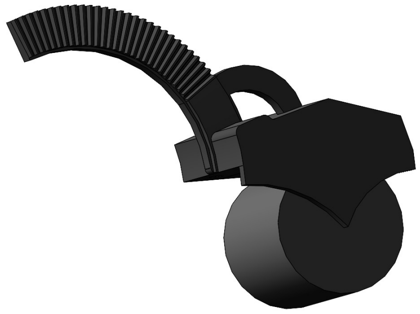


Figure 5.4: 3D design of the arms.

Figure 5.5: 3D printed version of the robotic arm.

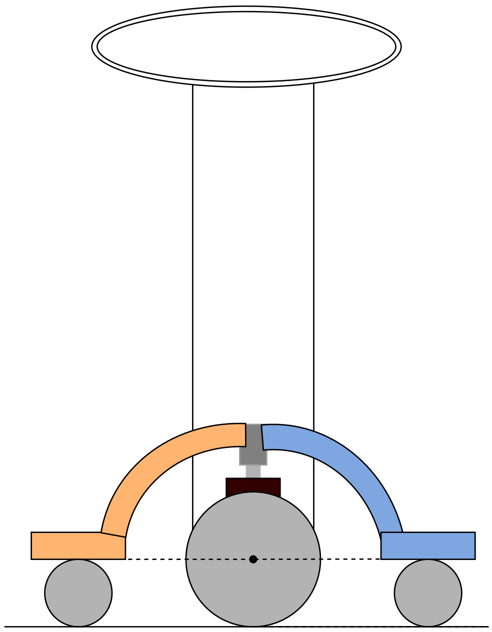


Figure 5.6: Static support system in static mode. The arms are fully extended.

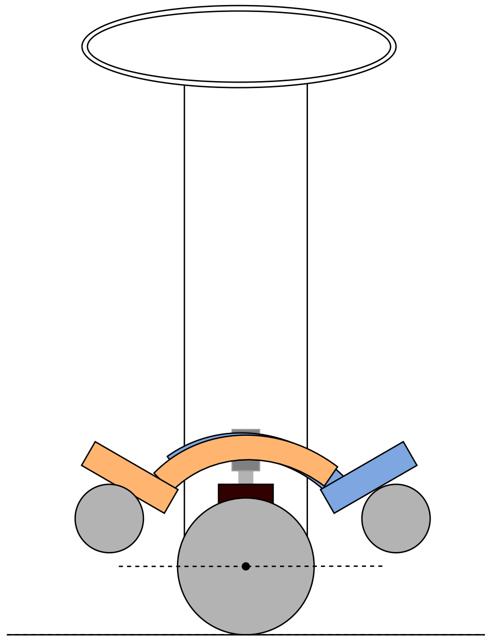


Figure 5.7: Static support system in dynamic mode. The arms are fully retracted.

The designed mechanism consists of three moving parts: a straight bevel pinion connected to the motor shaft, and two curved racks that form segments of a bevel gear (Figure 5.8). The actuator controlling the motor is positioned underneath the mechanism in the middle of the robot in order to lower the center of gravity and distribute the weight more evenly across both wheels. For reasons of simplicity and cost efficiency, the static support system uses the same type of stepper motor as the drivetrain.

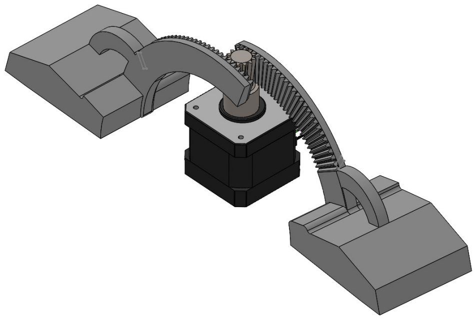


Figure 5.8: The arm system in static mode.

To ensure that the robot remains stable when the support arms are deployed, an evaluation was made to determine whether it would still resist tipping with an additional load placed on top. The full development of this evaluation is provided in Appendix B. This analysis confirms that the robot is stable within a tipping margin of approximately 15.4°.

5.1.2 Floor 1: Stepper motor drivers, the Lilygo LoRa32 and voltage converters

Level 1 consists of three important electrical components, the stepper motor drivers, Lilygo LoRa32 and the voltage converters.

5.1.2.1 Stepper motor drivers and PCB

The role of the stepper motor drivers is to convert the controller outputs into electrical signals capable of driving the stepper motors. The TMC2208 stepper motor drivers were chosen because they operate with almost no noise compared to other drivers. The motor driver is controlled through three logical input pins:

- **En (Enable)**: Enable when the pin is set to high. The motor can no longer spin. When the pin is set to low, the motor is disabled and can rotate freely without resistance.
- **Step**: Each rising edge on this pin triggers the motor to perform one step.
- **Dir (Direction)**: Defines the direction of rotation for the motor.

The driver outputs are connected to the motor windings via four terminals: M1A, M1B, M2A and M2B, corresponding to the motor inductance. This driver supports micro-stepping, allowing for finer position control and smoother motion. Additionally, the motor torque can be tuned using a built-in potentiometer that adjusts the current supplied to the stepper motor inductance.

For the stepper motor interface, a PCB designed in the master's thesis of Goens and Ponsard [22] is used to facilitate the connections between the different components. It connects the motor drivers, the main GRiSP 2 board, and the LilyGO LoRa32 module [22]. The features of the PCB are:

- **Eight logical input signals:** one `STEP` and one `DIR` input per motor, one dedicated `EN` signal for the central driver, and one shared `EN` signal for the two other drivers.
- **Logic supply voltage:** ranging from 3 V to 5 V.
- **Motor power supply:** supports input voltages up to 36 V.

The enable pins of the first and last stepper motor drivers are connected together, since they correspond to the motors used in the differential drive system and are always enabled simultaneously. The design of this PCB can be seen in Appendix E.

5.1.2.2 Lilygo LoRa32

To address the limitations of the GRiSP board in generating high-frequency control signals for stepper motors, an additional microcontroller, the Lilygo LoRa32, was integrated into the system [22]. Lilygo is an ESP32-based circuit board. This board offers both I^2C communication and LoRa wireless capabilities. Its main role is to receive actuation commands from the GRiSP via I^2C and translate them into motor control signals (step, direction, enable) for the three stepper drivers. Additionally, it handles wireless commands sent by a second Lilygo LoRa32 using LoRa communication. This secondary Lilygo captures user inputs and transmits them to the robot in real time, enabling remote control.

5.1.2.3 Voltage converters

The robot's electrical system requires two distinct voltage levels, which is why two buck converters were integrated. Boost converters are avoided because they tend to introduce electrical noise, which can interfere with communication buses [22]. A 12.8V $LiFePO_4$ battery is used as the main power source because it is the safest type of lithium-based battery [22]. The first voltage converter steps down the 12.8V to 10V to supply the stepper motor drivers. The second converter reduces the voltage to 5V, which is needed to power the logic components such as the GRiSP 2 boards and the Lilygo LoRa32. Using separate voltage converters ensures electrical isolation between high-current motor components and sensitive digital logic.

5.1.3 Floor 3: The GRiSPs, sonars and battery

5.1.3.1 GRiSP functions and positions

To ensure efficient obstacle detection across the entire width of the robot, three GRiSP boards are used, each playing a distinct role. Two GRiSP boards are mounted near the battery on either side of the robot. Each of these handles a single sonar sensor. These front-facing sensors provide the main detection capability for obstacles in the robot's path. A third sonar is installed at the rear of the robot, connected to the last GRiSP which is the main one (which is in the third part of the robot). The rear sonar, used when the robot moves backward, operates alone and therefore provides a narrower coverage angle of about 25°. This limitation is not critical, as backward motion is generally used for small maneuvers.

5.1.3.2 Optimization of sonar placement

To determine the optimal placement and orientation of the two sonar sensors on the robot, a Python script was developed (see Appendix D) based on trigonometric modeling of the sonar coverage area (Figure 5.9). The goal was to minimize the maximum vertical detection distance (h), which corresponds to the minimum distance at which an obstacle can be detected, in order to reduce blind spots in front of the robot.

The setup involves two front-facing Pmod MAXSONAR modules placed symmetrically around the robot's central axis. The sensors are rotated by α degrees with $\alpha \in [0°, 12°]$. The maximum angle was limited to 12° because beyond that, the projected distance h_2 becomes undefined (no intersection occurs). In addition to angular variation, the lateral distances (along the x-axis) from the robot's center were also varied using a variable offset x. The sonars are positioned with a fixed minimum of 55.5 mm from the wheels (the physical limit of the chassis) and a fixed maximum of 72.5 mm from the center.

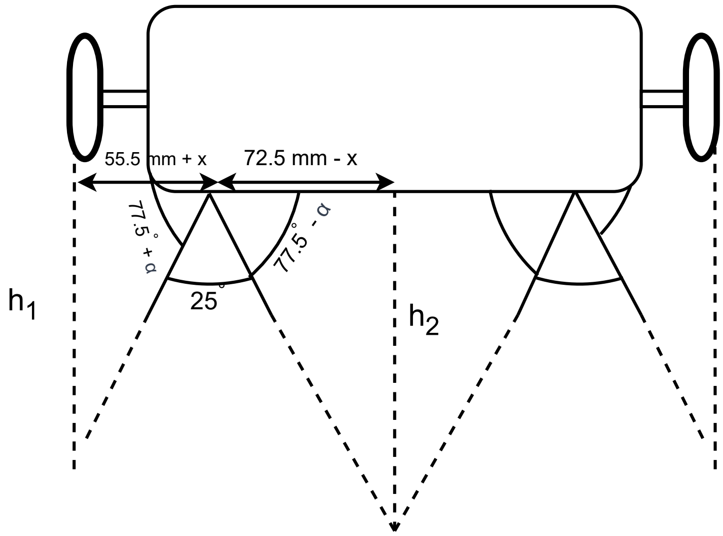


Figure 5.9: Bottom view of the robot for the optimization of sonar placement.

For each pair (α, x), the vertical detection distance is calculated using the tangent function:

$$h_i = \tan(\theta_i) \cdot d_i,$$

where $\theta_1 = 75° - \alpha$ and $\theta_2 = 75° + \alpha$, and d_1 and d_2 are the respective lateral distances from the robot's center. All feasible configurations were evaluated, and the maximum of the two distances h_1 and h_2 was recorded to account for the worst detection distance. The configuration that minimized this value was then selected.

The optimal configuration was found when $\alpha = 0°$, meaning both sensors are oriented straight ahead, and the lateral offset x is equal to 10 mm. This configuration results in a maximum detection of 233.25 mm, which is acceptable for this application. To implement this configuration, a 3D-printed part was created to securely attach the sonar sensors to the robot in the correct positions, as illustrated in Figure 5.10.

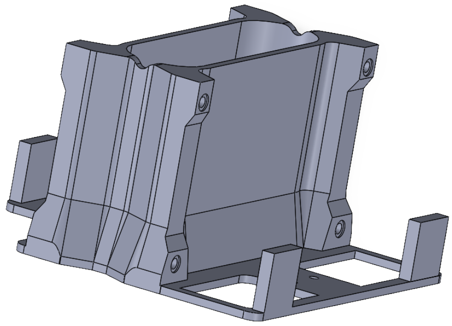


Figure 5.10: 3D design of the support of the front and back sonars, sonar-only GRiSPs and battery.

5.2 Part 2: Height structure of the robot

Figure 5.11: Side view of the height structure of the robot.

As shown in Figure 5.11, this structural part was designed to increase the height of the robot while minimizing additional weight and maintaining rigidity. It is composed of laminated rods chosen for their strength, although they proved too flexible on their own. To address this, they are reinforced with lightweight 3D-printed PLA supports that enhance stiffness without increasing too much the mass. Additional layers of wood were also added to further improve the overall rigidity of the structure.

5.3 Part 3: Main GRiSP casing

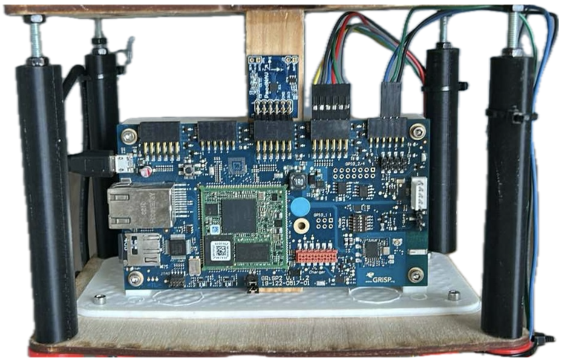


Figure 5.12: The main GRiSP with the Pmod NAV.

The main GRiSP is mounted at the top of the robot, and it is responsible for handling the Pmod NAV sensor and the data transmission to the Espressif Systems ESP32 Microcontroller (ESP32). This placement was chosen specifically to improve the sensitivity of the inertial measurements. Mounting the Pmod NAV higher makes it more responsive to subtle balance changes, improving reaction time and stability. An alternative approach using a bottom-mounted GRiSP with a cable extending the Pmod NAV to the top was tested, but it failed. The SPI port to which the Pmod NAV was connected could not detect the sensor properly over a serial cable extension, likely due to signal degradation or lack of proper shielding. In contrast to Pmod NAV, placing the Pmod MAXSONAR module at the bottom of the robot did not cause any communication issues, even though the cable length was approximately 70 cm. This is because the module communicates via UART, which is more tolerant of longer distances [20].

5.4 Top plate and counterweight design for dynamic stability

Figure 5.13: Side view of the top part of the robot. It is composed of a rounded plate and a counterweight.

To reduce the sensitivity of the stability system to load variations, a 1 kg counterweight was added just below the top plate of the robot. Its position and mass was chosen to keep the center of mass at a fixed height of approximately 41 cm above the wheel axis, just below the center of the robot. Since objects are placed on the table during use (books, glasses, keys, ...), this counterweight helps minimize the resulting changes in both the vertical center of mass position and the moment of inertia. By minimizing these variations, the physical parameters used by the Kalman filter remain nearly constant. This avoids the need for dynamic retuning of both the filter and controller parameters, ensuring that the system remains stable and predictable even when external loads are applied.

The top plate itself was made of lightweight plastic with a non-slip coating, ensuring both practical usability and minimal mass.

5.5 Overall electrical circuit

The diagram in Figure 5.14 provides a global overview of the Butler's electrical system, helping to visualize how the various components are connected and how they communicate. The top part of the schematic shows all the electrical elements located on the robot itself, while the lower section highlights the user input block, which consists of a remote emergency switch. This remote is built using a Lilygo LoRa32 module and an emergency push button [22]. Since the robot is inherently unstable by design, placing the emergency stop directly on the robot was considered unsafe. Instead, a remote solution was chosen. The remote is powered and receives user inputs via Universal Serial Bus (USB), which are then transmitted wirelessly through LoRa to the corresponding Lilygo on the robot.

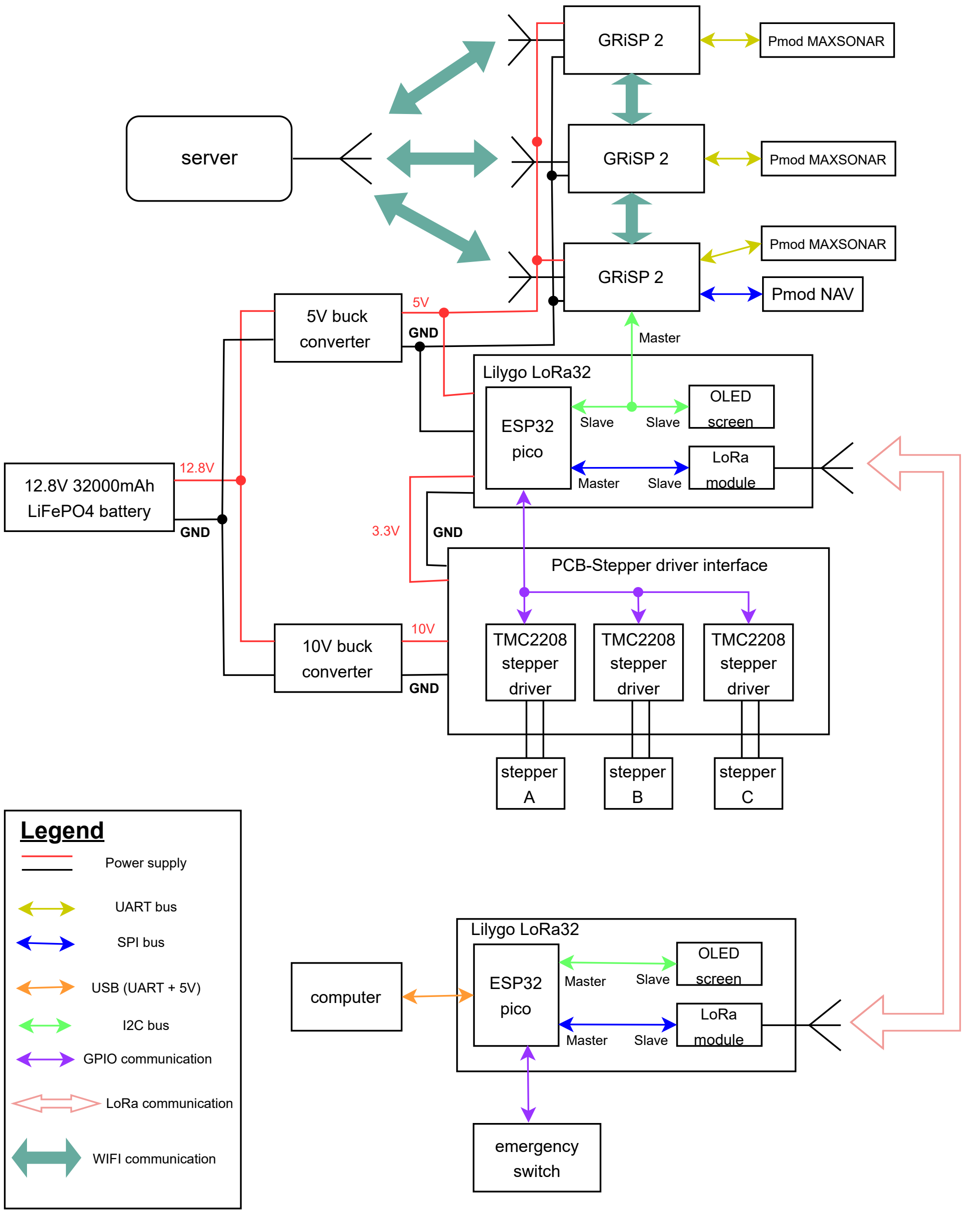


Figure 5.14: Global overview of the Butler's electrical system.

Chapter 6

Software implementation

6.1 Distributed architecture

The distributed architecture of the system is designed to handle 3 main functions: stability control, obstacle avoidance, and inter-board communication. To achieve this, the system integrates multiple GRiSP boards that share information and coordinate their actions, a central server that initiates the communications, and an ESP32 LyLiGo board that manages the motors (Figure 6.1).

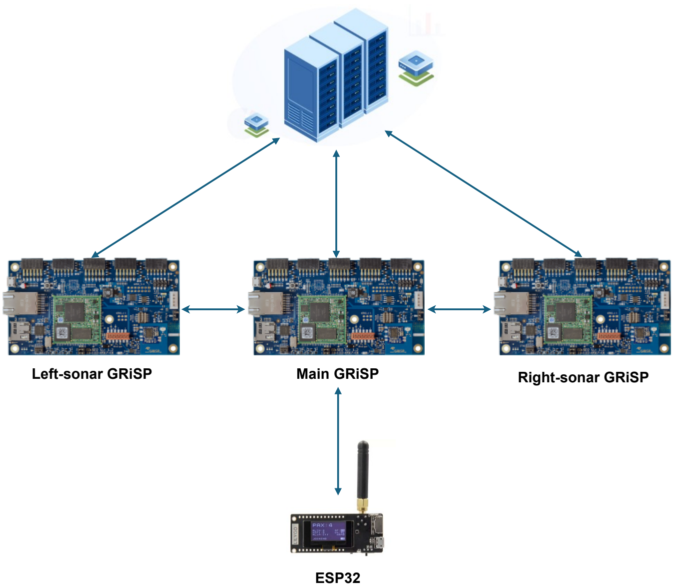


Figure 6.1: The distributed architecture of the different devices and their connections.

The server plays a key role during both initialization and operation. It is responsible for enabling device discovery at startup so that all GRiSP boards can identify and communicate with each other. It is also used for system monitoring and centralized logging. This ensures that diagnostic and runtime information is collected in one place. The GRiSP boards form the core of the system and handle the control logic. They are responsible for stability control and obstacle avoidance, as well as managing communication with each other once the system is initialized. The main controller GRiSP board is directly connected to the motors and to the user commands through the ESP32 board. This board serves as the interface between the user and the robot by relaying commands and managing motor control signals so that user inputs are reliably translated into actions.

The following sections will detail the software implementation of each of these elements. An explanation on how the GRiSP boards are structured, how the server manages system monitoring and discovery, and how the ESP32 board interfaces with the main controller to handle user commands and control the motors, is given.

6.2 GRiSP software architecture

The GRiSP software architecture was designed with two key features in mind: modularity and ease of use. To achieve this, the same software is installed on every GRiSP board to avoid the need to customize the code for each board. At startup, each board dynamically determines its role based on its DIP jumper configuration and connect to the network. Once done, the GRiSP boards begin executing their assigned functions. The description below provides an abstract overview of the software architecture and the control loops, highlighting the role of each board in the system (Figure 6.2).

The controller board is responsible for the core functions of the robot. This includes data fusion, obstacle avoidance, and stability control (i.e., the blue frame in Figure 6.2). Each loop iteration involves:

1. Acquire sensor data from the IMU and the sonar nodes.
2. Process and filter the data.
3. Run the obstacle avoidance logic and the stability controllers.
4. Send motor commands and schedule the sonar measurements.

Each sonar node board focuses on distance measurement (i.e., the orange frame in Figure 6.2). Their loop is simpler and follows a basic sequence:

1. Wait for authorization from the main controller to prevent sonar interference.
2. Perform a sonar measurement and filter the result.
3. Transmit the data back to the main controller.

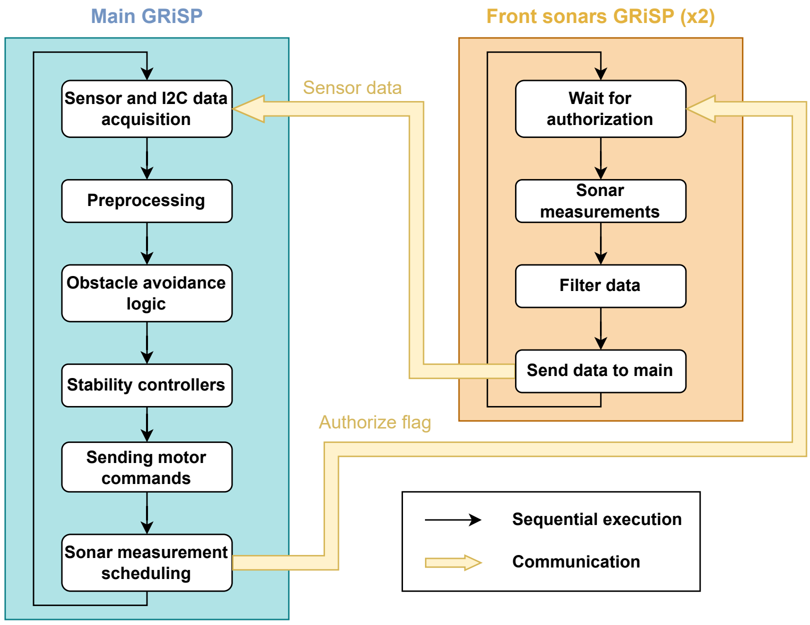


Figure 6.2: GRiSP architecture and sequential loops of the GRiSP boards.

After the initial discovery phase, the communication between the GRiSP boards occurs directly over UDP, without passing through the server. This guarantees low-latency data exchange and allows the main controller to maintain a high control frequency. It also allow the sonar nodes to operate asynchronously from the rest of the system. Of course, the actual implementation is more complex than this abstraction. In practice, the sequential execution on each GRiSP board is divided into multiple processes that communicate with each other internally. The following subsections provide a detailed review of each process.

6.2.1 Balancing control and communication handling

The process, called `balancing_robot`, is the highest-level process in the supervision tree of the software. It is launched when the GRiSP card boots up and it is responsible for launching the various processes according to the role assigned to each GRiSP board. Additionally, it also handles the sonar scheduling mechanism and configures the board for network communication. Once the configurations are done, it manages both inter-process and inter-GRiSP communication exchanges. This process is identical on all three GRiSP boards of the robot. Based on the configuration set using the jumpers, each GRiSP board is assigned a specific role, from 0 to 3, within the system (Figure 6.3).

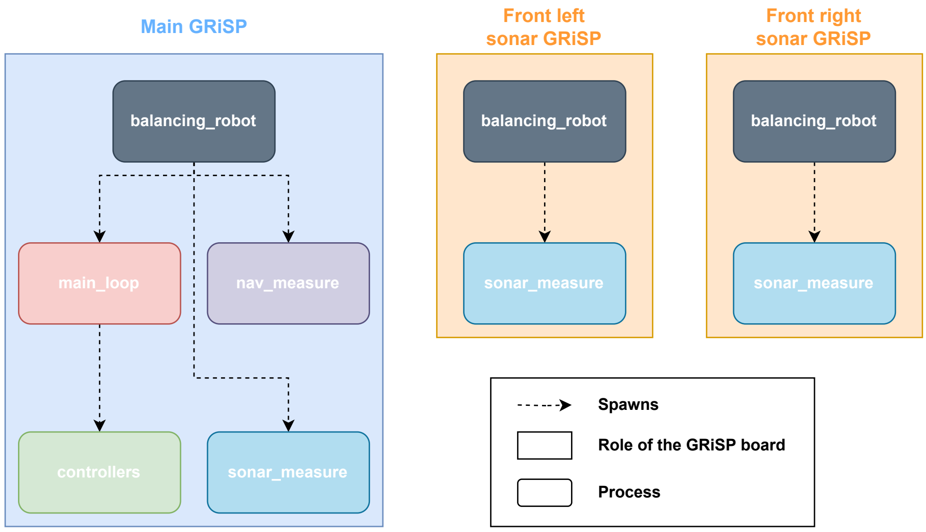


Figure 6.3: Process hierarchy based on the roles assigned to each GRiSP board.

6.2.1.1 Configuration of the GRiSP

At startup, the `balancing_robot` process determines the board's role based on the jumper configuration. Three roles are available: The first role corresponds to the main GRiSP board. This board is responsible for handling the stability loop (i.e., `main_loop` process) and running two `hera_measure` processes: the `nav_measure` and the `sonar_measure`. The second role is the front-left sonar board. It spawns only the `sonar_measure` process to handle sonar data collection. The last role is the front-right sonar board. Similar to the front-left sonar board, it only manages a `sonar_measure` process.

In addition to device initialization, the `balancing_robot` process configures the network connection. This configuration consists of two steps:

1. **WiFi connection:** The WiFi connection is handled entirely by the Hera framework. At startup, the board subscribes to Hera events with

hera_subscribe:subscribe(self()), which means it will receive notifications about the network status. Hera automatically attempts to join the preconfigured WiFi network. The board waits up to 18 seconds to receive a "connected" notification from Hera. After this timeout, it changes the LED color to two magenta LEDs and tries again. If it connects, the LEDs flash in white.

2. **Server discovery and handshake:** As shown in Figure 6.4, once connected to a network connection, the board listens for a "ping" message from the server. If received, the server's IP and port are stored in Hera, and the LEDs flash green. If the message is not received within 9 seconds, the process retries and the LEDs flash red. After receiving the ping message, the board sends a unicast "Hello" message to the server and waits for an "Ack" message. Upon success, the LEDs turn aqua to indicate that the system is fully connected. In parallel, a method called alive_loop/0 is spawned as a separate process.

This robust three-step procedure ensures that each GRiSP board is properly connected to the WiFi network and to the server before starting normal operations. Thanks to Hera, the application does not need to manually handle WiFi credentials or connection attempts: it only reacts to the hera_notify events indicating the current network state.

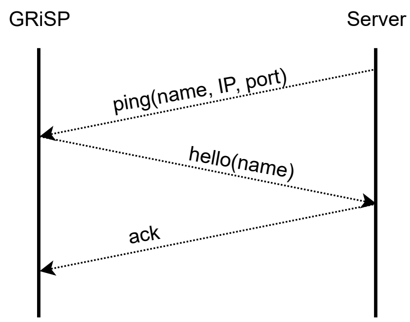


Figure 6.4: Handshake between a GRiSP board and the server.

6.2.1.2 Sonar scheduler for the obstacle avoidance mechanism

The sonar scheduler for the obstacle avoidance mechanism is also implemented in the main supervisor (Algorithm 1). Since the sonars cannot measure simultaneously to avoid interference, the sonar scheduler grants authorization to one sonar to take measurements every 75 ms. It is the time it takes for a sonar to perform a single measurement with a 25 ms margin added. This allows long-range echoes to return with minimal interference. Depending on whether the robot is moving forward or backward, the corresponding sonar is activated. When the robot is moving forward, two front sonars are used to cover a larger angle. The scheduler uses a round-robin approach and alternates between the two every 75 ms. The term round-robin refers to a scheduling and load-balancing method that distributes resources (e.g., time slices to processes or requests to servers) in a fair and circular manner [30]. When the robot is moving backward, only the rear sonar is used. In this case, the scheduler simply reauthorizes the same sonar every 75 ms as long as the robot continues to move backward.

6.2.1.3 Communication and message handling

The `balancing_robot` process also manages ongoing communications with the server and between GRiSP boards (Figure 6.5). Two main loops handle these tasks in `balancing_robot` :

- The **alive loop** is spawned after the server handshake. It periodically (every 20 seconds) sends a unicast message `"alive :  {name}"` to the server. This ensures the server is aware that the GRiSP board is still connected. It makes the system more robust, as a GRiSP board could be disconnected by crashing or losing the network connection. Even in such cases, it is able to recover and reconnect to the server without problems.

- The **message notification loop** waits for incoming `hera_notify` messages from the Hera framework or from other GRiSP boards. Each message is passed to the `handle_hera_notify/1` function, which processes the message based on its type:

  – **`"ping"`**: Ignored because it was already handled during the initial handshake.

  – **`"Add_Device"`**: Calls `add_device/3` to dynamically register a new GRiSP device with its IP and port in Hera.

  – **`"authorize"`**: Sends an `authorize` message to the sonar process, allowing it to start measurements.

  – **`"sonar_data"`**: This message is exclusive to the main GRiSP. It forwards the sonar distance from the other GRiSPs to the `main_loop` process.

  – **Unhandled messages**: Any other message types are logged for debugging purposes.

This design allows the main board to coordinate sonar boards dynamically. It allows the different GRiSP boards to maintain active communication with the server even after the initial connection is established. The full source code of the `balancing_robot` module is available in Listing M.1.

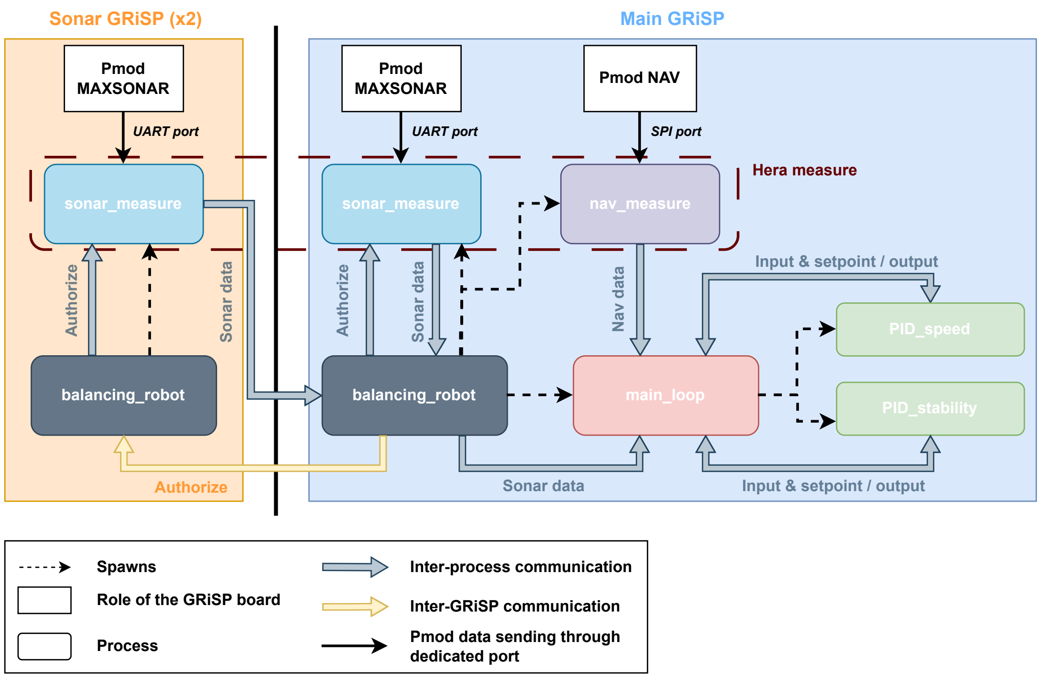


Figure 6.5: Diagram showing the different inter-process and inter-GRiSP communications.

6.2.2 High-level main loop

The `main_loop` module process is the robot's main execution thread, as it handles the entire control pipeline. It is divided into two parts: The initialization function and the real-time loop. Initialization of the main loop is critical. It runs only once at startup and is assigned the highest priority to ensure deterministic execution. This function performs the initial calibration of the robot by initializing the Kalman filter with a valid starting state (i.e., initial angle and velocity), spawning the PID controllers with their tuned parameters and defining the initial state of the control loop. Once these steps have been completed, the main loop function starts, continuously handling sensor readings, control updates and actuation in real time. The full implementation of the `main_loop` module can be found at Listing M.3. The following sequential steps describe the operations performed by the main loop function at each control iteration:

1) Time and Δt calculation At the beginning of each loop iteration, the current timestamp is recorded. The time step Δt is then computed as the difference between this timestamp and the one from the previous iteration.

2) Logging This step is not directly involved in the robot's control. It is used for debugging. Since the control loop runs at a high frequency, logs are recorded at fixed intervals. They are stored every 500 ms to limit the amount of data that is stored

3) Kalman filtering It handles the filtering of navigation data and produces a real-time estimate of the robot's tilt angle. As the Pmod NAV sensor data is processed in the `nav_measure` module, it is sent asynchronously to the main loop and handled through message reception. Since the main loop typically might run faster than the IMU sampling rate, the Kalman logic first checks whether a new message has been received. If one is available, a full Kalman update is performed (prediction + correction). Otherwise, only the prediction step is executed. This mechanism allows the control loop to maintain a high frequency even when sensor updates are delayed. The pseudo-code for this message-based Kalman update logic is detailed in Algorithm 3.

4) I^2C read Control inputs are collected from the remote controller through the I^2C interface. A total of five bytes are received from the ESP32 (Table I.1). The first four bytes represent two half-float values that encode the rotation speeds of the left and right wheels. The fifth byte is a control byte that encodes input flags from the user interface, such as commands for arm movement, forward/backward motion, or left/right turning. These binary signals are decoded and used to generate motion goals for the robot. They are also used to update its internal state. A full breakdown of the control byte is provided in Table I.2.

5) Sonar message handling This step processes the sonar readings to determine the proximity and direction of any obstacles. Similar to the Kalman filtering step, sonar data is received asynchronously from other processes via message passing. As a result, a new sonar measurement is not necessarily available at every loop iteration. If no new data is received, the previously stored value is reused.

6) Stability engine This is the most critical step of the main loop, as it handles the actual stability control of the robot. It takes as input all previously acquired and processed physical data (such as the estimated tilt angle, the linear speed, and sonar readings) and computes the required actuation command to maintain balance. The output of this step is an acceleration value that will be sent to the motors to adjust the robot's tilt.

7) State machine and I^2C write In this step, the next robot state is determined among `rest`, `preparing_static`, `static`, and `dynamic`. This new state is based on the current input flags. After the state has been determined, the acceleration, the desired turning speed, and the new robot state are sent to the ESP32 via the I^2C bus to update the control commands.

8) Frequency control The loop frequency is essential in the implementation of the robot's control. Indeed, a loop frequency that is too low prevents the robot from maintaining balance without moving, while a frequency that is too high causes abnormal behaviors.Therefore, this part of the loop stabilizes the frequency according to a target frequency (220 Hz in this case), which is set during the initialization of the main loop.

9) Server transmission The server transmission step is used for debugging. The logs are stored in a buffer and send to the server. The transmission is triggered when the robot is in the static state.

10) State update After completing all the previous steps, the control loop restarts using the updated state variables.

6.2.3 Navigation measurements

The `nav_measure` module implements the `hera_measure` behaviour and is responsible for continuously acquiring data from the Pmod NAV sensor. This process is launched during the initialization of the main GRiSP board in the `balancing_robot` process (Figure 6.3). It takes as parameters the process ID of the `main_loop` and the role of the board. The process ID is used to send navigation data and the role is mainly used for naming and reusability purposes, but in this case it always corresponds to `robot_main`, as it is the only board spawning this process.

A `hera_measure` process needs to define two methods: `init/1` and `measure/1`. The `init/1` function is called once when the process is started and is responsible for initializing the internal state and parameters (e.g., saving the process ID of the `main_loop` and the role of the board). The measurement function `measure/1` is called periodically by the Hera framework. It reads the three-axis acceleration and gyroscope data from the Pmod NAV sensor using the `pmod_nav:read/3` function.

Once the data is acquired, a message is sent directly to the main loop process via Erlang message passing. This allows the main control loop to immediately process the latest navigation data for the control algorithm. If a sensor read operation fails, the process catches the error, logs it, and continues measuring without crashing. This is important because this process is critical for maintaining the robot's stability. The decoupling of the Pmod NAV reading from the control logic helps reduce the workload and allows the control loop to run at a higher frequency. The full source code of the `nav_measure` module is available in Listing M.5.

6.2.4 Sonar measurements

The `sonar_measure` module is responsible for acquiring distance measurements from the Pmod MAXSONAR sensors. Unlike the `nav_measure` process, this module only performs a measurement when it receives an *authorize* message from the main controller. This scheduling avoids interference between the different sonar sensors, as explained earlier in the sonar scheduler. When the process receives an authorization signal, it performs the following steps:

1. It triggers a sonar reading that returns the raw distance measurement in inches. It is then converted into centimeters and rounded. Invalid readings outside the 15 to 648 cm range of the sonar specifications are discarded and replaced with the previous valid measurement.

2. Then, a simple low-pass filter is applied to the distance values to smooth them out:
$$D_{\text{filtered}} = \alpha \cdot D_{\text{previous}} + (1 - \alpha) \cdot D_{\text{new}} \tag{6.1}$$
where $\alpha = 0.2$ is the filter coefficient; D_{previous} is the previous measured distance; and D_{new} is the new measured distance. The value of α was chosen to be small enough to effectively smooth out measurement noise while avoiding significant latency in the detection of fast-changing distances.

3. Finally, the filtered measurement is sent back to the main controller and the server for logging purposes.

As with the navigation measurements, this process is fully decoupled from the main loop. This prevents any delay caused by the sonar hardware, such as waiting for the ultrasonic pulse, from blocking the execution of the control logic running on the `main_loop`. The complete code of the `sonar_measure` module is available in Listing M.6.

6.2.5 Stability controller engine

The stability controllers are implemented within the `stability_engine` module (Listing M.4). This module is responsible for maintaining the robot's upright posture and velocity tracking using two cascaded PID controllers. It also implements the object avoidance behavior based on sonar readings. The controller receives as input the system state `{Dt, Angle, Speed}`, sonar data, and the desired velocities for forward motion and turning. The function follows several steps:

1) Obstacle avoidance The first step is the obstacle avoidance logic. It is implemented into two main stages, as detailed in Algorithm 2. First, if the sonar reading falls below a critical threshold of 30 cm (i.e., the emergency threshold below which forward motion is no longer permitted), the controller triggers an immediate evasive maneuver by forcing a right turn at maximum turning speed. This acts as an emergency escape. Second, if an obstacle is detected at a distance of 65 cm (i.e., the safety threshold where braking begins) and is detected in the current direction of motion, the robot applies a virtual brake by setting the forward velocity to zero. If no valid sonar reading is available, the robot uses its previously planned velocities.

2) Acceleration saturation The second step of the stability engine saturates the acceleration based on the maximum velocity and acceleration. This ensures that the velocity profile remains trapezoidal. This profile helps prevent sudden changes in speed that could cause the robot to become unstable. The system applies a limiting function that restricts the rate at which both forward and angular velocities change. As a result, movement transitions stay fluid and physically achievable.

3) Cascade control with dual PID loops Finally, the architecture uses a two-stage PID control structure: The `Pid_Speed` compares the current linear velocity `Speed` with the reference velocity. It then outputs a desired tilt angle. The `Pid_Stability` uses this target angle and compares it with the measured angle to generate the actuation command `Acc`, which corresponds to the motor torque.

6.2.6 Debugging tools

To support development and parameter tuning, as well as to improve real-time visibility, several debugging tools are implemented. These tools have helped us to diagnose issues efficiently, monitor system behavior during execution and analyse sensor and control data over time.

First a logging system to monitor the robot's behavior was implemented. This includes the progression of the Kalman filter, the PID outputs, the robot's internal state, and the loop frequency. These logs are automatically sent to the server without any manual intervention being required, and they are ready to be used for plotting and performance analysis. Data is only transmitted when the robot is in a static state, preventing any loss of control loop frequency during critical phases, such as dynamic balancing or movement. This ensures that logging does not interfere with real-time control performance. The full implementation of the logging system can be found in Listing M.8.

Secondly, the on-board LEDs of the GRiSP were used to implement a useful debugging tool. These LEDs provide immediate visual feedback on the internal state of the robot. Since it is often difficult to know whether a process has crashed or is stuck without being connected to the robot via serial cable, the LEDs offer an efficient way to monitor system behavior in real-time. Each color or blinking pattern represents a specific status, such as startup, Wi-Fi connection, device initialization, or server discovery. This makes it easy to identify where the robot is in its boot or operational cycle without the need of any external tools. A table summarizing the different states can be found J.1.

6.3 Lyligo ESP32 software architecture

The LilyGO ESP32 module plays a fundamental role in the robot's embedded system. It handles the low-level operations of the GRiSP board and acts as a communication and actuation bridge between the user interface, the main GRiSP board, and the motors. The ESP32 is responsible of:

- Receiving user commands via LoRa.
- Exchanging data with the GRiSP board over the I^2C protocol in a slave configuration.
- Generating control signals for the stepper motors based on the output data provided by the GRiSP.

To manage these tasks efficiently, the ESP32 uses a dual-core architecture, as illustrated in Figure 6.6. The first core acts as the main execution loop, handling user commands received via LoRa. It also incorporates an interrupt-driven I^2C handler, as the LilyGO module operates as a slave device to the GRiSP board. When the GRiSP sends an interrupt request, the LoRa loop is temporarily interrupted to process it. There are two types of I^2C interruptions: `send request` and `receive`. The first one is triggered when the master wants to read data such as, user input flags, feedback flags, or speed measurements. The `receive` is triggered when the master sends data to the slave, such as acceleration values and control flags.

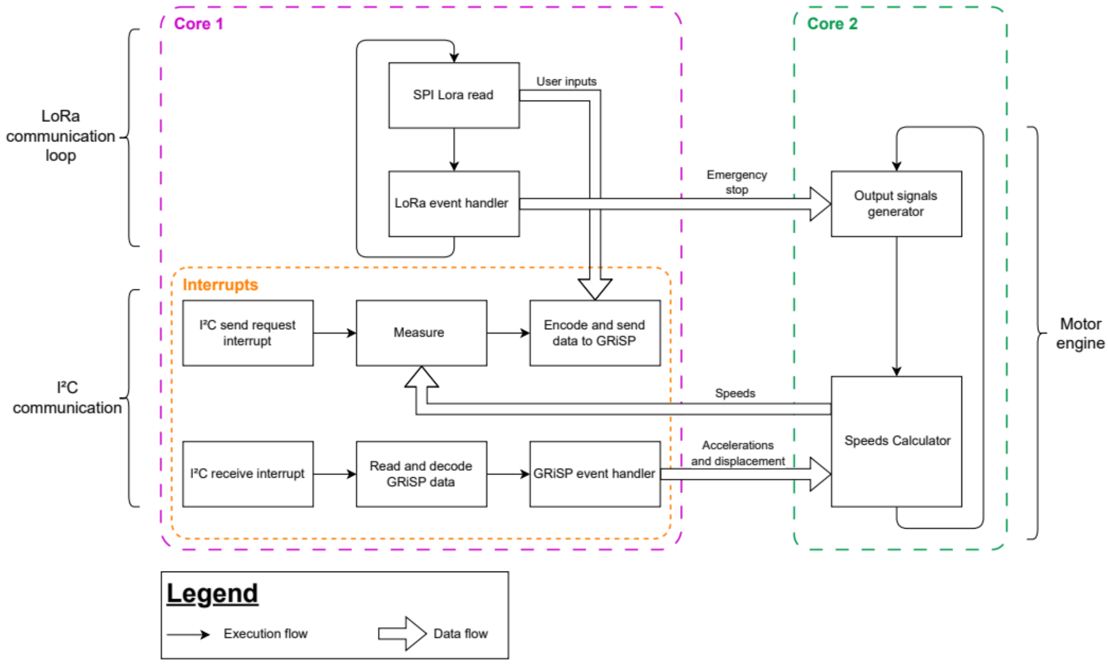


Figure 6.6: Overview of the ESP32 software architecture, showing task distribution across cores and communication interfaces [22].

The second core is responsible for executing the motor control logic (Figure 6.6). It is activated when an emergency stop is triggered or when movement and acceleration flags are received from the GRiSP. This core is divided into two parts: the signal generator and the speed calculator. The signal generator produces pulses at the correct frequency to drive the motor steps, manages the direction and activation signals needed to control the motors. The speed calculator computes the desired speed for each motor based on the acceleration and the differential speed. It then converts this speed into a step frequency using the following formula:

$$f = \frac{v}{2\pi r} \times \#\text{steps/turn} \times \#\text{microsteps}$$

Where v is the linear speed; r the wheel radius and the other terms represent the number of steps per turn and the number of microsteps of the stepper motors.

6.4 Server and user interface architecture

The last element of the global system architecture is the user interface (UI). This interface provides interactive control of the robot and is also designed to manage communication between the different GRiSP boards. To achieve this, the interface is structured into two main components: the UI and the server.

The graphical UI acts as the robot's control station. It operates as both a remote controller and a real-time graphical interface for the user. Through this interface, the operator can control the robot's movements, receive continuous feedback about its current state, view the control signals being applied and monitor messages transmitted from the robot to the server (figure A.1).

The server main purpose is to act as a network discovery and coordination hub for the GRiSP boards. In fact, the Hera framework does not allow GRiSP boards to send UDP broadcast messages. GRiSP boards cannot initiate communication without knowing the IP address of another board. The server is thus responsible for network discovery, aliveness monitoring, and logging. At start, it broadcasts its own IP address periodically over the network using a "ping" message. GRiSP boards that receive this message can extract the server's address and initiate a handshake process. During this process, the server stores the GRiSP board's information in a buffer. Once all three expected GRiSP boards have registered, the server sends the buffer containing details of all the devices on the network. This enables direct inter-GRiSP communication using Hera's unicast messaging model.

The server is also responsible for monitoring the different GRiSP boards. To achieve this, it maintains a dictionary of last-seen timestamps for each registered board. It then periodically checks if any board has been silent for more than 30 seconds. If so, it flags the GRiSP as lost and attempts to reestablish communication. The final use of the server is the logging system. GRiSP boards send debug logs using UDP messages. The server appends each of these logs to a file for later debugging.

Chapter 7

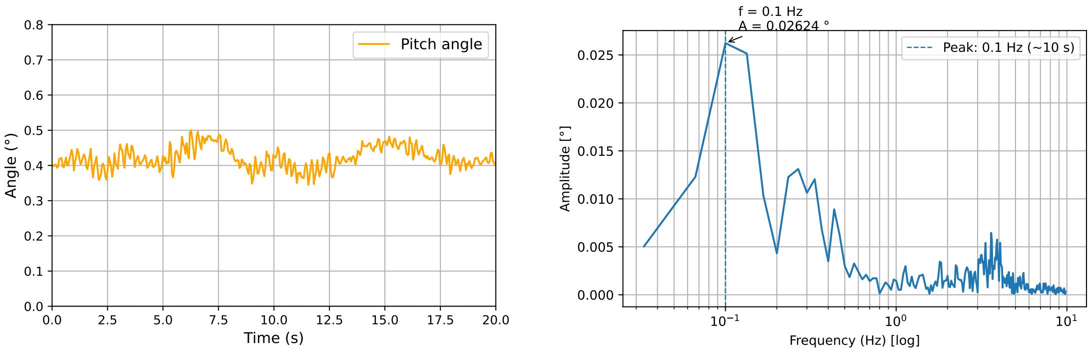
Evaluation

This chapter provides an evaluation of the robot's performance and identifies its limitations. The analysis is based on experimental measurements conducted under normal operating conditions, aiming to assess how the butler responds. Factors influencing the robot's behavior can originate from internal commands issued by the user or from external disturbances arising from the environment. Understanding these influences is essential to determine the system's operational boundaries and potential areas for improvement.

The first part of this evaluation focuses on the robot's stability in response to internal commands issued by the user. Tests are conducted under different scenarios: when the robot remains stationary, when moving forward, when turning, during transitions between static and dynamic modes, and while evaluating the performance of the obstacle-avoidance mechanism. The second part examines the robot's limitations. All analyses are based on the tilt angle estimated by the Kalman filter. The pitch angle from the Kalman filter is not a perfect reference, as it may be affected by sensor bias or suboptimal filter tuning, leading to small offsets or errors over time. Nevertheless, it remains sufficiently consistent to provide a meaningful basis for evaluating the robot's stability and identifying performance trends.

7.1 Stability reference case

The first test analyzes the stability of the robot in dynamic mode under normal conditions, meaning no external disturbances are applied and the test is conducted indoors on a flat parquet floor. This baseline scenario provides a solid point of comparison for other tests. Figure 7.1a shows that the robot's pitch angle oscillates between 0.3° and 0.5° with an oscillation of maximum 0.2°. Such a small variation is barely perceptible to the naked eye, indicating that the robot maintains a good degree of stability. It is also noticeable that the pitch angle does not oscillate around 0°. Instead it is centered near 0.4°, which corresponds to the equilibrium angle. This offset suggests that the robot is not perfectly mechanically symmetrical in height, likely due to the counterweight being slightly off-center or the Pmod NAV not being perfectly aligned with the vertical axis. However, this deviation is negligible as an equilibrium shift of 0.4° is too small to have any important impact on objects placed on the table top surface.



(a) Pitch angle variation over time when the robot is in a stable upright position.

(b) Fast Fourier transform of the pitch angle variation.

Figure 7.1: Analysis of the robot's pitch angle: (a) time-domain variation and (b) frequency-domain representation, under normal dynamic conditions.

Interestingly, the frequency-domain analysis of the pitch angle in this reference case, obtained using the Fast Fourier Transform (FFT) and shown in Figure 7.1b, reveals a distinct peak at 0.1 Hz with an amplitude of 0.02624°. This frequency corresponds to the natural frequency of the system in static conditions, with a period of approximately 10 seconds and a very small amplitude. Given this low oscillation, the observed pitch angle variation of 0.2° in the time domain is unlikely to originate from the control system itself. Instead, it is more plausibly explained by small structural flexibilities since the structure is not perfectly rigid or by external factors such as micro-irregularities on the floor surface.

7.1.1 Translation movement

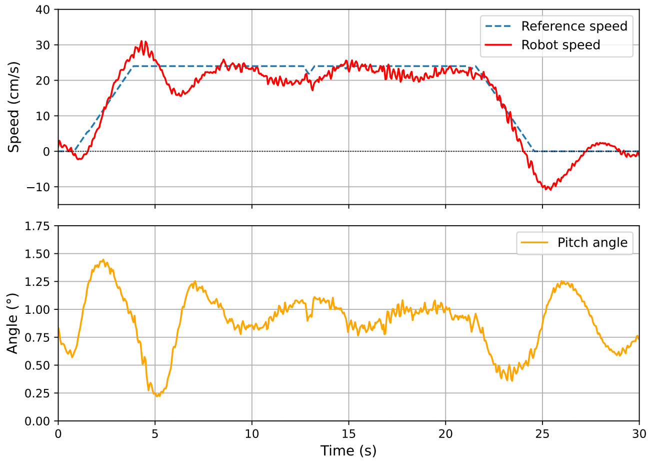


Figure 7.2: [Top graph] Robot speed variation and reference speed over time. [Bottom graph] Pitch angle variation over time when the robot is moving forward. The robot has an advancing speed of 24 cm/s and an advancing acceleration of 9 cm/s^2.

Figure 7.2 shows the variation of the robot's pitch angle when moving forward. Between 0 s and 4 s, the robot accelerates at a constant rate of 9 cm/s^2. During this acceleration, the pitch angle increases steadily, reaching a maximum of 1.4°. From 4 s to around 22 s, the robot maintains a nearly constant forward velocity of approximately 24 cm/s^2. After reaching this steady speed, the pitch oscillations gradually diminish and stabilize around 1°. Between 22 s and 25 s, the robot decelerates at the same rate as the acceleration when the forward command is released. The pitch angle exhibits increased oscillations during deceleration, although these are less pronounced than during acceleration. At around 2 s, the robot reaches a negative speed due to a slight backward motion caused by inertial effects during braking. This occurs because the robot's center of mass continues moving forward even after the wheels decelerate. This leads to a brief overshoot in the opposite direction before stabilization. After this point, the system begins to re-stabilize. In this forward movement test, the maximum variation in pitch angle is approximately 1.25°, with an absolute maximum angle of 1.5°. This is noticeably higher than the baseline case when the robot is stationary, likely due to the additional corrective torque required to counteract the inertial forces generated during forward motion. Nevertheless, this level of variation is well within acceptable limits for the intended application, as such small pitch angles do not displace objects on the tabletop.

7.1.2 Turning

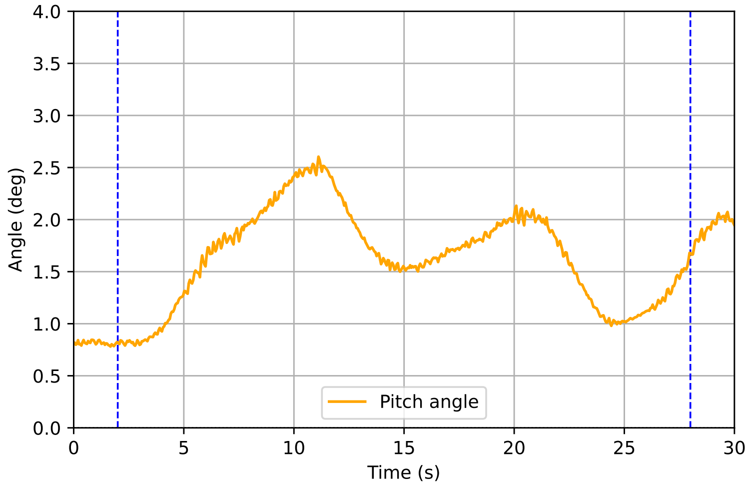


Figure 7.3: Pitch angle variation over time when the robot is turning for 25 seconds. The robot has a turning speed of 40 cm/s and a turning acceleration of 200 cm/s^2. The blue vertical lines designate the starting and ending of the turning operation.

For this test, the robot was continuously turning for 25 s. Figure 7.3 shows that the pitch angle variation displayed a clear sinusoidal pattern with a period of approximately 10 s and a peak-to-peak amplitude of about 1°. These oscillations may be linked to additional effects introduced during rotational movement such as, gyroscopic effect (i.e., rotation of the wheels generating a torque when their axis orientation changes), centripetal force (i.e., lateral force pulling the center of mass toward the turn's center), and the Coriolis effect (i.e., apparent deviation of internal motions within the rotating frame).

Another observation is that a larger maximum pitch angle of 2.5° was recorded compared to the other tests. This may be due to the turning velocity command not going through the controllers and being sent directly to the motor drivers. The PI controller is then not aware that it should adjust the equilibrium angle. As a result, it takes longer to compensate, allowing the pitch angle to increase further. Despite reaching a pitch angle of 2.5° and having a sinusoidal oscillation with a 10 s period, it remains acceptable for this application, as it does not significantly affect the stability of objects placed on the tabletop surface.

7.1.3 Dynamic/static mode transition

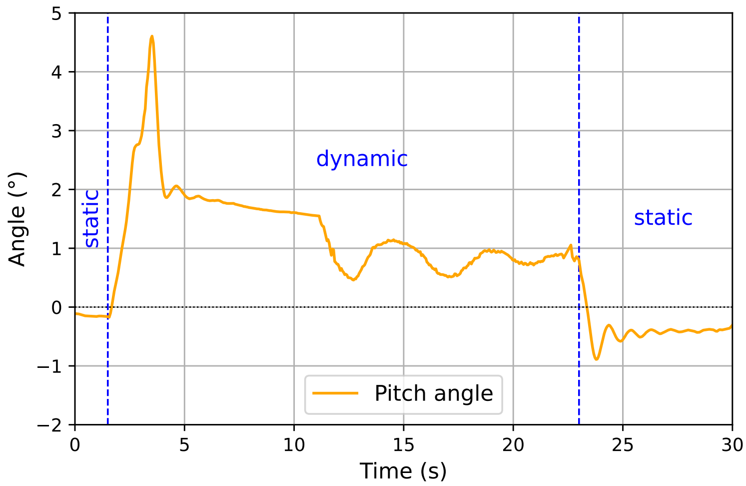


Figure 7.4: Pitch angle variation when the robot switches between the static to the dynamic mode and then again from the dynamic to the static.

Figure 7.4 shows that when transitioning from static to dynamic mode, the robot exhibits a pronounced peak in pitch angle of approximately 4.5° before settling around its equilibrium angle. This larger peak can be explained by the sudden release of the wheel locks in combination with the immediate engagement of the controllers, which must rapidly accelerate the wheels to achieve stability. The resulting inertial torque causes a brief overshoot before the control loop compensates. In contrast, the transition from dynamic to static mode also produces a peak in pitch variation, but of much lower magnitude. This can occur because the arms extend quickly but if the robot is not perfectly upright when they make contact with the ground, a small jolt occurs as one arm touches down before the other, causing this peak. The peak observed during the static-to-dynamic transition remains acceptable for the intended application. In practical testing with a full glass of wine, the wine oscillated slightly within the glass but not enough to spill a drop.

7.1.4 Performance of the obstacle avoidance mechanism

For this test, the robot was driven forward toward a couch, starting 4 meters away from it.The top graph of Figure 7.5 shows that at large distances (greater than 1.5,m), distance measurement errors appear. These errors fall below the braking threshold and cause false positives[1]. As a result, the robot believes it is close to an obstacle and brakes slightly, as seen in the speed graph. However, this effect is negligible, as the erroneous readings are too short in duration to produce a significant reduction in speed. At short distances (less than 1.5 meters), repeated tests showed that the sonars no longer produce incorrect readings. In this range, the braking behavior is consistent: the robot begins braking at 60 cm (the braking threshold) and continues decelerating until it stops at around 30 cm from the obstacle, leaving enough space for turning.

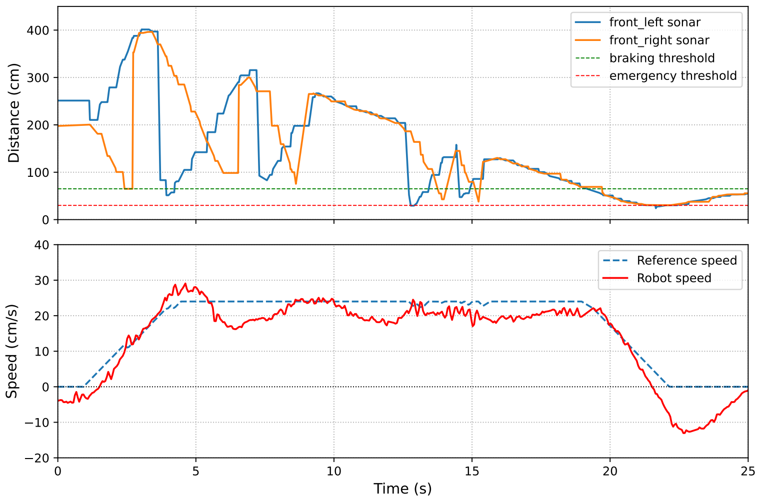


Figure 7.5: [Top graph] Sonar measures of the two front sonars. [Bottom graph] Robot speed variation and reference speed over time when the robot is going forward in the direction of an obstacle.

The distance measurement errors at long distances are likely caused the alternating operation of the two sonars. Long-distance measurements can be distorted by residual echoes or cross interference. In fact, the echo from a previous transmission may still be present when the other sonar is firing, or may be picked up by it. Results from a similar test for the backward sonar (Appendix L.1) are consistent with this hypothesis as it does not exhibit that kind of error because it operates on its own. An attempt to reduce this interference was made by increasing the delay between sonar firings and by implementing filters, such as the Hampel filter [31]. This did not help, as it introduced measurement errors at short distances. Therefore, occasional false positives were accepted rather than risking false negatives[2].

[1]False positive: the sonar detects an obstacle when there is none at the measured distance, causing the robot to brake unnecessarily.

[2]False negative = the sonar fails to detect an obstacle that is actually within the critical

7.2 Limitations

The aim of this section is to study the operational limits of the butler. There are many factors limiting the stability of the robot. A few were tested. Their characterization is divided into two categories:

- **External limitations** refer to external influences on the robot, such as the type of floor or the maximum payload it can carry.
- **Internal limitations** refer to parameters that are directly part of the control loop. This includes speed and acceleration commands and the processing frequency.
- **Additional limitations** refer to additional limitations in software and hardware components (e.g., sensor interference, process crashes, unsecured communications), along with some potential solutions.

7.2.1 External limitations

7.2.1.1 Types of floors

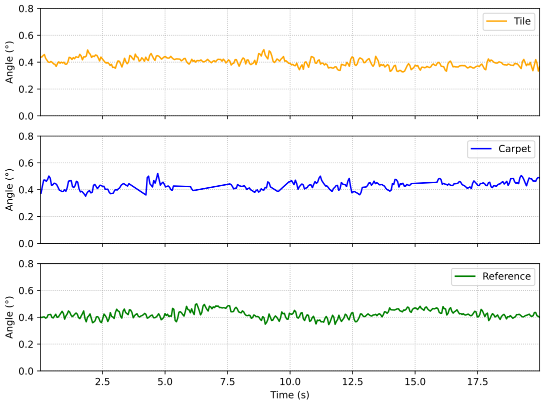


Figure 7.6: Pitch angle variation for 3 different types of floors tested when the robot is in a stable upright position. [Top graph] Pitch angle variation for tile floor. [Middle graph] Pitch angle variation for carpet floor. [Bottom graph] Pitch angle variation for parquet floor.

This test analyzes the robot's stability on three different types of floors. The reference case tests were conducted on a parquet floor. In addition to parquet,

distance, meaning the robot does not brake when it should.

tests were performed on carpet and tile floor, which are common in domestic environments. As shown in Figure 7.6, when no velocity command is issued, the robot's dynamic stability remains consistent across all tested surfaces, with no greater pitch variation observed. This demonstrates that the system is well adapted to operate on multiple floor types. Tests with internal commands issued by the user produced similar results, confirming that stability performance is not significantly affected by the type of floor. From a mechanical perspective, the limited influence of surface type can be explained by the robot's wheels, which provide sufficient grip and rolling efficiency on a variety of common indoor surfaces.

7.2.1.2 Impact of the payload

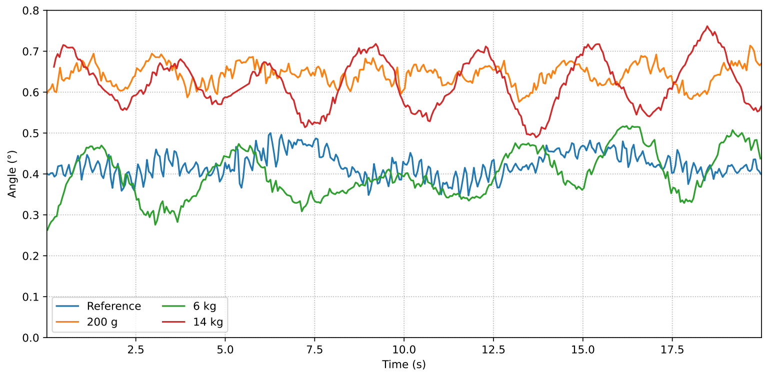


Figure 7.7: Pitch angle variation for the different payloads tested on the robot when the robot is at stable upright position.

This test evaluates the stability of the robot when payload is added. The first trial was conducted with a full glass of wine (200 g). Compared to the reference case, stability remains unchanged except that the pitch angle now oscillates around 0.65° instead of 0.4° (Figure 7.7). Across all payload tests, this offset varied slightly, most likely because the added weights were not placed exactly at the center, resulting in a shift in weight distribution.

The different payload tests demonstrated that the robot remains stable with up to 14 kg of payload, 2.8 times the weight of the robot. A slight increase in pitch angle variation was observed, although it was imperceptible to the naked eye. At 14 kg, The system remained stable, but the chassis began to deform slightly, indicating that the maximum weight capacity of the physical system had been reached. Pictures of the test setup can be found in Appendix K.

7.2.2 Internal limitations

7.2.2.1 Speed and acceleration commands

The selection of speed and acceleration commands required a careful compromise between performance and stability. On the one hand, the robot needed to move fast enough to be practical, while on the other, excessive speed reduced stability. The maximum achievable speed was found to be 60 cm/s, limited by an electromagnetic stall between the stepper motor's rotor and stator. Although higher speeds could theoretically be reached by increasing the motor current, this would cause the drivers to overheat and was therefore not considered viable. Acceleration proved to be an even more critical parameter. A higher acceleration allows the robot to reach its maximum speed quickly, but excessive values introduce large inertial shocks during acceleration and braking, which compromise stability. Moreover, the choice of speed and acceleration directly affects the braking distance: if too long, the robot risks colliding with obstacles unless the braking threshold is set unreasonably far away. After iterative testing, a speed of 24 cm/s combined with an acceleration of 9 cm/s^2 was selected. This configuration offered a good balance between responsiveness and stability. It also ensured a good braking distance for the obstacle avoidance system.

7.2.2.2 Processing frequency

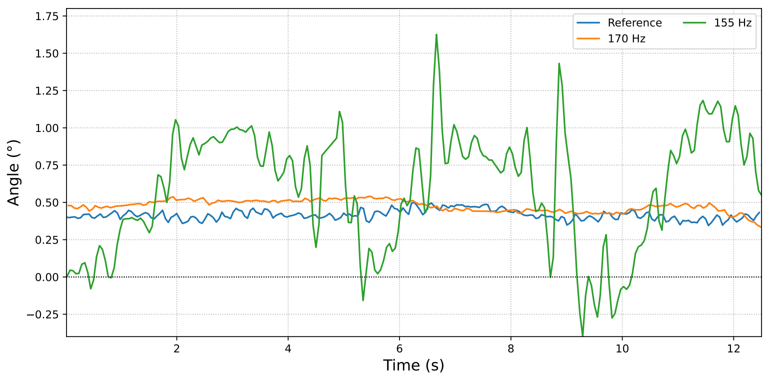


Figure 7.8: Pitch angle variation for the different frequencies tested on the robot when it is in a stable upright position.

The system's frequency is a critical constant. A low update frequency implies an unresponsive system and therefore compromises dynamic balancing. Tests have shown that the robot remains very stable between 220 (reference case) Hz and 200 Hz. Between 200 Hz and 180 Hz, the system is still stable but feels less reactive to user inputs. At lower frequencies, from 170 Hz to 155 Hz, the system becomes marginally stable, with large-amplitude oscillations. Under these conditions, the robot remains upright but is no longer able to withstand disturbances. Below 155 Hz, the robot is totally unstable and no longer able to maintain its equilibrium. The behavior of the robot in standstill condition at various frequencies (Figure 7.8).

7.2.3 Additional limitations

First, the Pmod NAV is not perfectly fixed in his position. In fact it often slightly moved in its pins due to the robot's vibrations inducing some small tilting, which directly impacted the obtained pitch angle. In addition, as it is located at 90 cm, it is more susceptible to fluctuations in the chassis. As a result, the calibration at start-up was not always sufficient and a small bias could appear, causing the robot to drift. A potential solution to this is to add a second Pmod NAV on one of the GRiSP boards at the bottom of the robot. With two IMUs, it would be possible to select the more stable signal or to fuse the two measurements for reduced drift and improved robustness.

Another limitation of the Pmod NAV is its occasional unexpected crashes. The ESP32 also crashed following abrupt power outages. Following these events, the GRiSP did not always detect the ESP32 upon reboot. A manual reboot was necessary for it to work properly again. These problems could be solved by taking greater advantage of the Erlang supervision architecture to implement automatic reboots of the affected processes.

7.3 Materials and costs

To evaluate the feasibility of commercializing this robot, it is essential to consider its total cost. The breakdown of the components and their prices is shown in Table C.1, with a total cost of 1,068.67 €. This cost is relatively high but the current version is a prototype. Thus, significant cost reductions could be achieved through bulk purchasing and the selection of less expensive components.

The three GRiSP 2 boards are the most expensive items. While these boards offer a wide range of functionalities, many of these are not required for this application. For instance, the GRiSP boards connected to the sonar modules can perform at a high level of efficiency, but the system only needs to take distance measurements every 50 ms and transmit this data. There are many unused features, such as multiple Pmod ports, SPI and UART interfaces, high-throughput networking capabilities, and additional GPIO pins. Even the processing power and memory capacity of the GRiSP boards far exceed the requirements for basic sonar acquisition and transmission. Developing and sourcing customized GRiSP boards designed specifically for the project's requirements could substantially reduce the overall cost. Similar optimization strategies could be applied at a larger scale to every part of the robot. Some components could be replaced with more cost-effective alternatives while maintaining the required performance.

7.4 Power consumption

An important point is the power consumption of the robot because it indicates how long the robot can run in its different modes without charging. The robot is powered by a 12.8 V, 32,000 mAh lithium battery pack, corresponding to approximately 410 Wh of usable energy. Power is distributed through two buck converters. The first delivers 5 V to the main GRiSP board (0.28 A), two auxiliary GRiSP boards (0.25 A each), and the LilyGO module (0.06 A). Together, these electronics draw 0.84 A at 5 V, or about 4.2 W. Considering a converter efficiency of 90%, this represents about 4.7 W from the battery.

The second buck converter provides 10 V to three TMC2208 stepper drivers powering NEMA17 motors. In dynamic mode, each motor consumes on average 0.7 A at 10 V, for a total of 21 W. With an assumed 90% efficiency, this corresponds to 23.3 W from the battery, which results in an autonomy of approximately 15 hours in dynamic mode. In static mode, the motors remain powered to lock the wheels, but their holding current (0.3 A) is reduced. The average draw is estimated at 9 W at the load or 10 W from the battery. This gives a autonomy in static mode of 28 hours. The power consumption results and the calculated autonomy are already impressive for a first domestic robot prototype. However, from a more practical and long-term perspective, it means the robot needs to be recharged every day, which is somewhat limiting.

7.5 Comparison with the previous master's thesis

As a reminder, this robot was a prototype developed based on the robot of François Goens and Cedric Ponsard. A small comparison of the performance of the two robots was performed and highlights interesting insights.

First, for similar reference cases where the two robots stand upright on a flat floor without any external disturbance, their smaller robot exhibited a pitch angle oscillation of less than 0.1°, while this taller robot oscillated around 0.2°. The slight increase is due to the greater height and weight of the robot, which make perfect stabilization more difficult. However, it is important to keep in mind that these oscillations are not visible to the naked eye and that a design trade-off had to be made between payload capacity and static stability. Then, during translational movement, both robots exhibit very similar behavior, with a slight speed overshoot and small angle variation. The speed overshoot is relatively smaller for the butler, due to the reduced acceleration. In fact, to allow stabilization of this taller robot, the acceleration was significantly reduced compared to theirs, being 9 cm/s^2 and 75 cm/s^2 respectively. Additionally, the behavior of the two robots at different frequencies is interesting to compare. In their master's thesis, the robot was capable of balancing at a frequency as low as 75 Hz, which is not the case here. In fact, the limiting frequency in this system is 155 Hz. This clearly shows that the increase in size and inertia at the top of the robot necessitates a higher control frequency. If the weight at the top is larger, a low frequency, meaning a longer interval between loop iterations, induces a larger "fall" movement at the tip of the robots, which is more difficult to counteract. Therefore, a higher frequency becomes inevitable to sustain stability, whereas for their smaller robot, the reduced frequency has less impact.

In their master's thesis, François Goens and Cedric Ponsard performed a change-in-inertia test, where they added a book on their robot. The results show that the robot stays upright but with large oscillations, highlighting that their robot was not adapted to inertia changes. In comparison, the butler, as demonstrated above, is capable of maintaining balance with a 14 kg payload. This confirms the design compromise made in this work, where priority was given to payload capacity over absolute static stability. It even far exceeded expectations, as the initial objective was simply to carry a full glass of wine.

Chapter 8

Conclusion and future works

The main objective of this thesis was to develop a practical device for domotics using a self-balancing two-wheeled robot prototype. This device required additional functionalities such as stable payload transport and obstacle avoidance capabilities. To achieve this goal, a fully designed two-wheeled butler was created. While keeping elements of the original robot that performed well, such as the motor and power hardware, the ESP32 design, the Kalman filter-based sensor fusion, the cascade PID stability loop, and the Hera framework implementation, the hardware and software were redesigned and expanded.

In terms of hardware, a taller, more robust chassis with a platform was constructed, retractable support arms were integrated, and three ultrasonic sonars were added to detect obstacles. Regarding the software, the existing stability framework to account for the physical model using an extended Kalman filter. The PID parameters were tuned, and a multi-sonar obstacle avoidance mechanism was integrated. Additionally, an increased control-loop frequency was attained by taking advantage of the Hera measurement framework strengths. The architecture was refactored into a distributed, multi-GRiSP system to enable synchronized sensor data exchange. Two operating modes were developed: a dynamic self-balancing mode for motion and a static mode with extendable wheels for increased stability and energy savings.

A series of tests were performed to evaluate the butler's performance under normal conditions and identify its limits. The experiments demonstrated stable operation with various payloads, ranging from a full glass of wine to 14 kg, without any loss of balance. The obstacle avoidance mechanism was verified through realistic scenarios. It confirmed the reliability of the braking and turning near obstacles despite occasional long-range sonar errors. The robot was tested on multiple surfaces and demonstrated that it maintained stability regardless of the flooring material. Finally, the measurement of the power consumption revealed an autonomy of approximately 15 hours in dynamic mode and 28 hours in static mode. This is promising for a prototype. However, it indicates the need for daily recharging for long-term use, which could be problematic. Several limitations were also identified. These include Pmod NAV sensitivity issues and the need for occasional calibration. Sonar-based obstacle detection was subject to interference. There were also ESP and Pmod module crashes that required manual restarts and the ESP's LoRa communication protocol lacked security. These issues highlight areas for improvement in future iterations of the system. Overall, this thesis has demonstrated the successful transformation of a two-wheeled robot prototype into a functional self-balancing butler capable of carrying heavy payloads, avoiding obstacles, and operating in realistic domestic environments. The robot's performance exceeded the objectives and expectations of this thesis, proving its potential as a foundation for future domestic robotic applications.

Looking ahead, several promising directions could further enhance the system. One important improvement would be the implementation of a low-power safety mode. This mode would allow the robot to stabilize or lock itself safely when the battery is nearly depleted. Another improvement would be the development of an adaptive controller that dynamically adjusts to payload variations, thereby reducing the need for static counterweights. The obstacle avoidance system could be more sophisticated to enable smoother and more reliable decision-making. One approach could be to combine different types of sensors, for example ultrasonic, infrared, or LiDAR, to increase reliability and reduce false detections. Another approach would be to implement Simultaneous Localization and Mapping (SLAM), which would allow the robot not only to avoid nearby obstacles but also to build a map of its environment and plan optimized trajectories. At the system level, fault detection and graceful-degradation strategies across GRiSP nodes could be implemented to mitigate sudden failures. Finally, by refining the industrial design and improving energy management, it would be possible to extend runtime and enhance the robot's safety and usability in long-term domestic operation.

Bibliography

[1] Erlang/OTP Team. *The Erlang Programming Language*. Accessed July 2025. Erlang.org. URL: https://www.erlang.org/.

[2] Wikipedia contributors. *Hot swapping*. Accessed July 2025. URL: https://en.wikipedia.org/wiki/Hot_swapping.

[3] Wikipedia contributors. *Erlang (programming language)*. Accessed July 2025. Wikipedia. URL: https://en.wikipedia.org/wiki/Erlang_(programming_language).

[4] Adabeat. *Why use Erlang for your next project*. Accessed July 2025. 2024. URL: https://adabeat.com/insight/why-use-erlang-for-your-next-project/.

[5] Rebar3 contributors. *Rebar3 Documentation*. Accessed July 2025. URL: https://www.rebar3.org/.

[6] Rebar3 contributors. *Rebar3*. Accessed July 2025. URL: https://github.com/erlang/rebar3.

[7] Lylian BRUNET and Basile COUPLET. “The best of both worlds–Fast numerical computation in Erlang”. MA thesis. Ecole polytechnique de Louvain, 2022. URL: https://thesis.dial.uclouvain.be/entities/masterthesis/8527cb73-6ebe-43bb-b5f7-0c7df3f5361a.

[8] Peer Stritzinger GmbH. *Technology for Embedded Systems, Industrial Automation & IoT Security*. Accessed June 2025. URL: https://stritzinger.com.

[9] GRiSP contributors. *Powering Embedded Systems with BEAM*. Accessed June 2025. Grisp.org. URL: https://www.grisp.org.

[10] GRiSP contributors. *GRiSP 2–Scalable Prototyping for Embedded Systems*. Accessed July 2025. Grisp.org. URL: https://www.grisp.org/hardware#grisp2-details-section.

[11] Digilent Inc. *Pmod NAV Reference Manual*. Accessed July 2025. 2017. URL: https://digilent.com/reference/_media/reference/pmod/pmodnav/pmod_nav_rm.pdf.

[12] Northern Digital Inc. *6DOF Explained*. Accessed July 2025. 2023. URL: https://www.ndigital.com/6dof-explained/.

[13] Digilent Inc. *Pmod NAV - 9-axis IMU Plus Barometer*. Accessed July 2025. 2024. URL: https://digilent.com/reference/pmod/pmodnav/start.

[14] Digilent Inc. *Pmod MaxSonar Reference Manual*. Accessed July 2025. 2024. URL: https://digilent.com/reference/pmod/pmodmaxsonar.

[15] Wikipedia contributors. *Communication protocol — Wikipedia, The Free Encyclopedia*. Accessed July 2025. 2024. URL: https://en.wikipedia.org/wiki/Communication_protocol.

[16] Texas Instruments. *Introduction to the I2C Bus*. Tech. rep. SBAA565. Accessed July 2025. Texas Instruments, 2022. URL: https://www.ti.com/lit/an/sbaa565/sbaa565.pdf.

[17] Newhaven Display. *I2C Communication Interface*. Accessed July 2025. 2021. URL: https://newhavendisplay.com/fr/blog/i2c-communication-interface/.

[18] Tobias Weltner. *SPI (Serial Peripheral Interface) – High-Speed Interface for Connecting Data-Hungry Peripherals to Microcontrollers*. Accessed July 2025. May 2024. URL: https://done.land/fundamentals/interface/spi/.

[19] Admin. *SPI Protocol - Prodigy Technovations*. Accessed July 2025. May 2023. URL: https://www.prodigytechno.com/spi-protocol#:~:text=SPI%20Protocol%3A%20Introduction&text=SPI%20is%20a%20full%20duplex,takes%20instructions%20from%20the%20master..

[20] Kaouthar Draif. "Les protocoles de communication SPI, I2C et UART - Moussasoft". In: (Apr. 2024). Accessed July 2025. URL: https://www.moussasoft.com/les-protocoles-de-communication-spi-i2c-et-uart/#:~:text=%C5%93uvre%20du%20syst%C3%A8me.-,Protocole%

20de%20communication%20SPI,une%20ligne%20de%20s%C3%A9lection%20distincte..

[21] Sebastien Kalbusch, Vincent Verpoten, and Peter Van Roy. "The Hera Framework for Fault-Tolerant Sensor Fusion with Erlang and GRiSP on an IoT Network". In: *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*. Erlang '21. Virtual, Republic of Korea: Association for Computing Machinery, Aug. 2021, pp. 1–13. ISBN: 978-1-4503-8612-8. DOI: 10.1145/3471871.3472962. URL: https://doi.org/10.1145/3471871.3472962.

[22] Goens Francois and Ponsard Cedric. "Dynamic balancing in the real world with GRiSP". Master thesis. Universite catholique de Louvain, 2024. URL: https://thesis.dial.uclouvain.cbe/entities/masterthesis/c11f8a20-c183-4f9e-b3ec-b14248faa61a.

[23] Wikipedia contributors. *Kalman filter*. Accessed July 2025. Wikipedia. URL: https://en.wikipedia.org/wiki/Kalman_filter.

[24] Wikipedia contributors. *Extended Kalman filter*. Accessed July 2025. Wikipedia. URL: https://en.wikipedia.org/wiki/Extended_Kalman_filter.

[25] TutorialsPoint contributors. *Control Systems–Introduction*. Accessed April 2025. TutorialsPoint. URL: https://www.tutorialspoint.com/control_systems/control_systems_introduction.htm.

[26] Wikipedia contributors. *Proportional–integral–derivative controller*. Accessed April 2025. Wikipedia. URL: https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller.

[27] *The PID controller & theory explained*. Accessed July 2025. Aug. 2006. URL: https://www.ni.com/en/shop/labview/pid-theory-explained.html?srsltid=AfmBOopRRGCQsFCgzFRvkS_ZBbMBgdX6HjITpx3cFv-rsnJwIHNeEnA3.

[28] Wikipedia contributors. *Ziegler-Nichols method*. Accessed May 2025. URL: https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method.

[29] Lauren Fuentes. *3D Printer Stepper Motor: All You Need to Know*. All3DP. Mar. 10, 2022. URL: https://all3dp.com/2/3d-printer-stepper-motor-what-to-consider-and-which-to-choose-2/ (visited on 08/16/2025).

[30] GeeksforGeeks. *Round Robin Scheduling in Operating System*. Accessed 28 Jul, 2025. 2025. URL: https://www.geeksforgeeks.org/operating-systems/round-robin-scheduling-in-operating-system/.

[31] Ronald K. Pearson et al. "Generalized Hampel Filters". In: *EURASIP Journal on Advances in Signal Processing* 2016 (2016). Article 87, pp. 1–18. DOI: 10.1186/s13634-016-0383-6. URL: https://link.springer.com/article/10.1186/s13634-016-0383-6.

[32] G. Campion J.-C. Samin and P. Maes. *FSAB 1202 Exercices de Mecanique*. UCLouvain, 2008.

Appendix A

User interface

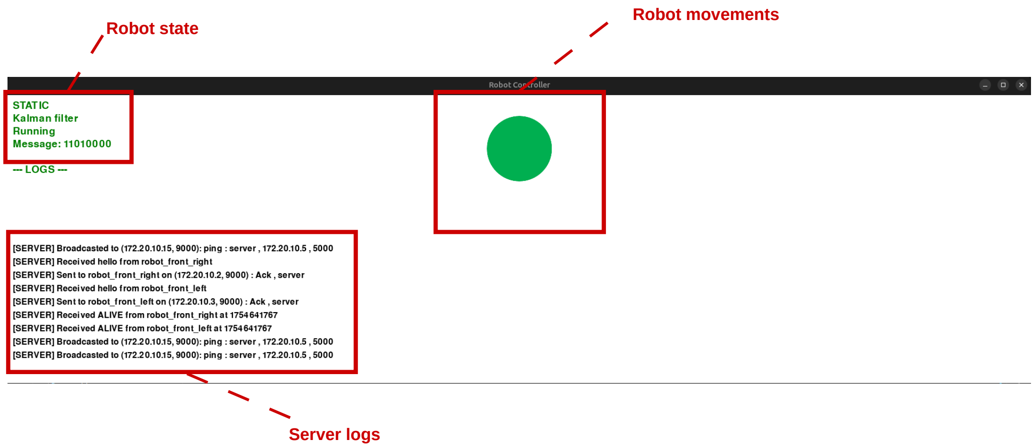


Figure A.1: User interface for robot control and logs visualization.

Appendix B

Evaluation static support system

o ensure that the robot remains stable when the support arms are deployed, an evaluation was made to determine whether it would still resist tipping with an additional load placed on top. For this purpose, a few assumptions were made: The two support arms are located at a height of 8.1 cm from the ground and extend 12.5 cm laterally on each side from the center, resulting in a total support base of 25 cm. This structure forms a triangular contact area with the ground on each side.

The effectiveness of this design was evaluated by calculating the critical tipping angle based on the robot's center of mass (CoM) and the dimensions of the support base. Since the robot is symmetrical, only the vertical position of the CoM needed to be determined. This was done using the balancing method, also known as the static equilibrium or bascule method: the robot was placed horizontally on a narrow edge and slowly moved until it reached a stable, balanced position. At that point, the center of mass lies directly above the edge. By measuring the vertical distance from the center of the wheels to the balance point, the height of the CoM was estimated. Initially, the CoM of the robot alone was found to be 41 cm. To determine the combined CoM of the robot and an additional payload placed at its top, the classical formula for the CoM of a system of discrete masses was applied as follow:

$$z_{\text{CoM}} = \frac{\sum_i m_i z_i}{\sum_i m_i}$$

where:

- m_i is the mass of the i-th component,
- z_i is the vertical position of the center of mass of the i-th component,

- z_{CoM} is the resulting combined center of mass.

The robot has a mass $m_r = 5\,\text{kg}$ and its center of mass is located at $z_r = 0.41\,\text{m}$. An additional payload of $m_p = 0.5\,\text{kg}$ is placed at the very top of the robot, at a height of $z_p = 0.9\,\text{m}$. Applying the formula:

$$z_{\text{CoM}} = \frac{m_r z_r + m_p z_p}{m_r + m_p} = \frac{(5 \times 0.41) + (0.5 \times 0.85)}{5 + 0.5} = \frac{2.05 + 0.425}{5.5} = 0.45\,\text{m}$$

Thus, the combined center of mass of the loaded robot is located at approximately 45 cm from the center of the wheels. The system can be analyzed using the principle that tipping occurs when the vertical projection of the center of mass lies outside the support polygon. The critical tipping angle θ is given by:

$$\tan(\theta) = \frac{\text{half base support}}{\text{height of CoM}} = \frac{0.125}{0.450} \Rightarrow \theta \approx 15.4^\circ$$

This means the robot can be tilted by up to 15.4° before the center of mass exceeds the boundary defined by the support arms, thereby triggering a fall. As illustrated in Figure B.1, the red dot represents the CoM of the loaded robot, and the red dashed line shows its vertical projection. The blue dashed line indicates the tipping boundary, the point where the robot would begin to lose balance if tilted further.

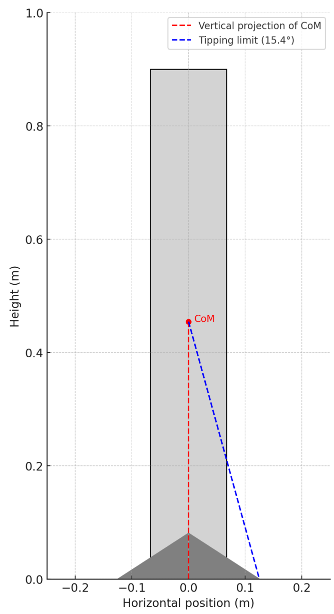


Figure B.1: Side view of the robot with the static support system. The support arms form a triangular base preventing tipping, even under a top load.

Appendix C

Materials and costs table

Table containing the materials used for the building of the robot along with their costs. It is also important to note that shipping fees were not included in the calculation, as the components were not ordered all at once and no cost optimization was applied for combined deliveries. As a result, the presented total is slightly lower than the actual cost of the prototype.

Component	**Cost [€]**	**Quantity**	**Total [€]**
Mechanical Components			
9 Laser-Cut Wooden Pieces (2 x 6 mm Wooden Boards)	3.30	2	6.60
36 Custom PLA Pieces	16 €/kg	± 790g	12.64
2 arms resin Clear v4 Pieces	169.40 €/L	± 0.2L	33.88
Wheels	11.90	2	23.80
Stepper Motor Attachments	3.00	2	6.00
Round Non-Slip Serving Tray	5.51	1	5.51
Counterweight	0.70	1	0.70
Metal Rods	2.50	4	10.00
Arm Wheels	2.20	4	8.80
Screws, Nuts, and Inserts	3.00	1	3.00
		Mechanical Subtotal	**110.93**

Electrical Components			
Stepper Motors (NEMA17)	10.52	3	31.56
GRiSP 2 Board	213.00	3	639.00
Buck Converters	6.99	2	13.98
12.8V $LiFePO_4$ Battery	65.00	1	65.00
Stepper Drivers	4.20	3	12.60
LilyGO LoRa32	18.75	2	37.50
Custom PCB	4.00	1	4.00
Micro USB Cables	6.00	4	24.00
Jumper Wires Pack	5.97	1	5.97
Pmod NAV	30.00	1	30.00
Pmod MAXSONAR	30.00	3	90.00
		Electrical Subtotal	**953.61**
Machine Usage			
30 Minutes Laser Cutting	4.13	1	4.13
		Machine Usage Subtotal	**4.13**
Total			**1068.67**

Table C.1: Table of materials for the butler robot and their costs.

Appendix D

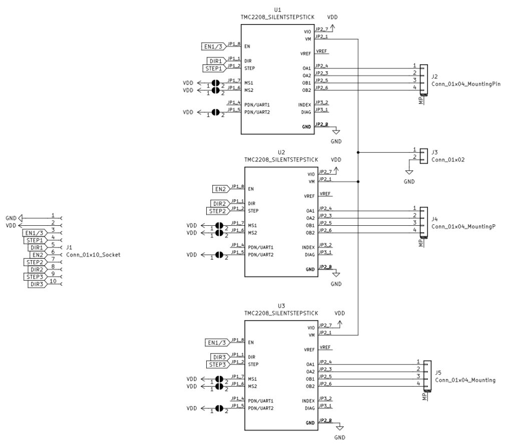
Sonar placement optimization code

```python
import math as mt

results = {}

# Distance and angle bounds
initial_distance1 = 145 / 2  # maximum distance to center in mm
initial_distance2 = 52.5     # minimal distance to the wheels in mm
max_delta = int(initial_distance1 - 1)  # minimum 1 mm between the sonars

# Loop over the distance offset (x in mm)
for x in range(0, max_delta + 1):
    distance1 = initial_distance1 - x
    distance2 = initial_distance2 + x

    # Loop over the angle alpha (from 0 to 12 degrees)
    alpha = 0
    while alpha <= 12:
        angle1 = 75 - alpha
        angle2 = 75 + alpha
        h1 = mt.tan(mt.radians(angle1)) * distance1
        h2 = mt.tan(mt.radians(angle2)) * distance2
        big_h = max(h1, h2)

        results[(alpha, x)] = big_h
        alpha += 0.5

optimal_config = min(results, key=results.get)
optimal_height = results[optimal_config]
```

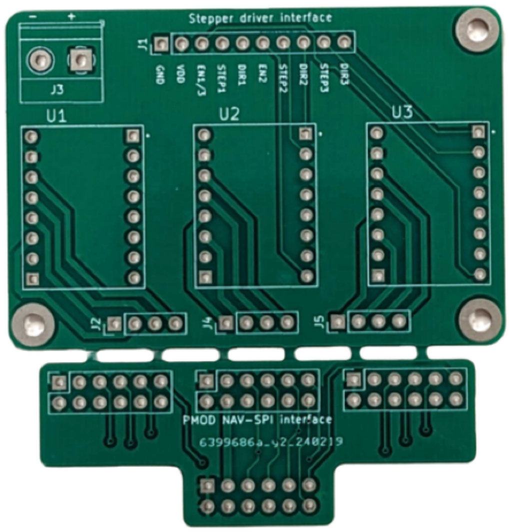
```
print(f"Meilleure configuration : angle = {optimal_config[0]}, decalage distance = {optimal_config[1]} mm")
print(f"Hauteur maximale minimale : {optimal_height:.2f} mm")
```

Appendix E

PCB design



(a) Electrical diagram of the stepper motor driver interface PCB (Kicad) [22].



(b) Physical PCB manufactured by JL-CPCB [22].

Figure E.1: Schematics and physical implementation of the PCB for stepper motor control.

Appendix F

Physical modelling

The purpose of this physical model is to give the relation between the input and the output of the physical system of the robot. As our robot is a physical extension in height of the robot developed in the Master's thesis by Ponsard and Goens [22], we were able to reuse the same physical model and underlying hypotheses, which are detailed below.

Hypotheses

Some hypotheses are necessary to simplify the physical model of the robot. They allow the derivation of a more simple mathematical model that can be used in the design of the Kalman filter. The following hypotheses are assumed for the development of the physical model [22]:

1. The robot behaves as a rigid body, with no structural deformation during motion.
2. The mass of the wheels is negligible and their rotational inertia is neglected.
3. Wheel slip is assumed to be negligible, so it is assumed a perfect rolling contact with the ground.
4. The effect of the torque generated by the wheels on the robot is negligible.
5. The translational motion resulting from changes in the robot's tilt is considered negligible, allowing a decoupling of rotational and translational dynamics.
6. Only two external forces are taken into account: the gravitational force and the ground reaction force, which includes static friction.

Scheme of the robot

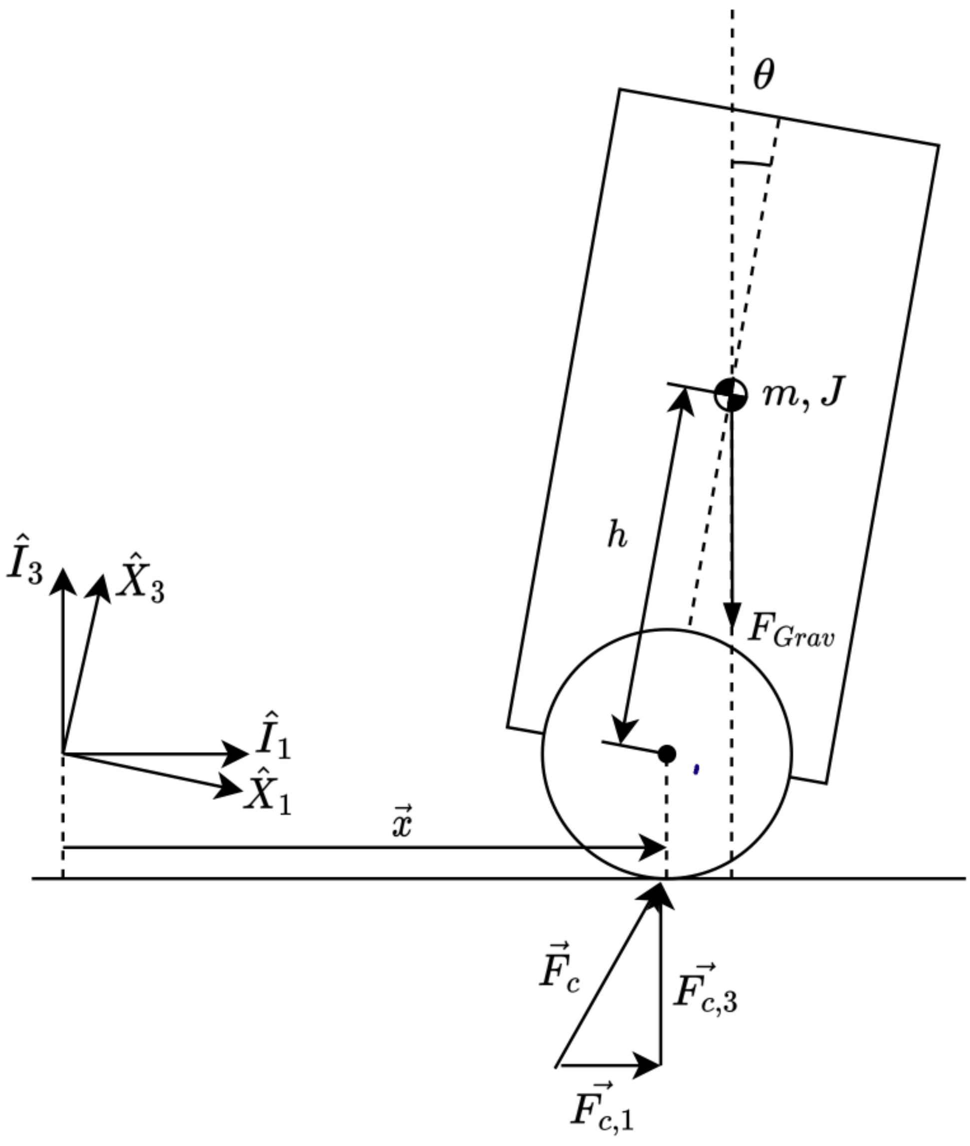


Figure F.1: Reference scheme of the robot for developing the physical model of the system [22].

Symbol	Description
θ	Tilt angle of the robot relative to the vertical axis [rad]
$\dot{\theta}$	Angular velocity of the robot [rad/s]
$\ddot{\theta}$	Angular acceleration of the robot [rad/s^2]
x	Position of the robot in the global frame [m]
$\ddot{x}$	Linear acceleration of the robot in the x-axis [m/s^2]
$\vec{F}$	Ground contact force applied to the robot [N]
h	Distance between the center of gravity (CG) and the rotation axis of the wheels [m]
m	Total mass of the robot [kg]
J	Moment of inertia of the robot at its center of mass [kg $\cdot$ m^2]
g	Gravitational acceleration [m/s^2]

Table F.2: List of variables and constants used in the physical model [22].

The model relies on two distinct reference frames. The first frame, attached to the robot, is defined by the axes $\hat{X}_3$ and $\hat{X}_1$. The axis $\hat{X}_3$ is aligned with the line connecting the center of gravity and the wheel axle, while $\hat{X}_1$ is perpendicular to $\hat{X}_3$ and oriented toward the front of the robot. The second frame corresponds to the ground reference frame, defined by the axes $\hat{I}_1$ and $\hat{I}_3$. In this frame, $\hat{I}_1$ is parallel to the ground, and $\hat{I}_3$ is perpendicular to it and oriented upwards.

Appendix G

Newton-Euler equations

This appendix presents the full derivation of the robot's mathematical representation using the Newton-Euler formalism. The goal is to obtain a compact expression linking the robot's angular acceleration to external inputs and forces. The derivation follows the approach proposed in ([22],p 89-91) and is inspired by classical mechanics principles presented in ([32], p.60).

The Newton-Euler formalism combines Newton's second law (force balance) and Euler's equation (torque balance) to describe the motion of a rigid body. To apply this method, we first need to compute:

- The position and acceleration of the center of gravity (CG),
- The angular momentum and its derivative,
- The external forces acting on the body,
- The torques applied about the CG.

1. CG position vector and derivatives

The position of the center of gravity is:

$$\vec{R} = x\hat{I}_1 + h\hat{X}_3$$

Using derivative rules in rotating frames:

$$\dot{\vec{R}} = \dot{x}\hat{I}_1 + h\dot{\theta}\hat{X}_1$$

$$\ddot{\vec{R}} = \ddot{x}\hat{I}_1 + h\ddot{\theta}\hat{X}_1 + h\dot{\theta}^2\hat{X}_3 \qquad \text{(D.1)}$$

Those are found from the vector derivative of a mobile basis formula :

$$\frac{d\vec{u}}{dt} = \left(\frac{d\vec{u}}{dt}\right)_{\text{rot}} + \vec{\omega} \times \vec{u}$$

2. Angular momentum relative to the CG and its derivative

$$\vec{H}_G = J\dot{\theta}\hat{X}_2, \quad \dot{\vec{H}}_G = J\ddot{\theta}\hat{X}_2 \tag{D.2}$$

3. External forces acting on the robot

- Gravity: $\vec{F}_g = -mg\hat{I}_3$
- Ground contact reaction: $\vec{F}_c = -F_{c,1}\hat{I}_1 - F_{c,3}\hat{I}_3$

Thus, the total force applied to the robot is:

$$\vec{F}_{\text{total}} = \vec{F}_g + \vec{F}_c = -mg\hat{I}_3 - F_{c,1}\hat{I}_1 - F_{c,3}\hat{I}_3 \tag{D.3}$$

4. Moment of forces about the CG

The torque generated by the ground contact forces about the CG is:

$$\vec{L}_G = -h\hat{X}_3 \times \vec{F}_c = h(F_{c,1}\cos\theta - F_{c,3}\sin\theta)\hat{X}_2 \tag{D.4}$$

Explanation: We project the torque along the axis perpendicular to the plane of motion.

5. Newton's second law

From $m\ddot{\vec{R}} = \vec{F}$ and projecting on $\hat{I}_1$ and $\hat{I}_3$:

$$m(\ddot{x} + h\ddot{\theta}\cos\theta - h\dot{\theta}^2\sin\theta) = -F_{c,1} \tag{D.5}$$

$$m(-h\ddot{\theta}\sin\theta + h\dot{\theta}^2\cos\theta) = -F_{c,3} + mg \tag{D.6}$$

These equations express the horizontal and vertical force balances.

6. Euler's rotational equation

From $\dot{\vec{H}}_G = \vec{L}_G$, we get:

$$J\ddot{\theta} = h(F_{c,1}\cos\theta - F_{c,3}\sin\theta) \tag{D.7}$$

7. Elimination of contact forces

To eliminate $F_{c,1}$ and $F_{c,3}$, we multiply (D.5) by $\cos\theta$, multiply (D.6) by $\sin\theta$, add the resulting expressions and substitute into (D.7).

After simplification:

$$m\ddot{x}\cos\theta + \left(mh + \frac{J}{h}\right)\ddot{\theta} = mg\sin\theta$$

$$\Rightarrow \ddot{x}\cos\theta + \left(h + \frac{J}{mh}\right)\ddot{\theta} = g\sin\theta \tag{D.8}$$

This equation will serve as the basis for the nonlinear state transition model used in the Extended Kalman Filter.

Appendix H

Algorithms

Sonar scheduler algorithm

Algorithm 1 Sonar scheduler for obstacle avoidance mechanism

```
Start the scheduler as a background task
while true do
    Wait 50ms
    if robot moving backward then
        Send Authorize to rear sonar
    else if robot moving forward then
        Alternate between left and right front sonars,
        Send Authorize to selected front sonar
    else
        Do nothing (no sonar measurement)
    end if
end while
```

Obstacle avoidance algorithm

Algorithm 2 Obstacle avoidance mechanism

Require: Sonar reading and movement intention
Ensure: Adjusted commands for safety

```
// Step 1: Emergency turn if too close
if distance to obstacle is very small then
    force a full-speed rotation
else
    keep the planned turning speed
end if
// Step 2: Stop forward motion if risky
if no sonar data then
    keep the planned forward speed
else if obstacle is detected at moderate range then
    if moving forward toward the obstacle then
        stop immediately
    else if moving backward toward the obstacle then
        stop immediately
    else
        keep moving
    end if
else
    no action needed
end if
```

Kalman message handling algorithm

Algorithm 3 Kalman message handling logic

Require: Previous state `(Xk, Pk)`, acceleration input `Acc`, timestep `Dt`
Ensure: Estimated tilt angle and updated Kalman state

```
Convert Acc from cm/s^2 to m/s^2
if new IMU message {nav_data, (Gy, Ax, Az)} is received then
    Perform prediction + correction using EKF
    Return filtered angle and updated state
else
    Perform prediction-only step using EKF
    Return predicted angle and updated state
end if
```

Appendix I

I2C packet structure

Table I.1: Structure of the 5-byte I^2C packet from the ESP32.

Byte Index	Content	Description
1–2	`Speed_Left`	Half-float encoded rotation speed of the left wheel
3–4	`Speed_Right`	Half-float encoded rotation speed of the right wheel
5	`Input Flags`	User control commands encoded as individual bits

Table I.2: Bit decomposition of the I^2C control byte (Byte 5).

Bit Index	Function
1	`Arm_Ready`: Lifting mechanism feedback
2	`Switch`: Unused
3	`Test`: Unused
4	`Get_Up`: Transition between dynamic and static mode
5	`Forward`: Command to move forward
6	`Backward`: Command to move backward
7	`Left`: Command to turn left
8	`Right`: Command to turn right

Appendix J

LED debugging indicators

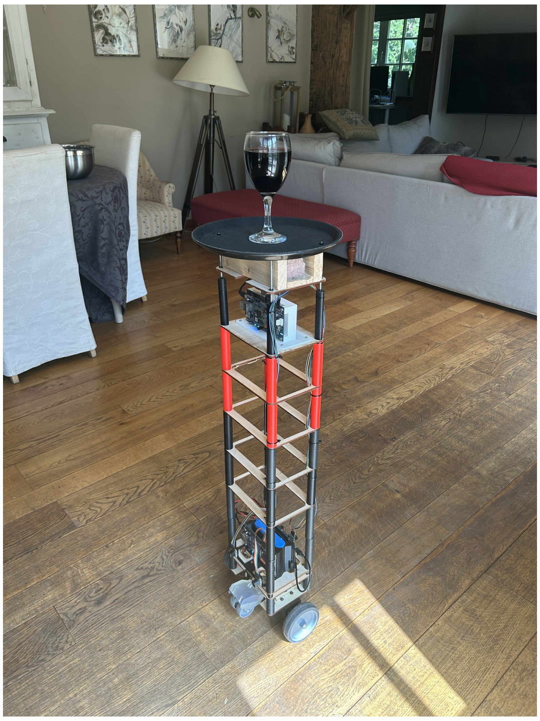
LED Color	**Pattern (ms)**	**Meaning**
Yellow	Flash 500	Balancing robot initialization
Yellow	Flash 250	Device added successfully
Red	Flash 250	Device registration failed
Red	Flash 1000	No ping from server (connection error)
Magenta	Flash 500	Wi-Fi setup failed, retrying connection
White	Flash 500	Wi-Fi connection established
Green	Flash 1000	Server discovered via ping
Aqua	Solid	Server acknowledged, robot ready

Table J.1: LED indicators used in `balancing_robot` for runtime debugging
.

Appendix K

Experimentation setup for payload tests

The 2 following pictures are pictures of the test setup of payloads.



(a) Picture of the robot during the test with a glass of wine.

(b) Picture of the robot during the test with 14 kg of payload.

Appendix L

Obstacle avoidance evaluation of the backward sonar

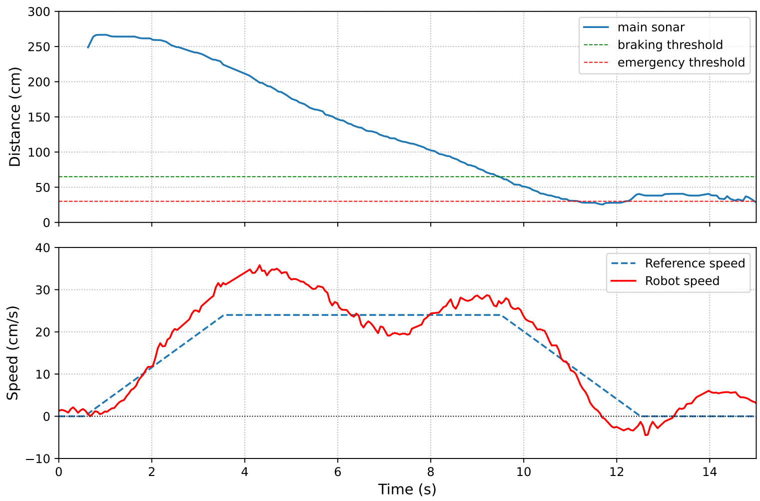


Figure L.1: [Top graph] Sonar measures of the main back sonar. [Bottom graph] Robot speed variation and reference speed over time when the robot is going backward in the direction of an obstacle (a couch).

Appendix M

Full source code

This appendix contains all the code needed to operate the system, in four main sections: the GRiSP code, the Lilygo LoRa32 code on the robot side, the Lilygo LoRa32 code on the emergency stop side, and the python user interface and server code. The latest version of the code can be found on our two GitHub repositories: `https://github.com/Artal44/Grisp_robot/tree/main`.

GRiSP board code

Full `balancing_robot.erl` source code

```
-module(balancing_robot).

-behavior(application).

-export([start/2, stop/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% STARTUP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start(_Type, _Args) ->
    {ok, Supervisor} = balancing_robot_sup:start_link(),
    [grisp_led:flash(L, yellow, 500) || L <- [1, 2]],

    %% Initialize log buffer
    log_buffer:init(100000),

    numerl:init(),
```

```
    hera_subscribe:subscribe(self()),
    persistent_term:put(server_on, false),

    %% Get GRiSP ID and start correct processes
    {ok, Id} = get_grisp_id(),
    init_grisp(Id),

    %% Setup WiFi and server discovery
    config(),

    %% Start alive loop in its own process
    spawn(fun alive_loop/0),

    %% Keep process alive to handle Hera messages
    hera_notify_loop(),

    {ok, Supervisor}.

stop(_State) ->
    persistent_term:get(name),
    case persistent_term:get(name) of
        robot_main ->
            ets:delete(adv_goal_tab);
        _ ->
            ok
    end,
    ok.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% INITIALIZATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

init_grisp(0) ->
    io:format("[BALANCING_ROBOT] GRiSP ID: 0, Spawning robot_main~n",
    ↪  []),
    persistent_term:put(name, robot_main),

    %% Initialize ETS table for adv_goal
    ets:new(adv_goal_tab, [set, public, named_table]),
    ets:insert(adv_goal_tab, {adv_goal, 0.0}),

    add_GRISP_device(spi2, pmod_nav),
```

```
    add_GRISP_device(uart, pmod_maxsonar),
    pmod_nav:config(acc, #{odr_g => {hz,238}}),
    timer:sleep(10000),

    Pid_Main = spawn(main_loop, robot_init, []),
    persistent_term:put(pid_main, Pid_Main),

    %% Start sonar scheduler
    persistent_term:put(current_sonar, robot_front_left),
    start_sonar_scheduler(),

    spawning_sonar(0, robot_main),
    hera:start_measure(nav_measure, [Pid_Main, robot_main]);

init_grisp(1) ->
    persistent_term:put(name, robot_front_left),
    add_GRISP_device(uart, pmod_maxsonar),
    timer:sleep(2000),
    spawning_sonar(1, robot_front_left);

init_grisp(2) ->
    persistent_term:put(name, robot_front_right),
    add_GRISP_device(uart, pmod_maxsonar),
    timer:sleep(2000),
    spawning_sonar(2, robot_front_right);

init_grisp(_) ->
    io:format("[BALANCING_ROBOT][ERROR] Unknown GRiSP ID~n", []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% DEVICES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

add_GRISP_device(Port, Name) ->
    case catch grisp:add_device(Port, Name) of
        {device, _, _, _, _} = DeviceInfo ->
            [grisp_led:flash(L, yellow, 250) || L <- [1, 2]],
            io:format("[~p] Device ~p added (info: ~p)~n",
                      [persistent_term:get(name), Name, DeviceInfo]);
        Other ->
            [grisp_led:flash(L, red, 250) || L <- [1, 2]],
            io:format("[~p] Failed to add device ~p: ~p~n",
                      [persistent_term:get(name), Name, Other]),
```

```
            timer:sleep(2000),
            add_GRISP_device(Port, Name)
    end.

get_grisp_id() ->
    JMPs = [jumper_1, jumper_2, jumper_3, jumper_4, jumper_5],
    Bits = [grisp_gpio:get(grisp_gpio:open(J, #{mode => input})) || J <- JMPs],
    {ok, lists:foldl(fun(B, Acc) -> (Acc bsl 1) + B end, 0, lists:reverse(Bits))}.

spawning_sonar(Id, Role) ->
    {ok, Pid_Sonar} = hera:start_measure(sonar_measure, [Role]),
    persistent_term:put(pid_sonar, Pid_Sonar),
    io:format("[BALANCING_ROBOT] GRiSP ~p spawned ~p (PID: ~p)~n",
              [Id, Role, Pid_Sonar]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% NETWORK / SERVER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

config() -> await_connection().

await_connection() ->
    io:format("[~p] WiFi setup starting...~n", [persistent_term:get(name)]),
    receive
        {hera_notify, "connected"} ->
            io:format("[~p] WiFi connected~n", [persistent_term:get(name)]),
            [grisp_led:flash(L, white, 500) || L <- [1, 2]],
            discover_server()

    after 18000 ->
        io:format("[~p] WiFi setup failed. Retrying...~n", [persistent_term:get(name)]),
        [grisp_led:flash(L, magenta, 500) || L <- [1, 2]],
        await_connection()
    end.

discover_server() ->
    receive
        {hera_notify, ["ping", Name, SIp, Port]} ->
```

```
            {ok, Ip} = inet:parse_address(SIp),
            [grisp_led:flash(L, green, 1000) || L <- [1, 2]],
            hera_com:add_device(list_to_atom(Name), Ip,
            ↪  list_to_integer(Port)),
            ack_loop()
    after 9000 ->
        io:format("[~p] No ping from server. Retrying...~n",
        ↪  [persistent_term:get(name)]),
        [grisp_led:flash(L, red, 1000) || L <- [1, 2]],
        discover_server()
    end.

ack_loop() ->
    Payload = "Hello from " ++ atom_to_list(persistent_term:get(name)),
    hera_com:send_unicast(server, Payload, "UTF8"),
    receive
        {hera_notify, ["Ack", _]} ->
            persistent_term:put(server_on, true),
            [grisp_led:color(L, aqua) || L <- [1, 2]],
            io:format("[~p] Received ACK from server~n",
            ↪  [persistent_term:get(name)])
    after 5000 ->
        ack_loop()
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOOPS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

alive_loop() ->
    Msg = "alive : " ++ atom_to_list(persistent_term:get(name)),
    hera_com:send_unicast(server, Msg, "UTF8"),
    timer:sleep(20000),
    alive_loop().

hera_notify_loop() ->
    receive
        {hera_notify, Msg} ->
            io:format("[~p] Received hera_notify: ~p~n",
            ↪  [persistent_term:get(name), Msg]),
            handle_hera_notify(Msg),
            hera_notify_loop();
        Other ->
```

```
            io:format("[~p] Unexpected message: ~p~n",
                      [persistent_term:get(name), Other]),
            hera_notify_loop()
    end.

handle_hera_notify(["ping", _, _, _]) -> ok;
handle_hera_notify(["Add_Device", Name, SIp, Port]) ->
    add_device(Name, SIp, Port);
handle_hera_notify(["authorize"]) ->
    persistent_term:get(pid_sonar) ! {authorize, robot_main};
handle_hera_notify(["sonar_data", Sonar_Name, D, Seq]) ->
    case persistent_term:get(pid_main, undefined) of
        undefined ->
            io:format("[~p] sonar_data ignored (no pid_main)~n",
                      [persistent_term:get(name)]);
        Pid_Main ->
            Pid_Main ! {sonar_data, list_to_atom(Sonar_Name),
                        [list_to_float(D), list_to_integer(Seq)]}
    end;
handle_hera_notify(Other) ->
    io:format("[~p] Unhandled hera_notify: ~p~n",
              [persistent_term:get(name), Other]).

add_device(Name, SIp, SPort) ->
    Self = persistent_term:get(name),
    case list_to_atom(Name) of
        Self -> ok;
        OName ->
            {ok, Ip} = inet:parse_address(SIp),
            hera_com:add_device(OName, Ip, list_to_integer(SPort)),
            io:format("[BALANCING_ROBOT] Added device ~p (IP: ~p, Port: ~p)~n",
                      [OName, Ip, SPort])
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% SONAR SCHEDULER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start_sonar_scheduler() ->
    spawn(fun sonar_scheduler/0).

sonar_scheduler() ->
```

```
    receive
        after 75 -> % run every 50 ms
            [{adv_goal, Adv_V_Goal}] = ets:lookup(adv_goal_tab,
            ↪  adv_goal),
            handle_sonar_authorization(Adv_V_Goal),
            sonar_scheduler()
    end.

handle_sonar_authorization(Adv_V_Goal) ->
    case Adv_V_Goal of
        V when V > 0 ->
            % Recule : only back sonar
            persistent_term:get(pid_sonar) ! {authorize,
            ↪  persistent_term:get(pid_main)};
        V when V < 0 ->
            % Avance : alternate front_left / front_right
            Sonar_Role = persistent_term:get(current_sonar),
            Next_Role = case Sonar_Role of
                robot_front_left  -> robot_front_right;
                robot_front_right -> robot_front_left
            end,
            persistent_term:put(current_sonar, Next_Role),
            spawn(fun() ->
                hera_com:send_unicast(Next_Role, "authorize", "UTF8")
            end);
        _ -> ok
    end.
```

Listing M.1: Full `balancing_robot.erl` code.

Full `balancing_robot_sup.erl` source code

```
% @private
% @doc balancing_robot top level supervisor.
-module(balancing_robot_sup).

-behavior(supervisor).

% API
-export([start_link/0]).

```

```
% Callbacks
-export([init/1]).

%--- API
↪ -----------------------------------------------------------------------

start_link() -> supervisor:start_link({local, ?MODULE}, ?MODULE, []).

%--- Callbacks
↪ -----------------------------------------------------------------

init([]) -> {ok, { {one_for_all, 0, 1}, []} }.
```

Listing M.2: Full `balancing_robot_sup.erl` code.

Full `main_loop.erl` source code

```
-module(main_loop).

-export([robot_init/0]).

-define(ADV_V_MAX, 24.0).
-define(TURN_V_MAX, 80.0).
-define(LOG_INTERVAL, 50). % ms

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INITIALISATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
robot_init() ->
    process_flag(priority, max),
    calibrate(),
    {X0, P0} = kalman_computations:init_kalman(),

    %I2C bus
    I2Cbus = grisp_i2c:open(i2c1),
    persistent_term:put(i2c, I2Cbus),

    % PIDs initialization with adjusted gains
    Pid_Speed = spawn(hera_pid_controller, pid_init, [-0.0635, -0.053,
    ↪ 0.0, -1, 15.0, 0.0]),
    Pid_Stability = spawn(hera_pid_controller, pid_init, [16.3, 0.0,
    ↪ 9.4, -1, -1, 0.0]),
```

```erlang
    persistent_term:put(controllers, {Pid_Speed, Pid_Stability}),
    persistent_term:put(freq_goal, 220.0),

    T0 = erlang:system_time()/1.0e6,
    State = #{
        robot_state => {rest, false}, % {Robot_State, Robot_Up}
        kalman_state => {T0, X0, P0}, % {Tk, Xk, Pk}
        move_speed => {0.0, 0.0}, % {Adv_V_Ref, Turn_V_Ref}
        frequency => {0, 0, 220.0, T0}, % {N, Freq, Mean_Freq, T_End}
        acc_prev => 0.0, % Acc_Prev
        sonar => {0, none, none}, % {Current_Seq, Prev_Dist, Prev_Direction}
        last_log_time => erlang:system_time(millisecond) % LastLog
    },

    robot_loop(State).

robot_loop(State) ->
    Start_log = erlang:system_time(millisecond),
    Start = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Dt Computation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    {Tk, Xk, Pk} = maps:get(kalman_state, State),
    T1 = erlang:system_time()/1.0e6,
    Dt = (T1 - Tk) / 1000.0,
    T_Dt = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% LOGGING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    LastLog = maps:get(last_log_time, State),
    {Robot_State, Robot_Up} = maps:get(robot_state, State),
    {DoLog, New_LastLog} = logging(T1, LastLog),
    {N, Freq, Mean_Freq, T_End} = maps:get(frequency, State),
    add_log({main_loop, Start_log, robot_frequency, [Freq, N, Mean_Freq]}, DoLog),
    T_Log = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% KALMAN LOGIC %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Acc_Prev = maps:get(acc_prev, State),
    {Angle, X1, P1} = kalman_message_handling(Xk, Pk, Acc_Prev, Dt),
    T_Kalman = erlang:system_time(microsecond),
```

```erlang

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% INPUT FROM I2C + CONTROLS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    {Speed, CtrlByte} = i2c_read(),
    [Arm_Ready, _, _, Get_Up, Forward, Backward, Left, Right] = hera_com:get_bits(CtrlByte),
    Adv_V_Goal = speed_ref(Forward, Backward),
    ets:insert(adv_goal_tab, {adv_goal, Adv_V_Goal}),
    Turn_V_Goal = turn_ref(Left, Right),
    T_I2C = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SONAR LOGIC %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    {Current_Seq, Prev_Dist, Prev_Direction} = maps:get(sonar, State),
    {Sonar_Data, New_Seq, New_Direction} = sonar_message_handling(Current_Seq, Prev_Dist, Prev_Direction),
    T_Sonar = erlang:system_time(microsecond),

   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% STABILITY ENGINE CONTROLLER %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    {Adv_V_Ref, Turn_V_Ref} = maps:get(move_speed, State),
    {Acc, Adv_V_Ref_New, Turn_V_Ref_New} = stability_engine:controller(
        {Dt, Angle, Speed},
        {Sonar_Data, New_Direction},
        {Adv_V_Goal, Adv_V_Ref},
        {Turn_V_Goal, Turn_V_Ref},
        {DoLog, Start_log}),
    T_Controller = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ROBOT STATE + I2C WRITE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Robot_Up_New = is_robot_up(Angle, Robot_Up),
    Next_Robot_State = get_robot_state({Robot_State, Robot_Up, Get_Up, Arm_Ready, Angle}),
    Output_Byte = get_output_state(Next_Robot_State),
    i2c_write(Acc, Turn_V_Ref_New, Output_Byte),
    T_Write = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FREQUENCY STABILISATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    {N_New, Freq_New, Mean_Freq_New} = frequency_computation(Dt, N, Freq, Mean_Freq),
    maximum_frequency(T1, T_End),
```

```
    T_Freq = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SERVER COMMUNICATION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    send_to_server(Robot_State),
    T_Server = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% STATE UPDATE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    T_End_New = erlang:system_time()/1.0e6,
    NewState = State#{
        robot_state => {Next_Robot_State, Robot_Up_New},
        kalman_state => {T1, X1, P1},
        move_speed => {Adv_V_Ref_New, Turn_V_Ref_New},
        frequency => {N_New, Freq_New, Mean_Freq_New, T_End_New},
        acc_prev => Acc,
        sonar => {New_Seq, Sonar_Data, New_Direction},
        last_log_time => New_LastLog
    },
    T_State = erlang:system_time(microsecond),

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TIMING LOGS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Total = T_State - Start,
    add_log({timing, erlang:system_time(millisecond), [
        {"Dt", T_Dt - Start},
        {"Logging", T_Log - T_Dt},
        {"Kalman", T_Kalman - T_Log},
        {"I2C_Read+Controls", T_I2C - T_Kalman},
        {"Sonar", T_Sonar - T_I2C},
        {"Controller", T_Controller - T_Sonar},
        {"I2C_Write+State", T_Write - T_Controller},
        {"Frequency_Stab", T_Freq - T_Write},
        {"Server", T_Server - T_Freq},
        {"State_Update", T_State - T_Server},
        {"Total", Total}
    ]}, DoLog),

    robot_loop(NewState).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ROBOT STATE LOGIC
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

get_robot_state(Robot_State) -> % {Robot_state, Robot_Up, Get_Up, Arm_ready, Angle}
    case Robot_State of
        % From rest
        {rest, true, _, _, _} -> dynamic;
        {rest, _, _, _, _} -> rest;

        % Dynamic âĘŠ static
        {dynamic, _, true, _, _} -> preparing_static;
        {dynamic, false, _, _, _} -> rest;
        {dynamic, _, _, _, _} -> dynamic;

        % Preparing static âĘŠ static
        {preparing_static, _, _, true, _} -> static;
        {preparing_static, _, _, false, _} -> dynamic;
        {preparing_static, _, _, _, _} -> preparing_static;

        % Static âĘŠ dynamic
        {static, false, _, _, _} -> rest;
        {static, _, false, _, _} -> dynamic;
        {static, _, _, _, _} -> static
    end.

get_output_state(State) ->
    % Output bits = [Power, Freeze, Extend, Robot_Up_Bit, Move_direction, 0, 0, 0]
    case State of
        rest                -> get_byte([0,0,0,0,0,0,0,0]);
        dynamic             -> get_byte([1,0,0,1,0,0,0,0]);
        preparing_static    -> get_byte([1,0,1,1,0,0,0,0]); % arms extending
        static              -> get_byte([1,1,1,1,0,0,0,0])
    end.

is_robot_up(Angle, Robot_Up) ->
    if
        Robot_Up and (abs(Angle) > 80) ->
            false;
        not Robot_Up and (abs(Angle) < 78) ->
            true;
        true ->
```

```erlang
            Robot_Up
    end.

get_byte(List) ->
    [A, B, C, D, E, F, G, H] = List,
    A*128 + B*64 + C*32 + D*16 + E*8 + F*4 + G*2 + H.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% I2C COMMUNICATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

i2c_read() ->
    %Receive I2C and conversion
    I2Cbus = persistent_term:get(i2c),
    case grisp_i2c:transfer(I2Cbus, [{read, 16#40, 1, 5}]) of
        [<<SL1,SL2,SR1,SR2,CtrlByte>>] ->
            [Speed_L,Speed_R] = hera_com:decode_half_float([<<SL1, SL2>>, <<SR1, SR2>>]),
            Speed = (Speed_L + Speed_R)/2,
            {Speed, CtrlByte};
        {error, Reason} ->
            io:format("[ROBOT][I2C ERROR] Error response: ~p~n", [{error, Reason}]),
            [grisp_led:color(L, red) || L <- [1, 2]],
            timer:sleep(5000),
            i2c_read();
        Other ->
            io:format("[ROBOT][I2C ERROR] Unexpected response: ~p~n", [Other]),
            timer:sleep(5000),
            i2c_read()
    end.

i2c_write(Acc, Turn_V_Ref_New, Output_Byte) ->
    I2Cbus = persistent_term:get(i2c),
    case hera_com:encode_half_float([Acc, Turn_V_Ref_New]) of
        [HF1, HF2] ->
            grisp_i2c:transfer(I2Cbus, [{write, 16#40, 1, [HF1, HF2, <<Output_Byte>>]}]);
        Error ->
            log_buffer:add({i2c_error, erlang:system_time(millisecond), Error})
```

```
    end.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% MISCELLANIOUS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
speed_ref(Forward, Backward) ->
    if
        Forward ->
            Adv_V_Goal = +?ADV_V_MAX;
        Backward ->
            Adv_V_Goal = -?ADV_V_MAX;
        true ->
            Adv_V_Goal = 0.0
    end,
    Adv_V_Goal.

turn_ref(Left, Right) ->
    if
        Right ->
            Turn_V_Goal = ?TURN_V_MAX;
        Left ->
            Turn_V_Goal = - ?TURN_V_MAX;
        true ->
            Turn_V_Goal = 0.0
    end,
    Turn_V_Goal.

frequency_computation(Dt, N, Freq, Mean_Freq) ->
    if
        N == 100 ->
            N_New = 0,
            Freq_New = 0,
            Mean_Freq_New = Freq;
        true ->
            N_New = N+1,
            Freq_New = ((Freq*N)+(1/Dt))/(N+1),
            Mean_Freq_New = Mean_Freq
    end,
    {N_New, Freq_New, Mean_Freq_New}.

maximum_frequency(T1, T_End) ->
    T2 = erlang:system_time()/1.0e6,
```

```
    Freq_Goal = persistent_term:get(freq_goal),
    Delay_Goal = 1.0/Freq_Goal * 1000.0,
    if
        T2-T_End < Delay_Goal ->
            wait(Delay_Goal-(T2-T1));
        true ->
            ok
    end.

wait(T) ->
        Tnow = erlang:system_time()/1.0e6,
        wait_help(Tnow,Tnow+T).
wait_help(Tnow, Tend) when Tnow >= Tend -> ok;
wait_help(_, Tend) ->
    Tnow = erlang:system_time()/1.0e6,
    wait_help(Tnow,Tend).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LOGGING
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
logging(Now, Last_Log_Time) ->
    case Now - Last_Log_Time > ?LOG_INTERVAL of
        true ->
            {true, Now};
        false ->
            {false, Last_Log_Time}
    end.

add_log(Log, DoLog) ->
    case DoLog of
        true ->
            log_buffer:add(Log);
        false -> ok
    end.

send_to_server(Robot_State) ->
    case Robot_State of
        static ->
            case persistent_term:get(server_on) of
                true ->
                    log_buffer:flush_to_server(server,
                     ↪  persistent_term:get(name));
                _ ->
```

```
                    ok
            end;
        _ -> ok
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SONAR
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sonar_message_handling(Current_Seq, Prev_Dist, Prev_Direction) ->
    receive
        {sonar_data, Sonar_Name, [D, Seq]} ->
            D_M = D / 100.0,
            add_log({main_loop, erlang:system_time(millisecond), new_sonar_measure, [Sonar_Name, D_M]}, true),
            % io:format("[ROBOT][SONAR] Sonar data received: ~p from ~p~n", [D_M, Sonar_Name]),
            Direction = case Sonar_Name of
                robot_front_left -> front;
                robot_front_right -> front;
                _ -> back
            end,
            {D_M, Seq, Direction}
    after 0 ->
        {Prev_Dist, Current_Seq, Prev_Direction}
    end.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% KALMAN
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
kalman_message_handling(Xk, Pk, Acc_Prev, Dt) ->
    Acc_SI = Acc_Prev / 100.0, % Convert acceleration from cm/s^2 to m/s^2
    {Angle, X1, P1} =
    receive
        {nav_data, [Gy, Ax, Az]} ->
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            % KALMAN PREDICTION + CORRECTION
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            [Angle1, {X1a, P1a}] = kalman_computations:kalman_angle(Dt, Ax, Az, Gy, Acc_SI, Xk, Pk),

            {Angle1, X1a, P1a}
```

```
    after 0 ->
        kalman_predict_only(Xk, Pk, Dt, Acc_SI)
    end,
    {Angle, X1, P1}.


kalman_predict_only(Xk, Pk, Dt, Acc_SI) ->
    % Kalman prediction
    [Angle1, {X1a, P1a}] = kalman_computations:kalman_predict_only(Dt, [Xk, Pk], Acc_SI),

    {Angle1, X1a, P1a}.

calibrate() ->
    N = 500,
    Y_List = [pmod_nav:read(acc, [out_y_g]) || _ <- lists:seq(1, N)],
    Gy0 = lists:sum([Y || [Y] <- Y_List]) / N,
    persistent_term:put(gy0, Gy0).
```

Listing M.3: Full `main_loop.erl` code.

Full `stability_engine.erl` source code

```
-module(stability_engine).

-export([controller/5]).

-define(ADV_V_MAX, 24.0).
-define(ADV_ACCEL, 8.0).

-define(TURN_V_MAX, 40.0).
-define(OBSTACLE_TURN_V_MAX, 80.0).
-define(TURN_ACCEL, 200.0).

-define(MIN_SONAR_DIST, 0.30).
-define(MAX_SONAR_DIST, 0.65).

controller({Dt, Angle, Speed}, {Sonar_Data, Direction}, {Adv_V_Goal, Adv_V_Ref}, {Turn_V_Goal, Turn_V_Ref}, {DoLog, Time}) ->
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % CONTROLLER
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    {Pid_Speed, Pid_Stability} = persistent_term:get(controllers),

    % ÃĽvitement automatique si bloquÃĽ
    Turn_V_Goal_Avoid =
        case Sonar_Data =< ?MIN_SONAR_DIST andalso (Adv_V_Goal > 0 orelse Adv_V_Goal < 0) of
            true -> ?OBSTACLE_TURN_V_MAX;  % tourne Ãă droite (ou -?TURN_V_MAX Ãă gauche)
            false -> Turn_V_Goal
        end,

    % Applique un freinage si lâĂŹobstacle est dÃĽtectÃĽ
    Adv_V_Goal_Safe =
        case Sonar_Data of
            none ->
                Adv_V_Goal;  % Pas de donnÃĽe sonar, on garde la vitesse
            _ ->
                case Sonar_Data < ?MAX_SONAR_DIST of
                    true ->
                        case Adv_V_Goal of
                            V when V < 0.0 andalso Direction =:= front ->
                                0.0;  % Stop en marche avant
                            V when V > 0.0 andalso Direction =:= back ->
                                0.0;  % Stop en marche arriÃĺre
                            _ ->
                                Adv_V_Goal
                        end;
                    false ->
                        Adv_V_Goal
                end
        end,

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % ACCELERATION SATURATION
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Adv_V_Ref_New = saturate_acceleration(Adv_V_Goal_Safe, Adv_V_Ref, Dt, ?ADV_ACCEL, ?ADV_V_MAX),
    Turn_V_Ref_New = saturate_acceleration(Turn_V_Goal_Avoid, Turn_V_Ref, Dt, ?TURN_ACCEL, ?TURN_V_MAX),
```

```

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % ADVANCED SPEED CONTROLLER
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Pid_Speed ! {self(), {set_point, Adv_V_Ref_New}},
    Pid_Speed ! {self(), {input, Speed}},
    receive {_, {control, Target_Angle}} -> ok end,

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % STABILITY CONTROLLER
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Pid_Stability ! {self(), {set_point, Target_Angle}},
    Pid_Stability ! {self(), {input, Angle}},
    receive {_, {control, Acc}} -> ok end,

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % LOGGING
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    add_log({controller, Time, speed_controller, [Adv_V_Ref_New,
    ↪ Turn_V_Ref_New, Speed, Target_Angle]}, DoLog),
    add_log({controller, Time, stability_controller, [Target_Angle,
    ↪ Angle, Acc]}, DoLog),

    {Acc, Adv_V_Ref_New, Turn_V_Ref_New}.

% Saturates the acceleration based on the goal and reference speed.
% If the goal is positive, it accelerates towards the goal, if negative,
↪ it decelerates.
% If the goal is zero, it checks the reference speed and applies a
↪ deceleration or acceleration based on the current speed.
saturate_acceleration(Goal, Ref, Dt, Accel, V_Max) ->
    case Goal of
        G when G > 0.0 ->
            hera_pid_controller:saturation(Ref + Accel * Dt, V_Max);
        G when G < 0.0 ->
            hera_pid_controller:saturation(Ref - Accel * Dt, V_Max);
        _ ->
            case Ref of
                R when R > 0.05 ->
                    hera_pid_controller:saturation(Ref - Accel * Dt,
                    ↪ V_Max);
                R when R < -0.05 ->
```

```
                    hera_pid_controller:saturation(Ref + Accel * Dt,
                     ↪  V_Max);
                _ ->
                    0.0
            end
    end.

add_log(Log, DoLog) ->
    case DoLog of
        true ->
            log_buffer:add(Log);
        false -> ok
    end.
```

Listing M.4: Full `stability_engine.erl` code.

Full `nav_measure.erl` source code

```
-module(nav_measure).
-behavior(hera_measure).

-export([init/1, measure/1]).

init([Pid, Role]) ->
    Name = list_to_atom("NAV_" ++ atom_to_list(Role)),
    io:format("[NAV] Starting ~p~n", [Role]),
    {ok, #{seq => 1, target => Pid, role => Role}, #{
        name => Name,
        iter => infinity,
        timeout => 5 % ODR is 238 Hz, the sensor updates every ~4.2 ms.
      }}.

measure(State) ->
    try
        [Gy, Ax, Az] = pmod_nav:read(acc, [out_y_g, out_x_xl, out_z_xl],
         ↪  #{g_unit => dps}),
        Pid = maps:get(target, State),
        Pid ! {nav_data, [Gy, Ax, Az]},
        Seq = maps:get(seq, State),
        NewState = State#{seq => Seq + 1},
        {ok, [Gy, Ax, Az], nav_measure, maps:get(role, State),
         ↪  NewState}
```

```
    catch
        _:Error ->
            io:format("[NAV][ERROR] ~p~n", [Error]),
            {ok, [], maps:get(role, State), State}
    end.
```

Listing M.5: Full nav_measure.erl code.

Full sonar_measure.erl source code

```
-module(sonar_measure).
-behaviour(hera_measure).

-export([init/1, measure/1]).

-define(ALPHA, 0.15).
-define(MIN_CM, 15.0).
-define(MAX_CM, 648.0).

init([Role]) ->
    Name = list_to_atom("SONAR_" ++ atom_to_list(Role)),
    io:format("[SONAR] Starting ~p~n", [Role]),
    State = #{
        seq => 1,
        role => Role,
        last_distance => none
    },
    {ok, State, #{name => Name, iter => infinity, timeout => 50}} .

measure(State) ->
    Role = maps:get(role, State),
    receive
        {authorize, Sender} ->
            RawD0 = measure_distance(),

            %% Hard gate
            Valid = (RawD0 >= ?MIN_CM) andalso (RawD0 =< ?MAX_CM),
            PrevD = maps:get(last_distance, State),
            RawD  = case Valid of true -> RawD0; false -> (PrevD =/=
            ↪   none andalso PrevD) orelse RawD0 end,

```

```
            % Lowa-pass filter + hampel + smoothnig
            LPF_filtered = low_pass_filter(RawD, PrevD, ?ALPHA),

            Final = round_to(LPF_filtered, 2),
            Seq = maps:get(seq, State),
            send_to_main(Role, Sender, Final, Seq),
            send_to_server(Role, Final, Seq),

            NewState = State#{
                    seq => Seq + 1,
                    last_distance => Final,
                    hampel_buffer => LPF_filtered
            },
            {ok, [Final], NewState}
        after 0 ->
            {ok, [-1], State}
    end.

%% --- helpers ---

measure_distance() ->
    Dist_inch = pmod_maxsonar:get(),
    Dist_inch * 2.54.

round_to(Value, Precision) ->
    Factor = math:pow(10, Precision),
    round(Value * Factor) / Factor.

low_pass_filter(RawD, LastD, Alpha) ->
    case LastD of
      none -> RawD;
      _    -> Alpha * LastD + (1-Alpha) * RawD
    end.

send_to_server(Role, D, Seq) ->
    Msg = "sonar_data , " ++ atom_to_list(Role) ++ " , " ++
    ↪  float_to_list(D) ++ " , " ++ integer_to_list(Seq),
    hera_com:send_unicast(server, Msg, "UTF8").

send_to_main(Role, Sender, D, Seq) ->
    case Role of
        robot_main ->
            Sender ! {sonar_data, Role, [D, Seq]};
```

```
        _ ->
            Msg = "sonar_data , " ++ atom_to_list(Role) ++ " , " ++
            ↪  float_to_list(D) ++ " , " ++ integer_to_list(Seq),
            hera_com:send_unicast(Sender, Msg, "UTF8")
    end.
```

Listing M.6: Full `sonar_measure.erl` code.

Full `kalman_computations.erl` source code

```
-module(kalman_computations).

-export([init_kalman/0, old_init_kalman/0, update_with_measurement/4,
↪  old_kalman_angle/6, kalman_predict_only/3, kalman_angle/7]).

-define(RAD_TO_DEG, 180.0/math:pi()).
-define(DEG_TO_RAD, math:pi()/180.0).

-define(g, 9.81). % Gravity in m/sÂš
-define(M, 5.3). % Mass of the robot (kg)
-define(h, 0.41). % Height of the robot center of mass (m)
% Poid en haut -> un metre de g
-define(width, 0.185). % Width of the robot (m)
-define(height, 0.95). % Height of the robot (m)
-define(I, ?M * (math:pow(?width, 2) + math:pow(?height, 2)) / 12). % I
↪  = M * (wÂš + hÂš) / 12 (rectangular parallelepiped)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% KALMAN INITIALIZATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
calibrate_initial_state() ->
    N = 500, % Increase the number of samples for better accuracy
    Measurements = [pmod_nav:read(acc, [out_x_xl, out_z_xl, out_y_g])
    ↪  || _ <- lists:seq(1, N)],
    {Ax_Sum, Az_Sum, Gy_Sum} = lists:foldl(
        fun ([Ax, Az, Gy], {Ax_Acc, Az_Acc, Gy_Acc}) ->
            {Ax_Acc + Ax, Az_Acc + Az, Gy_Acc + Gy}
        end,
        {0.0, 0.0, 0.0},
        Measurements
    ),
```

```erlang
    Ax_Avg = Ax_Sum / N,
    Az_Avg = Az_Sum / N,
    Gy_Avg = Gy_Sum / N,

    % Compute the initial angle and angular velocity
    Initial_Angle = math:atan(Az_Avg / (-Ax_Avg)),
    Initial_Angular_Velocity = (Gy_Avg - persistent_term:get(gy0)) * ?DEG_TO_RAD, % Subtract gyroscope bias
    log_buffer:add({main_loop, erlang:system_time(millisecond), kalman_calibration, [Initial_Angle* ?RAD_TO_DEG, Initial_Angular_Velocity]}),
    {Initial_Angle, Initial_Angular_Velocity}.

init_kalman() ->
    % Adjusted Kalman constants
    R = mat:matrix([[3.0, 0.0], [0, 3.0e-6]]),
    Q = mat:matrix([[1.0e-6, 0.0], [0.0, 2.5]]),

    % Model constants
    G = ?g,
    Hh = ?h + (?I / (?M * ?h)),

    Jh = fun (_) -> mat:matrix([[1, 0], [0, 1]]) end,
    persistent_term:put(kalman_constant, {R, Q, Jh, G, Hh}),

    % Initial State and Covariance matrices
    {Initial_Angle, Initial_Angular_Velocity} = calibrate_initial_state(),
    X0 = mat:matrix([[Initial_Angle], [Initial_Angular_Velocity]]),
    P0 = mat:matrix([[0.01, 0], [0, 0.01]]), % Slightly increased initial covariance
    {X0, P0}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% KALMAN COMPUTATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
update_with_measurement(Gy, Ax, Az, [Xk, Pk]) ->
    {R, _Q, Jh, _G, _Hh} = persistent_term:get(kalman_constant),
    H = fun (X) -> [Th, W] = mat:to_array(X), mat:matrix([[Th], [W]]) end,
    Z = mat:matrix([[math:atan(Az / (-Ax))], [(Gy - persistent_term:get(gy0)) * ?DEG_TO_RAD]]),
    {X1, P1} = hera_kalman:ekf_correct({Xk, Pk}, H, Jh, R, Z),
```

```
    [Th_Kalman, _] = mat:to_array(X1),
    Angle = Th_Kalman * ?RAD_TO_DEG,
    [Angle, {X1, P1}].

kalman_predict_only(Dt, [Xk, Pk], Acc) ->
    {_R, Q, _Jh, G, Hh} = persistent_term:get(kalman_constant),
    F = fun (X, U) ->
        [Th, W] = mat:to_array(X),
        Th1 = Th + W * Dt,
        W1 = W + ((G / Hh) * math:sin(Th) - (U / Hh) * math:cos(Th)) * Dt,
        mat:matrix([[Th1], [W1]])
    end,
    Jf = fun (X) ->
        [Th, _] = mat:to_array(X),
        DW_dTh = ((G / Hh) * math:cos(Th) + (Acc / Hh) * math:sin(Th)) * Dt,
        mat:matrix([[1, Dt], [DW_dTh, 1]])
    end,
    {X1, P1} = hera_kalman:ekf_predict({Xk, Pk}, F, Jf, Q, Acc), % {X1, P1}.
    [Th_Kalman, _] = mat:to_array(X1),
    Angle = Th_Kalman * ?RAD_TO_DEG,
    [Angle, {X1, P1}].

kalman_angle(Dt, Ax, Az, Gy, Acc, X0, P0) ->
    {R, Q, Jh, G, Hh} = persistent_term:get(kalman_constant),

    % Nonlinear state model (digital twin)
    F = fun (X, U) ->
        [Th, W] = mat:to_array(X),
        Th1 = Th + W * Dt,
        W1 = W + ((G / Hh) * math:sin(Th) - (U / Hh) * math:cos(Th)) * Dt,
        mat:matrix([[Th1], [W1]])
    end,

    % Jacobian of F
    Jf = fun (X) ->
        [Th, _W] = mat:to_array(X),
        DW_dTh = ((G / Hh) * math:cos(Th) + (Acc / Hh) * math:sin(Th)) * Dt,
        mat:matrix([[1, Dt],
```

```
                                [DW_dTh, 1]])
        end,

        % Observation function
        H = fun (X) ->
            [Th, W] = mat:to_array(X),
            mat:matrix([[Th], [W]])
        end,

        % Measurement vector: angle from accelerometer, angular velocity from gyro
        Z = mat:matrix([[math:atan(Az / (-Ax))], [(Gy - persistent_term:get(gy0)) * ?DEG_TO_RAD]]),
        {X1, P1} = hera_kalman:ekf_control({X0, P0}, {F, Jf}, {H, Jh}, Q, R, Z, Acc),

        [Th_Kalman, _W_Kalman] = mat:to_array(X1),
        Angle = Th_Kalman * ?RAD_TO_DEG,
        [Angle, {X1, P1}].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OLD KALMAN COMPUTATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

old_init_kalman() ->
        % Initiating kalman constants
        R = mat:matrix([[3.0, 0.0], [0, 3.0e-6]]),
        Q = mat:matrix([[3.0e-5, 0.0], [0.0, 10.0]]),
        Jh = fun (_) -> mat:matrix([          [1, 0],
                                              [0, 1]
                                              ])
                     end,
        persistent_term:put(old_kalman_constant, {R, Q, Jh}),

        % Initial State and Covariance matrices
        X0 = mat:matrix([[0], [0]]),
        P0 = mat:matrix([[0.1, 0], [0, 0.1]]),
        {X0, P0}.

old_kalman_angle(Dt, Ax, Az, Gy, X0, P0) ->
        Gy0 = persistent_term:get(gy0),
```

```
    {R, Q, Jh} = persistent_term:get(old_kalman_constant),

    F = fun (X) -> [Th, W] = mat:to_array(X),
                            mat:matrix([         [Th+Dt*W],
                                                            [W
                                                            ↪   ]
                                                            ↪   ])
                end,
    Jf = fun (_) -> mat:matrix([           [1, Dt],
                                                                [0,
                                                                ↪   1
                                                                ↪   ]
                                                                ↪   ])
                 end,
    H = fun (X) -> [Th, W] = mat:to_array(X),
                            mat:matrix([         [Th],
                                                            [W ] ])
                end,

    Z = mat:matrix([[math:atan(Az / (-Ax))], [(Gy-Gy0)*?DEG_TO_RAD]]),
    {X1, P1} = hera_kalman:ekf({X0, P0}, {F, Jf}, {H, Jh}, Q, R, Z),

    [Th_Kalman, _W_Kalman] = mat:to_array(X1),
    Angle = Th_Kalman * ?RAD_TO_DEG,
    [Angle, {X1, P1}].
```

Listing M.7: Full `kalman_computations.erl` code.

Full `log_buffer.erl` source code

```
-module(log_buffer).
-export([init/1, add/1, flush_to_server/2]).

%% Create two ETS tables: one for logs, one for metadata (max, idx)
init(MaxSize) ->
    ets:new(logs,     [named_table, set, public]),
    ets:new(log_meta, [named_table, set, public]),
    ets:insert(log_meta, {max, MaxSize}),
    ets:insert(log_meta, {count, 0}),
    %% Start idx at -1 so first increment yields 0
    ets:insert(log_meta, {idx, -1}).
```

```


add(Entry) ->
    [{max, Max}] = ets:lookup(log_meta, max),
    NewIndex = ets:update_counter(log_meta, idx, {2, 1, Max, 0}),
    ets:insert(logs, {NewIndex, Entry}),
    _ = ets:update_counter(log_meta, count, {2, 1, Max, Max}), % saturating at Max
    ok.

dump() ->
    [{idx, Last}]   = ets:lookup(log_meta, idx),
    [{count, Cnt}]  = ets:lookup(log_meta, count),
    [{max, Max}]    = ets:lookup(log_meta, max),
    case Cnt of
        0 -> [];
        _ ->
            Start = ((Last - Cnt + 1) rem Max + Max) rem Max,
            Indices = [((Start + I) rem Max) || I <- lists:seq(0, Cnt - 1)],
            [ {I, E} || I <- Indices, [{I, E}] <- [ets:lookup(logs, I)] ]
    end.

to_string(Value) when is_binary(Value) ->
    binary_to_list(Value);
to_string(Value) when is_atom(Value) ->
    atom_to_list(Value);
to_string(Value) ->
    io_lib:format("~p", [Value]).

flush_to_server(ServerRole, SelfRole) ->
    L = dump(),
    lists:foreach(fun({Index, Entry}) ->
        LogStr = format_log_entry(Entry),
        Msg = "log : " ++ atom_to_list(SelfRole) ++ " , " ++ lists:flatten(LogStr),
        hera_com:send_unicast(ServerRole, Msg, "UTF8"),
        ets:delete(logs, Index)  %% <-- delete the entry after printing
    end, L).

format_log_entry({Level, Timestamp, Category, Message}) ->
    io_lib:format("[~s] ~p | ~s | ~s",
```

```
        [string:to_upper(atom_to_list(Level)), Timestamp, to_string(Category), to_string(Message)]);
format_log_entry({Level, Timestamp, Message}) ->
    io_lib:format("[~s] ~p | ~s",
        [string:to_upper(atom_to_list(Level)), Timestamp, to_string(Message)]);
format_log_entry({Level, Timestamp}) ->
    io_lib:format("[~s] ~p", [string:to_upper(atom_to_list(Level)), Timestamp]);
format_log_entry(Other) ->
    io_lib:format("~p", [Other]).
```

Listing M.8: Full `log_buffer.erl` code.

Full `numerl.erl` source code

```
-module(numerl).
%-on_load(init/0).
-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]).

%Matrices are represented as such:
%-record(matrix, {n_rows, n_cols, bin}).

init()->
    ok  = erlang:load_nif(atom_to_list(?MODULE), 0).

%Creates a random matrix.
rnd_matrix(N)->
    L = [[rand:uniform(20) || _ <- lists:seq(1,N) ] || _ <- lists:seq(1,N)],
    matrix(L).

%Combine multiple functions.
eval([L,O,R|T])->
    F = fun numerl:O/2,
    eval([F(L,R) |T]);
eval([Res])->
    Res.
```

```erlang

%%Creates a matrix.
%List: List of doubles, of length N.
%Return: a matrix of dimension MxN, containing the data.
matrix(_) ->
    nif_not_loaded.

%%Returns the Nth value contained within Matrix.
at(_Matrix,_Nth)->
    nif_not_loaded.

%%Returns the matrix as a flattened list of ints.
mtfli(_mtrix)->
    nif_not_loaded.

%%Returns the matrix as a flattened list of doubles.
mtfl(_mtrix)->
    nif_not_loaded.

%%Returns a value from a matrix.
get(_,_,_) ->
    nif_not_loaded.

%%Returns requested row.
row(_,_) ->
    nif_not_loaded.


%%Returns requested col.
col(_,_) ->
    nif_not_loaded.


%%Equality test between matrixes.
equals(_, _) ->
    nif_not_loaded.


%%Addition of matrix.
add(_, _) ->
    nif_not_loaded.


```

```erlang
%%Subtraction of matrix.
sub(_, _) ->
    nif_not_loaded.


%% Matrix multiplication.
mult(A,B) when is_number(B) -> '*_num'(A,B);
mult(A,B) -> '*_matrix'(A,B).

'*_num'(_,_)->
    nif_not_loaded.

'*_matrix'(_, _)->
    nif_not_loaded.

%Matrix division by a number
divide(_,_)->
    nif_not_loaded.


%% build a null matrix of size NxM
zeros(_, _) ->
    nif_not_loaded.

%%Returns an Identity matrix NxN.
eye(_)->
    nif_not_loaded.

%Returns the transpose of the given square matrix.
transpose(_)->
    nif_not_loaded.

%Returns the inverse of asked square matrix.
inv(_)->
    nif_not_loaded.


%------CBLAS--------

%nrm2
%Calculates the squared root of the sum of the squared contents.
nrm2(_)->
    nif_not_loaded.
```

```
% : dot product of two vectors
% Arguments: vector x, vector y.
%   x and y are matrices
% Returns the dot product of all the coordinates of X,Y.
vec_dot(_, _)->
    nif_not_loaded.

% dgemm: A dot B
% Arguments: Matrix A, Matrix B.
%   alpha, beta: numbers (float or ints) used as doubles.
%   A,B,C: matrices.
% Returns the matrice resulting of the operations alpha * A * B + beta
↪  * C.
dot(_,_)->
    nif_not_loaded.
```

Listing M.9: Full `numerl.erl` code [22].

Full user interface and server code

User interface code: `RC_robot_UI.py`

```
import pygame
import pygame_gui
import sys
import numpy as np
import serial
from Server import Server
import socket
import threading

class User_interface:
    # App General State
    WIDTH, HEIGHT = 1920, 540 # Screen Size
    running = True
    in_popup = False
    active_popup = None
    temp_origin = None
    x = 0
    string = ""
    image_dict = {}
    rect_dict = {}
```

```
    current_action = ""

    # Log settings
    logs = []
    MAX_LOGS = 10
    log_messages = []
    LOG_PORT = 5001  # Port for receiving logs from the robot

    # Robot state
    message = 0  # Message to send to the robot
    run = True
    stand = False
    kalman = True  # Kalman filter is always enabled and cannot be
 disabled
    release_space = True
    release_enter = True
    release_t = True

    def __init__(self):

        pygame.init()
        self.ser = serial.Serial(port="/dev/ttyACM0", baudrate
=115200)

        self.screen = pygame.display.set_mode((self.WIDTH, self.
HEIGHT), pygame.RESIZABLE)
        pygame.display.set_caption("Robot Controller")

        self.manager = pygame_gui.UIManager((self.WIDTH, self.
HEIGHT))
        self.clock = pygame.time.Clock()
        self.clock.tick(200)

        self.server = Server()

        self.load_figures()

        self.ui_socket = socket.socket(socket.AF_INET, socket.
SOCK_DGRAM)
        self.ui_socket.bind(("0.0.0.0", 6000))  # Port UI listens
on

        self.log_thread = threading.Thread(target=self.
listen_to_logs, daemon=True)
        self.log_thread.start()

    def load_figures(self):
        arrow_img = pygame.image.load('./img/arrow.png')
        arrow_img = pygame.transform.scale(arrow_img, (arrow_img.
```

```
get_width() // 4, arrow_img.get_height() // 4))

        circle_img = pygame.image.load('./img/point.png')
        circle_img = pygame.transform.scale(circle_img, (circle_img.get_width() // 2, circle_img.get_height() // 2))

        stop_img = pygame.image.load('./img/Stop_sign.png')
        stop_img = pygame.transform.scale(stop_img, (stop_img.get_width() // 10, stop_img.get_height() // 10))

        self.image_dict["stop"] = stop_img
        self.image_dict["arrow"] = arrow_img
        self.image_dict["circle"] = circle_img

######################################################## TRIGGER CHECK ################################################

    def event_handler(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            self.manager.process_events(event)


######################################################## KEYBOARD FUNCTIONS ################################################

    def check_keys_movement(self, keys):
        if keys[pygame.K_SPACE]:
            if self.release_space:
                self.release_space = False
                if self.message < 10000000:
                    self.run = True
                else:
                    self.run = False
                    self.current_action = ""
        elif keys[pygame.K_z] or keys[pygame.K_UP] or self.current_action == "front":
            self.x += -1
        elif keys[pygame.K_s] or keys[pygame.K_DOWN] or self.current_action == "back":
            self.x += 1
        elif keys[pygame.K_q] or keys[pygame.K_LEFT] or self.current_action == "left":
            self.x += 1j
        elif keys[pygame.K_d] or keys[pygame.K_RIGHT] or self.current_action == "right":
            self.x += -1j
        elif keys[pygame.K_ESCAPE]:
```

```python
            self.running = False

        else:
            self.release_space = True

    def check_keys_kalman(self, keys):
        # Kalman filter is always enabled and cannot be disabled
        self.kalman = True

    def check_test(self, keys):
        if keys[pygame.K_t] and self.release_t:
                self.test, self.release_t = True, False
        else:
            self.test, self.release_t =  False, True

    def check_standing(self, keys):
        if keys[pygame.K_RETURN] or self.current_action == "stand":
            if self.release_enter:
                self.stand = not self.stand
                self.release_enter = False
        else:
            self.release_enter = True

######################################################### DRAWING FUNCTIONS #################################################

    def update_screen_size(self):
        self.WIDTH, self.HEIGHT = self.screen.get_size()
        self.manager.set_window_resolution((self.WIDTH, self.HEIGHT))

    def draw_move_ctrl(self):
        if self.message < 10000000:
            self.draw_image("stop", self.WIDTH //2, 100)
        elif abs(self.x) == 0:
            self.draw_image("circle", self.WIDTH //2, 100)
        else:
            angle = np.angle(-1*self.x, deg=True)
            rotated_arrow = pygame.transform.rotate(self.image_dict.get("arrow"), angle)
            rotated_rect = rotated_arrow.get_rect(center = (self.WIDTH//2, 100))
            self.screen.blit(rotated_arrow, rotated_rect.topleft)

    def draw_string(self):
        font = pygame.font.Font(None, 28)
        self.string += "DYNAMIC\n" if not self.stand else "STATIC\n"
```

```
        self.string += "Kalman filter\n" if self.kalman else ""
        self.string += "Running\n" if self.run else "Stopped\n"
        self.string += "Message: " + str(self.message) + "\n"

        lines = self.string.split("\n") + ["--- LOGS ---"] + self.log_messages

        for i, line in enumerate(lines):
            text = font.render(line, True, (0, 128, 0))
            self.screen.blit(text, (10, 10 + i * 24))


######################################################### SERIAL COMM FUNCTIONS #################################################

    def serial_comm(self):
        data = self.run << 7 | self.kalman << 6 | self.test << 5 | self.stand << 4 | (self.x.real == 1) << 3 | (self.x.real == -1) << 2 | (
                        self.x.imag == 1) << 1 | (self.x.imag == -1)
        self.ser.write(bytes([data]))

        Content = self.ser.readline()
        Content = Content.decode().replace("\r\n", "")
        self.message = int(Content)

############################################################ HELPER FUNCTIONS ######################################################
    def is_click_image(self, name, event):
        return self.rect_dict.get(name) != None and self.rect_dict.get(name).collidepoint(event.pos)

    def load_image(self, room_num, object, side):
        img = pygame.image.load(object.img)
        img = pygame.transform.scale(img, (img.get_width() // 5, img.get_height() // 5))

        name = object.type + "_" + side + "_" + str(room_num)
        self.image_dict[name] = img

    def draw_image(self, name, x, y):
        plus_rect = self.image_dict.get(name).get_rect(center = (x, y))
        self.screen.blit(self.image_dict.get(name), self.image_dict.get(name).get_rect(center=plus_rect.center))
        self.rect_dict[name] = plus_rect

```

```python
    def close_popup(self):
        self.active_popup.kill()
        self.active_popup = None

########################################### LOG LISTENER ###############################################

    def listen_to_logs(self):
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
            s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            try:
                s.bind(('', self.LOG_PORT))
            except OSError as e:
                print(f"[ERROR] Cannot bind UDP log socket on port {self.LOG_PORT}: {e}")
                return

            while True:
                try:
                    data, _ = s.recvfrom(1024)
                    msg = data.decode().strip()
                    self.logs.append(msg)
                    if len(self.logs) > self.MAX_LOGS:
                        self.logs.pop(0)
                except:
                    continue

    def draw_logs(self):
        font = pygame.font.Font(None, 24)
        base_y = self.HEIGHT - 25 * self.MAX_LOGS - 10
        for i, line in enumerate(self.logs):
            text = font.render(line, True, (0, 0, 0))
            self.screen.blit(text, (10, base_y + i * 25))


######################################################## MAIN LOOP ##########################################################

    def main_loop(self):
        while self.running:
            self.screen.fill((255, 255, 255))

            self.event_handler()
            self.update_screen_size()

            keys = pygame.key.get_pressed()
```

```
            self.x = 0
            self.string = ""

            if not self.in_popup:

                self.check_keys_movement(keys)
                self.check_keys_kalman(keys)
                self.check_test(keys)
                self.check_standing(keys)

            self.draw_move_ctrl()
            self.draw_string()
            self.draw_logs()

            self.manager.update(self.clock.tick(60)/1000)
            self.manager.draw_ui(self.screen)
            pygame.display.flip()
            self.serial_comm()

        # Quit
        pygame.quit()
        sys.exit()

if __name__ == '__main__':
    ui = User_interface()
    ui.main_loop()
```

Code Listing M.1: Python RC robot interface

Server implementation: Server.py

```
import socket
import threading
from Sonar import Sonar
import time

class Server:
    HOST = "172.20.10.5" # "172.20.10.4"

    UI_PORT = 5001  # Port sur lequel le UI Ãľcoute
    PORT = 5000  # Use the same port on both the server and the GRiSP device
    robot_sonar = {}
    last_seen = {}  # role -> last timestamp
    lost_robots = set()

    def __init__(self):
        # Create robot_debug.txt if it doesn't exist otherwise clear it
```

```python
        with open("robot_debug.txt", "w") as log_file:
            log_file.write("LOGGING STARTED\n")
        self.rcvServer = threading.Thread(target=self.rcv_server, daemon=True)
        self.rcvServer.start()
        self.pinger = threading.Thread(target=self.ping_server, daemon=True)
        self.pinger.start()
        self.alive_checker = threading.Thread(target=self.check_alive_loop, daemon=True)
        self.alive_checker.start()

    def ping_server(self):
        while not len(self.robot_sonar) == 3:
            time.sleep(5)
            message = "ping : server , " + self.HOST + " , " + str(self.PORT)
            self.send(message, "brd")

    def broadcast_devices(self):
        for role, sonar in self.robot_sonar.items():
            msg = f"Add_Device , {role} , {sonar.ip} , {sonar.port}"
            self.send(msg, "brd")
            self.log(f"[SERVER] Broadcasted device info for {role}")

    def uni_devices(self, receiver):
        for role, sonar in self.robot_sonar.items():
            msg = f"Add_Device , {role} , {sonar.ip} , {sonar.port}"
            self.send(msg, "uni_sonar", role=receiver)
            self.log(f"[SERVER] Broadcasted device info for {role}")

    def rcv_server(self):
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as server_socket:
            server_socket.bind((self.HOST, self.PORT))
            self.log(f"[SERVER] Listening for UDP packets on {self.HOST}:{self.PORT}")

            while True:
                data, addr = server_socket.recvfrom(1024)
                try :
                    data = data.decode()
                    if data[:16] == "Hello from robot":
                        role = data[11:]
                        is_reconnect = role in self.robot_sonar
```

```
                    self.robot_sonar[role] = Sonar(addr[0], addr[1], role)
                    self.last_seen[role] = int(time.time())
                    self.send("Ack , server", "uni_sonar", role=role)
                    self.log("[SERVER] Received hello from " + role)

                    if len(self.robot_sonar) == 3:
                        self.broadcast_devices()
                    elif is_reconnect:
                        self.lost_robots.discard(role)
                        self.log(f"[SERVER] Reconnected robot {role}, rebroadcasting peers")
                        self.uni_devices(role)
                elif data.startswith("log :"):
                    log_message = data[6:].strip()
                    with open("robot_debug.txt", "a") as log_file:
                        log_file.write(log_message + "\n")
                elif data.startswith("alive :"):
                    role = data.split(":", 1)[1].strip()
                    self.last_seen[role] = int(time.time())
                    if role not in self.robot_sonar:
                        self.robot_sonar[role] = Sonar(addr[0], addr[1], role)
                        if len(self.robot_sonar) == 3:
                            self.broadcast_devices()
                    self.log(f"[SERVER] Received ALIVE from {role} at {self.last_seen[role]}")
                elif data.startswith("sonar_data ,"):
                    parts = data.split(",")
                    if len(parts) == 4:
                        role = parts[1].strip()
                        distance = float(parts[2].strip())
                        seq = int(parts[3].strip())
                        self.log(f"[SERVER] Received sonar data from {role}: {distance} cm, seq: {seq}")
                        if role in self.robot_sonar:
                            self.robot_sonar[role].update_distance(distance, seq)
                            self.last_seen[role] = int(time.time())
                        else:
                            self.log(f"[SERVER] Unknown role in sonar data: {role}")
                else :
                    self.log(f"[SERVER] Received unknown message: {data}")
```

```python
                except :
                    pass

    def send(self, message, type, id=None, role=None):
        if type == "brd":
            threading.Thread(target=self.brd_server, args=(message,), daemon=True).start()
        elif type == "uni_sonar":
            threading.Thread(target=self.uni_server_sonar, args=(message, role), daemon=True).start()
        elif type == "uni_main":
            threading.Thread(target=self.uni_server_main, args=(message,), daemon=True).start()

    def brd_server(self, message):
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as srv_socket:
            srv_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

            broadcast_ip = '172.20.10.15'
            port = 9000

            srv_socket.sendto(message.encode(), (broadcast_ip, port))
            self.log(f"[SERVER] Broadcasted to ({broadcast_ip}, {port}): {message}")

    def uni_server_sonar(self, message, role):
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as srv_socket:
            sonar = self.robot_sonar.get(role)
            ip = sonar.ip
            port = sonar.port
            srv_socket.sendto(message.encode(), (ip, port))
            self.log("[SERVER] Sent to " + str(sonar.role) + " on (" + str(ip) + ", " + str(port) + ") : " + str(message))

    def uni_server_main(self, message):
        robot_main = self.robot_sonar.get("robot_main")
        if robot_main:
            with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as srv_socket:
                ip = robot_main.ip
                port = robot_main.port
                srv_socket.sendto(message.encode(), (ip, port))
                self.log(f"[SERVER] Sent to robot_main ({ip}, {port}) : {message}")

```

```python
    def check_alive_loop(self):
        while True:
            now = int(time.time()) # Current time in seconds
            for role, last_time in self.last_seen.items():
                if now - last_time > 30:
                    self.log(f"[SERVER] Lost contact with {role}! Last seen {now - last_time} seconds ago.")
                    if role not in self.lost_robots:
                        self.lost_robots.add(role)
                    self.try_reconnect(role)
                else:
                    self.lost_robots.discard(role)  # It's alive again
            time.sleep(5)

    def try_reconnect(self, role):
        self.log(f"[SERVER] Attempting to reconnect {role}...")
        msg = "ping : server , " + self.HOST + " , " + str(self.PORT)
        self.send(msg, "brd")

    def log(self, msg):
        print(msg)
        self.send_to_ui(msg)

    def send_to_ui(self, message):
        if self.HOST and self.UI_PORT:
            with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock:
                sock.sendto(message.encode(), (self.HOST, self.UI_PORT))

    def get_local_ip():
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        try:
            # dummy connect to find your IP
            s.connect(("8.8.8.8", 80))
            IP = s.getsockname()[0]
        except Exception:
            IP = "127.0.0.1"
        finally:
            s.close()
        return IP

if __name__ == '__main__':
    serv = Server()
```

Code Listing M.2: Python server

Sonar class: Sonar.py

```python
import time

class Sonar :
    def __init__(self, IP, Port, Role):
        self.ip = IP
        self.port = Port
        self.role = Role
        self.last_seen = int(time.time())
        self.distance = 0.0

    def update_distance(self, distance, seq):
        self.distance = distance
        self.seq = seq
        self.last_seen = int(time.time())
```

Code Listing M.3: Python sonar class

Full source code of the lilygo on the robot side

Full lilygo_bot.ino source code

```cpp
#include <SPI.h>
#include <LoRa.h>
#include <Arduino.h>
#include <Wire.h>

#include "motor_engine.h"

#define BAND 433E6
#define CONFIG_MOSI 27
#define CONFIG_MISO 19
#define CONFIG_CLK  5
#define CONFIG_NSS  18
#define CONFIG_RST  23
#define CONFIG_DIO0 26

#define SDCARD_MOSI 15
#define SDCARD_MISO 2
#define SDCARD_SCLK 14
#define SDCARD_CS   13

#define I2C_SLAVE_ADDR 0x40

```

```
// trapezoidal speed command parametter for turning, migration on GRiSP for this part
#define max_turn_speed 80
#define turn_acc 400


float I2C_command[2] = {0.0, 0.0}; // value received from GRiSP : {wheels acceleration , turn speed}

float freq_lim [13] = {300,200,175,150,125,100,90,80,70,60,50,40,30};
int size_test_freq = sizeof(freq_lim)/sizeof(freq_lim[0]);
int index_lim = 0;

// time mesure variable
unsigned long t_GRiSP;
unsigned long t_LORA;
unsigned long t_test;
unsigned long t_ESP;

// freq and period variable
float dt_GRiSP = 10;
float freq_GRiSP = 200;
float dt_ESP = 0;

// control byte received
byte cmd = 0; // received from LoRa communication and transfered to GRiSP
byte GRiSP_flags = 0; // Received from GRiSP

//control flag
bool new_cmd =false;
bool test = false;
bool disturb = false;
bool ext_end = true;



void setup() {
  Serial.begin(115200);
  // SPI - LoRa init
  SPI.begin(CONFIG_CLK, CONFIG_MISO, CONFIG_MOSI, CONFIG_NSS);
  LoRa.setPins(CONFIG_NSS, CONFIG_RST, CONFIG_DIO0);
  if (!LoRa.begin(BAND)) {
```

```
    Serial.println("Starting LoRa failed!");
    while (1);
  }

  // I2C Slave init, work with IRQ so no need to incorporate into the
   ↪ main loop
  Wire.begin(I2C_SLAVE_ADDR);
  Wire.onReceive(GRiSP_receiver);
  Wire.onRequest(GRiSP_sender);

  // motor init, works on core nÂř2
  engine_init();
  delay(1000);
  set_speed(0, 0);
  set_acceleration(0, 0);

  // time init
  t_GRiSP = millis();
  t_LORA = t_GRiSP;
  t_ESP = t_GRiSP;
}


void loop() {
  unsigned long new_t_ESP = millis();
  dt_ESP = (new_t_ESP - t_ESP) / 1000.0;
  t_ESP = new_t_ESP;
  LoRa_receiver();
  Event_handle();
  delay(1);
}



void GRiSP_receiver(int howMany) {
  if(howMany == 5){ // check if the packet match the expected lenght
    unsigned long new_t_GRiSP = millis();
    dt_GRiSP = (new_t_GRiSP - t_GRiSP) / 1000.0;
    freq_GRiSP = freq_GRiSP * 0.99 + 1.0 / dt_GRiSP * 0.01;
    t_GRiSP = new_t_GRiSP;

    byte A;
    byte B;
```

```
    if (Wire.available()) {

      // read and decode the wheel acceleration
      A = Wire.read();
      B = Wire.read();
      I2C_command[0] = decoder(A, B);

      // read and decode the differential turn speed
      A = Wire.read();
      B = Wire.read();
      I2C_command[1] = decoder(A, B);

      // read the flags
      GRiSP_flags = Wire.read();
    }

    //set acceleration
    if(!disturb && bitRead(GRiSP_flags, 4) && !bitRead(GRiSP_flags, 6)){
      set_acceleration(I2C_command[0], I2C_command[0]);
    }

    // Freeze
    if (bitRead(GRiSP_flags, 6)) {
        set_acceleration(0, 0);
        set_speed(0, 0);
    }

    // extension/retraction of the rising system
    if(bitRead(GRiSP_flags, 5)){
      ext_end = stand(-30,60.0);
    } else {
      ext_end = stand(0,60.0);
    }

  }

  // Empty the stack
  while(Wire.available()){
    Wire.read();
  }
}

```

```
void GRiSP_sender()
{
  byte v[5];
  float* speeds = get_speed();
  encoder(v, speeds[0]);
  encoder(v+2, speeds[2]);

  // control byte send to GRiSP witth : finsish extension/retraction
  ↪  flag and the command inputs
  v[4] = (cmd & 127) | (is_ready() * 128);

  //send
  Wire.write((byte*) v, sizeof(v));
}

double decoder(byte X, byte Y) {
  // decode half float to double

  byte A = (X & 192);
  if ((A & 64) == 0) A = A | 63;  // fill the missing exponent bytes
  ↪  with the right value
  byte B = ((X << 2) & 252) | ((Y >> 6) & 3);
  byte C = ((Y << 2) & 252);

  byte vals[] = { 0x00, 0x00, 0x00, 0x00, 0x00, C, B, A };
  double d = 0;
  memcpy(&d, vals, 8);

  return d;
}

void encoder(byte* res, double X){
  // encode double to half float
  byte vals[8];
  memcpy(vals, &X,8);
  byte A = vals[7];
  byte B = vals[6];
  byte C = vals[5];

  res[0] = (A&192)|((B>>2)&63);
  res[1] = ((B<<6)&192)|((C>>2)&63);

  return ;
```

```
}

void LoRa_receiver(){
// receiption of LoRa packets
  if (LoRa.parsePacket()) {
    if(LoRa.available()>=2){
      byte cmd1 = LoRa.read();
      byte cmd2 = LoRa.read();
      if(cmd1 == cmd2){
        cmd = cmd1;
        new_cmd = true;
      }
    }
    while (LoRa.available()){
      LoRa.read();
    }
  }
}

void Event_handle(){

  //emergency stop
  emergency(!bitRead(cmd, 7) || !bitRead(GRiSP_flags, 7));
  if (!bitRead(cmd, 7) || !bitRead(GRiSP_flags, 7)){
    set_acceleration(0, 0);
    set_speed(0, 0);
  }


  // start test procedure
  if(bitRead(cmd, 5)){
    test = true;
    t_test = millis();
  }
  if(test){
    //start the disturbance 500ms to let the record start
    if(millis()> t_test + 500){
      //disturb = true;
      //set_acceleration(40, 40);
    }
    // the disturbance is only applied between t=500 and t=800
    if(millis()> t_test + 800){
      disturb = false;
```

```
      test = false;
    }
  }

  set_turn(I2C_command[1]);

  new_cmd =false;
}
```

Listing M.10: Full `lilygo_bot.ino` code.

Full `motor_engine.cpp` source code

```
#include "motor_engine.h"
TaskHandle_t Motor_engine_task;

int long total_stepA = 0;
int long total_stepB = 0;
int long total_stepC = 0;

//motor dir
int dirA = 0;
int dirB = 0;
int dirC = 0;

//motor dt
int dt_MA = 0;
int dt_MB = 0;
int dt_MC = 0;

//motor avance speed, rot/s
float v_MA = 0;
float v_MB = 0;
float v_MC = 0;



// Total wheel speed
float v_tot_A = 0;
float v_tot_B = 0;
float v_tot_C = 0;
```

```
float v_diff =0.0;

//motor accelerations, only applied on motor avance speed, rot/sÂš
float a_MA = 0;
float a_MB = 0;
float a_MC = 0;

//motor dist
float total_dist_A = 0;
float total_dist_B = 0;
float total_dist_C = 0;

//motor freq
double f_MA = 0;
double f_MB = 0;
double f_MC = 0;

//Rise system parameters
int target_step = 0;
float Speed_stand =0;

float v_max_ang = v_max /(PI * diameter);

void engine_init() {

  pinMode(MOTOR_AC_EN_PIN, OUTPUT);
  pinMode(MOTOR_B_EN_PIN, OUTPUT);
  digitalWrite(MOTOR_AC_EN_PIN, LOW);
  digitalWrite(MOTOR_B_EN_PIN, LOW);

  pinMode(MOTOR_A_STEP_PIN, OUTPUT);
  pinMode(MOTOR_B_STEP_PIN, OUTPUT);
  pinMode(MOTOR_C_STEP_PIN, OUTPUT);
  digitalWrite(MOTOR_A_STEP_PIN, LOW);
  digitalWrite(MOTOR_B_STEP_PIN, LOW);
  digitalWrite(MOTOR_C_STEP_PIN, LOW);

  pinMode(MOTOR_A_DIR_PIN, OUTPUT);
  pinMode(MOTOR_B_DIR_PIN, OUTPUT);
  pinMode(MOTOR_C_DIR_PIN, OUTPUT);
  digitalWrite(MOTOR_A_DIR_PIN, LOW);
  digitalWrite(MOTOR_B_DIR_PIN, LOW);
```

```
  digitalWrite(MOTOR_C_DIR_PIN, HIGH);


  xTaskCreatePinnedToCore(
    Motor_engine,       /* Task function. */
    "Motor_engine",     /* name of task. */
    10000,              /* Stack size of task */
    (void*)NULL,        /* parameter of the task */
    1,                  /* priority of the task */
    &Motor_engine_task, /* Task handle to keep track of created task */
    1);                 /* pin task to core 1 */
}


void Motor_engine(void* Parameters) {
  //setup
  Serial.print("Motor Engine running on core ");
  Serial.println(xPortGetCoreID());


  unsigned long t_MA = 0;
  unsigned long t_MB = 0;
  unsigned long t_MC = 0;

  unsigned long t_motor = micros();
  unsigned long dt_motor;
  unsigned long new_t_motor;

  //loop
  while (true) {

    //time calculation
    new_t_motor = micros();
    dt_motor = new_t_motor - t_motor;
    t_motor = new_t_motor;
    t_MA += dt_motor;
    t_MB += dt_motor;
    t_MC += dt_motor;


    // incrementation of the stepper motors
    if (dirA != 0 && t_MA > dt_MA) {
      digitalWrite(MOTOR_A_STEP_PIN, HIGH);
```

```
      total_stepA += dirA;
      t_MA = 0;
    } else {
      digitalWrite(MOTOR_A_STEP_PIN, LOW);
    }

    if (dirB != 0 && t_MB > dt_MB) {
      digitalWrite(MOTOR_B_STEP_PIN, HIGH);
      total_stepB += dirB;
      t_MB = 0;
    } else {
      digitalWrite(MOTOR_B_STEP_PIN, LOW);
    }

    if (dirC != 0 && t_MC > dt_MC) {
      digitalWrite(MOTOR_C_STEP_PIN, HIGH);
      total_stepC += dirC;
      t_MC = 0;
    } else {
      digitalWrite(MOTOR_C_STEP_PIN, LOW);
    }

    engine_update(dt_motor); // uptate each motor step period
  }
}



void set_speed(float vA, float vC) {
  v_MA = vA/(PI * diameter);
  v_MC = vC/(PI * diameter);
}

void set_angular_speed(float vA, float vC) {
  v_MA = vA;
  v_MC = vC;
}

void set_acceleration(float aA, float aC) {
  a_MA = aA/(PI * diameter);
  a_MC = aC/(PI * diameter);
}

```

```
void set_angular_acceleration(float aA, float aC) {
  a_MA = aA;
  a_MC = aC;
}

float list_speed[3] = { 0.0, 0.0, 0.0 };
float* get_speed() {
  list_speed[0] = v_tot_A*(PI * diameter);
  list_speed[1] = v_tot_B * lever_ratio;
  list_speed[2] = v_tot_C*(PI * diameter);
  return list_speed;
}

float list_dist[3] = { 0.0, 0.0, 0.0 };
float* get_dist() {
  list_dist[0] = total_stepA * 1.0 / (steps * microsteps) * diameter * PI;
  list_dist[1] = total_stepB * 1.0 / (steps * microsteps) * lever_ratio;
  list_dist[2] = total_stepC * 1.0 / (steps * microsteps) * diameter * PI;
  return list_dist;
}

//low speed loop
void engine_update(unsigned long dt_loop) {

  //acceleration calculation of motor A with limit to +/- vmax
  v_MA += a_MA * dt_loop * 1e-6 ;
  if (v_MA > v_max_ang) {
    v_MA = v_max_ang;
  } else if (v_MA < -v_max_ang) {
    v_MA = -v_max_ang;
  }

  //displacement calculation of motor B to set the extension/retraction speed
  //and the wheel counter rotation
  if (total_stepB > target_step){
    v_MB = Speed_stand / lever_ratio;

  } else if (total_stepB < target_step){
    v_MB = -Speed_stand / lever_ratio;
```

```
  } else {
    v_MB = 0;
  }

  //acceleration calculation of motor A with limit to +/- vmax
  v_MC += a_MC * dt_loop * 1e-6;
  if (v_MC > v_max_ang) {
    v_MC = v_max_ang;
  } else if (v_MC < -v_max_ang) {
    v_MC = -v_max_ang;
  }

  //total speeds
  v_tot_A = v_MA + v_diff;
  v_tot_B = v_MB;
  v_tot_C = v_MC - v_diff;


  //compute freq from speed
  f_MA = v_tot_A * steps * microsteps;
  f_MB = v_tot_B * steps * microsteps;
  f_MC = v_tot_C * steps * microsteps;

  //compute the increment period from freq and set the direction
  if (f_MA < 0) {
    dt_MA = 1e6 / abs(f_MA);
    digitalWrite(MOTOR_A_DIR_PIN, HIGH);
    dirA = 1;
  } else if (f_MA > 0) {
    dt_MA = 1e6 / abs(f_MA);
    digitalWrite(MOTOR_A_DIR_PIN, LOW);
    dirA = -1;
  } else {
    dirA = 0;
  }


  if (f_MB > 0) {
    dt_MB = 1e6 / abs(f_MB);
    digitalWrite(MOTOR_B_DIR_PIN, LOW);
    dirB = -1;
  } else if (f_MB < 0) {
```

```
    dt_MB = 1e6 / abs(f_MB);
    digitalWrite(MOTOR_B_DIR_PIN, HIGH);
    dirB = 1;
  } else {
    dirB = 0;
  }

  if (f_MC > 0) {
    dt_MC = 1e6 / abs(f_MC);
    digitalWrite(MOTOR_C_DIR_PIN, HIGH);
    dirC = -1;
  } else if (f_MC < 0) {
    dt_MC = 1e6 / abs(f_MC);
    digitalWrite(MOTOR_C_DIR_PIN, LOW);
    dirC = 1;
  } else {
    dirC = 0;
  }
}

void reset_dist() { //reset function
  total_stepA = 0;
  total_stepB = 0;
  total_stepC = 0;
}

void set_turn(float angular_speed){ // apply a differential speed on
↪  the wheel to turn
  v_diff = (lenght_btw_wheels*PI*angular_speed / (180.0*2))/(PI *
  ↪  diameter) ;
}

void emergency(bool stop){ //shut off the power
  if(stop){
    digitalWrite(MOTOR_AC_EN_PIN, HIGH);
    digitalWrite(MOTOR_B_EN_PIN, HIGH);
  }else{
    digitalWrite(MOTOR_AC_EN_PIN, LOW);
    digitalWrite(MOTOR_B_EN_PIN, LOW);
  }
}

bool stand(float angle, float speed){
```

```
  target_step = angle/ (360 * lever_ratio) * steps * microsteps;
  Speed_stand = speed*1.0/360.0 ; // go from deg/s to rot/s
  return total_stepB == target_step;
}


bool is_ready(){
  return total_stepB == target_step;
}
```

Listing M.11: Full `motor_engine.cpp` code.

Full `motor_engine.h` source code

```
#pragma once

//#include <stdint.h>
#include <Arduino.h>

#define sgn(x) ((x) < 0 ? -1 : ((x) > 0 ? 1 : 0))

#define MOTOR_AC_EN_PIN 14
#define MOTOR_A_STEP_PIN 12
#define MOTOR_A_DIR_PIN 13
#define MOTOR_B_EN_PIN 15
#define MOTOR_B_STEP_PIN 2
#define MOTOR_B_DIR_PIN 0
#define MOTOR_C_STEP_PIN 4
#define MOTOR_C_DIR_PIN 25

#define v_max  100 // speed max, in cm/s
#define microsteps 4
#define steps 400
#define diameter 10 // cm
#define lever_ratio 0.09090909090909090909090909090909 //    12/132
#define lenght_btw_wheels 24 // cm

void engine_init();

void Motor_engine(void *);

```

```
void engine_update(unsigned long dt_loop);

void set_speed(float,float);

void set_angular_speed(float,float);

void set_acceleration(float,float);

void set_angular_acceleration(float,float);

float* get_speed();

float* get_dist();

void reset_dist();

void set_turn(float);

void emergency(bool);

bool stand(float,float);

void raise_dir(int);

bool is_ready();
```

Listing M.12: Full `motor_engine.h` code.

Full source code of the lilygo on the user side

Full `liygo_user.ino` source code

```
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>


#define Pin 15
#define Buzz 13

#define LORA_PERIOD 433
```

```
#define BAND 433E6


#define CONFIG_MOSI 27
#define CONFIG_MISO 19
#define CONFIG_CLK  5
#define CONFIG_NSS  18
#define CONFIG_RST  23
#define CONFIG_DIO0 26



unsigned long t ;
int state = 0;
int prevstate = 0;
byte cmd;

void setup()
{
  // init serial
  Serial.begin(115200);
  while (!Serial);


  // init LoRa
  SPI.begin(CONFIG_CLK, CONFIG_MISO, CONFIG_MOSI, CONFIG_NSS);
  LoRa.setPins(CONFIG_NSS, CONFIG_RST, CONFIG_DIO0);
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }

  //init Pin
  pinMode(Pin, OUTPUT);
  pinMode(Pin, INPUT_PULLUP);
  pinMode(Buzz, OUTPUT);
  digitalWrite(Buzz, LOW);

  prevstate = esp_sleep_get_wakeup_cause()!=ESP_SLEEP_WAKEUP_TIMER &&
  ↪ !digitalRead(Pin);
  t = millis();
}
```

```
int count = 0;

//main loop
void loop(){
  Keyboard_input();
  LoRA_sender();
}

void LoRA_sender(){

  state = digitalRead(Pin);

  if(state==HIGH){
    cmd = cmd & 127;
  }

  LoRa.beginPacket();
  // send two packet for redundancy
  LoRa.write(cmd);
  LoRa.write(cmd);
  LoRa.endPacket();
  Serial.println(cmd, BIN);


  //buzzer logic
  if(state == HIGH && prevstate == LOW){
    digitalWrite(Buzz, HIGH);
    delay(100);
    digitalWrite(Buzz, LOW);
  }


  if(state == LOW && prevstate == HIGH){
    digitalWrite(Buzz, HIGH);
    delay(100);
    digitalWrite(Buzz, LOW);
  }
  prevstate = state;
}


void Keyboard_input(){
  if (Serial.available() > 0) {
```

```
    cmd = Serial.read();
  }

  while(Serial.available()){
    Serial.read();
  }
}
```

Listing M.13: Full `liygo_user.ino` code [22].

Full `board_def.h` source code

```
#include <Arduino.h>

#define LORA_V1_0_OLED  0
#define LORA_V1_2_OLED  0
#define LORA_V1_6_OLED  1
#define LORA_V2_0_OLED  0

// Change here whether you define Sender or Receiver.
// The rest should be the same
//#define LORA_SENDER 0
#define LORA_SENDER 1

#define LORA_PERIOD 433
// #define LORA_PERIOD 915
//#define LORA_PERIOD 433

#if LORA_V1_0_OLED
#include <Wire.h>
#include "SSD1306Wire.h"
#define OLED_CLASS_OBJ  SSD1306Wire
#define OLED_ADDRESS    0x3C
#define OLED_SDA    4
#define OLED_SCL    15
#define OLED_RST    16
#define CONFIG_MOSI 27
#define CONFIG_MISO 19
#define CONFIG_CLK  5
#define CONFIG_NSS  18
#define CONFIG_RST  14
```

```
#define CONFIG_DIO0 26
// !     There are two versions of TTGO LoRa V1.0,
// !     the 868 version uses the 3D WiFi antenna, and the 433 version uses the PCB antenna.
// !     You need to change the frequency according to the board.

#define SDCARD_MOSI -1
#define SDCARD_MISO -1
#define SDCARD_SCLK -1
#define SDCARD_CS   -1

#elif LORA_V1_2_OLED
//Lora V1.2 ds3231
#include <Wire.h>
#include "SSD1306Wire.h"
#define OLED_CLASS_OBJ  SSD1306Wire
#define OLED_ADDRESS    0x3C
#define OLED_SDA    21
#define OLED_SCL    22
#define OLED_RST    -1
#define CONFIG_MOSI 27
#define CONFIG_MISO 19
#define CONFIG_CLK  5
#define CONFIG_NSS  18
#define CONFIG_RST  23
#define CONFIG_DIO0 26

#define SDCARD_MOSI -1
#define SDCARD_MISO -1
#define SDCARD_SCLK -1
#define SDCARD_CS   -1

#define ENABLE_DS3231

#elif LORA_V1_6_OLED
#include <Wire.h>
#include "SSD1306Wire.h"
#define OLED_CLASS_OBJ  SSD1306Wire
#define OLED_ADDRESS    0x3C
#define OLED_SDA    21
#define OLED_SCL    22
#define OLED_RST    -1

```

```
#define CONFIG_MOSI 27
#define CONFIG_MISO 19
#define CONFIG_CLK  5
#define CONFIG_NSS  18
#define CONFIG_RST  23
#define CONFIG_DIO0 26

#define SDCARD_MOSI 15
#define SDCARD_MISO 2
#define SDCARD_SCLK 14
#define SDCARD_CS   13

#elif LORA_V2_0_OLED
#include <Wire.h>
#include "SSD1306Wire.h"
#define OLED_CLASS_OBJ  SSD1306Wire
#define OLED_ADDRESS    0x3C
#define OLED_SDA    21
#define OLED_SCL    22
#define OLED_RST    -1

#define CONFIG_MOSI 27
#define CONFIG_MISO 19
#define CONFIG_CLK  5
#define CONFIG_NSS  18
#define CONFIG_RST  23
#define CONFIG_DIO0 26

#define SDCARD_MOSI 15
#define SDCARD_MISO 2
#define SDCARD_SCLK 14
#define SDCARD_CS   13

#else
#error "please select board"
#endif


#if LORA_PERIOD == 433
#define BAND 433E6
#elif LORA_PERIOD == 868
#define BAND 868E6
#elif LORA_PERIOD == 915
```

```
#define BAND 915E6
#else
#error "Please select the correct lora frequency"
#endif
```

Listing M.14: Full `board_def.h` code [22].

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**