"Dynamic balancing in the real world with GRiSP"

Goens, François ; Ponsard, Cédric

ABSTRACT

Unstable systems are present in many engineering domains, such as industrial plants, energy production, aeronautics, transportation and medical. Such systems are really challenging to control in order to achieve specific missions, such as controlling complex chemical processes, putting a satellite into orbit, restarting a human heart, or even deploying legged or self-balancing robots. It has been, and still is, at the heart of many innovations. The objective of this master thesis is to control an unstable system using a GRiSP board. This compact circuit board features a wide range of ports and connectivity while running under Erlang, a multi-paradigm language that offers a number of features of great interest in the world of control. This master thesis builds on top of the Hera framework developed over several past master theses to enable the use of Kalman filter-based sensor fusion, a very powerful tool for measurement enhancement. As a concrete, yet representative and generalisable use case, a real-world unstable system was selected and built: the two-wheeled self-balancing robot. After designing such a device to be compatible with GRiSP, a control strategy was established to make it dynamically balanced. The software was then created and optimised to accommodate this theoretical control loop and the various associated sub-components. A Kalman filter, fed by multiple sensors, was implemented, along with a simpler complementary filter, allowing for performance comparison. Additional features that make this robot stand out are its lifting mechanism and the possibility of ...

CITE THIS VERSION

UCLouvain

epl

École polytechnique de Louvain

Dynamic balancing in the real world with GRiSP

Two-wheeled self-balancing robot with Hera

Authors: **François Goens, Cédric Ponsard**
Supervisor: **Peter Van Roy**
Readers: **Benoît Herman, Peer Stritzinger**
Academic year 2023–2024
Master [120] in Electro-mechanical Engineering

Abstract

Unstable systems are present in many engineering domains, such as industrial plants, energy production, aeronautics, transportation and medical. Such systems are really challenging to control in order to achieve specific missions, such as controlling complex chemical processes, putting a satellite into orbit, restarting a human heart, or even deploying legged or self-balancing robots. It has been, and still is, at the heart of many innovations.

The objective of this master thesis is to control an unstable system using a GRiSP board. This compact circuit board features a wide range of ports and connectivity while running under Erlang, a multi-paradigm language that offers a number of features of great interest in the world of control. This master thesis builds on top of the Hera framework developed over several past master theses to enable the use of Kalman filter-based sensor fusion, a very powerful tool for measurement enhancement.

As a concrete, yet representative and generalisable use case, a real-world unstable system was selected and built: the two-wheeled self-balancing robot. After designing such a device to be compatible with GRiSP, a control strategy was established to make it dynamically balanced. The software was then created and optimised to accommodate this theoretical control loop and the various associated sub-components. A Kalman filter, fed by multiple sensors, was implemented, along with a simpler complementary filter, allowing for performance comparison. Additional features that make this robot stand out are its lifting mechanism and the possibility of being remotely controlled. This system is multilevel with two levels: an outer executive loop deciding how the robot moves around, and an inner stability loop ensuring stability.

This results in the creation of a totally autonomous robot capable of positioning itself in its balancing state. This system is very powerful and reliable. It is able to move agilely while withstanding most external disturbances that were experimentally characterised and compared with other solutions. The whole architecture supports the development of similar systems in various fields, such as robotics, automation or IoT, through the reuse and adaptation of parts like the outer state machine loop or the inner control loop. It is also well documented and comes with a set of development and debugging tools.

Acknowledgements

Many people helped us to achieve this master thesis and without them it would probably not have been possible, so we would like to thank them, in particular:

Our ***families*** *and* ***friends*** *for their unconditional support throughout our studies and particularly during the completion of our master thesis.*

Nicolas Isenguerre *for his collaboration on the common part of our master thesis. As well as the other students in the workspace, such as* ***Zoé***, ***Pauline***, ***Marion*** *and* ***Camille***, *for their advice and support, as well as for the good times we had together.*

Peter Van Roy, *our supervisor, for his enthusiasm, interest and invaluable advice throughout the master thesis.*

Peer Stritzinger and his team, *who provided technical support on GRiSP and Erlang. Their advice and their vision of the future of our work helped to guide us in the realisation of our project.*

All the members of the ***CREDEM*** *platform, in particular* ***Simon De Jaeger*** *who helped us with the prototyping of our system.*

Gianluca Bianchin *and* ***Guillaume Fontaine*** *for giving us access to the linear automation course lab to carry out experiments.*

Vanessa Maons *for taking care of our administrative files and providing us access to an openspace, the* ***cleaning team*** *for keeping this space clean throughout the year.*

More generally, we would like to thank ***the education provided by the EPL***, *most of whose course have been useful during the realisation of this master thesis. As well as to all people directly or indirectly linked to the realization of this work.*

AI use disclaimer

No LLM or generative AI was employed in this master thesis, with the exception of the graphical interface in Python for command display.

Translation and grammar correcting tools using non-generative AI, namely DeepL and QuillBot, were used to improve the level of English of the text.

Acronyms

AWGN Additive White Gaussian Noise

BIBO Bounded-Input Bounded-Output

CAD Computer-Aided Design

CG Center of Gravity

DIY Do It Yourself

DoF Degree of Freedom

ESC Electronic Stability Control

ETS Erlang Term Storage

FDM Fused Deposition Modelling

FSM Finite-State Machine

GPIO General Purpose Input Output

I^2C Inter-Integrated Circuit

IIR Infinite Impulse Response

IMU Inertial Measurement Unit

IoT Internet of Things

JTAG Joint Test Action Group

LQR Linear-quadratic regulator

NIF Native Implemented Function

OLED Organic Light-Emitting Diode

OS Operating System

OTP Open Telecom Platform

PID Proportional-Integral-Derivative

Pmod Peripheral module

ROS Robot Operating System

ROSiE Robot Operating System in Erlang

RTEMS Real-Time Executive for Multiprocessor Systems

SPI Serial Peripheral Interface

UART Universal Asynchronous Receiver-Transmitter

Contents

Introduction

Context

Instabilities are phenomena encountered in many situations, whether in everyday life or in more specific cases. Often, instability is synonymous with insecurity, as it refers to the absence of predictability and control. These instabilities can take many forms: a car on an extremely slippery road, a plane in a stall, a nuclear reaction... Fortunately, humans have learned to master these instabilities and put them to good use in many areas. Cars are equipped with Electronic Stability Control (ESC) systems, aircraft are equipped with stall warning systems, nuclear fission is harnessed by control rods, etc. Although humans themselves are unstable by nature, they are capable of balancing dynamically on their own two feet, using their various senses. It is a concept that is not unique to humans: many high-tech objects are equipped with dynamic self-balancing systems, from boat stabilizers to anti-seismic systems in skyscrapers, and even small personal transportation devices such as gyropods and electric unicycles.

These instabilities are handled by control systems, their purpose is to set the actions that must be applied to the system to stabilise it. However, in most cases, such controllers need to know the current state of the system: as mentioned above, human beings must use their senses to maintain equilibrium. From a technological point of view, sensors are used but their practical use is often limited by a number of factors such as measurement frequency, accuracy and noise. The "garbage in, garbage out" principle also applies to control systems, so it is essential to have low-noise measurements at the controller input.

Filters are commonly employed to improve sensor measurements. A further concept for achieving noise reduction is sensor fusion, it involves combining data from different sensors. For example, a human being not only uses his inner ear for balance, he also uses his vision, proprioception and sense of touch.

Sensor fusion is what is offered by Hera, an Erlang framework designed for the Internet of Thing (IoT) on the GRiSP circuit board. Hera is based on the Kalman filter, a very special tool that combines sensors dynamically using a predictive model.

Objective

The main objective of this master thesis is to create a library for controlling an unstable system using Hera functionalities (see Section 1.3). In addition, the creation of an unstable system serves as a proof of concept for an autonomous robotic system using

GRiSP (see Section 1.1) and real time sensor fusion with Hera. A more precise description of the objectives is given in Section 2.1.2. It involves the selection of a concrete yet representative and generic enough application case: a two-wheeled self-balancing system.

Previous developments

The Erlang programming language, in the context of IoT, is a highly suitable language for creating distributed systems. Its fault-tolerancy and its ability to manage hotcode loading is particularly useful for systems that must be running continuously.

Using this programming language, a framework has been created, the Hera platform, with the purpose of developing IoT devices functionalities. It is an open-source state-of-the-art fault-tolerant platform. Hera has been developed to be used on a GRiSP board, an IoT platform with Pmod GPIO ports and an Erlang-based software platform developed by Stritzinger GmbH.

The Hera platform enables sensor fusion by providing Kalman filtering which reduces noise on the measures collected by a set of sensors. This is critical in a real-time application where noise can have a high impact on the system. Hera also provides support for communication between multiple GRiSP boards which makes it a good candidate for constructing an IoT network.

Contributions

The targeted objectives are reached thanks to a large number of contributions made throughout this master thesis. These contributions include :

- The design and prototyping of a two-wheeled robot with a lifting system enabling to raise the robot after a fall, allowing it to return to a state of self-balance by its own efforts.
- The design and implementation of a software stability engine to keep the robot in balance.
- Implementation of a multilevel system with an executive loop supervising a stability loop[1].
- The use of a Kalman filter and complementary filter in such a system to improve the measurements made by the sensors.
- The implementation of a stepper motor driver specific to this application, but general enough to be adapted to other applications.
- The implementation of a half-precision float representation of decimal values for faster communications.
- Remote control for robot displacement and emergency stop system via LoRa communication.
- A wide range of tests to determine the behaviour and limits of the system

[1]This follows the Erlang philosophy, but applied to hardware.

All these contributions have made it possible to prove that both the hardware (GRiSP) and software (Erlang and Hera) tools used are powerful enough to run such a real time system. The designed robot provides a good basis for use in many future applications. For example, by adding actuators so that it can interact with the world around it, or by having several of these robots communicating to carry out a common task.

A short video of the robot executing different movements can be found at this url: `https://youtu.be/-GYXGzXmlVE`. All the code developed for the robot can be found in this GitHub repository: `https://github.com/FrancoisGgg/balancing_robot`.

Roadmap

This master thesis begins by explaining the context to understand the subject. Once the context has been established, the various tools developed previously and used in this mater thesis are detailed.

The rest of the content is then divided into three different parts:

- Part I explains the general design of the project. Firstly, by clearly redefining the objectives to be achieved. This part then moves on to the physical design of the robot, in Chapter 3, before explaining the design of the four main parts of the robot in the four chapters, Chapter 4, 5, 6 and 7, that follow: preprocessing, controller, motor driver and executive controller.
- Part II deals with the robot's software. This part begins with Chapter 8, which describes the software implementation of the robot components covered in chapters 4 to 7. Then all the problems that led to software optimisation are highlighted in Chapter 9.
- In the last part, Part III, the robot is pushed to its limits to evaluate its behaviour and to be characterised in different situations detailed in Chapter 10. The results are also discussed and compared with other similar work in Chapter 11.

This master thesis ends on the conclusion and possible future work, describing the avenues to be explored for future projects in which the robot can make a contribution.

Some documents, such as the source code, but also more detailed mathematical developments can be found in the appendices or in the bibliography at the end of this master thesis in Part IV.

Chapter 1

Background

Throughout this master thesis, a variety of theoretical and technological concepts are used. In this section, a number of relevant projects, researches and tools developed previously are presented.

1.1 GRiSP board

GRiSP boards are circuit boards .

The GRiSP board [1], shown in Figure 1.1, is a compact circuit board developed by Stritzinger GmbH[2], with plenty of different ports. This board can be programmed in the Erlang programming language. It offers plenty of possibilities regarding the subject of IoT. It has been designed to be a prototyping tool. Two types of GRiSP boards exist: the GRiSP-Base board and the GRiSP 2 board. The GRiSP 2 board is the second generation of GRiSP boards. In addition to its multitude of ports such as Pmod [3], it can also communicate via Wi-Fi.

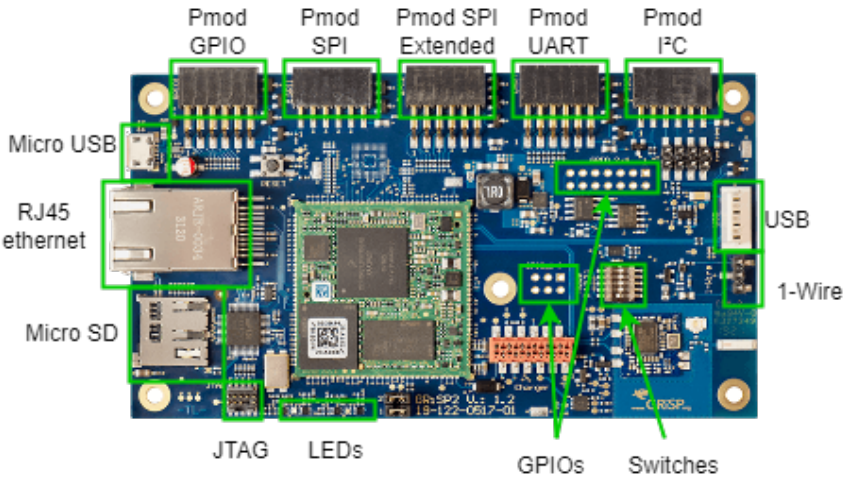


Figure 1.1: GRiSP 2 circuit board, modified pictures from [1].

RTEMS

RTEMS is an operating system OS used in embedded devices such as in the medical industry, for space flights and for networking [4]. It stands for Real-Time Executive for Multiprocessor Systems (RTEMS). Running Erlang on RTEMS allows for very little latency between something happening in the real world and the reaction to this change in Erlang. This Operating System (OS) allows to run an Erlang shell on the GRiSP and to run programs written in Erlang.

Pmod sensors

The GRiSP 2 board has six different Digilent Pmod compatible connectors, like the one shown in Figure 1.2. Pmods are small circuit boards allowing to extend the capabilities of embedded systems. It can give access to new ports like microSD or VGA. It can implement communication modules like Wi-Fi or Bluetooth. Another option is to have input and output devices like a joystick or an Organic Light-Emitting Diode (OLED) screen and it also gives access to many different sensors like a gyroscope or a colour sensor.

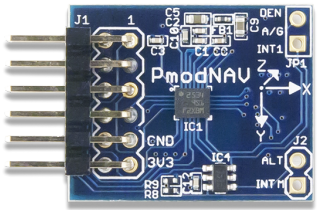


Figure 1.2: Digilent Pmod NAV sensor [5].

1.2 Erlang programming language

Erlang/OTP[1] is a multiparadigm programming language designed for building scalable distributed systems [6]. Erlang is a fault-tolerant language based on lightweight processes. This means that it allows some parts of the system to crash while keeping the rest of the system running. Furthermore, it allows the system to restart its crashed processes with the use of supervisors. These are processes with the only purpose of restarting crashed processes. This is where the tagline of Erlang comes from: "*Let it crash*".

Another useful feature of Erlang is called hotcode loading. This allows to update the code of a system without shutting it down. This is particularly important for systems that must be highly available.

The last major advantage of Erlang resides in its handling of concurrency. As Erlang is designed with this potential issue in mind it allows to run easily thousands of individual processes on the same core without having to care about concurrency issues.

Erlang is a high-level real-time language, which makes programming easier but is not optimal for calculation performance. To solve this problem, Erlang allows the use of native implemented functions (NIFs). NIFs are functions written in the C language but

[1]Open Telecom Platform (OTP)

that can be called in an Erlang program. This allows to speed up the program, which is critical in a real-time application.

Finally Erlang runs on a virtual machine called BEAM with compilation to a specific bytecode.

1.3 Hera framework

The Hera platform is an Erlang framework allowing sensor fusion with multiple sensors and GRiSP boards with asynchronous measurements. This framework was developed and improved by several previous master theses [7]–[10]. Sensor fusion inside Hera is made possible by the use of a Kalman filter. This filter is detailed in Section 1.7.

Hera has a simple supervision tree composed of two different supervisors and three different kinds of processes. As shown in 1.3, these are: **hera_sup** which supervises **hera_data** and **hera_com** and **hera_measure_sup** which supervises every **hera_measure** procedure. To keep it simple, **hera_data** is used to handle the storage of data and **hera_com** makes the communication of data between the different GRiSP boards and from and to each GRiSP board itself. The only files that a user of Hera must provide are the measurement files that extend **hera_measure**.

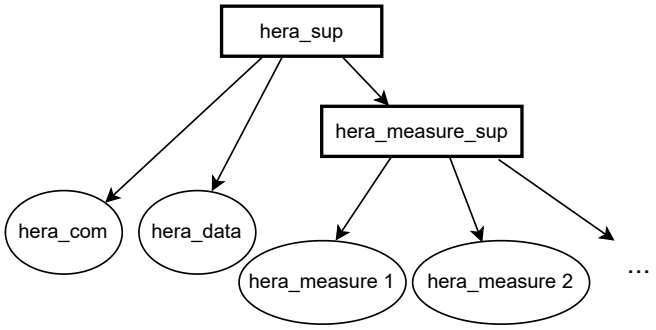


Figure 1.3: Hera supervision tree.

This framework is the only one available on GRiSP. Other more well-known and more general frameworks existed on the market before this one. Here are a few examples:

- The Thymio robot with its VPL framework for educational purposes [11].
- The ROS framework[2] offers many tools to develop the software part of a robot [12]. A version of ROS has been developed to run in the Erlang language. This framework is called ROSiE for ROS in Erlang [13].
- The Orocos, standing for Open Robot Control Software, framework is a set of libraries written in C++ with the aim of developing advanced machine and robot control [14].

Other examples are the RT-middleware, Robot Framework and MoveIt frameworks [15]–[17].

[2]Robot Operating System

1.4 Self balancing devices

The concept of dynamic balancing is widely used in control theory, and more specifically in the field of robotics. There are a multitude of applications requiring dynamic balance control. For example:

- **Aerospace systems**: the general stability of airplanes is designed to be naturally stable thanks to their aerodynamic characteristics, but this is not the case for rockets, where a notion of stability control is required. This applies mainly to fin control, thrust vectoring, etc. The control of such vehicles is very complex as there are at least 3 unstable degrees of freedom (DoF) : pitch, roll and yaw. Furthermore, such a controller should be compliant with the evolution of aerodynamic behaviour during the flight. More accessible flying systems, such as unmanned aerial vehicles, also require stabilisation control while being autonomous.
- **Mobile robotics**: the two-wheeled self-balancing robot is a well know inverted-pendulum system and is is used in many applications such as the self-balancing scooters by Segway for people transportation [18]. The I-bot [19] or Kim-e [20] wheelchair enables people with reduced mobility to move around thanks to a self-balancing system, while allowing them to be positioned at the same height as standing people. As for payload transport, there's evoBOT [21] by Fraunhofer or Handle [22] developed by Boston dynamics. All these applications have the major advantage of being compact. It's this compactness that makes them unstable, as it is very difficult to keep the center of gravity above the very small area of contact with the ground, then requiring balancing control on their one unstable DoF.
- There are many other types of robots requiring dynamic balancing, but they have no real application. The ball-moving platform system, the reaction wheel balancer, the actuated double pendulum and the mobile base balancer are well-known study cases but don't have many applications.

A picture of the Handle robot made by Boston Dynamics is shown Figure 1.4.

Figure 1.4: Handle robot of Boston Dynamics.

Two-wheeled self-balancing robots are very popular. They are easy to manufacture and are affordable. Many kits supplying this type of systems are available on the market.

1.5 Two-wheeled self-balancing devices

As said previously, these robots are very popular and widespread. Literature research focused on robots / prototypes built for research and educational purposes, in order to have a coherent basis for comparison with the system designed for this master thesis. Such robots exist under very different shapes and forms and can be constructed in many different ways. The differences are expressed on various points: the type of hardware, the type of actuator used, but also the type of controller or even the type of filter used in preprocessing. They also have common points such as the use of Inertial measurement unit(IMU) senors in order to compute the leaning angle.

Robots based on the same type of computing hardware often share similar characteristics. They have therefore been grouped into 3 categories:

- **Programmable control boards**: These boards are easy to use if having some bases in programming. Once the program has been written, the boards must be flashed so that they can run the code. The best-known examples of these boards are the Arduinos control boards. There are other lesser-known boards of this type, such as the Pololu Balboa 32U4, the NI myRIO-1900 and the OpenCR boards. Several examples of robots with these boards have been built, as described in [23]–[30].
- **LEGO Mindstorm**: These boards can be programmed with scratch or a scratch-like language. This makes the LEGO Mindstorm [31] one of the easiest ways of constructing a robot without having any programming knowledge. It is a plug-and-play system, which means that every component can simply be connected and will work without any effort. A robot using the Mindstorm is sold as a kit. This robot is the Gyroboy from LEGO [32], [33].
- **Small embedded computer**: These boards allow to take much more complete applications and have much better performances due to the fact that they are computers. An exemples of these boards is the RaspberryPi. A robot using this board can be found in the ADRC robot[34]. The swarm robot[30] uses a nvidia jetson agx computer.

Through literature, it has been noticed that almost all of them require the intervention of a human to place the device in its balancing state, which means that they are not very autonomous.

1.6 Control strategies

The controller play a key role in the design of a two-wheeled self-balancing robot. It is the brains of the loop computing the actuation required to stabilize the robot.

- **Proportional-Integral-derivative controllers (PID)** are very popular and easy to use. Many control architectures use PID controllers as a building block. A more advanced technique based on the PID is the fuzzy PID controller. In the case of this master thesis, the use of PID controllers is at the heart of the stabilization algorithm, as explained in Section 5.2.
- **Linear-quadratic regulator (LQR)** [35, p. 25] Linear-quadratic regulators are

also frequently used to control self-balancing devices. It's a calculation method for determining the different parameters of a state regulator based on the minimization of a performance criterion.

- **Neural network** can also be used to control this type of robot. These neural networks have a highly abstract operating logic, and are not directly encoded as they are. A period of training is required to find a set of parameters called weights that describe the neural network. The system must therefore also be modeled numerically in order to be able to train the network in a virtual environment. In addition, it is necessary to design an evaluation function so that the algorithm generating the neural network can identify good and bad behaviour.

1.7 Filters

This section describes different filters that can be used to combine sensors in order to enhance the data measured by the sensors. As will be explained in more detail in Chapter 4, the sensors used are the accelerometer and the gyroscope. The following explanations on the different filters will thus be limited to the scope of these two sensors.

1.7.1 Complementary filter

The complementary filter [36] combines different signals carrying the same information but affected with different types of noise. The combination is done by summing each signal after passing through individual filters $F(s)$ as depicted in Figure 1.5. Complementary means that those filters are designed in a way that the sum of their transfer function is 1 [37]:

$$\sum F_k(s) = 1 \tag{1.1}$$

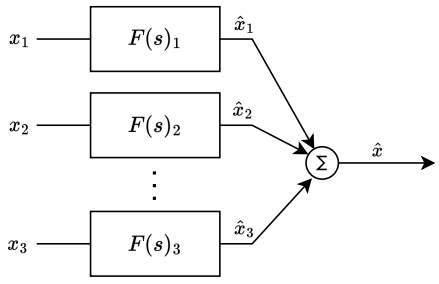


Figure 1.5: Complementary filter architecture: the filter results of a sum of multiple signals affected by other filters, inspired from [37].

The noise on the accelerometer and on the gyroscope are not alike. The accelerometer has a tendency to have high frequency noise while the gyroscope has low frequency noise. This is due to the fact that the accelerometer is strongly affected by vibrations. The gyroscope, on the other hand, works as an integrator which makes it vulnerable to constant value errors, or in other words, low frequency noise.

All the sensors indirectly measure the same quantity but are all affected by a different noise. In the case of an accelerometer and a gyroscope, the complementary does a great job of filtering these different types of noises due to its way of segmenting the frequency

domain. As can be seen in Figure 1.6, for two sensors, the complementary filter allows to filter the whole frequency domain while still keeping every frequency of the signal intact, by combining both sensors.

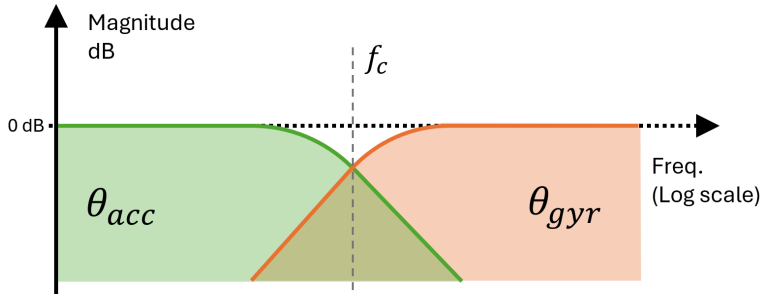


Figure 1.6: Complementary filter transfer function: the angle measured by the accelerometer is affected by a low-pass filter and the one measured by the gyroscope is affected by a high-pass filter, inspired from [37].

1.7.2 Kalman filter

The Kalman filter [35, p. 41] is a highly intelligent filter linked to a mathematical model that predicts the evolution of the measured state. This enables filtering out sensor noise with great precision while remaining highly reactive. It is based on confidence levels between the predicted state and the various sensor observations. One of its great strengths is its ability to adapt, in real time, the confidence levels of its sensors according to the accuracy of their measurements. It is therefore a great tool for sensor fusion, which explains why it is part of the Hera framework [38].

The Kalman filter is executed in two steps at each iteration: the prediction and the update [39]. Each iteration of this filter evaluates a vector and a matrix [40]:

- $\mathbf{x}$ the vector that represents the state of the system.
- $\mathbf{P}$ the covariance matrix of the error on the state $\mathbf{x}$. More intuitively, $\mathbf{P}$ represents the expected inaccuracy of the computed state vector. It is then used to measure the confidence level on the estimation of the state $\mathbf{x}$.

The prediction phase predicts what the next state of the system will be based only on the physical model of the system and on the command given to the system.

$$\hat{\mathbf{x}}_k = \mathbf{F}_k \mathbf{x}_{k-1}$$

$$\hat{\mathbf{P}}_k = \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^T + \mathbf{Q}_k$$

The update phase takes both the measurement and the prediction into account to generate a better estimation of the state $\mathbf{x}$ and the confidence on the estimation $\mathbf{P}$.

$$\mathbf{K}_k = \hat{\mathbf{P}}_k \mathbf{H}_k^T . (\mathbf{H}_k \hat{\mathbf{P}}_k \mathbf{H}_k^T + \mathbf{R}_k)^{-1}$$

$$\mathbf{x}_k = \hat{\mathbf{x}}_k + \mathbf{K}_k . (\mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_k)$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) . \hat{\mathbf{P}}_k$$

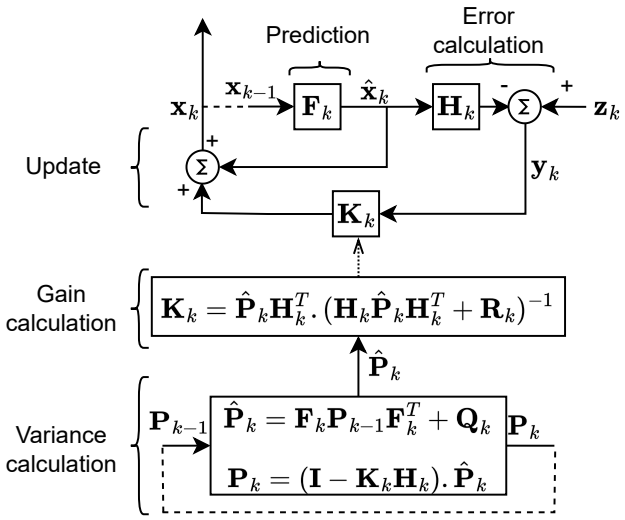


Figure 1.7: Bloc diagram of the Kalman filter showing the computation of the variances and the gain and their role in the update part of the filter.

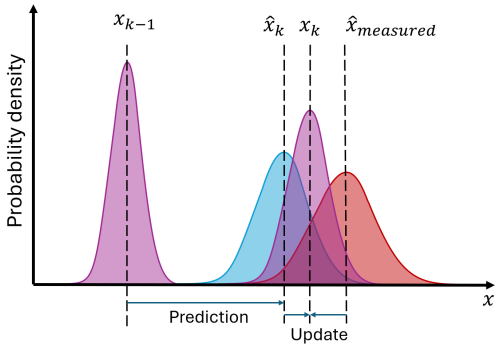


Figure 1.8: Gaussian combination in Kalman filtering. The prediction and the update phase as well as the result of the product of the Gaussians.

- **F**, State transition model: the matrix representing the evolution model of the system. It describes the evolution of the state based on a mathematical model of its natural behaviour.
- **Q**: the covariance matrix describing the evolution of the noise from the prediction. It represents the noise that should be added to **P** after the prediction due to the inaccuracies of the mathematical model.
- **R**: the covariance matrix describing the evolution of the noise from the sensors. It represents the noise or the inaccuracies of the measurements.
- **H**, Observation model: the matrix mapping the state of the system to the measurement of the sensors. It describes what the sensor should measure knowing the predicted state of the system.
- **z**: the vector representing the measurements of the sensors.
- **K**: the calculated matrix representing the Kalman gain. It's where magic happens. The Kalman gain takes into account the confidence on the sensor and confidence in the prediction to know how to combine them in order to lower the variance of P after the update.

A block diagram Figure 1.7 allows to better understand the links between the different parts of the Kalman filter.

One of the concepts that makes the kalman filter so powerful is the product of two Gaussian curves: the result is a new Gaussian curve with a lower variance, or in other words, less noise. This is a real asset for sensor fusion. An example is shown in Figure 1.8.

The extended Kalman filter is a variant of the Kalman filter. It allows the use of nonlinear equations for the predictive model and the observation model. The associated Jacobians must therefore be specified.

1.7.3 High-pass filter

Some devices, like the Pololu Balboa bot [28], use only the data from gyroscope integration to calculate its angle, passing it through a high-pass filter. This ensures that the average angle is zero. Otherwise, the robot would fall.

Such filter can be very easily implemented with a digital filter used, known as the infinite impulse response (IIR) filter. It's implementation is very simple:

$$\theta_n = k \times \left(\theta_{n-1} + \int_{t_{n-1}}^{t_n} \dot{\theta}_{gyro} dt \right) \tag{1.2}$$

To set the cut-off frequency f_c, the factor $k \in [0; 1]$ must be determined. Starting from the time constant equation for this filter [41]:

$$\begin{aligned} \tau &= \frac{k \cdot \Delta t_{loop}}{1 - k} \\ \Leftrightarrow k &= \frac{\tau}{\tau + \Delta t_{loop}} \qquad (1.3) \\ \Leftrightarrow k &= \frac{\frac{1}{2\pi \cdot f_c}}{\frac{1}{2 \cdot \pi f_c} + \frac{1}{f_{loop}}} \\ \Leftrightarrow k &= \frac{1}{1 + 2\pi \frac{f_c}{f_{loop}}} \qquad (1.4) \end{aligned}$$

This system is very simple but is very quickly limited in terms of performance. However, it also simplifies controllers because it ensures that the stabilisation angle is $0°$[3].

[3]Controllers that use an absolute angle need a part reserved for calculating the stabilisation angle. This is something more complex, but it also makes them more robust.

Part I

Design

Chapter 2

Overall design

The aim of this chapter is to present the direction followed during this master thesis by precisely defining the objectives to be achieved, as well as presenting the overall structure on which the system has been built.

2.1 Objectives specification

The purpose of this master thesis is to develop a library that allows dynamic stabilisation of real-life objects through the use of actuators. This section describes the type of application that was decided to explore, the objectives pursued and the constraints imposed on the implementation of such a system.

2.1.1 Application selection

As seen in the background chapter, Chapter 1, dynamic control of unstable systems is a very large domain with lots of different applications and control strategies. The choice of a two-wheeled self-balancing robot as an application for this master thesis is a good choice for multiple reasons:

- Generic: a two-wheeled robot is an application of inverted pendulum balancing. The dynamic of such a system is very similar to that of other unstable devices, making the stabilisation library versatile for other applications.
- Scalable: mechanically, a two-wheeled robot is a simple system that keeps the same dynamic behaviour when scaled up or down. This allows the stabilisation library to be directly usable in other applications with different sizes and weights by adjusting some parameters.
- Real-life application in IoT: Hera is a long-term project that aims to develop a new generation of robust IoT networks. In the IoT, such robots can have many different applications. Examples include payload transport in industry, assistance with various domestic tasks, remote monitoring, people transportation, etc.
- Mobile robotics: the two-wheeled robots are devices able to move freely in their environment. This opens a new research domain for GriSP, Hera, and the Erlang community.

- Ease of prototyping and use: the two-wheeled robot is a well-known concept that can easily be prototyped at low cost.

2.1.2 Objectives and constraints

The main objective of this master thesis is to develop a stability engine running on the GRiSP and that can easily be implemented on different devices. This very general and abstract task comes along with other more precise objectives:

- Create a device using the stability engine. The chosen type of device are inverted-pendulum like devices. More precisely, a two-wheeled self-balancing robot. Just like systems running under Erlang, the device should be able to recover from a crash without human intervention.
- Use Hera as an interface on the whole system for IoT purposes.
- Integrate the Kalman filter in the control algorithm. To efficiently use the Kalman filter, multiple sensors must be used on the device to make use of sensor fusion.
- Characterise the performance of the GRiSP board and Hera in a real-time application

The detailed technical specifications are provided in Appendix B

2.2 Overall system diagram

The first step, in order to correctly implement the system on the basis of the set objectives, is to create a draft of the overall system. It is built from several interconnected blocks, each with a specific purpose within the control loops, as seen in Figure 2.1.

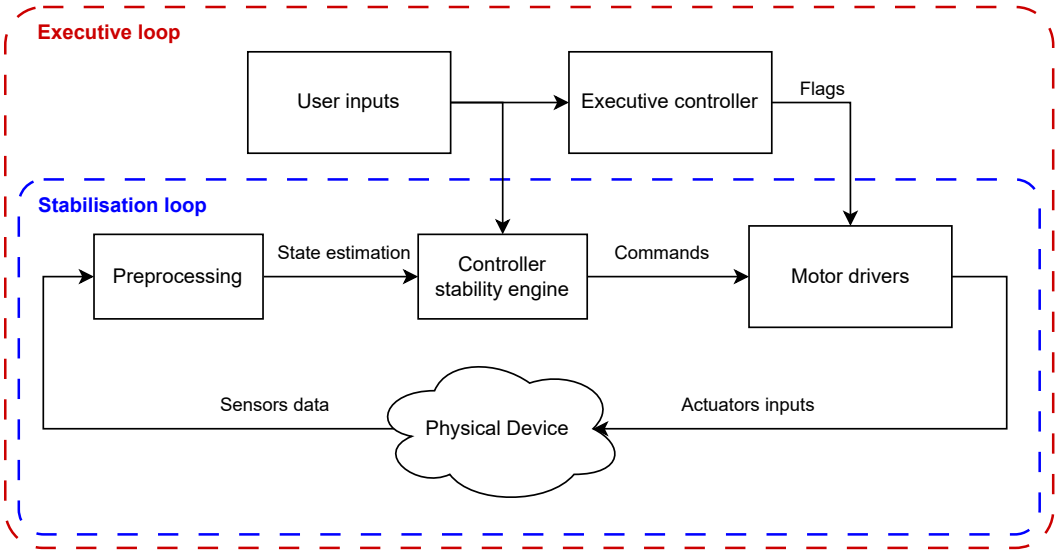


Figure 2.1: Simple system diagram: the stability control loop is nested within the high-level control loop. Numerous blocks, each with a specific function, are interconnected in order to make the system work as intended.

First of all, there are two loop levels, with a loop controlling the other [1] :

- **The stability loop**: this is a low-level loop whose function is to stabilise the robot when placed in its operating zone.
- **The executive loop**: this loop is used to manage all the events modifying the robot's state, like entering safety mode or lifting the robot up.

Each loop consists of several subsystems interacting with each other:

- **The physical system**: the body of the robot itself, with all its mechanical components and electronics having their own specific behaviour. Explanations on the design of the physical system are given in Chapter 3.
- **The preprocessing**: its purpose is to estimate the state of the physical device on the basis of noisy data from multiple sensors. Explanations on the design of the preprocessing are given in Chapter 4.
- **The controller**: its purpose is to compute the actuation commands needed to reach the desired state based on the estimated state from the preprocessing. Explanations on the design of the controller are given in Chapter 5.
- **The motor drivers**: their purpose is to translate the controllers' commands into relevant actuation signals. Explanations on the design of the physical system are given in Chapter 6.
- **The user inputs**: the set of control signals generated by the user, who may be a human via a remote control system or a higher-level system.
- **The high-level controller**: its purpose is to make high-level decisions to modify the system's mode of operation. Explanations on the design of the physical system are given in Chapter 7.

The complete and detailed diagram shown in Figure 2.2 is the result of the whole design process. The design and function of each of these blocks is explained hereunder in the following chapters.

The creation of such a system is an iterative process in which modifications and improvements are made step by step. These successive modifications impact both the internal workings of individual blocks and their interactions with other blocks.

[1] This architecture coincides perfectly with erlang's philosophy, where processes called supervisors monitor and restore other processes. In this system, the executive loop is like a supervisor and the stability engine is like a process.

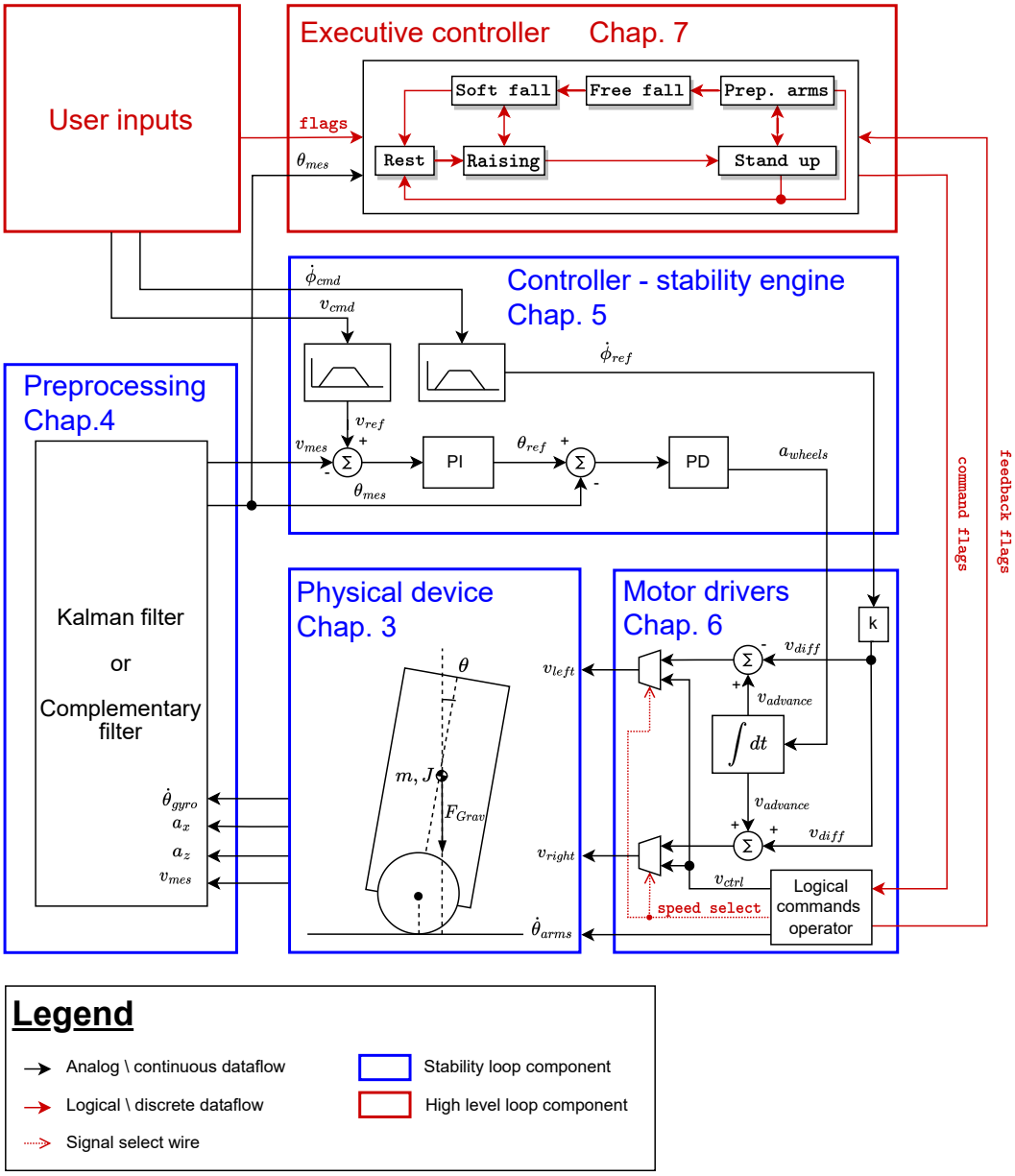


Figure 2.2: Detailed global system diagram, allowing to see the internal working of each block, their inputs and their outputs.

Chapter 3

Physical device

This chapter focuses on the design and open-loop behaviour of a two-wheeled self-balancing robot. The design part, Section 3.1, presents the different technological choices made from an electronic and mechanical point of view in order to make this system controllable. The physical modelling part, Section 3.2, explains the system's natural tendency to instability through mathematical developments.

3.1 Device design

The aim of this section is to describe the design process followed and the technological choices that were made in order to obtain a system that meets the objectives.

The whole system has been designed by following a set of technical specifications which are referenced in Appendix B. The most important points are :

- Balancing with a two-wheel drive train.
- Putting itself into self-balancing from a rest/post crash position.
- Embed the GRiSP 2 as the main controller.

These requirements lead to the need for three actuators. Two actuators will take care of the drive train. A differential configuration is the most adapted to be able to move backwards and forwards but also to rotate. The third actuator is used for getting up from a rest position.

In the following subsections, the electronic and mechanical design is presented. However, a common basis for these two sections concerning actuation technology choice must first be established.

3.1.1 Actuation

The robot must be equipped with several actuators to convert electrical signals into movements. In order to design the electrical and mechanical parts of the robot appropriately, it is essential to know which actuator technology will be used.

For reasons of cost and simplicity, it is preferable to use only one type of motor technology in the robot, and preferably the same model of actuators. Here's a list of

the different types of electric motors:

- **Induction and synchronous motor**: useful for high-power applications, but requires a highly advanced control and additional installation for speed and position control. Not suitable for our application
- **DC motor**: motor corresponding to the expected dimensions of the robot, but coupled to a gearbox that brings backlash, which could be very annoying for our application, where the motors will often change direction and therefore pass through the backlash zone. In addition, an encoder system must be installed to provide feedback control on their speed or position.
- **Servo motor**: some types of servos, such as dynamixel, are not limited in their travel, are easy to control, but are often limited by their speed, are rather noisy and also suffer from backlash. In addition, servo motors adapted to our use would have a significant cost.
- **BLDC motor**: brushless DC motors have the advantage of being very quiet and can be mounted without a gearbox, so there's no backlash, but they require highly advanced control strategies such as vector control, and very specific drivers that can be expensive. This type of motor constantly consumes energy, even at a standstill.
- **Stepper motor**: Stepper motors are fairly compact and can be easily controlled in open-loop, but require drivers. Both motors and drivers are easy to find on the market. As with BLDCs, stepper technology consumes energy at rest. Moreover, they are easy to upgrade, as most of them use the same mounting system.

Given the constraints of the project, the stepper motor is the most suitable technology for the robot. It is easy to control, has a good torque/speed ratio for this application, is easy to install and is relatively inexpensive.

The NEMA 17 serie stepper motor is a great candidate for the project. It is a widely used motor in the 3D printing industry, where the speed and torque requirements are similar to those of the robot.

3.1.2 Electrical design

The electrical design was carried out in two steps:

1. The logic part: all the components and their means of communication are selected to meet expectations.
2. The power-supply part: takes care of supplying the various components with the right voltage and the necessary current flow.

Logical circuit design

The starting point of the electrical logic design is the GRiSP 2 board, as it should remain the central element of the robot. Another thing already known is that the robot will use stepper motors. At a number of three (two for the wheels and one for the lifting system), those motors are each controlled by a stepper driver.

Based on a GPIO frequency benchmark test of the GRiSP 2 board, it was clear that the board could not supply a high-frequency signal for the stepper driver control through its GPIOs with the actual firmware. The solution was to use a microcontroller between the GRiSP 2 and the stepper drivers.

The choice of the sensor is straightforward, as the GRiSP supports the Pmod NAV sensor, which provides accelerometers and gyroscope measurements.

In case of emergency, remote control via the emergency stop button is used to stop the robot. It's the most suitable option, as the robot may flip over and the button may be difficult to reach if installed directly on the robot. For this purpose, a LoRa communication technology was chosen for its range and reliability.

Components description

Here is a brief description of the components depicted depicted in Figure 3.1:

- **GRiSP 2** is the main controller of the system. Its purpose is to compute the high-level controls as well as the low-level commands sent to the Lilygo LoRa32 through I^2C.
- **TMC2208 stepper motor driver** makes almost no noise compared to cheaper drivers. The driver has three logical input pins:
  - En (enable): when this pin is set to high, the motor is enabled. The motor can no longer spin freely.
  - Step: at each rising egde, the motor takes one step.
  - Dir (direction): this pin allows the motor to change the spinning direction.

  The four output pins going to the motor are M1A, M1B, M2A and M2B and correspond to the motor inductances. This driver allows micro-stepping, which provides more precise position control. Finally, the torque of the motor can be modified with a potentiometer adjusting the current going through the stepper's inductances.
- **Lilygo LoRa32** is an ESP32-based circuit board. It embeds multiple useful features, such as I^2C communication and an SPI LoRa emitter-receiver. Its purpose is to directly interact with the steppers, receive LoRa commands and behave as an I^2C slave with the GRiSP.
- **Pmod NAV**. This Digilent Pmod communicates through SPI and has multiple sensors, such as an accelerometer and a gyroscope.
- **Stepper driver interface PCB**[1] is especially designed for this master thesis in order to easily connect three TMC2208 stepper drivers to a GRiSP 2 or a Lilygo LoRa32. It features:
  - Eight logical inputs: one step and one direction signal for each motor, one enable signal for the driver in the middle and one for the remaining drivers together.

[1] Printed circuit board (PCB)

  - Logical supply voltage, ranging from 3V to 5V.
  - Motor power supply input voltage that can go up to 36V.

  The enable ports of the first and last stepper drivers have been connected together as they will be connected to the motors used for the differential wheel system and are therefore enabled at the same time. This reduces the number of ports by one, allowing it to be connected entirely to a double-row GPIO Pmod for eventual future use.

Robot power supply design

Within this circuit, two supply networks are required:

- A 5V supply for powering the logic elements.
- A supply of 12V or less is needed to power the motors.

By using a 12V LiFePO4 battery, which is the safest type of lithium-based battery, these two networks can be supplied using buck converter regulators. Boost converters should be avoided as they can induce noise on bus communications.

Emergency switch

As the robot is naturally unstable, it was decided to install a remote emergency stop button instead of putting it directly on the robot. This emergency stop button does not meet industry standards, but for low power levels, such a system is not mandatory.

This remote switch is made out of an emergency button and a Lilygo LoRa32. The switch can communicate with the user via universal serial bus (USB) to retrieve user inputs. It can communicate these inputs via LoRa to the robot.

Overall electrical circuit

Figure 3.1 shows all the connections between the selected components. The upper part of the figure shows the robot with all its components while the lower part shows the remote emergency button.

3.1.3 Mechanical design - two wheeled robot with lifting mechanism

The mechanical design is divided into several parts, which must fit together:

- The drive train: the entire wheel drive system in differential configuration.
- The lifting system: its purpose is to raise the robot up in order for it to get into dynamic balancing.
- Electronic components.
- The chassis: the main structure that holds everything together.

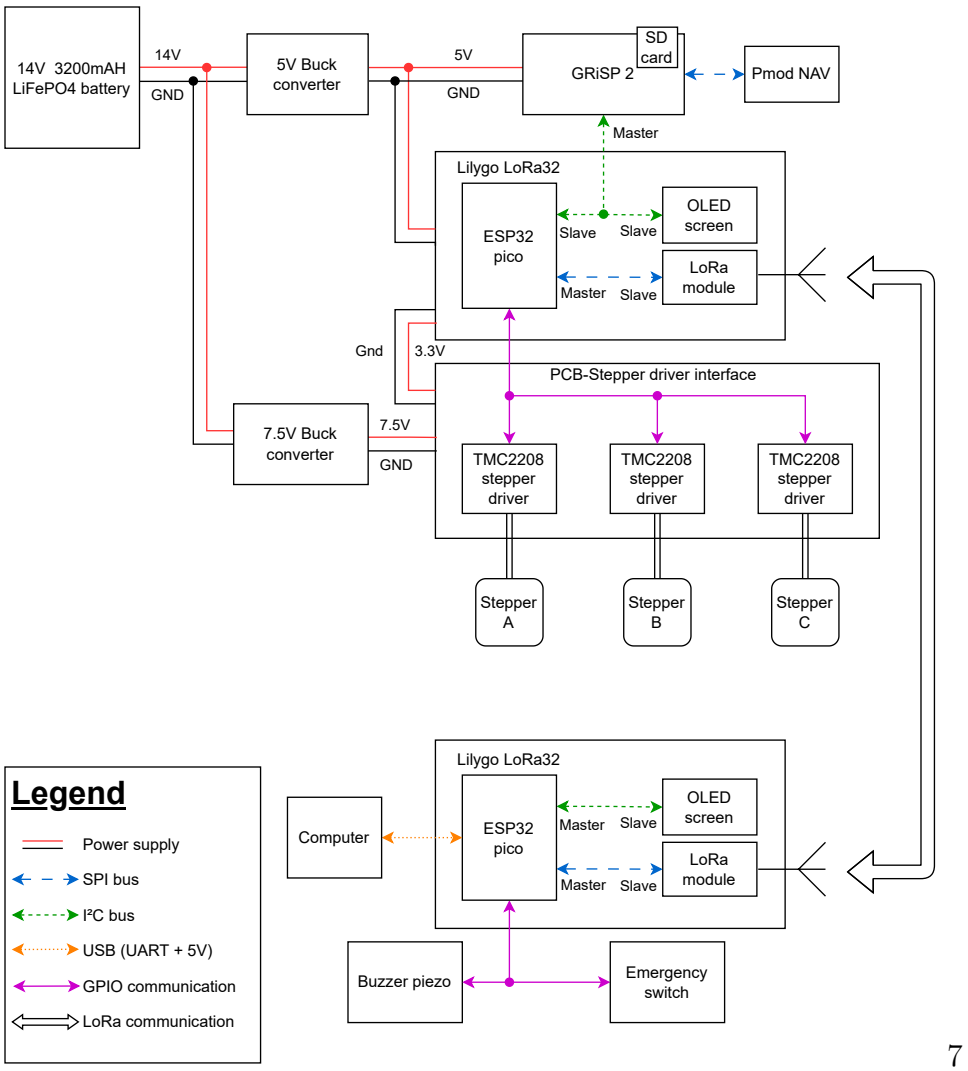


Figure 3.1: Electrical diagram, the power lines and the communication between the components are represented.

Drive train

The drive train is composed of two stepper motors sharing the same virtual axis each having one wheel on their shaft. The wheels are 70mm in diameter and made out of steel with a rubber tire to provide a good grip on the ground. Those wheels are locked on the stepper shaft with the use of a pressure screw.

The two stepper motors are mounted on their own right-angle bracket to be attached to the chassis.

Lifting mechanism

The lifting system is engineered with two aspects in mind: it must be compact and be able to raise the robot up to 80° with respect to the ground.

The designed mechanism is made up of three moving parts: a straight bevel pinion connected to the motor shaft and two curved racks, which are segments of a bevel gear, as shown in Figure 3.3. The 90° bevel gear allows the motor to be positioned below

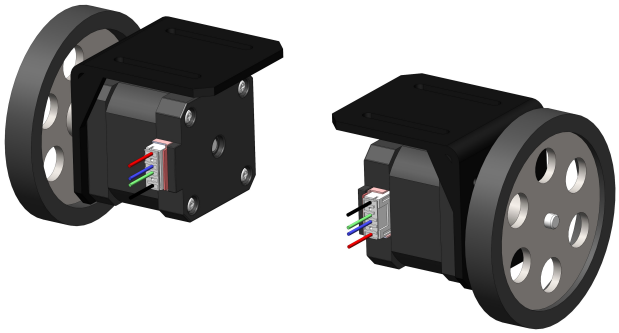
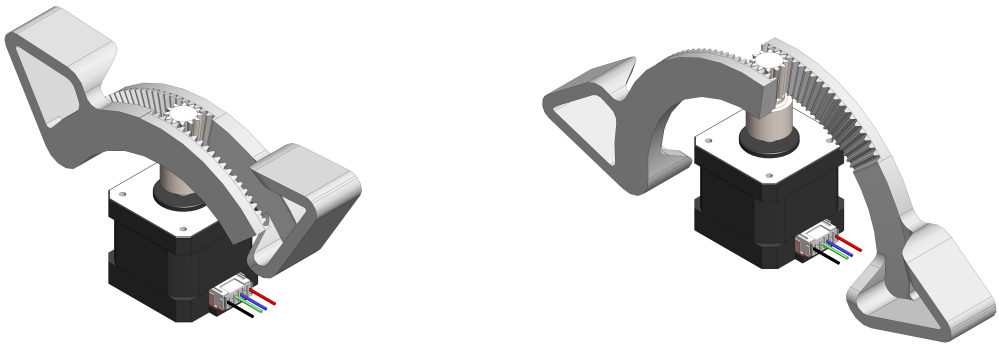


Figure 3.2: Drive train with two stepper motors, two wheels and two brackets.

the mechanism, leaving space above and on the right and left sides.

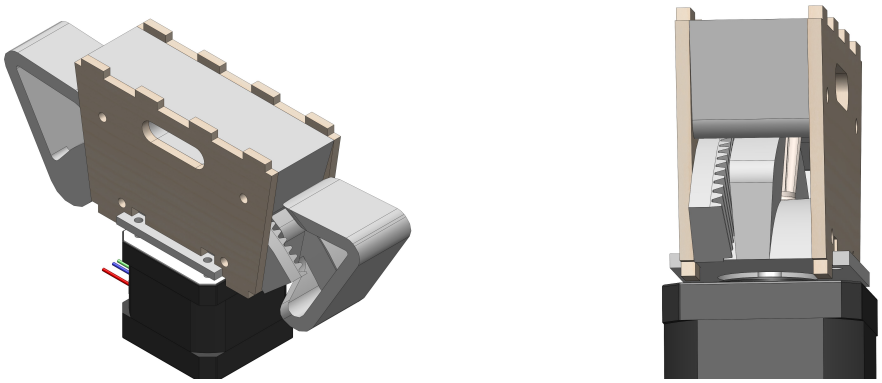


(a) Retracted lifting mechanism.

(b) Extended lifting mechanism.

Figure 3.3: Lifting mechanism with bevel gear, both arms extend in opposite directions to raise the robot up.

In order to maintain good gear contact, slots have been designed into the casing to guide the sliding of the lifting arms, as shown in Figure 3.4.



(a) Casing around the lifting mechanism.

(b) Lifting mechanism with one arm removed allowing to see the slots used to guide the arms.

Figure 3.4: Lifting mechanism with its casing.

One of the main design constraints was to keep the contact area permanently tangent

to the wheel circumference. This implies having the rotation axis of the arms collinear with the axis of rotation of the wheels. For simplicity, the two arms simultaneously extend and retract, as shown in Figure 3.5.

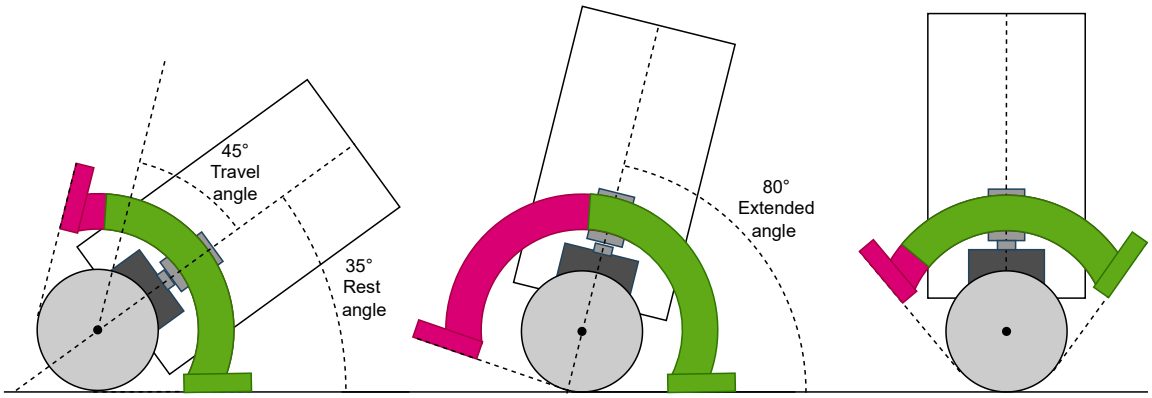


Figure 3.5: Lifting process: on the left, the robot is down with the arms retracted. In the middle, the arms are extended and the robot is still resting on them. On the right, the arms are retracted and the robot is dynamically balancing.

At rest with its arms retracted, the robot is at an angle of 35° with respect to the ground. The arms have a travel angle of 45° . When fully extended, the arms hold the robot at an angle of 80° . Figure 3.5 shows the different steps of the lifting process.

All of those pieces have a very specific geometry. Plastic Fused Deposition Modeling (FDM) 3D printing is a great prototyping option to produce these pieces due to its simplicity, speed and affordability.

This highly innovative system, which keeps the lifting arms tangent to the ground, comes with a problem: the robot can slip during lifting. If the wheel motors are blocked during lifting, the wheels will rotate with the body, causing it the body to move. To avoid this slipping effect, a counter-rotation must be applied to the wheels: the angular speed of the wheels must be equal and opposite to the angular rate of the lift.

Electronic components

Electronic components can take up a lot of space and need to be distributed appropriately. It is important to keep the power and logic parts of the system separate to avoid induced voltages in system communications, which can cause errors in communication signals.

Chassis

The chassis holds all the components together and must be robust enough to withstand the various shocks. It is during the design of the chassis that the distribution of mass is considered. In the context of a self-balancing robot, this aspect is very important. The robot's center of gravity (CG) must be high enough to have slower time constants, and the weight distribution between the left and right wheels must be similar enough to have similar grip and wear on the two wheels.

The robot is designed with five different levels, as shown in Figure 3.6 to be able to fit all the mechanical and electronic components mentioned above.

Each level has its own components and use:

- Level 0: the drive train, fixed to the chassis by the right angle bracket, and the lifting system actuator.
- Level 1: made up of three parts, the voltage converters, the lifting mechanism and the motor controllers.
- Level 2: space reserved for cable management.
- Level 3: used to attach two GRiSP 2 boards, their Pmods and the battery with a 3D-printed support.
- Level 4: platform for additional payload.

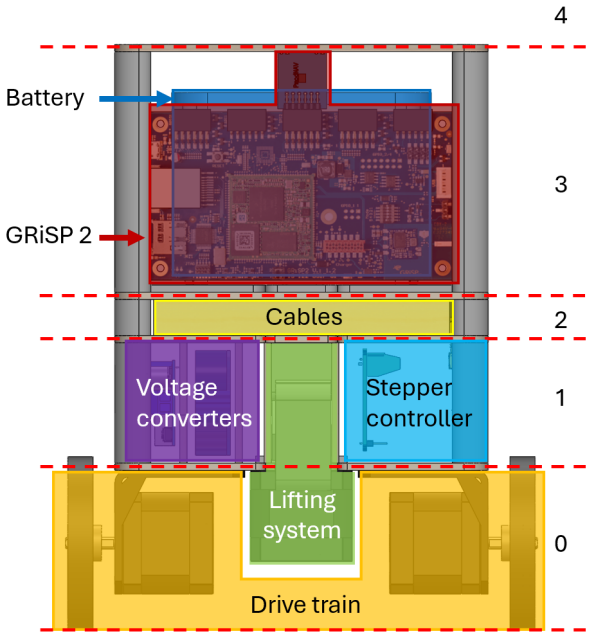


Figure 3.6: Chassis layout in five different levels.

Laser-cut wooden boards are used to separate the floors. To secure the various components, vertical wooden panels are placed between the horizontal boards using a mortise and tenon system. This system enables rapid assembly, but only holds together if put into compression. To secure the assembly, vertical screws operating in tension apply compression to the mortise and tenon joints, holding all the assembly together.

3.2 Physical modelling

To understand the behaviour of the system, to know how to properly control and enhance the Kalman filter, it's important to develop a mathematical representation of the system. The purpose of this model is to describe the relationship between the input and output of the physical system based on a set of hypotheses.

3.2.1 Hypotheses

The following hypotheses are assumed for the development of the physical model:

- The robot is a rigid body.
- The mass of the wheels is negligible.
- The slip of the wheels is negligible.
- The effect of the torque of the wheels on the robot is negligible.
- The movement, speed and acceleration of the robot due to a change of the chassis angle is negligible.
- There are only two external forces applied to the robot: the weight and the reaction force of the ground, including the friction force.

3.2.2 Frame representation

All the quantities used in the development of the model are defined in Figure 3.7.

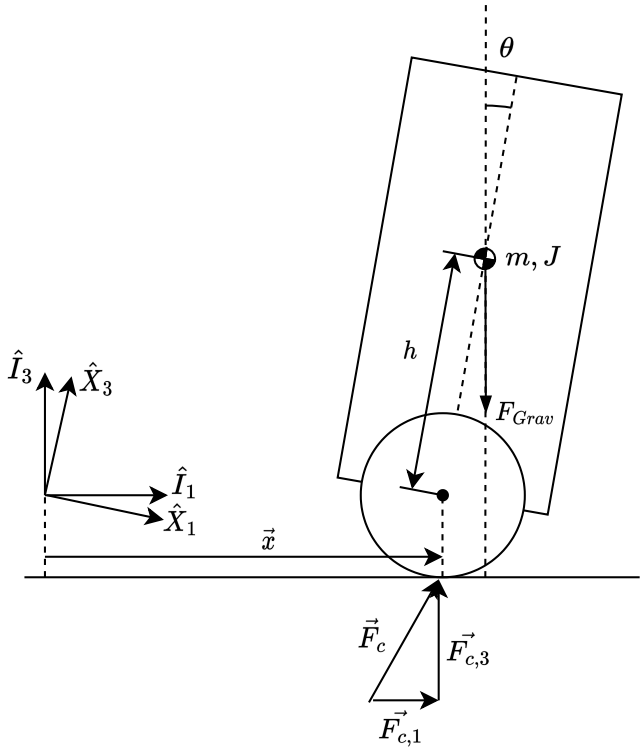


Figure 3.7: Reference scheme of the bodies and the different quantities for developing the physical model of the system.

This model uses two different reference frames:

- $\hat{I}$ is attached to the robot and is defined by $\hat{X}_3$ and by $\hat{X}_1$. $\hat{X}_3$ is aligned to the axis passing through the CG and wheel center. $\hat{X}_1$ is perpendicular to the previous axis and pointing to the front of the robot.

- The reference frame of the ground is defined by $\hat{I}_1$ and $\hat{I}_3$. $\hat{I}_1$ is parallel to the ground. $\hat{I}_3$ is perpendicular to the ground and pointing upward.

All the variables are shown in Table 3.1.

θ	The angle of the robot relative to the vertical axis in $[rad]$
$\dot{\theta}$	The angular speed of the robot in $[rad/s]$
$\ddot{\theta}$	The angular acceleration of the robot in $[rad/s^2]$
x	The position of the robot in $[m]$
$a_x = \ddot{x}$	the acceleration of the robot in the x-axis in $[m/s^2]$
$\vec{F}$	The contact force of the ground on the robot in $[N]$

Table 3.1: List of variables used in the model.

Finally, some constants are used in the model. These are described in Table 3.2.

h	The distance between the CG and the axis of rotation in $[m]$
m	The mass of the robot in $[kg]$
J	The moment of inertia of the robot in $[kg.m^2]$
g	The gravitational acceleration in $[m/s^2]$

Table 3.2: List of constants used in the model.

3.2.3 Movement equations

The full development using Newton-Euler equations is provided in Appendix D. It results in the following formula:

$$(h + \frac{J}{mh})\ddot{\theta} = g\sin(\theta) - \ddot{x}\cos(\theta) \tag{3.1}$$

According to the previously stated hypotheses, the effect of the motors is manifested through $\ddot{x}$. Equation (3.1) shows that the lateral acceleration induces an angular acceleration $\ddot{\theta}$.

Chapter 4

Preprocessing

The preprocessing unit is used to clean and interpret the data collected by the sensors. In this application, the preprocessing uses the inertial measurement unit (IMU) sensors' data, which are the accelerations from the accelerometer and the angular rate from the gyroscope, to output an estimation of the pitch angle (θ). In most cases, the raw data obtained from the sensors is not directly usable for two main reasons:

- The data obtained from sensors is affected by interference and noise, which can be more or less significant depending on the type of sensor and its surrounding environment. For this reason, the data is subjected to a filtration process.
- The data measured by the sensor, for example, accelerations, can only be indirectly linked to a quantity describing the system, for example, the angle. An intermediate transformation is thus required in order to contribute, either partially or entirely, to the calculation of the quantity of interest.

The following sections will present two distinct preprocessing strategies:

- The complementary filter in Section 4.1.
- The Kalman filter in Section 4.2.

Both strategies support sensor fusion, a process which combines data from multiple sensors to enhance the measurements.

4.1 Complementary filter

As explained in Section 1.7, the complementary filter is a combination of several measurements affected by specific cutoff frequency filters in order to be efficient on every frequencies.

In this application, the filter is the result of averaging two terms with the use of two weights k and $1 - k$:

$$\theta_{cf} = k \cdot \theta_{gyr} + (1 - k) \cdot \theta_{acc} \tag{4.1}$$

- θ_{acc} is the angle measured by the accelerometer based on the gravity vector. In this application, it is computed with a_x and a_z, the x and z-axis accelerations measured by the sensor:
$$\theta_{acc} = \arctan\left(\frac{a_z}{-a_x}\right)$$

- θ_{gyr} is computed from the previous angle, $\theta_{cf,n-1}$, incremented by the integration of the angular velocity, $\dot{\theta}_{gyr}$ and measured by the gyroscope on the sampling time:
$$\theta_{gyr} = \theta_{cf,n-1} + \int_0^{\Delta t} \dot{\theta}_{gyr} dt$$

There is therefore a low-pass effect on the angle measured by the accelerometer, which gets rid of the significant noise induced by the robot movements. On the other hand, there is a high-pass filter effect on the gyroscope measurement, which avoids the DC error that accumulates during integration. In this case, the cut-off frequency was selected to be 0.127 Hz, i.e. a $rate = 1.25$rad/s. This enables the low-pass part to average the accelerometric measurements over a fairly long period, and keeps the high-pass part very dynamic. To compute k the equation (1.4) applies

The complementary filter is widely used in IMU measurements. It is directly provided by Digilent on the Pmod NAV product page [5]. It is also used in many do it yourself (DIY) self-balancing applications, such as [42]. But in both cases the frequency is fixed and therefore it doesn't automatically compute the k factor from the loop frequency of the system.

4.2 Kalman filter

The Kalman filter, as explained in Section 1.7.2, is a very powerful tool that enables adaptive sensor fusion. In other words, it is capable of modifying its interest in one sensor or another by continuously comparing them with a physical model. In this context, it is particularly useful, as data from several Pmod NAV sensors must be interpreted in the best possible way for the controller to work at its best.

4.2.1 Simple model

The Kalman filter makes predictions based on a mathematical model. This model can be very basic, allowing few calculations, or very complex, which can affect performances but makes better estimates.

In this very simple model, most of the physics of the robot is ignored. The remaining equations are the integration of angular speed to find the angle (4.2) as well as Newton's first law applied to rotating objects without external forces (4.3):

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

$$\dot{\theta}_k \approx \dot{\theta}_{k-1} \tag{4.3}$$

The state vector and the state transition matrix are defined as:

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} \quad ; \quad F = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{4.4}$$

The measure vector, which is in fact identical to the one used for the complementary filter. The observation matrix is simple as x and z are similar:

$$z = \begin{bmatrix} \theta_{acc} \\ \dot{\theta}_{gyr} \end{bmatrix} \quad ; \quad H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.5}$$

The covariance matrix of the additional noise for measurement and for prediction is based on the sensor datasheet and from rules of thumb and empirically tuned.

$$Q = \begin{bmatrix} 3.0\text{e}{-5} & 0 \\ 0 & 1.0\text{e}1 \end{bmatrix} \quad ; \quad R = \begin{bmatrix} 3.0\text{e}0 & 0 \\ 0 & 3.0\text{e}{-6} \end{bmatrix} \tag{4.6}$$

The order of magnitude of the values of the matrix Q, describing the evolution of variance on the prediction model, shows that there is a lot of confidence in the integration part, thanks to the low variance defined by $Q[1,1]$, and little confidence in the part linked to Newton's first law, due to the high variance of $Q[2,2]$.

Regarding the matrix R, describing the evolution of the variance of the measurements, $R[1,1]$ shows that there is little confidence in the angle values coming from accelerometer measurements, which is expected as it is a very noisy measure. On the other hand, the order of magnitude of $R[2,2]$ shows that there is greater confidence in gyroscope data, which is known to be quite precise [5].

4.2.2 Advanced "digital twin" model

The model described below is much more precise, taking more phenomena into account than the simple model. The predictive model is directly derived from the equations describing robot behaviour detailed in Appendix D.

A digital twin is a model of the physics of a system running in real time. It is used in many advanced control systems to map all the state parameters of the system. The development of a partial digital twin as a model for the Kalman filter is described below. It is called a partial as it does not map the entire robot behaviour, but a part of it. It focuses on the θ related state variable and the wheel acceleration command input u. As the equations are not linear, the extended Kalman filter must be used.

The state vector remains the same as the one from the simple model:

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} \tag{4.7}$$

The state transition model and its Jacobian is obtained from the physical model equation discretisation:

$$f = \begin{bmatrix} \theta_k + \dot{\theta}_k \Delta t \\ \dot{\theta}_k + \left(\frac{g}{h + J/mh} \sin(\theta) - \frac{u}{h + J/mh} \cos(\theta) \right) \Delta t \end{bmatrix} \tag{4.8}$$

$$J_f = \left.\frac{\partial f}{\partial x}\right|_{x,u} = \begin{bmatrix} 1 & \Delta t \\ \left(\frac{g}{h+{}^{J}/_{mh}}\cos(\theta) + \frac{u}{h+{}^{J}/_{mh}}\sin(\theta)\right)\Delta t & 1 \end{bmatrix} \tag{4.9}$$

Regarding the measure vector, it remains the same:

$$z = \begin{bmatrix} \theta_{acc} \\ \dot{\theta}_{gyr} \end{bmatrix} \tag{4.10}$$

The observation model also remains the same as x maps to z in the same way as in the simple model:

$$h = \hat{x} \quad ; \quad J_h = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{4.11}$$

In the previous models, a computed angle from the accelerometer data, a_x and a_z, is used as an input for the Kalman filter. To enhance this advanced model, the acceleration measurements are directly provided as sensor input. This captures a wider range of physical phenomena than just gravity, as the system is constantly moving.

The observation model should be rebuilt to map the accelerometer inputs. The acceleration of any point on the robot's vertical axis, $\hat{X}_3$, is a combination of four different phenomena, still referring to Figure 3.7:

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

The total acceleration perceived by the sensor axis is the sum of the preceding effects:

$$\hat{a}_x = \underbrace{u\cos(\theta)}_{\text{lateral acc.}} + \underbrace{\left(\frac{g}{h+{}^{J}/_{mh}}\sin(\theta) - \frac{u}{h+{}^{J}/_{mh}}\cos(\theta)\right)r}_{\text{angular acc.}} + \underbrace{g\sin(\theta)}_{\text{gravity acc.}} \tag{4.12}$$

$$\hat{a}_z = \underbrace{u\sin(\theta)}_{\text{lateral acc.}} + \underbrace{-\dot{\theta}^2 r}_{\text{centripetal acc.}} - \underbrace{g\cos(\theta)}_{\text{gravity acc.}} \tag{4.13}$$

The observation model can be written as:

$$h = \begin{bmatrix} \hat{a}_x \\ \hat{a}_z \\ \hat{\omega}_y \end{bmatrix} = \begin{bmatrix} u\cos(\theta) + \left(\frac{g}{h+{}^{J}/_{mh}}\sin(\theta) - \frac{u}{h+{}^{J}/_{mh}}\cos(\theta)\right)r + g\sin(\theta) \\ u\sin(\theta) - \dot{\theta}^2 r - g\cos(\theta) \\ \dot{\theta} \end{bmatrix} \tag{4.14}$$

The Jacobian of the measurement model is:

$$
\begin{aligned}
J_h &= \left.\frac{\partial h}{\partial x}\right|_{x,u} \\
&= \begin{bmatrix} -u\sin(\theta) + \left(\frac{g}{h+{}^J/_{mh}}\cos(\theta) + \frac{u}{h+{}^J/_{mh}}\sin(\theta)\right) r + g\cos(\theta) & 0 \\ u\cos(\theta) + g\sin(\theta) & -2\dot{\theta}r \\ 0 & 1 \end{bmatrix}
\end{aligned} \tag{4.15}
$$

Since the simple model surpasses all expectations, see Part III, it was not felt necessary to integrate this more advanced model in order to spend more time optimizing other blocks. However, a detailed discussion on the choice of covariance evolution matrices is given below.

The matrix Q, describing the evolution of the variance of prediction, is expected to be much smaller than the one of the simple model. The same would apply to the matrix R, especially for variances related to the accelerometer data. However, even if the model predicts the robot's behaviour perfectly, the matrix Q must still retain a certain level of variance in order to adapt to unpredictable variations. In other words, the prediction model is a deterministic system that cannot predict the arrival of disturbances occurring in the real world, which means that a level of uncertainty remains.

4.2.3 Overall system with Kalman filter

The schematic representation of the overall system using the Kalman filter for its preprocessing is shown in Figure 4.1

4.3 Preprocessing filters comparison

Kalman and complementary filters are very different but are used in the same way. Below is a comparison of the two filters to understand their differences. :

- Complexity: the mathematical computations required to deploy such filters and the difficulty of implementation are very different. The complementary filter is very easy to instantiate, and few parameters are required to make it work properly. The Kalaman filter, on the other hand, requires a special physical model to be designed and multiple covariance matrices to be chosen. Furthermore, implementation requires matrix computation.
- Noise processing: these two filters deal with noise in different ways. On the one hand, Kalman is very flexible and is able to favor one sensor or the other on the basis of its prediction. On the other hand, the complementary filter does not adapt its way of working according to observations. In the particular case of this application, the Kalman filter behaves in a way like a complementary filter, but is able to adapt its cut-off frequency dynamically.

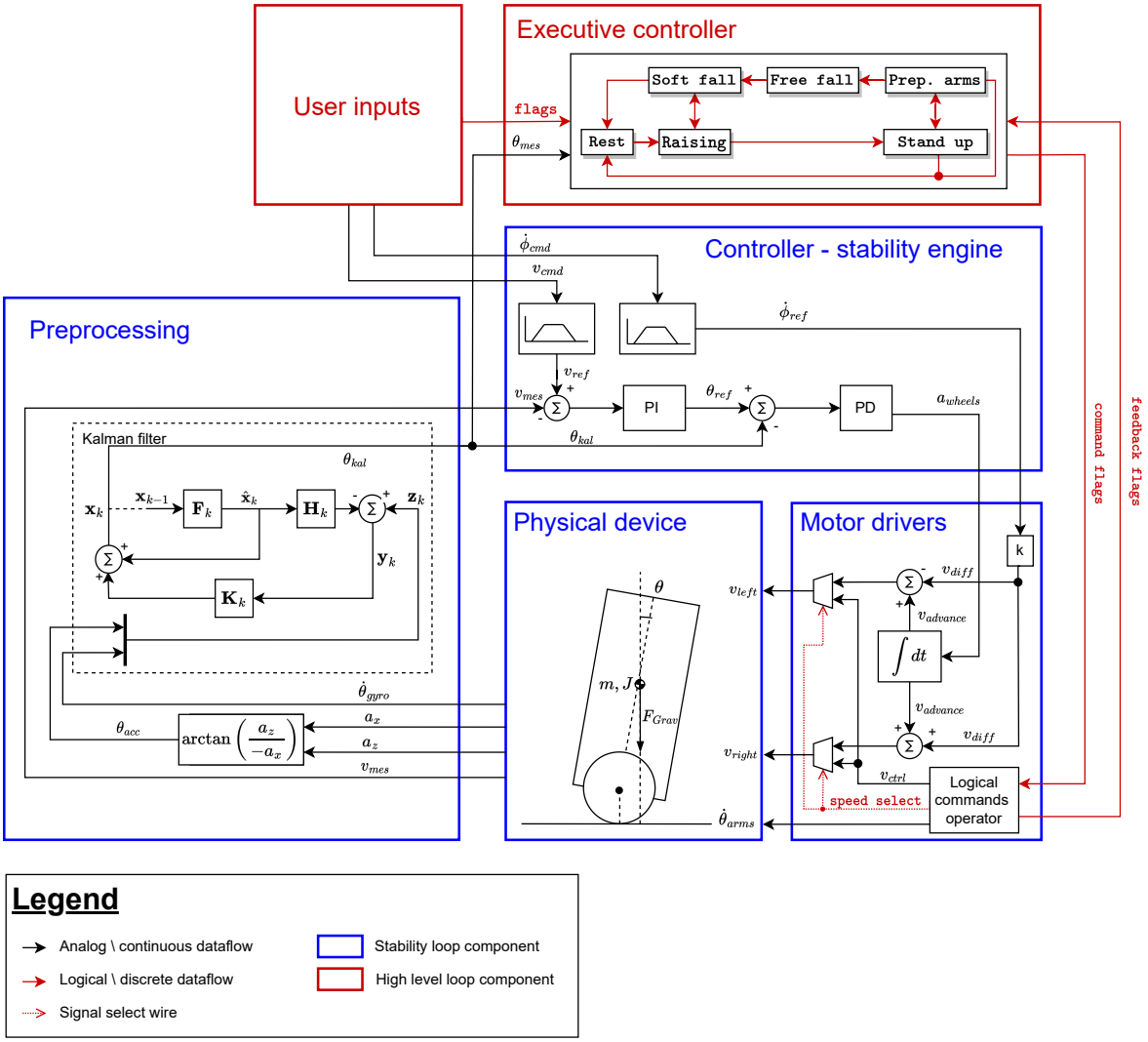


Figure 4.1: Overall system representation using a Kalman filter. It shows the computation of an absolute angle from the accelerometer data outside the Kalman filter, as it is used as one of its inputs. The wheel speed measure is not filtered at all.

Chapter 5

Controller

The purpose of the controller is to modify the state of a system in order to align it with a reference input. The aim of the controller in this case is to stabilise a system that is by its nature unstable. When the controller uses a measurement of the system to generate its output, it is said to be a feedback controller or closed-loop controller. Controllers that do not use an observation on the system to make a control are called feed-forward controllers or open-loop controllers.

5.1 PID controller

PID controllers are widespread feedback control systems. In this application, such controllers are used as fundamental building blocks for system control, as described in Section 5.2.

These controllers generate their output command on the basis of an error, the difference between a reference value and a measurement of the controlled variable.

$$e = x_{ref} - x_{mes}$$

The aim of such a controller is to make this error tend towards 0, thanks to the contributions of three terms contributing to the output u:

- **Proportional**: the contribution is proportional by a factor K_p to the error.
- **Integral**: the contribution proportional by a factor K_i to the integral error.
- **Derivative**: the contribution is proportional by a factor K_d to the derivative of the error.

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

It is usual to add a limiter on the integral term, also known as anti-windup, in order to avoid too much overshoot when big set point variations occurs and when the command output saturates.

The transfer function of such a controller is expressed as:

$$\frac{U}{E} = K_p + \frac{K_i}{s} + K_p \cdot s$$

This means that with such a controller, the pole[1] of a system can be moved around in a smart way through the choice of the K_p, K_i and K_d terms.

5.2 Stability engine conception

Designing a control system is not a trivial task. Especially when the system being controlled is not a stable bounded-input bounded-output (BIBO) system, which is the case with the self-balancing robot. In order to design the stability engine, it was essential to correctly define the inputs and outputs of the control system and to identify the various phenomena involved.

5.2.1 Controller for any reference angle

The designed controller allows the robot to reach any given reference angle by producing accelerations at the wheels' motors. A simple PD controller is sufficient for this. This controller is represented in Figure 5.1.

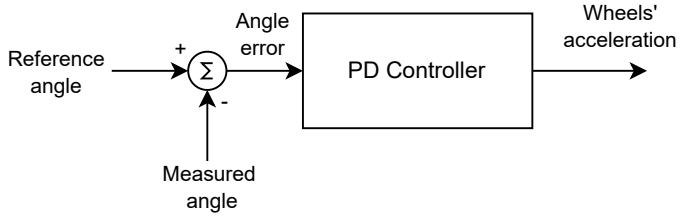


Figure 5.1: PD control system with the error computed from a reference angle set by the user and the measured angle, and return a command for the wheels' accelerations.

5.2.2 Controller with equilibrium point reference

This first control system has a major problem: if the angle imposed is not precisely the angle of equilibrium, a small force will appear. This would result in the need of a constant acceleration. This is not suitable for the hardware, as the motors would saturate in speed quite quickly. Furthermore, the robot would be in constant motion, which is very annoying for this type of application.

The solution is therefore to impose the reference angle as the angle of equilibrium. This evolution of the controller is shown in Figure 5.2.

5.2.3 Finding the equilibrium point

The angle of equilibrium is not straightforward to find. It is defined as the pitch angle at which all the forces acting on the robot cancel each other out. There are several ways to find it:

[1] In control theory, the poles of a state space representation is used to show the stable or unstable behaviour of a system and the way it oscillates.

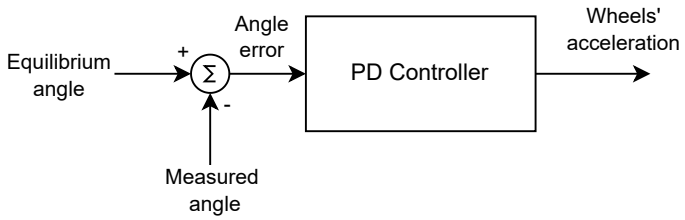


Figure 5.2: PD control system with the error computed from the equilibrium angle and the measured angle and return a command for the wheels' acceleration.

- **Calculating the balance point**: based on the robot's computer-aided design (CAD), the robot's theoretical balance angle can be calculated. The problem is that this measurement is not exact, and there will be a small variance with the actual value. This will inevitably imply constant acceleration.
- **Measuring the equilibrium point**: this can be done by turning the robot upside down, and it will naturally stabilize at its stable equilibrium point. By adding 180° to this measurement, the unstable equilibrium point is obtained. The measurement can be made directly by the robot's sensor. However, certain problems can still be identified:
  - Components could move when the robot is turned upside down, falsifying the measurement.
  - The equilibrium point is set as a constant, but in reality it may vary over the robot's use. If the robot is used to transport goods, its center of mass will vary depending on the payload, and so will its equilibrium point. It could also vary depending on the external environment. For example, if the robot is positioned on a slope, its point of contact on the wheel changes. As the ground's reaction force is applied to it, the point of equilibrium is also modified.

  This method is therefore not suitable for practical use either.
- **Continuously updating the equilibrium point**: the control system detects by itself when the robot moves away from its equilibrium zone, and evolves its equilibrium point accordingly. This strategy, when implemented correctly, is the most robust, as it allows the robot's behaviour to be adapted to multiple disturbances affecting its equilibrium point.

The most advantageous strategy is the third one. It is represented in Figure 5.3.

5.2.4 Equilibrium angle controller

A specific controller can be dedicated to dynamically compute the angle of balance. Its output is therefore the angle of equilibrium, although the choice of input is more complex. For this application, the wheels' speed as inputs has been adopted:

- Speed is a good indicator of the robot's equilibrium: when the robot starts moving at high speed in one direction, it is highly likely that the angle of equilibrium is on the other side of the robot.

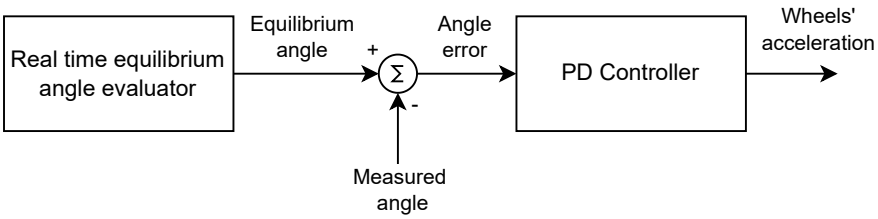


Figure 5.3: PD control with the error computed from equilibrium angle and the measured angle and return a command for the wheels' acceleration. The equilibrium angle is continuously updated.

- Speed control is perfect for making the robot move with the desired speed, as the user could wish. It also prevents the robot from going beyond the speed limits of what the hardware is able to deliver.

To do it a PI controller is used as shown in Figure 5.4 .

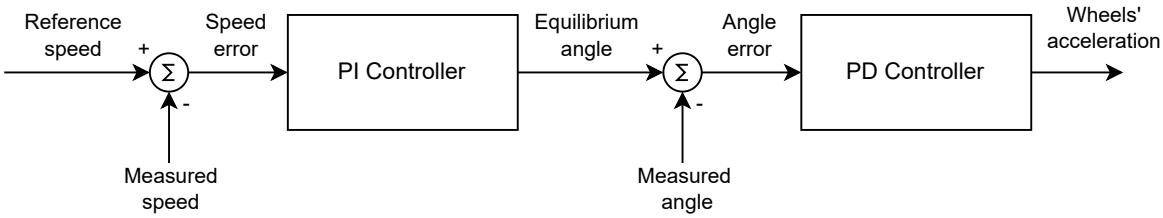


Figure 5.4: PD control for stabilisation with the error computed from the equilibrium angle and the measured angle and return a command for the wheels' acceleration. The equilibrium angle is previously computed from a PI controller, which takes a reference speed and the measured speed as input.

In this way, the proportional term has an effect on the speed limit. The I term is linked to the integral of the speed and therefore to a notion linked to distance. In this way, stabilization takes place over the distance. The result is a behaviour similar to that of a marble settling at the bottom of a bowl. A limiter on the I term has been installed to prevent the robot from tilting too much when translated manually. The proportional term P is linked to speed, and is used to tilt the robot in the right way to obtain the desired speed.

5.3 Enhanced stability engine

5.3.1 Advance speed profile

To avoid disturbing the system too abruptly, the speed reference is incremented progressively on the basis of a trapezoidal profile. This means that the speed evolves from an initial speed to a final speed based on a constant acceleration during the transition. This avoids oscillations in the robot due to the way the controllers operate when changing the reference speed. The trapezoidal speed profile is represented in Figure 5.5.

There are also other profiles that could be used, such as 5^{th} degree interpolation, which allows the acceleration, and therefore the forces, to be continuous during the transient speed period.

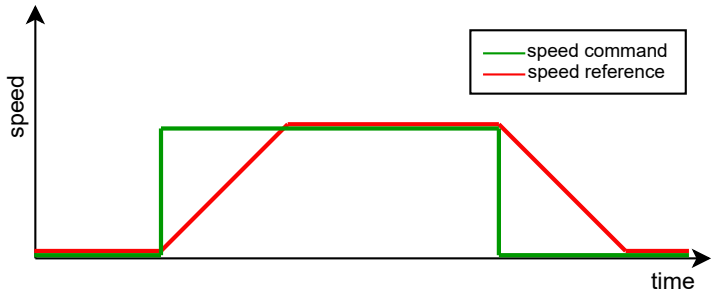


Figure 5.5: Trapezoidal profile for speed reference transition.

5.3.2 Rotation speed profile

Even if there is no direct link between turning speed and stability, a similar speed profile can be applied for rotation speed control.

Such a trapezoidal speed profile will produce fewer sharp movements and therefore induce less noise on the measurements and avoid wheel slip.

5.4 Controller block representation

The controller used in the theoretical representation of the overall system is composed of the stability engine in itself and the trapezoidal speed profile limiters. This block is represented in Figure 5.6:

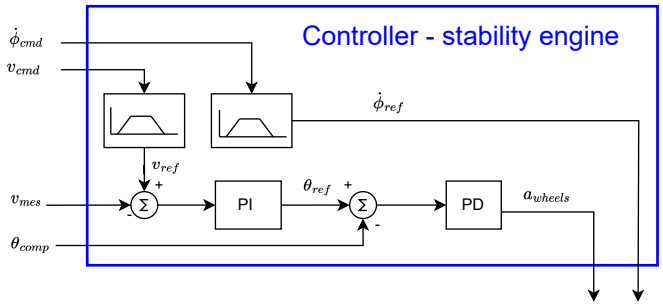


Figure 5.6: The schematic representation of the controller block. The PI and PD sub-controllers are represented with their respective error calculation. The two speed profiles are shown in the upper left corner, in order to compute v_{ref} and $\dot{\phi}_{ref}$.

Chapter 6

Motor drivers

The purpose of the motor drivers is to generate commands that can be interpreted by the actuators on the basis of commands generated by the controller.

There are two types of commands used to control the robot:

- Wheels acceleration a_{wheels}.
- Turning speed of the robot $\dot{\phi}$.

On the other hand the physical device needs two inputs:

- The left wheel speed v_{left}.
- The right wheel speed v_{right}.

Therefore, the two commands should be combined in a way that the two inputs of the motors operate the command as expected. In addition, a logic controller system must be added to comply with the specific commands of the high-level controller.

6.1 Wheel speed computation

In this section, the computation of motors inputs based on acceleration and rotation components will be described.

6.1.1 Acceleration integration

To make the robot follow a given acceleration by providing the speed, a continuous integration is required:

$$v_{advance} = \int a_{wheels} dt$$

6.1.2 Rotation control

To make the robot turn, a speed difference between the wheels is required. To link the turning speed (°/s) with the differential wheel speed (cm/s), the following equation has to be used:

$$\dot{\phi} = v_{diff} \frac{L}{2} \cdot \frac{180}{\pi} \tag{6.1}$$

$$\Leftrightarrow v_{diff} = \phi \frac{2}{L} \frac{\pi}{180} \tag{6.2}$$

6.1.3 Left and right wheel speed

The formula for the left and right speeds is just a simple addition or difference of the advance and differential terms.

$$v_{\text{left}} = v_{advance} - v_{diff} \tag{6.3}$$

$$v_{\text{right}} = v_{advance} + v_{\text{diff}} \tag{6.4}$$

With $v_{advance}$ the instantaneous speed from the acceleration command to balance the robot. The differential speed concept is well represented in Figure 6.1.

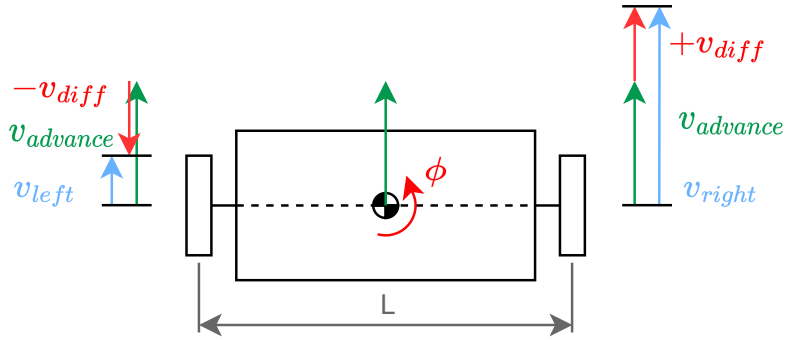


Figure 6.1: Top view of the robot while a differential speed is applied to the wheels in order to perform a turn.

6.2 Logical command operator

The purpose of the logical command operator is to control the behaviour of the driver according to the flags given by the high-level logic. It is used to:

- Control the speed/position of the lifting arms.
- Deactivate the motors in the event of an emergency stop or at rest.
- Impose a speed on the wheels, this is useful for two reasons:
  - Induce a free fall by setting the speed to 0.
  - Operating a counter-rotation when the lifting mechanism is in action, either in a raising or a lowering sequence. This allows to not move the wheel-ground contact point during these sequences. To achieve that, the wheels' angular velocity should match the lifting angle rate.

It also provides feedback to the high-level controller on the lifting arms position.

6.3 Motor drivers block representation

The theoretical implementation of the motor drivers block is represented in Figure 6.2.

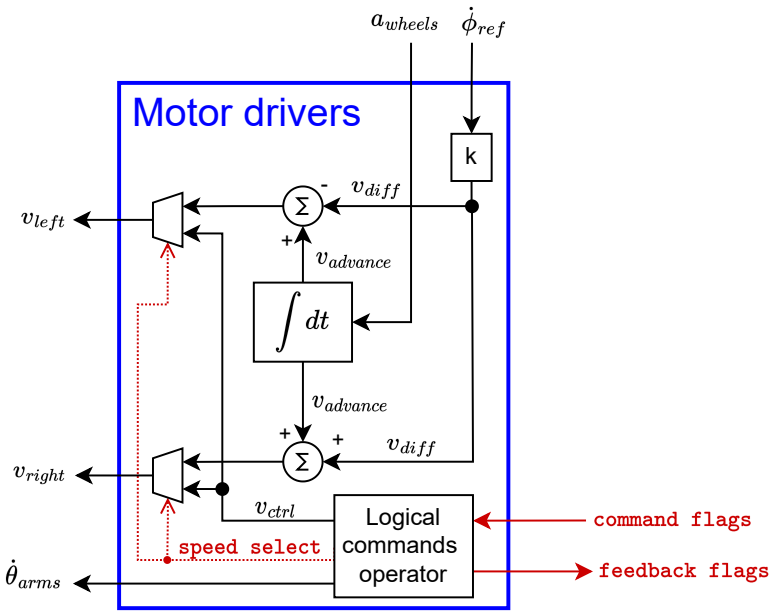


Figure 6.2: The schematic representation of the driver block. The acceleration is integrated in order to get the advance speed, which is then provided to the sum blocks in providing the left and right speeds. The logical command operator imposes speeds to the wheels and controls the lifting mechanism.

Chapter 7

Executive controller

The purpose of the executive controller is to switch between several different operating states according to requirements from the application domain. There are several ways of implementing such a high-level controller, such as a series of logic functions or a finite-state machine (FSM). The FSM was chosen for its ease of creation, implementation, understanding, extension and reuse. In this self-balancing use case, it is mainly used to execute the initial lifting and final lowering sequences but the proposed framework is able to capture any kind of finite state machine and associate events from the designed system to trigger transitions. This section start by describing the FSM framework before illustrating the FSM of the self-balancing robot.

7.1 Finite State Machine Framework

The FSM is a simple directed graph connecting states and transitions which can easily be represented graphically in modelling languages such as UML as pictured in Figure 7.1. They do not allow sub-states inside a state.

- `States` are used defined and identified by a labelling name. A specific state is the initial state in which the system is at startup. A possible final state can capture the system at shutdown. When up and running, control system transition between different operating modes (possibly nominal or degraded). The states are completely connected by transitions with at least one incoming and outgoing transition except possibly for the initial and final state.
- `Transitions` enable to move from a state to a connected state. They are specified by a set of logical rules (or guards) that must be met depending of the actual state. In our framework, those rules are based on a set of flags of different natures:
  - user inputs: a specific command to move the robot in some direction
  - sensor input: some key condition detected in the environment (robot is falling)
  - actuator feedback: giving information about progress or completion of some action (e.g. arm extension) A more precise description of the flag mechanism is given in Chapter 8.1.

7.2 Robot's Finite State Machine

In the case of this self balancing robot, the different operating states are:

- `Rest/emergency`: motors are disabled , they have no holding current.
- `Raising`: the robot is lifted using its lifting mechanism.
- `Stand up`: dynamic balancing state, the lifting mechanism retracts.
- The lowering sequence takes place in three parts:
  1. `Prepare arms`: the robot is still in dynamic balancing and extends its lifting mechanism.
  2. `Free fall`: the robot sets the motors' speeds to 0 in order to fall into the extended lifting mechanism.
  3. `Soft fall`: the robot retracts its lifting mechanism.

Those states are connected as shown in Figure 7.1:

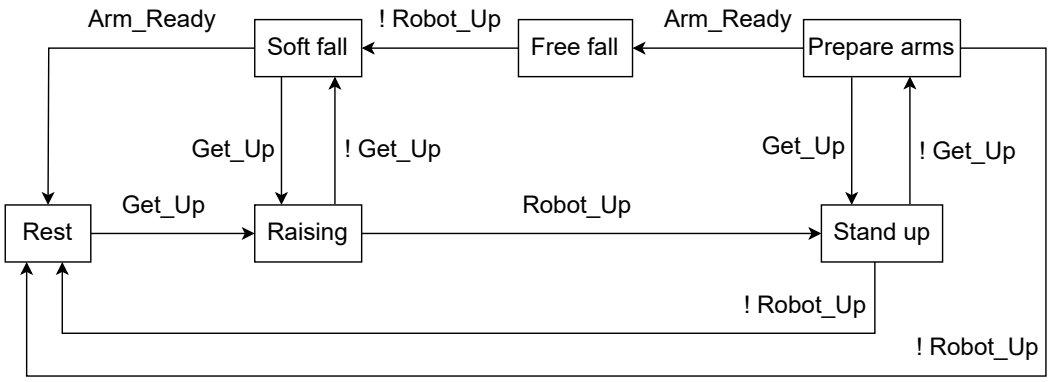


Figure 7.1: Bloc diagram of the FSM representing the state of the robot with each transition.

The transitions are associated with flags of the following kinds:

- User inputs such as Get_Up or !Get_Up (get down). All user flags are listed in AppendixE
- Angle trigger: Robot_Up will allow to detect the robot is up and know when to start the dynamic balancing or when to start retracting the lifting mechanism during the lowering sequence.
- Motor driver feedback: Arm_Ready enables to know when the lifting arms are fully extended or retracted.

All details relating to the implementation of the FSM are given in Section 8.1

Part II

Implementation

Chapter 8

Software architecture

This section describes how the different blocks have been implemented on the various electronic boards, explaining their software architectures. The blocks of the overall system detailed in Part I are highlighted with different colours in Figure 8.1 to show on which board they run (GRiSP or ESP32).

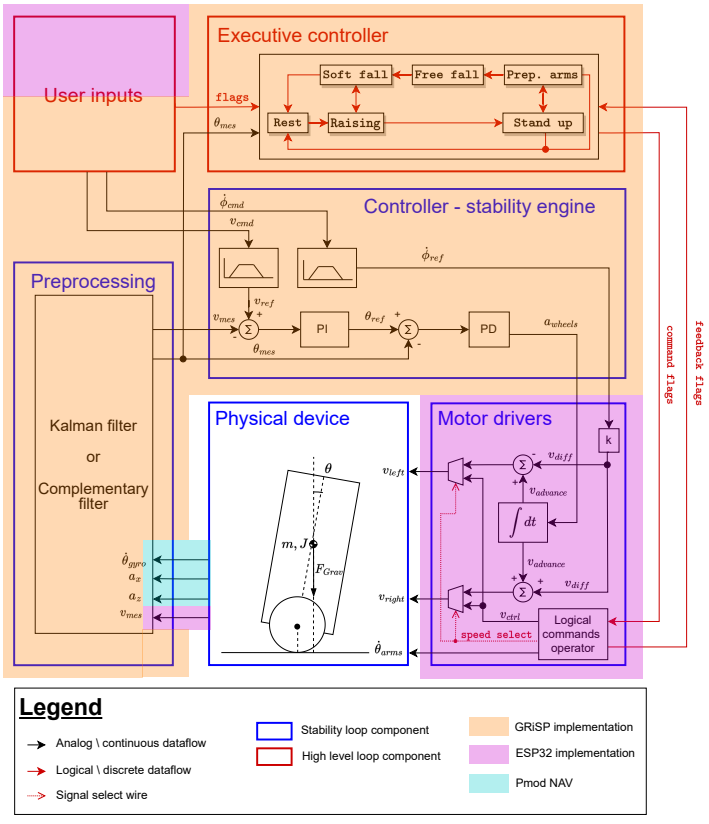


Figure 8.1: Global system with highlighted tasks assigned to GRiSP and ESP32. Most tasks are assigned to GRiSP. The parts relating to interaction with the motors, i.e. drivers and speed measurement, are dedicated to the ESP32 only. User input is shared between these two boards, as user data is received by LoRa on ESP32 and directly transferred to GRiSP via I^2C.

Figure 8.2 shows the overall software architecture, resulting from the various opti-

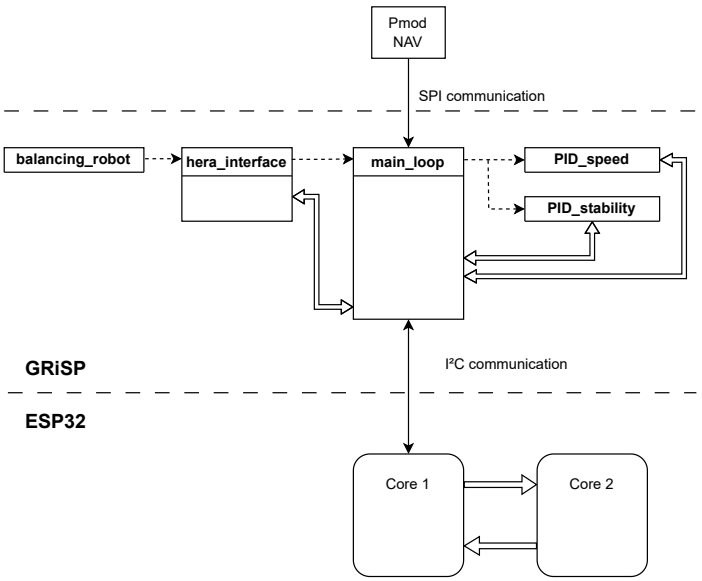


Figure 8.2: Overall software architecture.

misations (see Chapter 9). Communications between processes, between cores and between electronic boards are also detailed. The aim of this chapter is to link the overall theoretical system (as shown in Figure 8.1) to the software architecture.

8.1 GRiSP software architecture

The GRiSP runs on Erlang. Programming with this language is very particular, as Erlang favors the construction of multiple processes. An Erlang process can be considered as a very light thread, i.e. a piece of code that runs autonomously, "living" simultaneously among other processes. Processes can interact with each other by sending messages. They can also spawn new processes.

From this point of view, a process becomes the ideal tool for hosting a control loop, where running independently and at an adapted frequency is a key factor.

As seen in Figure 8.3, the software part of the robot running on GRiSP is composed of four different processes:

- The GRiSP application with the process of `balancing_robot`.
- Hera with the process `hera_interface`.
- The robot with the process `main_loop`.
- The PID controllers with the processes `PID_speed` and `PID_stability`.

8.1.1 GRiSP application process: balancing_robot

This process is the first to be launched after the the GRiSP has booted up. Once **balancing_robot** has been spawned, it calls its **start/2** function. Which initialises the supervisors, the Pmod NAV and the Numerl library. It also spawns the second process, which is **hera_interface**.

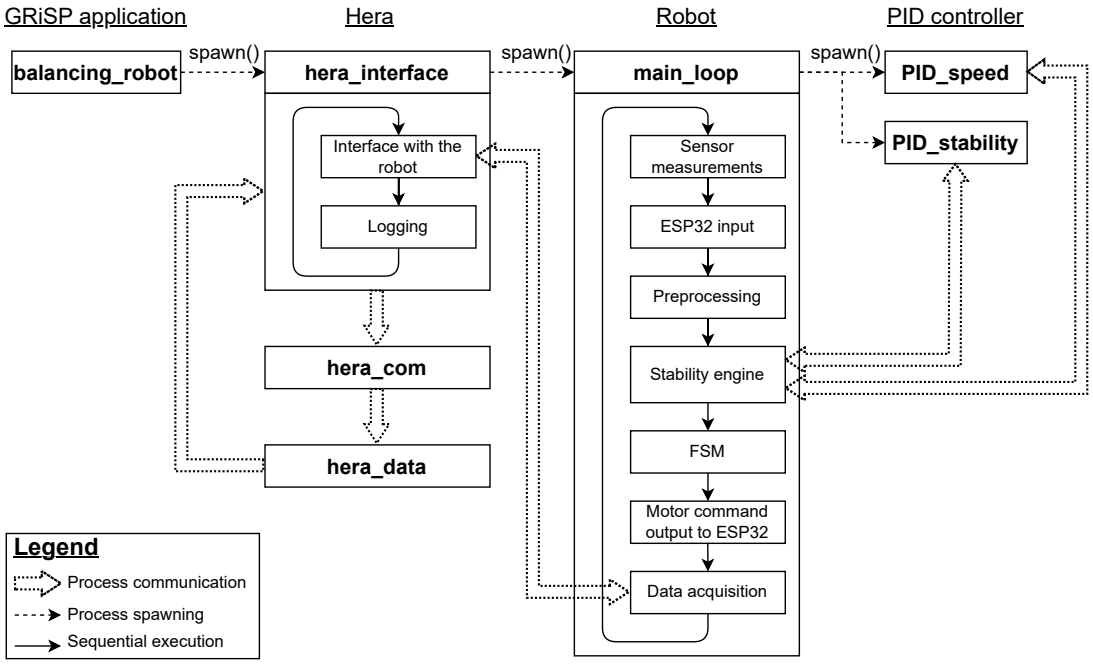


Figure 8.3: Process architecture running on GRiSP.

8.1.2 Hera process: hera_interface

This second process is a hub used to communicate between Hera and the robot process. It allows to keep the functionalities of Hera while letting the **main_loop** process run by itself. This choice is an optimisation detailed in Section 9.7. It uses the behaviour of **hera_measure** which has two callback functions that must be implemented: an initialisation function and a looping function [7] [43].

- **Initialisation**: This function creates the **main_loop** process and saves its process ID to be able to communicate with it. It also initialises a function to start the loop, as required by **hera_measure**.
- **Loop**: This function has two tasks: retrieving data from the robot and logging the values when asked to. The first task is done by using the process ID of **main_loop** and sending messages to it. Any query made by the user can be implemented in **main_loop**. The second task is also done by exchanging messages. To better fit the demands of the project, modified functions for logging are used instead of using the ones implemented in Hera. At each new logging sequence, a new file is created with the logged values.

8.1.3 Robot process: main_loop

This process hosts the loop responsible for the self balancing. As the previous process, it is divided into two parts: the initialisation part and the looping part. The loop executes many different tasks detailed hereafter.

- **Initialisation**: an Erlang term storage (ETS) table, used to store and retrieve data inside a process, is initialized to allow dynamic modification of specific

parameters. This is very useful for testing purposes. The gyroscope is calibrated to get rid of its DC offset, ensuring accuracy of measurement. This is followed by the launch of the I²C communication and the initialisation of the Kalman filter. Finally the two last processes for the PID controllers are spawned and their process IDs are saved for communication between the robot and the controllers.

- **Loop**: The loop is composed of a set of sequential functions:

1) Sensors measurements Three values are measured by the Pmod NAV at each loop: the angular velocity with respect to the y-axis of the Pmod NAV and the acceleration in the x-axis and z-axis of the Pmod NAV.

2) I²C input Five bytes are received from the ESP32 through I²C. The four first bytes represent two half-float values for the rotation speed of the left and right wheels of the robot. The last byte is made of eight flags, each represented by one bit. This is shown in Table 8.1.

1	2	3	4	5
Speed left		Speed right		Input flags

Table 8.1: Bytes received via I²C from ESP32.

The variable v_{mes} shown on the global diagrams corresponds to the average of the left and right speed measurements. This average value could have been calculated directly and sent from the ESP32, saving two bytes in the transfer, but for future position calculation purposes it was considered worthwhile to supply both velocity values. Regarding last byte, the eight bits are detailed in Table 8.2

1	2	3	4	5	6	7	8
`Arm_Ready`	`Switch`	`Test`	`Get_Up`	`Forward`	`Backward`	`Left`	`Right`

Table 8.2: Bit decomposition of the I²C input flags byte.

Bit number one is the feedback flag from the lifting mechanism, the seven remaining bits are users commands. `Switch` is used to switch from the Kalman (1) filter to the complementary filter (0). When `Test` is set to 1, data recording starts. `Get_Up` is an instruction used to switch between the self-balancing (1) and the resting position (0). The four last bytes are direction commands.

3) Preprocessing This task takes care of the computation of the angle of the robot as explained in Chapter 4. At each iteration, it first computes the angle as seen from the accelerometer data. It is then used to feed the Kalman filter as well as the complementary filter.

```
%Angle based directly on the sensors
Angle_Accelerometer = math:atan(Az / (-Ax))*?RAD_TO_DEG,

%Kalman filter computation
{X1, P1} = kalman_angle(Dt, Ax, Az, Gy, Gy0, X0, P0),
```

```
[Th_Kalman, _W_Kalman] = mat:to_array(X1),
Angle_Kalman = Th_Kalman*?RAD_TO_DEG,

%Complementary angle computation
K = 1.25/(1.25+(1.0/Mean_Freq)),
{Angle_Complem_New, Angle_Rate_New} = complem_angle({Dt, Ax, Az, Gy, Gy0, K, Angle_Complem,
     Angle_Rate}),

%Select angle between kalman or complementary
Angle = select_angle(Switch, Angle_Kalman, Angle_Complem),
```

Finally, it selects which angle should be used in the control loop between Kalman and complementary based on the `Switch` input flag mentioned above.

4) Stability engine As explained in Chapter 5, its purpose is to compute the required acceleration to balance the robot in real-time, based on the measured angle and the robot's speed. To do this, a first communication with the **PID_speed** process is established in order to get the reference angle. This value is then communicated to the **PID_stability** process to retrieve the acceleration required for stability.

The stability engine also computes trapezoidal speed profiles for both the reference advance speed, given to **PID_speed**, and the reference rotation speed, given directly to the motor drivers.

5) Executive control FSM The implementation of the FSM is shown in Figure 7.1, which repeats the basics explained in Chapter 7, and specifies the set of trigger flags for state changes.

All the transitions are triggered based on three different flags: Get_Up, Arm_Ready and Robot_Up. These flags are detailed in Table 8.3. The first two flags are inputs from the ESP32 as mentioned above, and are provided by the user. The last flag depends on the measured angle. To avoid problems around the switching point of the value of Robot_Up, a Schmidt trigger has been implemented. This means that the trigger to set the flag to 0 is higher than the trigger to set the flag to 1.

Flag	Value of 1	Value of 0
Get_Up	The robot must get up.	The robot must get down.
Arm_Ready	The arm of the robot is either fully extend or fully retracted.	The arm of the robot is in a transition phase.
Robot_Up	The angle of the robot is smaller than 18°.	The angle of the robot is greater than 20°.

Table 8.3: State flags of the robot.

Each state returns output flags that are later interpreted by the ESP32. These different output flags are: Power, Freeze, Extend and Robot_Up_Bit, F_B as detailed in Table 8.4

Flag	Value of 1	Value of 0
Power	The motors must be turned on	The motors must be turned off
Freeze	The motors must be in holding state	No action on the motors
Retract	The arm must be in the extended state	The arm must be in the retracted state
Robot_Up_Bit	The robot is in dynamic balancing	The robot is resting on the lifting arms
F_B	The robot is tilted forward: Angle $> 0°$	The robot is tilted backward: Angle $\leq 0°$

Table 8.4: Output flags.

The three first flags, Power, Freeze and Retract, are command flags that are interpreted by the logical command operator of the motor drivers (see Section 6.2). The two remaining flags are indicator flags also used by the logical command operator. Their purpose is related to the wheels' counter-rotation during lifting/lowering sequences (see Section 3.1.3).

- The F_B flag is used to know in which direction it is required to turn the wheels, as it depends on the leaning side of the robot.
- The Robot_Up_Bit indicates when the robot is in dynamic balancing. This is used in order to stop the wheels' counter-rotation after lifting up.

6) I²C output to ESP32 Five bytes are sent through I²C to the ESP32. As seen in Table 8.5. The first four bytes represent two half-float values: the acceleration command of the motors and their differential speed command. Both are computed in the stability engine. The last byte represents the four output flags explained in Table 8.4. These flags are written on the five leftmost bits of this byte, as seen in Table 8.6.

1	2	3	4	5
Acceleration		Differential speed		Output flags

Table 8.5: Bytes transmitted to ESP32 via I²C.

1	2	3	4	5	6	7	8
`Power`	`Freeze`	`Retract`	`Robot_Up_Bit`	`F_B`			

Table 8.6: Bit decomposition of the I²C output flags byte.

7) Data acquisition The last task executed, before returning to the beginning of the loop, is the communication with the Hera interface. The outgoing data is useful for logging, or any other application required by the user outside the scope of this master thesis. This is done by sending a message at the start, and one at the end, of a logging sequence. During the logging sequence, messages with data are sent to the interface. In the interface, the data is retrieved and written into a file for later use.

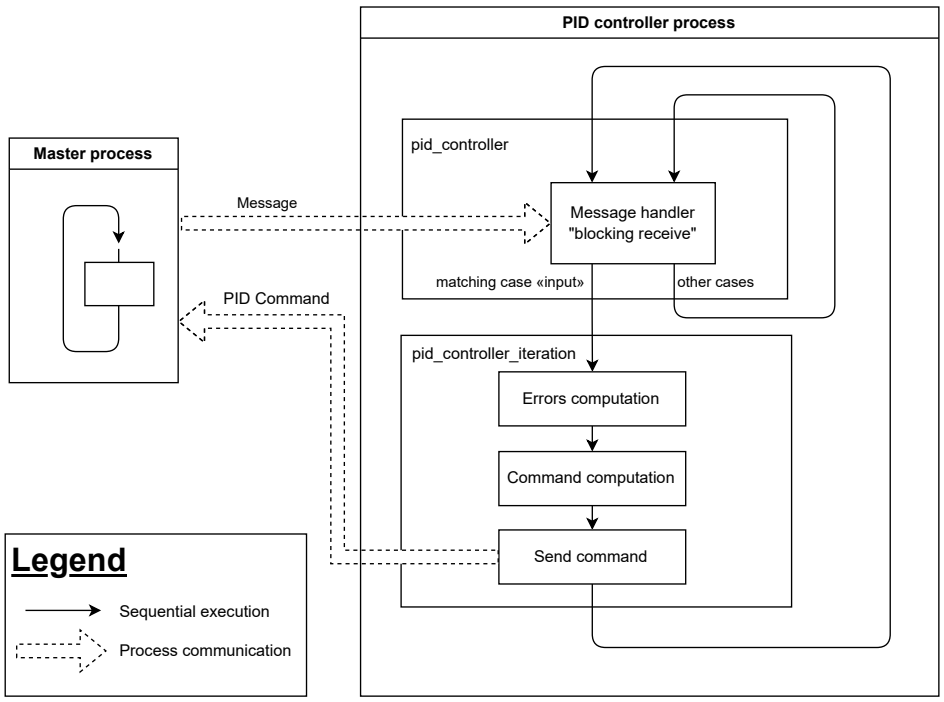


Figure 8.4: PID controller process architecture: the **pid_controller** waits until a master process sends the "input" message to get back the resulting command through the **pid_controller_iteration** function. Other cases could also be used to transmit data to modify parameters, in theses cases no command is expected, then **pid_controller** loop back on itself.

8.1.4 PID controller processes: PID_speed and PID_stability

The PID controller has been implemented in such a way that it can be instantiated by several processes running completely independently. Its architecture is shown in Figure 8.4 and more detailed below:

Each process running the PID controller starts with an initialization function and continues with a looping function:

- **Initialization**: all the internal parameters are initialized, and the loop is started.
- **Loop**: this takes place in two stages, via two functions: pid_controller and pid_controller_iteration.
  - pid_controller works like a big mailbox. It waits to receive a message from a master process. Once the message has been received, it interprets it and decides either to call pid_controller_iteration to calculate a control command or to modify internal variables based on the message received. The Table 8.7 lists all the commands that can be interpreted by this function.
  - pid_controller_iteration is the calculation part of the process, where proportional, integral and derivative errors are calculated. Based on these values, the command is computed taking into account the effect of the limiters. The command is sent to the process master, and the function ends by calling pid_controller again, closing the loop.

Atom matching case	Description	Next called function
`exit`	Terminate the process	None
`kp`	Modify the K_p parameter	pid_controller
`ki`	Modify the K_i parameter	pid_controller
`kd`	Modify the K_d parameter	pid_controller
`limit`	Set a new value on the output command value limiter[1]	pid_controller
`intlimit`	Set a new value to the integral error limiter[1]	pid_controller
`setpoint`	Set a new setpoint	pid_controller
`input`	Provide a new input value in order to compute the resulting PID command	pid_controller_iteration
`reset`	Set the integral error to 0	pid_controller

Table 8.7: Atom cases matching logic inside the pid_controller function.

8.2 ESP32 software architecture

The code running on the ESP32pico CPU is written in ino and C++, and runs on two cores [44]. It is made out of three parts:

- LoRa communication through SPI, to receive messages from the user.
- I^{2}C communication, to receive commands from the GRiSP and send back measures and user inputs. The ESP32 behaves as a slave.
- Motor engine, to generate the signals required to run the motors from the GRiSP inputs.

The code overview running on ESP32 is shown in Figure 8.5.

8.2.1 LoRa communication loop

This is the main loop of the ESP32 core 1. Its purpose is to constantly retrieve the user inputs transmitted over LoRa. This is done by communicating with the embedded LoRa module via SPI. Each message is sent twice and compared to reduce the risk of message corruption. Some flags sent by the user, such as the emergency stop command or the test trigger, can be directly interpreted by the LoRa event handler module. Two libraries are used to implement this module: `<SPI.h>` [45] and `<LoRa.h>` [46].

8.2.2 I^{2}C communication

The ESP32 acts as a slave on the I^{2}C. On an incoming request from the master, it must interrupt its execution sequence to process that requests coming from the GRiSP. These interruptions are instantiated in core 1 and therefore only affect this core.

There are two types of interruptions:

[1] A non-positive value deactivates the limiter.

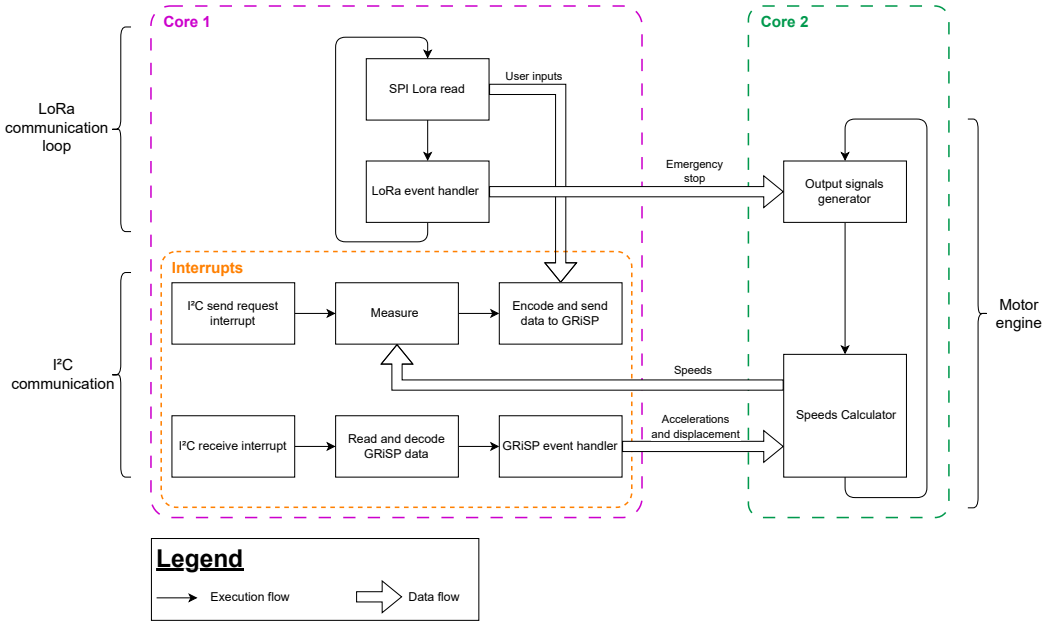


Figure 8.5: Bloc diagram of the code structure running on the ESP32.

- Send request: when the master wants to read information from the slave. In this case, user input flags, feedback flags and speed measurements are involved.
- Receive: the master transfers a series of data to the slave, in this case acceleration and command flags.

The `<Wire.h>` [47] library is used to implement this module.

8.2.3 Motor engine

The Motor engine generates a set of signals for each motor. They consists of two parts:

- signal generator: generates signals at the right frequency to control the step inputs. It is also responsible for controlling the direction signal and the enable signal used to activate/deactivate the motor.
- speed calculator: this block performs all calculations related to motor speeds (integration of acceleration, application of differential speed, moving the lifting arms). To obtain the desired speed for each motor it requires to calculates the frequencies of the step signal of each motor :

$$f_{step} = \underbrace{\frac{v}{2\pi \times r}}_{\text{Speed to rev per second}} \times \underbrace{\#steps\ per\ turn \times \#microsteps}_{\text{Total number of microsteps per rev}} \tag{8.1}$$

Chapter 9

Problems and optimisations

The system as described in its final version throughout this master thesis is, among other things, the result of multiple optimisations and problem solving. Some of these optimisations are crucial for achieving dynamic balancing, while others have contributed to improving the system's performance. In this section, these optimisations and their effects are presented. They are summarised as hatched areas in Figure 9.1 representing the overall software architecture.

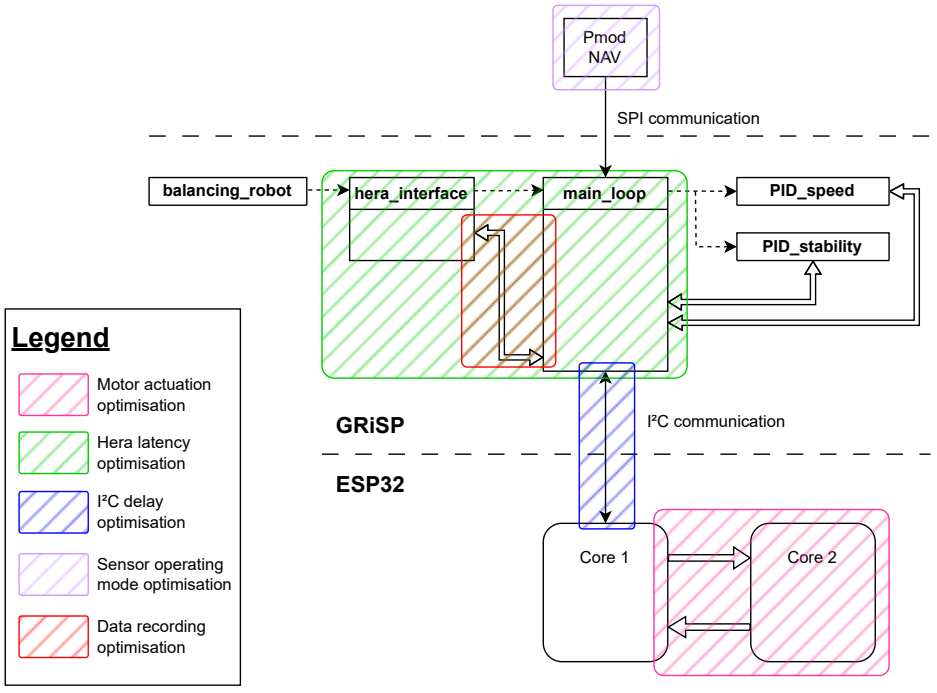


Figure 9.1: Diagram of the overall software architecture. The parts suffering from problems and corrected by optimisation are hatched.

The objective of these optimisations is always the same: **increase the update frequency of the stability loop**. However, evaluating the performance of each optimisation on the basis of frequency gain is not a good metric. Indeed, comparing two successive optimisations on the basis of their frequency gain does not make it possible to determine which one is the most efficient. This can be seen in the example shown

in Figure 9.2. **Instead of the frequency, the gain over the period of the loop should be used in order to keep a comparable basis for all the optimisations**.

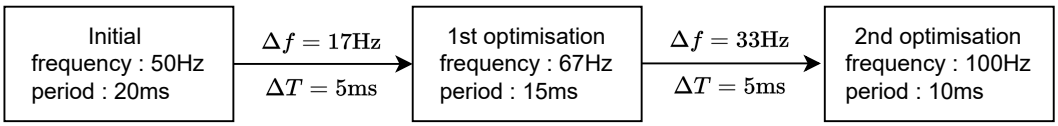


Figure 9.2: Example of successive optimisation resulting in different frequency gains. The spared time period, ΔT, is the same but the frequency, Δf, gain is different.

Another key factor to take into account when optimising is the delay. The delay corresponds to the end-to-end time between the measurement and the application of the resulting command to the actuators. When using a system with serial processes communicating with each other, as is very easily done in Erlang, it is possible to have a high update frequency but relatively long end-to-end delays. To ensure good controlability, when a measurement is taken, this new information must pass as quickly as possible through all the successive loops to reach the actuator in the form of a command.

9.1 Available GRiSP optimizations

There are already a few optimisations provided by the GRiSP :

- The upgrade from GRiSP to GRiSP 2 offers huge performance gains.
- The use of the Numerl library, offering a set of NIFs to speed up matrix computations [9].

9.2 GRiSP GPIO frequency

Problem statement: at the beginning of the design period, it was considered to directly feed the stepper driver cards via signals generated by the GRiSP GPIOs. As shown in Appendix G, the Erlang functions available under GRiSP are not able to generate proper signals at the required frequencies. Using f_{step} formula of Subsection 8.2.3 results in a velocity of 2910 steps per second, thus a frequency of 2.9 kHz which is too high.

Solution: to overcome the problem of the speed of actuation of the motors, another circuit board than the GRiSP has been used. The circuit board used is an ESP32. This circuit board has a maximum GPIO switching frequency of 122 kHz. which is more than enough to control the motors.

9.3 Motor actuation

Problem statement: in early versions of the code running on the ESP32, lots of vibration was generated by the motors, to the point where some nuts and screws became loose. This problem is caused by fluctuations in frequency generation within the ESP32. After investigation, it was determined that the main cause lay in the fact that the

implementation of the communication part and the engine were running on the same core. Hence, during each I2C communication interrupt, it was not possible to generate the proper signal required to make the motors spin as expected. A similar problem can occur with LoRa communication.

Solution: to counter this problem in subsequent versions of the code, motor signal generation is assigned to a second core entirely dedicated to it.

Problem statement: another source of fluctuation in frequency generation has also been identified: the use of serial print. These are very useful for debugging but have a harmful effect on the code execution on both cores at the same time. Using a serial print on one core affect the generation of GPIO signals on the other core. This should then be avoided at all costs when attempting to generate a signal to control stepper motors.

Solution: by placing the motor engine on the 2^{nd} core and banning the use of serial prints, the motors run very smoothly, as explained in 8.2

9.4 Hera measure latency

Problem statement: Initially only one **hera_measure** process executed all the tasks of the **main_loop** process explained in Chapter 8. This is the easiest way of implementation but it doesn't make use of the potential of Hera and Erlang. To fully use this potential, the measurement task was separated from the other tasks and moved to an additional **hera_measure** process. Communication between these two processes are provided by **hera_com** and **hera_data**. This is how Hera is expected to be used. The purpose of doing this is to avoid the whole system from crashing due to Pmod NAV crashes.

This way of setting up the Hera files led to poor robot performance. This was due to the delay between the starting point of a measurement in the Pmod NAV and the arrival of this measure in the Kalman filter. In between these two actions, different tasks are executed, like sending the data through **hera_com** to the **hera_measure** process with the Kalman filter, receiving the input from the ESP32 through I^2C etc. Every little task takes some time to execute, which adds up to the total delay. The problem with an excessive delay is the reactivity of the motors to the falling motion of the robot. A delay that is too high will prevent the motors from reacting fast enough for the robot to stay in equilibrium.

Let's sum up the different delays that have to be taken into account:

- t_1: The time for the Pmod NAV to make the measurement.
- t_2: The time for the Pmod NAV to send the measurement to the GRiSP board.
- t_3: The time for the first **hera_measure** process to make its calculations.
- t_4: The time for **hera_com.erl** to send the values to the second. **hera_measure** process
- t_5: The time to make the angle computation of the kalman filter in the second hera_measure process.

The delays t_1 and t_2 are not easily measurable but are negligible with respect to the rest of the delays. The total delay of the measure from the Pmod NAV to the Kalman filter is approximately 20.34 ms. By looking at each delay individually, one delay stands out. The delay for the communication between both hera_measure processes through hera_com takes approximately 15.19 ms. This represents almost 75% of the total delay.

The maximal delay that has been empirically observed, by simulation, to lead to a marginal stability, is approximately 20 ms. The said simulation is detailed in Appendix F. This limit makes it impossible to use the communication between Hera processes in the application of stabilising a falling object like the robot of this master thesis.

Solution: fall back to initial design with a single measure process. Using Hera and Erlang to their potential would have been useful, but it is not possible in such an application. Nevertheless, this does not mean that Hera cannot be used at all, as explained in Chapter 8.

9.5 Bus communication latency

Problem statement: one of the tasks of the **main_loop** process is to exchange data with the ESP32 through I^2C. This type of communication can be quite slow as its throughput is limited by its clock frequency of 100 kHz. This means that the maximum throughput of the I^2C bus is 12.5 kB/s. The delay of sending 1 byte is thus 0.08 ms.

Each iteration of the loop transfers a particular amount of data between GRiSP and ESP32, two transfers occurs :

- from ESP32 to GRiSP, the payload consist of : left speed (float) + right speed (float) + input flags
- from GRiSP to ESP32 the payload consist of : acceleration (float) + differential speed (float) + output flags

For each communication, the slave address must also be specified during transfer.

With the double precision float as used in Erlang, each float is 8 bytes wide, the flags and address being 1 byte wide each. Each transfer is therefore 18 bytes long and takes delay of 1.44 ms. this result in a total of 36 bytes at each iteration and thus a delay of 2.88 ms.This is a huge delay for such a system.

Solution: in order to make the optimal use of the limited bandwidth, the amount of bytes sent can greatly be reduced by trading off some bytes in exchange for a lower precision. The type of floats sent has been reduce to the half-precision floats[48]. This allows to send 2 bytes per float instead of 8 bytes for double-precision floats. Reducing the total amount of bytes sent to a total of 2×6 bytes which result in a total delay of 0.96 ms. This optimisation reduces transit times by a factor of 3, as shown in Figure 9.3

Double-precision floats are written on 64 bits: 1 sign bit, 11 exponent bits and 52 mantissa bits. In comparison, half-floats are written on 16 bits: 1 sign bit, 5 exponent bits and 10 mantissa bits[49]. There is a trade-off between the amount of bytes storing the information and the precision and range of possible values. Using half-floats reduces the amount of bytes sent, but it also reduces the precision of the values and the range of

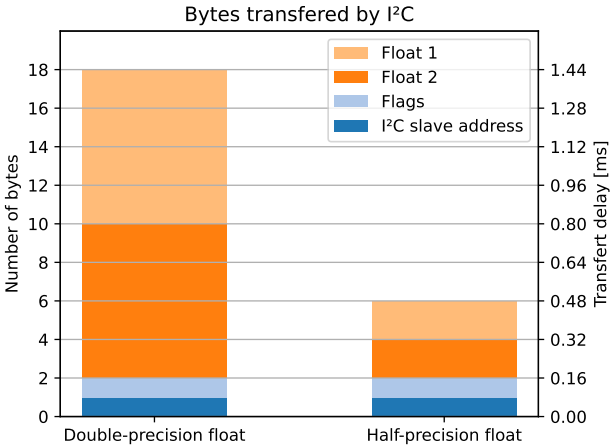


Figure 9.3: Number of bytes transferred via I²C at each communications. before and after using half float representation.

possible values sent. As a general formula depending on n, values between 2^n and $2^{(n+1)}$ have a precision of $2^{(n-10)}$. This is more than enough in case of this master thesis.

9.6 Sensor operating mode

One of the main bottlenecks that has been solved late in the timeline of the project is the configuration of the Pmod NAV. The default operating mode of the Pmod NAV is to take measures at a rate of 59.5 Hz. When data is read from the Pmod NAV at a faster rate, it returns the same value as if it had made the same measurement. This sort of puts a boundary on the frequency of the whole system. To solve this problem, the Pmod NAV was configured at 238 Hz which is higher than the maximal frequency of the robot process.

Between two sampling times, the Pmod NAV averages the data before returning this average. This acts as a low-pass filter by averaging out the high frequencies. Reducing the noise before the Kalman filter reduces its performance. Even if the robot would have been able to stay somewhat in equilibrium, increasing the Pmod NAV sample frequency was necessary both reasons.

9.7 CPU load during data recording

Problem statement: in order to evaluate the system and study its performance and limitations, a number of tests were carried out (see Part III). A range of data should then be collected during these tests. However, it is essential that these tests represent the reality as accurately as possible: they cannot have any impact on the robot's operation. In addition, for test analysis, it is essential to collect as much data as possible. The sampling frequency should be as high as possible, ideally the same as **main_loop**.

Initially, the test method had no impact on the robot's performance because its sampling rate was very low. The initial architecture consists of **hera_interface** sending a

data request to **main_loop** at each of its loops and writing directly on receipt. This made the frequency of **hera_interface** drop to about 60 Hz. The data request frequency dropped to the same amount, which led to a lot of data lost as shown in Figure 9.4

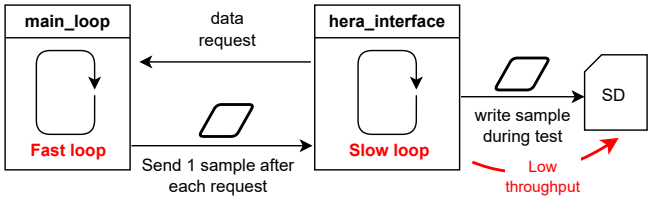


Figure 9.4: Log records before optimisation, the sampling is as fast as **hera_interface** frequency.

The solution to collect data at higher frequency without losing data was to buffer and send a packet of samples from **main_loop** instead of a single sample. These packets contain all the data from previous cycles not yet collected by **hera_interface**. The frequency drop due to the file writes cannot be fixed without using a second GRiSP that would take care of that task. After implementing this method, a large amount of data was generated as expected.

However, this led to a new problem: writing such a large amount of data in real time placed an additional load on the GRiSP CPU, causing the frequency of all the processes to drop. This is shown in Figure 9.5

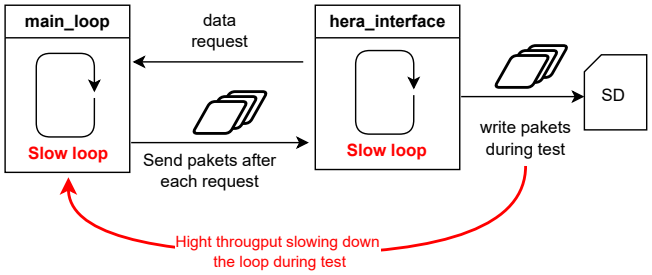


Figure 9.5: Log records after first optimisation, the sampling is as fast as **main_loop** frequency which is slow down due to data writing.

To solve this second problem, it is decided to write the data after the logging sequence. To achieve this, the data is stored in an array in **main_loop** during the logging sequence. At the end of this sequence, the array containing all the data is sent to **hera_interface** which then writes everything in a file. This is shown in Figure 9.6

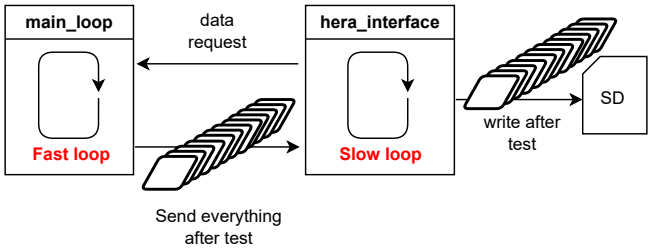


Figure 9.6: Log records after second optimisation, the sampling is as fast as **main_loop** frequency with lower frequency drop.

This second optimisation comes with its own problems: appending new data to a large array and writing all the data at once. These two problems cause a frequency drop from 200 Hz to about 140 Hz during almost 30 seconds. This frequency drop is small enough for the robot to be able to stay in equilibrium during this time span and does not longer affect the logging period.

Part III

Evaluation

Chapter 10

Characterisation

The system as a whole is expected to behave in a well-defined way when faced with stimuli. This chapter presents a study of the robot's behaviour based on experimental measurements under normal operating conditions and explore the system's limitations.

Two different types of causes can have an impact on the robot's behaviour, either an internal command (from the user) or an external disturbance (from the environment):

- Internal command:
  - Move forward/backward.
  - Turn at a certain angular speed.
  - Raise up and lie back down.
  - Emergency stop.
- External disturbance:
  - Frontal impulse.
  - Additive mass, with and without offset.
  - Inclined surface.

The tilt angle values used to explain the robot's behaviour come directly from the measurements made by the filters. No external angle measurement tools were used as absolute reference, so the measurements could be biased because the angles are seen from different filters, which could produce different values for the same angle measured. For example, they could have filtered out angle spikes that may actually occur and thus not perfectly represent the real situation.

Each test is described in Section 10.2. It is preceded by a first reference test in a-self-balancing position without any stimuli applied. By characterising the robot's limitations, it is possible to determine the system's operating zone. This is done in Section 10.3. Throughout this chapter, the Kalman and complementary filters are compared.

10.1 Reference case

The preliminary tests show how the robot behaves while in dynamic equilibrium without being subjected to any intentional disturbance. This is useful to see the order of magnitude of multiple parameters under the best possible conditions, a flat and smooth surface with no external forces, in order to have a great comparative basis for the other tests.

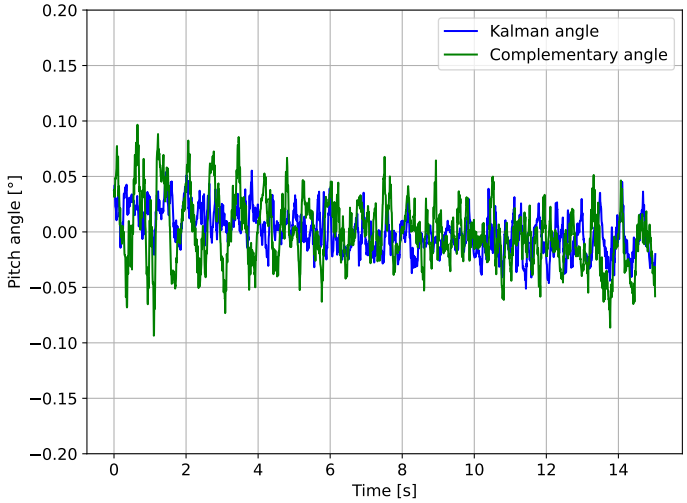


Figure 10.1: Evolution of the angle of the robot while in a stable upright position.

Figure 10.1 depicts two tests capturing the "natural evolution" of the robot's angle when it is not subjected to disturbances. The first is carried out using Kalman filter and the second with data from the complementary filter.

The first point to note is the order of magnitude: oscillations are less than 0.1° which is extremely low. It is very difficult for any observer to even see a slight angle and displacement variation. This shows the robot's perfect mastery of stability and attests to the great work of the sensors, filters and control system.

Closer observation reveals that the complementary filter is slightly more subject to oscillations than the Kalman filter. The use of a Fourier transform allows us to better observe this phenomenon by associating an amplitude to each frequency on the same test samples. The results of the Fourier transform are shown in Figure 10.2. It clearly shows that the frequency content of the test with the Kalman filter is much lower than with the complementary filter.

In both cases, a low frequency peak is observed. One at 1.664 Hz for Kalman and one at 1.398 Hz for the complementary. That is the system's natural frequency.

Finally, a last spike is observed around 30 Hz. This is due to the rigidity of the chassis.

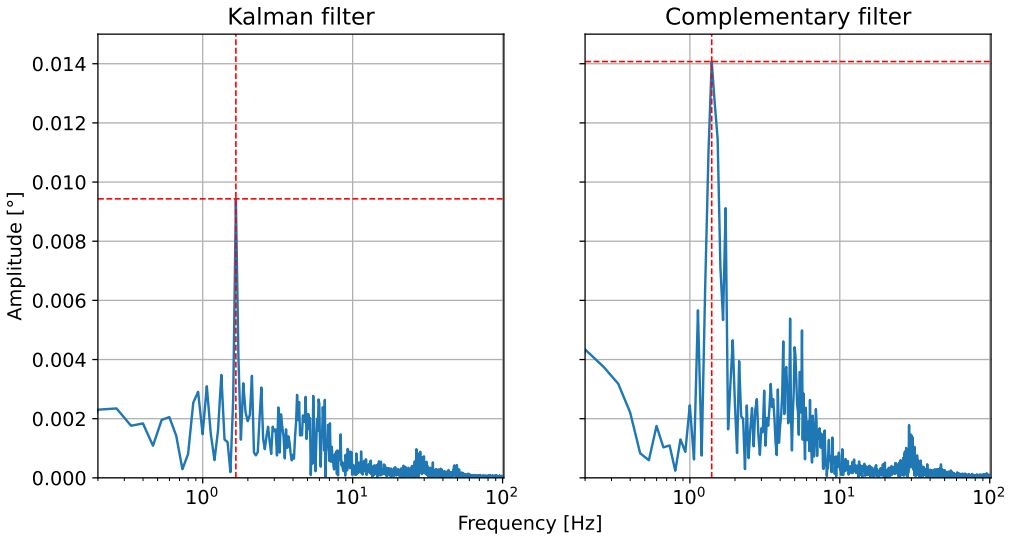


Figure 10.2: Fourier transform of the two signals in Figure 10.1.

10.2 Behavioural study

10.2.1 Straight movement

This test shows the robot's transition from rest to the speed desired by the user, which is 30 $^{cm}/_{s}$ in this case. It also shows the importance of using trapezoidal speed profiles.

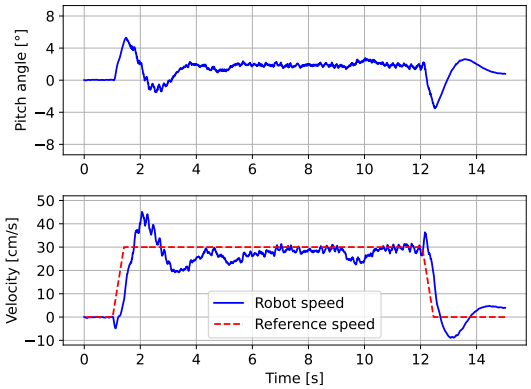


Figure 10.3: Evolution of the angle and velocity of the robot when going forward with a trapezoidal velocity profile.

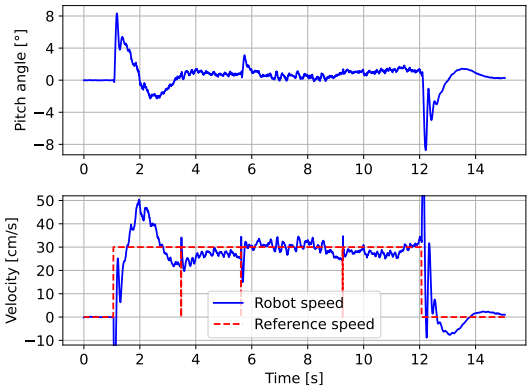


Figure 10.4: Evolution of the angle and velocity of the robot when going forward with an instantaneous velocity profile.

A first observation is the large variation in angle and speed after applying a new command. This is due to the need for acceleration in order to change speed. As demonstrated in Section 3.2, acceleration implies a force impacting the robot's angle.

Immediately after a speed change command, it can be observed that the wheels' speed is opposite to the direction of the desired speed. This behaviour allows the robot to angle itself in the right direction in order to be able to accelerate without falling.

It is clearly visible that the stabilization angle, once the robot is moving at 30 $^{cm}/_{s}$, is slightly greater than 0. This indicates the presence of additional forces when the

robot is moving at constant speed compared to when at rest. These forces have several origins: wheel-ground friction, motor shaft friction and air friction.

Another observation at a non-null constant speed reference is that the variation of the angle is greater than the one measured at a null velocity. This can be caused by irregularities of the wheel and the ground, but might also be due to the set of forces described above.

The use of the trapezoidal speed profile has a very positive impact: the comparison of Figures 10.3 and 10.4 shows that without the trapeze, there is a peak angle of 8°, whereas it is 5° degrees with the trapeze.

The speed profile also has a filtering effect on potential glitches coming from the user's speed command. These glitches have several origins, but the main one is due to signals induced in the communication with LoRa and I^2C. Their effect can be seen in dotted red on Figure 10.4. The test with the trapeze was also affected by such glitches on the command speed but it did not impact the reference speed thanks to this filtering effect.

10.2.2 Rotation movement

This test was carried out in normal conditions, with only a rotation command applied 1 second after the start of the test and deactivated 12 seconds later.

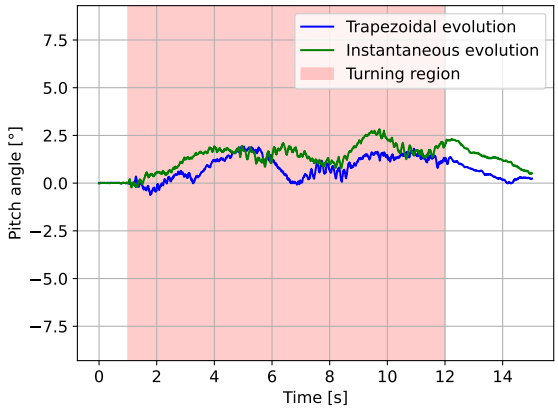


Figure 10.5: Graph showing the angle of the robot when the robot is turning left.

It can be seen that the angle of inclination is affected by this change, with much greater fluctuations than when the robot is not rotating. These fluctuations are probably due to the hypothesis of being able to decouple the robot's rotation from its stabilization, which is therefore not entirely accurate. Robot rotation also induces new phenomena: the gyroscopic effect, the centripetal force and the Coriolis effect. As a result, the robot no longer operates under the same conditions, and the controller is no longer as well adapted as it would be for normal conditions without rotation speed.

However, although the fluctuation is greater than in normal conditions, it remains acceptable for this application and does not place the robot in a zone of vulnerability.

Slight differences can be seen in the application of a trapezoidal profile with lower trapezoidal fluctuations. However, these differences are barely noticeable and probably depend more on slightly different test conditions.

10.2.3 Raise up and lie back down

This test shows the robot's behaviour in the various operating phases described in Chapter 7.

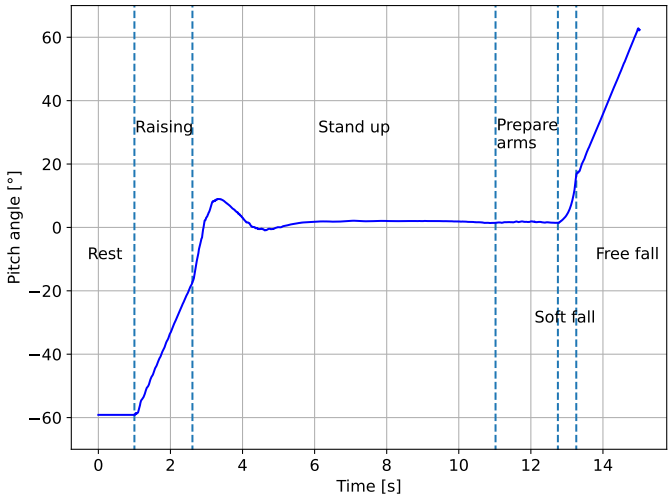


Figure 10.6: Evolution of the robot's angle when getting up and going back down.

In this test, the robot rises on one side (-60°) and gets down on the other (+60°). Each phase has its own specific behaviour:

- $[0.0; 1.0[$ `Rest` phase: the robot lies on one of the retracted arms of the lifting mechanism.
- $[1.0 : 2.61[$ `Raising` phase: characterized by the robot leaning on the lifting mechanism. The angle evolution is very linear since the angular velocity is directly that of the lifting mechanism, which is constant.
- $[2.61; 11.02[$ `Stand up` phase: starting at an angle of -15° to the vertical, the robot gradually stabilizes until it finds its equilibrium point while retracting its lifting mechanism.
- $[11.02; 12.75[$ `Prepare arms` phase: the robot extends its lifting mechanism, inducing small and barely visible vibrations caused by the lifting mechanism's deployment.
- $[12.75; 13.26[$ `Free fall phase`: the body is subject only to gravity due to the motors' speed set to 0.
- $[13.26; 15]$ `Soft fall` phase, the robot is gradually lowered. Its linear behaviour is the same as during the raising phase.

10.2.4 Emergency stop

This test shows how the robot behaves when it is suddenly stopped, i.e. when the wheels are free to spin, with the emergency button hit after 1 second.

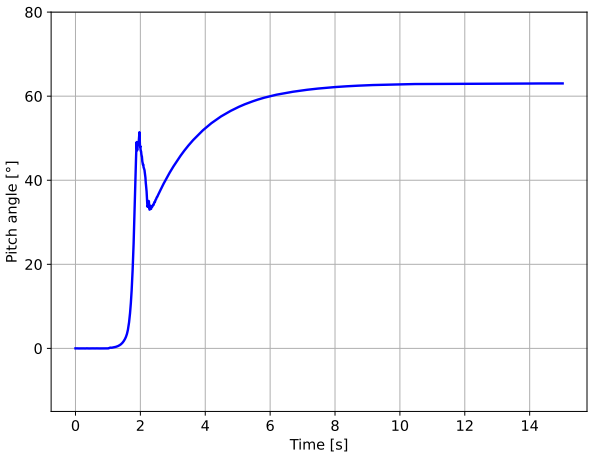


Figure 10.7: Evolution of the pitch angle as seen from the Kalman filter after emergency stop.

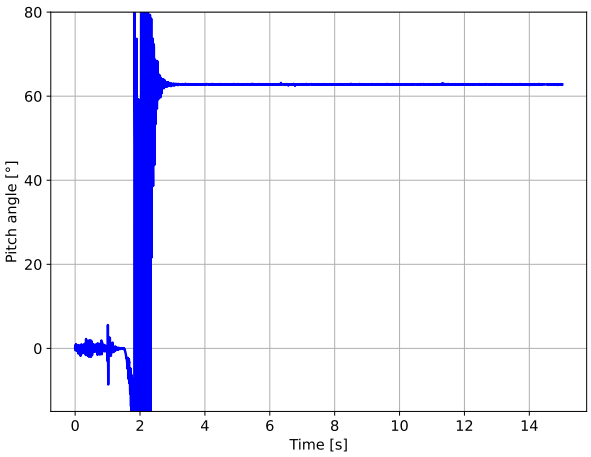


Figure 10.8: Evolution of the pitch angle computed from the accelerometer data after emergency stop.

The graph shows a very strange behaviour:

1. The angle is 0°.
2. The angle rises from 0° to 50° quite rapidly.
3. The angle decreases from 50° to 35°.
4. The angle returns from 45° to 60°.

This is not at all what can be seen during the test: the falling robot bounces by a few degrees due to the collision at around 60° and is directly at rest at 60° less than 2 seconds after the start of the fall. This strange behaviour of the angle measurement results from a combination of two phenomena:

- The gyroscope saturation.
- The effect of the Kalman filter as implemented.

During the fall, the angular velocity exceeded the maximum velocity measurable by the gyroscope of 245 °/s [5]. So even when the system kept accelerating, the gyroscope kept reporting that the rotation speed was the speed at which it saturated.

The Kalman filter, which had a high degree of confidence in the gyroscope (due to its low noise), decided to continue considering its data. The Kalman filter therefore assumed that the body was falling at saturation speed, which indeed was lower than the actual speed. Kalman naturally underestimated the angle after saturation. This is why the 2 second spike corresponding to collision is at 50° and not 60° as observed in reality. Figure 10.8 shows a behaviour closer to reality. The data of the accelerometer doesn't make much sense during the fall, but once the robot is still, the accelerometer stabilises almost directly at the right angle.

Collision induces a rebound, during which the angular rate falls back below the saturation value of the gyroscope and can therefore be well interpreted by Kalman, although it is still screwed by believing it bounced at 50° instead of 60°. The filter estimates the bounce angle at 35° while in reality it is more like 45°. It is only at rest that the effect

of the accelerometer measurement starts to be taken into account, which explains the slow transition to 60°.

The complementary filter suffers exactly from the same problem. However, more advanced implementations of the Kalman filter can avoid this phenomenon by doing:

- Prediction enhancement: use of a complete physical model of the robot, not just an integration model, in order for the robot to take the control input into account so that it could know more accurately how the robot will behave.
- Saturation detection: when the gyroscope is saturated, the system should either exclude it directly from the data or dramatically increase its variance (in order to decrease the confidence) in the **R** matrix.
- Operating mode selection: it consists in changing the operating mode of the sensor to measure higher rates once the saturation speed has passed. Therefore, the associated variance values in the Kalman filter should be adapted.

10.2.5 Frontal impulse

The aim of this test is to introduce a disturbance similar to an impulse acting on the robot's body and to observe its behaviour according to the type of filter used. To make the test repeatable, a significant acceleration of 40 cm/s² was applied to the wheels during the interval [0.5; 0.8[as shown in Figure 10.9.

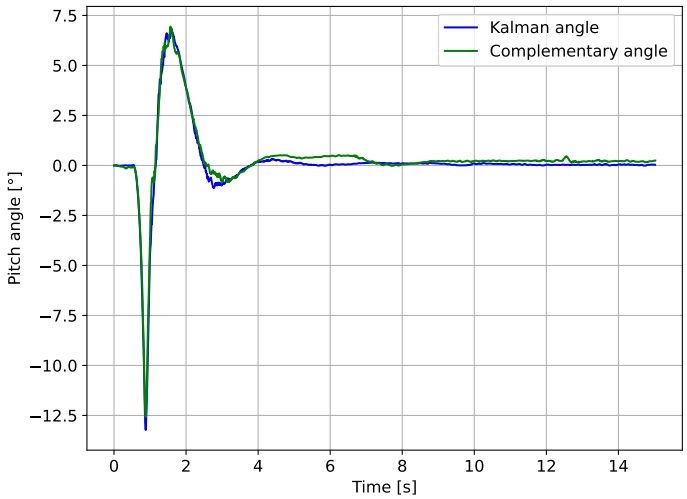


Figure 10.9: Evolution of the robot's angle when reacting to a frontal impulse.

To reject this perturbation, the system must face two effects:

- The angle of inclination, measured at around 13°. The gravitational force has more effect than at stability.
- The angular velocity: the system must counteract the rotational kinetic energy that the body contains.

The results are clear: the disturbance is well rejected and the system returns to its equilibrium position in a very short time of about 3 seconds. However, the system

reacts strongly after the disturbance, leading to an overshoot. This overshoot is quickly damped.

There is very little difference in the behaviour of the system depending on the choice of filter. The small visible differences are probably due more to environmental variability than to the effects of the filters. This means that the measurement bias pointed at the start of this chapter would have a very limited impact when facing large variations, as observed in this test where the measurements follow each other very closely.

10.2.6 Moment of inertia increase

This test aims to observe how the robot's behaviour is affected when its moment of inertia increases, possibly due to carrying a payload. A picture of the test setup can be found in Appendix H.

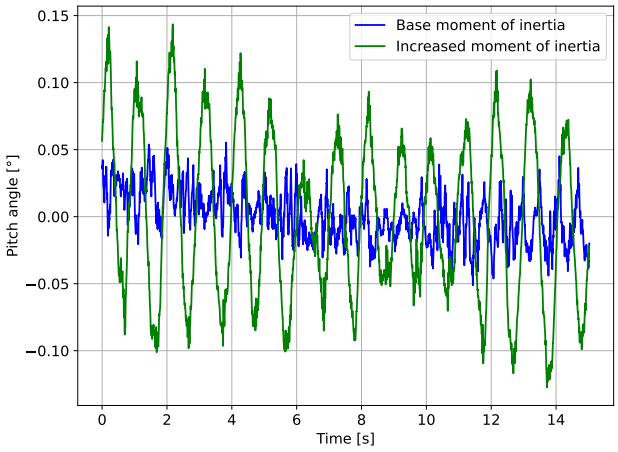


Figure 10.10: Graph showing the evolution of the pitch angle during dynamic balancing.

The experiment indicates that with an increased moment of inertia, the system's performance around its equilibrium point deteriorates: the oscillations become larger and slower compared to the initial state.

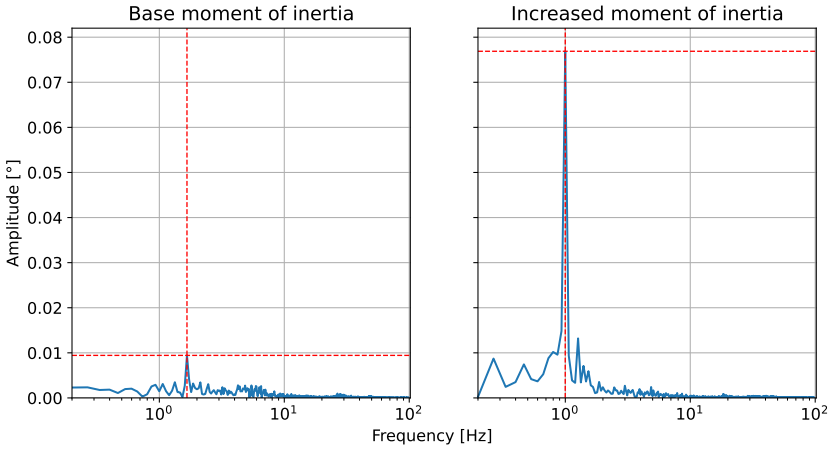


Figure 10.11: Fourier transform of the two signals in Figure 10.10.

Frequency analysis further confirms these findings, showing a significant increase in frequency content, particularly at low frequencies.

The main reason for these differences is that the increased inertia alters the time constants of the system. As the controller parameters remain unchanged between tests, the controller is less capable of adapting to this new scenario.

Another phenomenon observed during testing was the rigidity of the chassis. When tested without any load, the robot barely flexed when moving. But with some added weight to the top of the robot, flexion of the chassis during balancing is visible. This is due to a lack of stiffness of the chassis itself and bad fixation points. This probably contributed to the development of oscillations.

10.2.7 Slope

This test shows how the robot behaves when placed on an inclined surface. The inclined surface implies a displacement of the wheel-ground contact point, impacting the stabilization angle. It is possible to calculate the robot's stabilization angle based on knowledge of the center of gravity, wheel radius and slope angle.

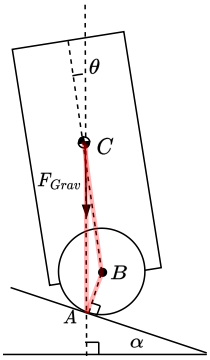


Figure 10.12: Self-balancing robot at equilibrium on a slope.

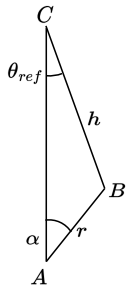


Figure 10.13: Triangle between contact point, wheel center and center of gravity to determine balancing angle.

Based on the triangle shown in Figure 10.13, the stabilisation angle θ_{ref} is calculated using the sine relationship:

$$r \cdot \sin(\alpha) = h \cdot \sin(\theta_{ref}) \Leftrightarrow \theta_{ref} = \arcsin\left(\frac{r\sin(\alpha)}{h}\right) \tag{10.1}$$

The experiment was carried out by positioning the robot on an inclined plane directly facing the slope. The plane was gradually tilted to establish several equilibrium points. The results are shown in Figure 10.14. It reflects the expected result which is linear over the tested slope range with a factor r/h at first order. The gap below 5 degree is probably related to higher adherence.

These results show and confirm the ability of the robot to adapt its position and movement to its environment.

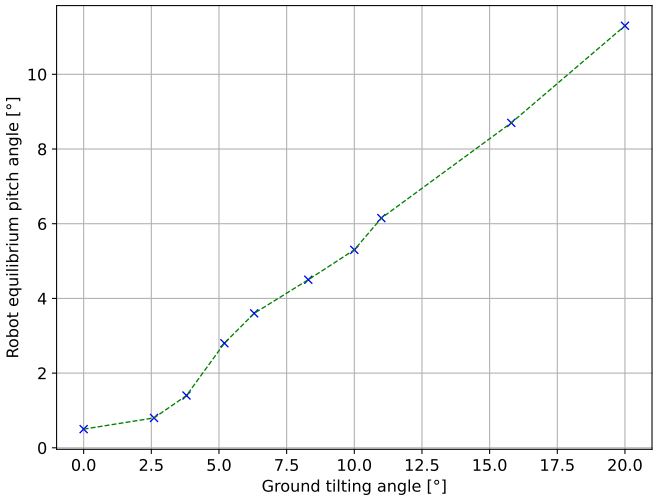


Figure 10.14: Graph showing the equilibrium angle of the robot on a slope.

10.2.8 Offset loading

This test shows how the robot reacts when an asymmetrical load with respect to the robot is added. The purpose of this test is to show that the robot can easily adapt its angle of balance to any type of load it has to carry. A picture of the test setup can be found in Appendix H.

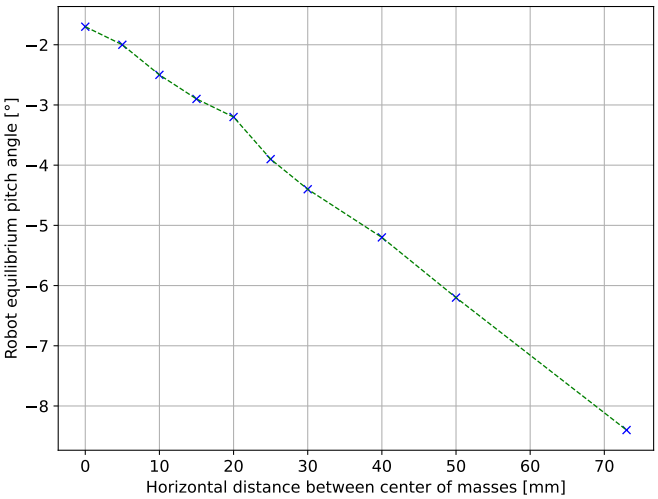


Figure 10.15: Graph showing the equilibrium angle of the robot with an offset mass.

The graph shows the value of the robot's equilibrium angle for a mass that is progressively shifted to the front of the robot. This graph confirms what was expected: the controller can adapt the equilibrium angle dynamically without any additional external actions required.

10.3 Limitations

The aim of this section is to study the limits of the system. There are many factors limiting the stability of the device. Their characterisation is split into two categories:

- Internal limitations refers to parameters that are directly part of the control loop. This includes: speed command, start angle, motor voltage and processing frequency.
- External limitations refers to a series of parameters that are part of the external environment in which the system evolves such as slope and noise.

10.3.1 Speed command

The aim of this test is to measure the maximum feed speed that the robot can achieve while remaining stable. In standard settings, the robot moves at a speed of 30 $^{cm}/_{s}$. It was realized on the basis of successive tests in which the control speed was progressively increased.

The maximum speed reached is 60 $^{cm}/_{s}$ and is due to an electromagnetic stall between the stepper's rotor and stator. Increasing the current sent to the motors would allow higher speeds to be reached but would cause the drivers to overheat.

However, from a speed of 40 $^{cm}/_{s}$ onwards, it is possible to put the robot into resonance by making it move forward and backward, leading to the fall of the device. This effect can be compensated for by adjusting the slope of the trapezoidal profile.

10.3.2 Start angle

The robot has been designed to lift up to 10° from the vertical with its lifting mechanism as explained in 3.1.3, in order to switch to self-balancing by itself. The aim of this test is to determine the start angle from which the robot is capable of getting into self-balancing.

After the test sequence, a maximum angle of 27° was found, beyond which the torque produced by the motors to provide the required acceleration was insufficient. This led to an electromagnetic stall of the rotor.

10.3.3 Motor voltage

Motor voltage allows current to flow through the motor windings. The selected voltage must be able to overcome two simultaneous phenomena:

- The back electromotive force (emf) linked to rotation speed.
- The joule effect due to winding resistance.

Voltage is easily adjusted on the buck converter by means of a potentiometer. The test was carried out by progressively lowering the voltage while the robot was in dynamic balancing. The robot held its balance without any problems until the end of the test. The test finished at 4.5 V as the voltage was not lowered below this value, corresponding to the lower limit of the stepper driver's operating range.

10.3.4 Frequency

The system's frequency is a critical constant that has been the subject of numerous optimizations detailed in Section 9. A low update frequency implies an unresponsive

system and therefore compromises dynamic balancing.

Initially, the control loop operated at 200 Hz. In order to conduct the tests, it had to be slowed down. An artificial delay was therefore implemented to act as a loop frequency controller. Of course, this function is only capable of adding delay and thus lowering the frequency.

Robot tests have shown that the robot remains very stable between 100 Hz and 200 Hz. Between 100 Hz and 75 Hz, the system is still stable but feels less reactive to the user when playing with it. At lower frequencies, from 75 Hz to 55 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 55 Hz, the robot is totally unstable and no longer able to maintain its equilibrium. The behaviour of the robot in standstill condition[1] at various frequencies, is shown in Figure 10.16.

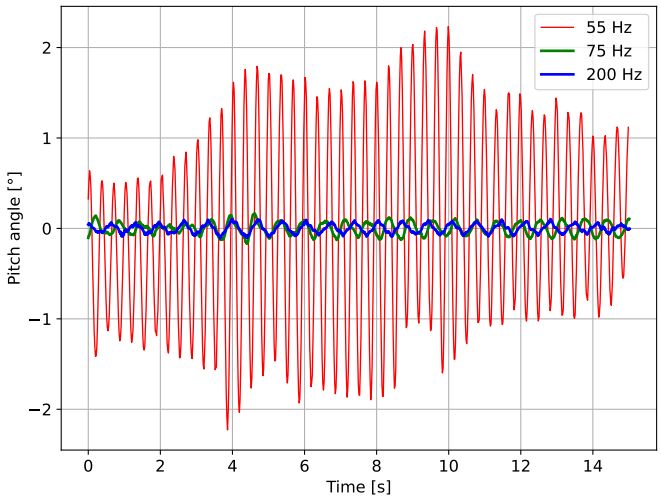


Figure 10.16: Graph showing the evolution of the angle of the robot at different update frequencies for the main loop.

10.3.5 Slope

This test is directly related to the test performed in Section 10.3.5, and is used to assess the angle of the maximum slope at which the robot can hold its balance.

At a gradient of 6°, the robot would start to roll in the direction of the gradient while maintaining its balance. After investigation, it was determined that this behaviour was due to the effect of the limiter on the speed controller's integral error. The slope was increased up to a limit of 20°, this time the cause being the limit of adhesion of the robot's wheels.

10.3.6 Sensor noise

This test consists of injecting noise into the sensors to see the limits of the filters used. The noise was added in a numerical way as Additive White Gaussian noise (AWGN) to

[1]These tests must have been performed on a table, which is not as stiff as the floor, so the observed behaviour may be slightly different.

ensure the same effect on all frequencies. The test was carried out in two phases: noise injection on the accelerometer only and then on the gyroscope only.

Figure 10.17 shows the filter output angle measurement based on artificially noisy accelerometer data. With an enormous AWGN of 10 g standard deviation, the robot is still able to keep its balance, although with greater difficulty. This results in a standard output angle variation of 1.703° for the Kalman filter and 3.159° for the angle coming from the complementary filter. The Kalman filter, even in its simplest implementation, outperforms the angle measured by the complementary filter. By increasing the variance of the accelerometer's AWGN to near infinite values, the robot was still in equilibrium with both filters. This is due to the fact that the Kalman filter understands that the accelerometer is not reliable so it relies only on the gyroscope. On the other hand, the complementary filter acts as a low-pass filter on the accelerometer, so it filters out all that rapidly varying noise, as explained in Section 1.7.

On the other hand, the AWGN on the gyroscope measurements has a much greater impact. Tests have shown that the noise limit is in the region of $\sigma_{gyro} = 32°/s$ for both filters. Beyond this value, the system moves from a marginally stable state to an unstable state. At this ultimate σ_{gyro}, the standard deviations of the filters are similar, with 2.217° for Kalman and 2.412° for the complementary, i.e. two orders of magnitude above the variations of the reference case as explained in Section 10.1. This test is shown in Figure 10.18.

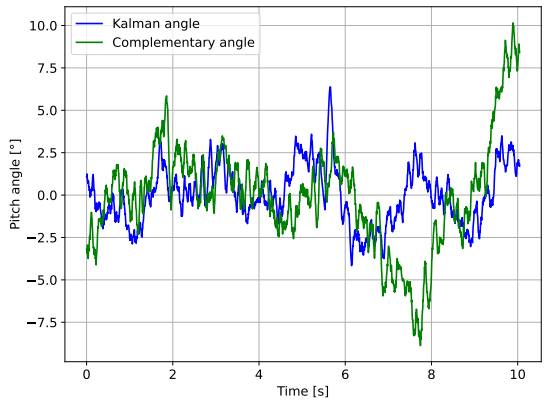


Figure 10.17: Measured angle as seen from the Kalman filter and the complementary filter. The additive noise on both accelerometer axes has a $\sigma^2_{acc} = 100$, i.e. a standard deviation of $\sigma_{acc} = 10g$.

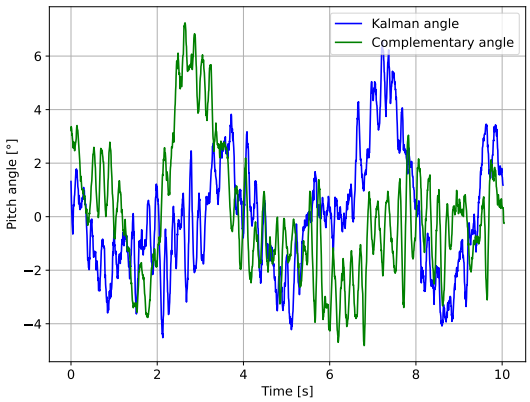


Figure 10.18: Measured angle as seen from the Kalman filter and the complementary filter. The additive noise on the gyroscope has a $\sigma^2_{gyro} = 1000$, i.e. a standard deviation of $\sigma_{gyro} = 31.62°/s$.

Chapter 11

Discussion

This chapter discusses the system's performance, with the aim of linking it to design choices and establishing comparisons with similar control systems developed by others. An identification of critical points of the design is also covered.

All the tests carried out in the previous chapter led to an analysis of the robot's behaviour under a wide range of conditions. In terms of performance, the robot is very stable and smooth. Its movements are barely noticeable to the naked eye when in dynamic balancing mode. The system also performs well in terms of command response and resilience to disturbance. The system's ability to find a new equilibrium point outperforms expectations and demonstrates the importance of having a control system to calculate the equilibrium point as explained in Chapter 5.

The results found in Chapter 4 show to importance of sensor fusion. This is also confirmed by the fact that most of the robots mentioned in Chapter 1 use sensor fusion to improve the measurements. Visually, the Kalman filter performs better than the complementary filter. This is confirmed by the results obtained in 10. The Kalman filter can still be improved much more, as explained in the future work section.

Compared with the educational robots tested in this master thesis, the LEGO robot [33] and the Pololu Balboa robot [28], the GRiSP robot, developed in this master thesis, has a **much smoother feel with less vibrations. It also has a much larger stability zone, capable of withstanding a wide range of disturbances. In terms of control, it is also faster and more responsive**. These differences can be explained by hardware disparities: such as motor power, the use or absence of encoders, natural frequency, computing power, etc. Another difference may be due to the software and architecture of filters and controllers. The device in this master thesis uses high-performance filters such as the Kalman filter. In the other robots, the filters used are more elementary, but for educational robots this makes sense, as these filters are much simpler to understand and modify than the Kalman filter. The major parameter impacting stability is the loop update frequency. The Pololu Balboa robot being a smaller robot and, therefore, having a higher natural frequency, should also have a higher control loop frequency. In reality, it runs at a frequency of 100 Hz, which is quite low. Comparing this to the 200 Hz obtained after optimization in this master thesis explains why the performance of the GRiSP robot are much better.

The GRiSP robot developed in this master thesis has a feature that most other robots do not: a lifting system. This system is very robust and can be partially adapted to the environment in which the robot is located. The Pololu Balboa robot also has a way of getting up that is simpler than the GRiSP robot, but it is not as robust. Its means of getting up is simply to make a very quick back and forth movement when the robot is in a resting position. The advantage of the GRiSP robot regarding this feature, is that its lifting system adapts to inclined planes unlike the Pololu Balboa robot.

In addition to the operating limitations documented in Section 10.3, there are a few known problems with the robot, which can lead to undesirable behaviour or reduced performance. These can create a certain amount of uncertainty when using the robot and would require further work. such problems are:

- Lack of supervisors: most parts of the code, except for Hera, are not covered by supervisors. This lack can lead to the following problems:
  - Pmod NAV crashes: this problem occurs approximately one time out of ten when the robot is launched. It completely shuts down the program and requires the robot to be restarted manually.
  - Process crashes: only the **hera_interface** process is supervised through Hera. Every other process, all described in Chapter 8, is not supervised and could cause the whole robot to stop working.
  - ESP32 crashes: the ESP32 could also crash which would imply an error in the robot due to I²C. This problem could be solved in GRiSP by detecting that the I²C has an error and sending a signal to the restet pin of the ESP32. This would avoid having to manually restart the robot.
- Unsecured LoRa communication. At the moment, any other device could send data via LoRa. This means that anyone could take control of the robot. Security was of course not the scope of the work and a specific security analysis should be considered to address it.
- Performance drops: instantaneous drops in performance are observed at fairly regular and frequent intervals. These can lead the robot to fall if these spikes appear when the robot is operating at lower frequencies.
- Motors speed limitation: as explained in Section 10.3, if the motors are requested to turn to a speed that is too high, the motors stall and stop turning.
- Pmod NAV positioning: the Pmod NAV is rather loose in the Pmod connector. This leads to a variable DC-bias added to the angle. This bias changes at each collision of the robot. This problem could be solved by calibrating the DC-bias each time the robot is down.

Conclusion and future work

The initial objectives of this project, detailed in the introduction and Section 2.1.2 was to develop a stability engine running on the GRiSP and that can be easily implemented on different devices. This has been achieved through the following specific sub-objectives which have been fully documented in this document.

- Creating of a two-wheeled self-balancing robot. Just like systems running under Erlang, the device is able to recover from a crash without human intervention.
- Successful use of Hera as an interface on the whole system for IoT purposes.
- Integration of a Kalman filter in the control algorithm to make sensor fusion multiple sensors for noise reduction.
- Characterisation of the performance of the GRiSP board and Hera in a real-time application

In addition to these initial objectives, other subsidiary objectives were identified and achieved as they contributed to the main objectives, improved performance or enabled the work to be reused. These are summarised below:

- The design of a two-wheeled robot with a lifting system to to be 100% autonomous
- The design and implementation of a stability engine to balance the device.
- Implementation of a multilevel system with an executive loop to monitor the stability loop.
- Use and comparison of 2 sensor fusion strategies: Kalman filter and complementary filter
- The implementation of a stepper motor driver for ESP32.
- The implementation of a half-precision float for faster I^2C communications.
- Remote control for robot displacement and emergency stop system via LoRa communication.
- A wide range of tests to establish the system's behaviour and limits.

The process followed is of course more complex than the main design, implementation and characterisation steps reported here. Before the robot was able too keep its balance, a number of problems prevented stability. One major problem, linked to Pmod NAV, was capping the robot's frequency to a value of 60 Hz, which is too low for robust stability. In the quest for robot stability, many other problems were corrected and

optimised with the aim of achieving stability. It was only when the Pmod NAV problem was fully identified and corrected that the robot held its balance robustly. Fixing all the other problems beforehand meant that once the last problem had been solved, the robot was immediately very robust, because all the other sources of slowdown had already been corrected.

All these optimisations and developments, in addition to the basic objectives, have resulted in a very complete and finished project. **The robot's performance exceeded all expectations. The result was a very stable system, resistant to disturbance, responding easily to commands and totally autonomous.**

One of the most important features of this project is its modifiability and modularity. This means the project has strong foundations for being extended in many different ways. From its improvement to its deployment in industry or education, many improvements can be considered to give the robot more purposes. Here is an non exhaustive list of such work:

- **Include other topics of mobile robotics**: Depending on the application in which the robot is used, some sensors like a temperature sensor or an humidity sensor could be added. Some applications might require the robot to interact with its surroundings. To do that, the robot could have an end effector like a gripper or a piston to move things around.

  Adding new sensors like a GPS sensor or an ultra wideband sensor can make it possible to have new features like position control of the robot. Having a good estimation of the position of the robot allows to make the robot follow a trajectory. Adding proximity or distance sensors like a LiDAR sensor or an IR sensor allow to implement obstacle avoidance by having a vision of the surroundings of the robot. To avoid the obstacles, different algorithms exist to find a clear trajectory like potential field and A-star.

  For better performances of the robot, multiple sensors can be added and integrated to the sensor fusion in the Kalman filter. These sensors could be placed in different spots on the robot to have data of the whole robot. Another improvement on the Kalman filter is to make a digital twin of the robot for the physical model, as explained in Section 4.2. This would enable the Kalman filter to have an accurate estimation of the state of the robot in any position.

- **Multiplexer for the SPI port**: One idea that has been explored but not yet fully implemented is to be able to multiplex the SPI port with 12 pins on the GRiSP so that three different Pmods can be connected. A circuitboard has been designed for this but has not yet been tested. The modifications to the code have not yet been made either. This addition to the GRiSP may be useful if several Pmods using SPI are to be used with a single GRiSP board. For the moment, the solution would be to use a GRiSP card for each additional sensor, which would incur unnecessary costs and delays.

- **Reduce the amount of hardware**: Currently different circuit boards are used in the robot. The only purpose of having the ESP32 is for its frequency to control the motors. This increases the complexity of the robot and makes it less robust against crashes. A good improvement would be to have all the functionality on

only one circuit board, the GRiSP. To achieve this, a Pmod driver for the stepper motors must be used to interface the motors directly with the GRiSP board.

- **Include ROSiE to the software**: As explained in Chapter 1, there are many frameworks for robotics other than Hera. One of these frameworks could be adapted to the robot. It is the ROSiE framework. Based on the existing ROS framework, ROSiE could provide a range of features that Hera does not have.
- **Communicating robots**: With the robot's IoT and communication capabilities, a network of these robots can be created, all communicating with each other. These robots could then perform a common task by all working together, including considering resilience issues to compensate robots failing to achieve specific tasks due to internal (breakdown) or external causes like obstacle or deliberate attacks.

Bibliography

[1] D. P. P. S. GmbH, *GRiSP*. [Online]. Available: `https://www.grisp.org/` (visited on 08/18/2024).

[2] D. P. P. S. GmbH, *Stritzinger.com — Home page*. [Online]. Available: `https://www.stritzinger.com/` (visited on 08/18/2024).

[3] Digilent Inc., *Digilent Pmod™ Interface Specification*, Oct. 2020. [Online]. Available: `https://digilent.com/reference/pmod/start`.

[4] RTEMS Project, *RTEMS Real-Time Operating System*, `https://www.rtems.org/`, Accessed: 2024-08-18, 2024.

[5] Digilent Inc., *Pmod nav*, Accessed: 2024-08-17, 2024. [Online]. Available: `https://digilent.com/reference/pmod/pmodnav/start`.

[6] J. Armstrong, "Erlang—a survey of the language and its industrial applications," in *Proc. INAP*, vol. 96, 1996, pp. 16–18.

[7] N. Guillaume and B. Julien, "Sensor fusion at the extreme edge of an internet of things network," Ecole polytechnique de Louvain, Université catholique de Louvain, 2020, Van Roy, Peter, Prom.

[8] K. Sébastien and V. Vincent, "The hera framework for fault-tolerant sensor fusion on an internet of things network with application to inertial navigation and tracking," Ecole polytechnique de Louvain, Université catholique de Louvain, 2021, Van Roy, Peter, Prom.

[9] L. Tanguy, "Numerl : Efficient vector and matrix computation for erlang," Ecole polytechnique de Louvain, Université catholique de Louvain, 2022, Van Roy, Peter, Prom.

[10] N. Lucas, "Low-cost high-speed sensor fusion with grisp and hera," Ecole polytechnique de Louvain, Université catholique de Louvain, 2023, Van Roy, Peter, Prom.

[11] Thymio Team, *Thymio robot official website*, Accessed: 2024-08-17, 2024. [Online]. Available: `https://www.thymio.org/`.

[12] M. Quigley, K. Conley, B. Gerkey, *et al.*, "Ros: An open-source robot operating system," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Accessed: 2024-08-17, vol. 3, IEEE, 2009, pp. 5–10. [Online]. Available: `https://www.ros.org/`.

[13] R. project, `https://github.com/rosie-project`, 2022.

[14] Orocos Project, *Orocos: Open robot control software*, Accessed: 2024-08-17, 2024. [Online]. Available: `https://orocos.org/`.

[15] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Rt-middleware: Distributed component middleware for rt (robot technology)," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 3933–3938. DOI: 10.1109/IROS.2005.1545521.

[16] Robot Framework Foundation, *Robot framework*, Accessed: 2024-08-17, 2024. [Online]. Available: https://robotframework.org/.

[17] MoveIt Development Team, *Moveit: The motion planning framework for ros*, Accessed: 2024-08-17, 2024. [Online]. Available: http://moveit.ros.org/.

[18] S. Inc., *About us - segway official store*, https://store.segway.com/about-us, Accessed: 2024-08-14, 2023.

[19] MeetiBOT, *Meet ibot - your personal mobility device*, https://meetibot.com/, Accessed: 2024-08-14, 2023.

[20] C. Robotics, *Chronus robotics - kim-e, the self-balancing personal mobility robot*, https://chronusrobotics.com/, Accessed: 2024-08-14, 2024.

[21] F. I. for Material Flow and L. (IML), *Evobot - the evolution of autonomous mobile robotic systems*, https://www.iml.fraunhofer.de/en/fields_of_activity/material-flow-systems/iot-and-embedded-systems/evobot.html, Accessed: 2024-08-14, 2024.

[22] RobotsGuide, *Handle robot*, Accessed: 2024-08-17, 2024. [Online]. Available: https://robotsguide.com/robots/handle.

[23] O. M. Mohamed Gad, S. Z. M. Saleh, M. A. Bulbul, and S. Khadraoui, "Design and control of two wheeled self balancing robot (twsbr)," in *2022 Advances in Science and Engineering Technology International Conferences (ASET)*, 2022, pp. 1–6. DOI: 10.1109/ASET53988.2022.9735004.

[24] W. Vega, J. D. Garcia, K. Mollan, *et al.*, "Conceptual mechatronics design and prototyping of autonomous inverted pendulum-system applied on two-wheeled mobile robot," in *2023 Third International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, 2023, pp. 1–6. DOI: 10.1109/ICAECT57570.2023.10118071.

[25] S. Kim and K. Yeom, "Development of a hand-fan-shaped arm and a model predictive controller for leg crossing, walking, and one-legged balancing of a wheeled-bipedal jumping robot," *Machines*, vol. 12, no. 5, 2024, ISSN: 2075-1702. DOI: 10.3390/machines12050284. [Online]. Available: https://www.mdpi.com/2075-1702/12/5/284.

[26] C.-C. Tsai, W.-T. Hsu, F.-C. Tai, and S.-C. Chen, "Adaptive motion control of a terrain-adaptive self-balancing leg-wheeled mobile robot over rough terrain," in *2022 International Automatic Control Conference (CACS)*, 2022, pp. 1–6. DOI: 10.1109/CACS55319.2022.9969857.

[27] N. Hasanah *et al.*, "Adaptive motion control of two-wheeled robot based on complementarity filter and fuzzy logic controller," *International Journal of Robotics and Automation*, vol. 9, no. 1, pp. 53–60, 2020. DOI: 10.11591/ijres.v9.i1.pp53-60. [Online]. Available: https://ijres.iaescore.com/index.php/IJRES/article/view/20676/pdf.

[28] P. Corporation, *Balboa 32u4 balancing robot*, https://www.pololu.com/product/3575, Accessed: 2024-08-14, 2024.

[29] M. Han, K. Kim, D. Y. Kim, and J. Lee, "Implementation of unicycle segway using unscented kalman filter in lqr control," in *2013 10th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, 2013, pp. 695–698. DOI: 10.1109/URAI.2013.6677427.

[30] A. Petrovsky, I. Kalinov, P. Karpyshev, D. Tsetserukou, A. Ivanov, and A. Golkar, "The two-wheeled robotic swarm concept for mars exploration," *Acta Astronautica*, vol. 194, pp. 1–8, 2022, ISSN: 0094-5765. DOI: https://doi.org/10.1016/j.actaastro.2022.01.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0094576522000340.

[31] STMicroelectronics, *Hardware developer kit*, Accessed: 2024-08-17, 2016. [Online]. Available: https://www.mikrocontroller.net/attachment/338591/hardware_developer_kit.pdf.

[32] LEGO Education, *Ev3 program description: Gyroboy*, Accessed: 2024-08-17, 2015. [Online]. Available: https://assets.education.lego.com/v3/assets/blt293eea581807678a/blt6442d49d724f3570/5f8803d01c5db60f7d0ae38e/ev3-program-description-gyroboy.pdf?locale=en-us.

[33] T. H. Hughes, G. H. Willetts, and J. A. Kryczka, "Lqg controller for the lego mindstorms ev3 gyroboy segway robot," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 17 282–17 287, 2020, 21st IFAC World Congress, ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.1811. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896320324216.

[34] G. A. Muñoz-Hernandez, J. Díaz-Téllez, J. Estevez-Carreon, and R. S. García-Ramírez, "Adrc attitude controller based on ros for a two-wheeled self-balancing mobile robot," *IEEE Access*, vol. 11, pp. 94 636–94 646, 2023. DOI: 10.1109/ACCESS.2023.3308948.

[35] R. Weber, *Optimization and control*, 2010.

[36] M. N. Öz, S. Budak, E. Kurnaz, and A. Durdu, "Orientation Determination in IMU Sensor with Complementary Filter," *Turkish Journal of Forecasting*, vol. 06, no. 1, pp. 34–39, Aug. 2022, ISSN: 2618-6594. DOI: 10.34110/forecasting.1126184. [Online]. Available: http://dergipark.org.tr/en/doi/10.34110/forecasting.1126184 (visited on 08/18/2024).

[37] M. Salwa and I. Krzysztofik, "Application of filters to improve flight stability of rotary unmanned aerial objects," *Sensors*, vol. 22, p. 1677, Feb. 2022. DOI: 10.3390/s22041677.

[38] S. Kalbusch, V. Verpoten, and P. Van Roy, "The Hera framework for fault-tolerant sensor fusion with Erlang and GRiSP on an IoT network," en, in *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang*, Virtual Republic of Korea: ACM, Aug. 2021, pp. 15–27, ISBN: 9781450386128. DOI: 10.1145/3471871.3472962. [Online]. Available: https://dl.acm.org/doi/10.1145/3471871.3472962 (visited on 08/13/2024).

[39] R. Renaud, *Lecture notes in robot modelling and control lelme2732*, 2022.

[40] Wikipedia, *Filtre de kalman*, https://fr.wikipedia.org/w/index.php?title=Filtre_de_Kalman&oldid=216332692, [accessed January-2024], 2024.

[41] Wikipedia, *Low-pass filter — Wikipedia, the free encyclopedia*, [accessed May-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Low-pass_filter&oldid=1234242174.

[42] A. Arhutich, *Balancingwii*, https://github.com/mahowik/BalancingWii, 2020.

[43] B. Benavides. "'erlang behaviors'. erlang battleground." (), [Online]. Available: https://medium.com/erlang-battleground/erlang-behaviors-4348e89351ff. [accessed 14-08-2024].

[44] Espressif, *Esp32 pico series datasheet*, 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-pico_series_datasheet_en.pdfs.

[45] Arduino contributors, *Spi.h library*, [accessed November-2023], 2023. [Online]. Available: https://www.arduino.cc/reference/en/language/functions/communication/spi/.

[46] S. Mistry, *Wire.h library*, [accessed November-2023], 2023. [Online]. Available: https://github.com/sandeepmistry/arduino-LoRa.

[47] Arduino contributors, *Wire.h library*, [accessed November-2023], 2023. [Online]. Available: https://www.arduino.cc/reference/en/language/functions/communication/wire/.

[48] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[49] Wikipedia contributors, *Half-precision floating-point format — Wikipedia, the free encyclopedia*, [Accessed : 11-05-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Half-precision_floating-point_format&oldid=1235367421.

[50] J.-C. Samin, G. Campion, and P. Maes, *FSAB 1202 Exercices de Mécanique*, Feb. 2008.

Part IV

Appendices

Appendix A

Developed device

The following figures provide an overview of what the robot looks like.

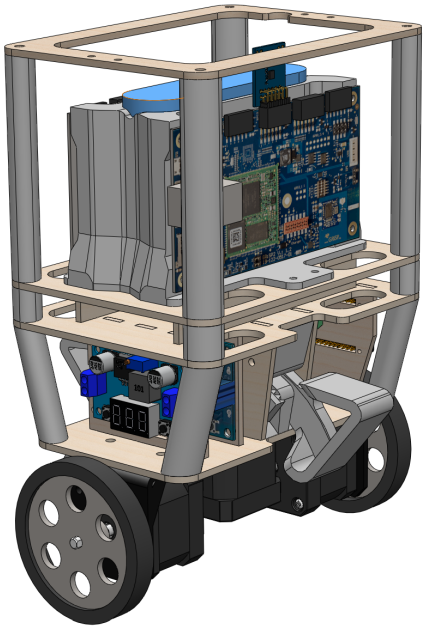


Figure A.1: Screenshot of the complete SolidWorks model of the robot.

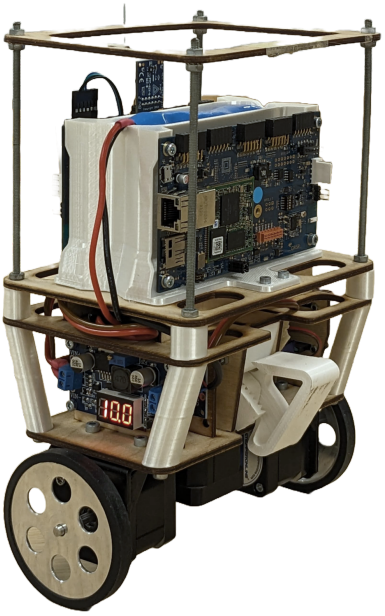


Figure A.2: Picture of the robot while in equilibrium.

Appendix B

Technical specifications

A set of criteria based on use, manufacture and safety are taken into account to define the technical specifications as shown on TableB.1.

Functions	
F1	Device to test a dynamic balancing algorithm using Hera on GRiSP
F2	Usage for further research in IoT with GRiSP
Functional Requirements	
FR1.1	Mechanically unstable device
FR1.2	Use of actuation to stabilize the device
FR1.3	Self-rising/recovery from fall
FR1.4	Autonomous : no need for external supply
Constraints	
C1	Usage of GRiSP 2 board
C2	Easy to prototype and replicate
C3	Safe to use
Constraints Requirements	
CR1.1	Supply of 5V for GRiSP
CR1.2	Limited type and number of communication bus
CR2.1	Usage of “on the shelf” pieces, 3D printing, Laser cutting and Custom PCB
CR3.1	Emergency stop
CR3.2	Voltages lower than 20V
CR3.3	Non-flammable batteries
CR3.4	Weigh less than 5Kg

Table B.1: Specifications of the self-balancing robot, F : Function , FR : Functional Requirement, C : Constraint, CR : Constraint Requirement.

Appendix C

PCBs

Two PCBs have been created:

- Stepper driver interface: enables all the stepper drivers to be mounted on the same board, the power circuit to be routed and the connections for the controller to be brought together.
- PMOD NAV-SPI interface: allows the IMUs of 3 Pmod NAVs to be used on the same Pmod SPI extended port, sacrificing certain other Pmod NAV functions.

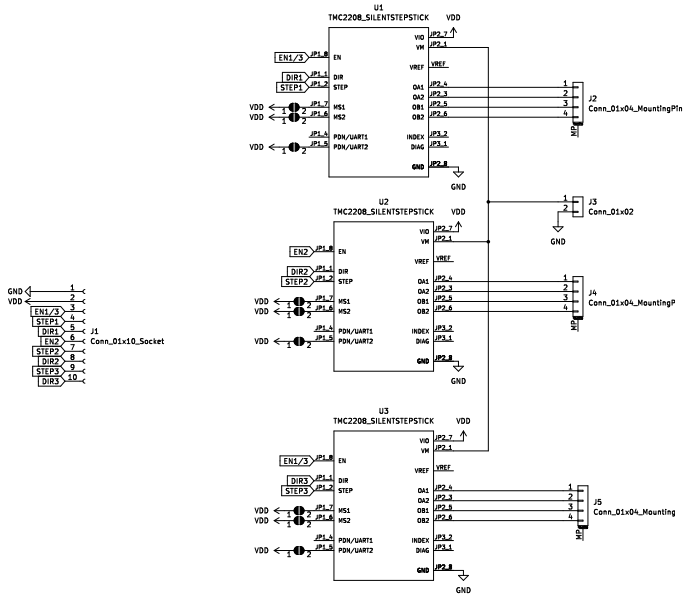


Figure C.1: Electrical diagram of Stepper driver interface with Kicad.

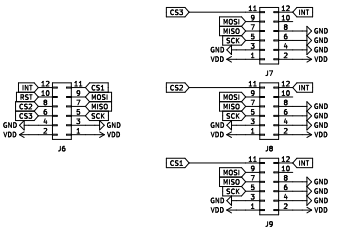


Figure C.2: Electrical diagram of PMOD NAV-SPI interface with Kicad.

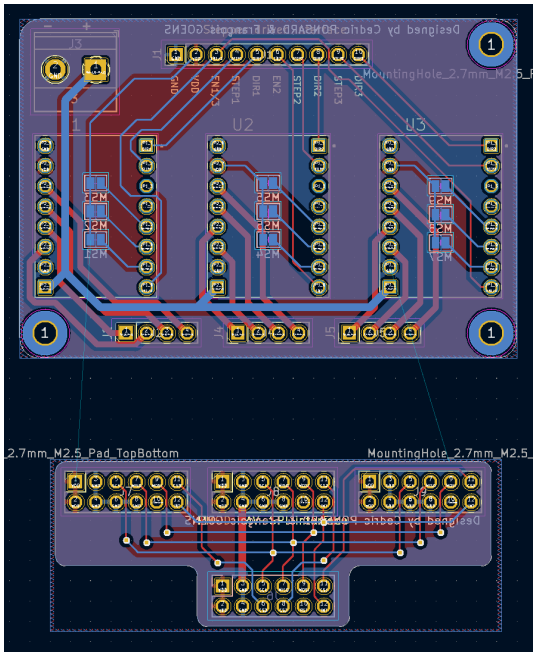


Figure C.3: Routing PCBs in Kicad.

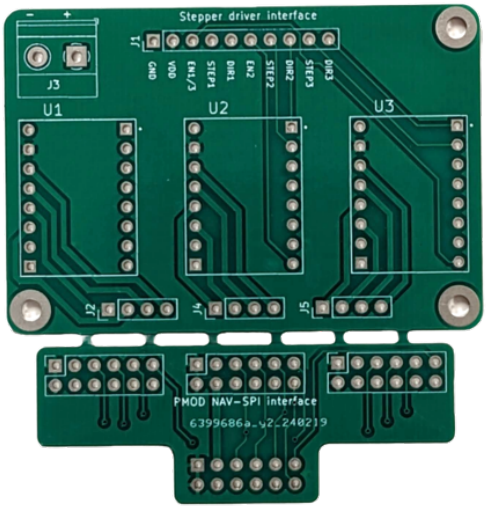


Figure C.4: PCBs manufactured by JLC PCB.

Appendix D

Physical model development

This appendix details all the calculations providing a physical model of the robot. The Newton-Euler method has been chosen for this purpose. This development was inspired by the one developed in [50, p. 60]

This model use all the reference presented in the main text and used in Figure D.1

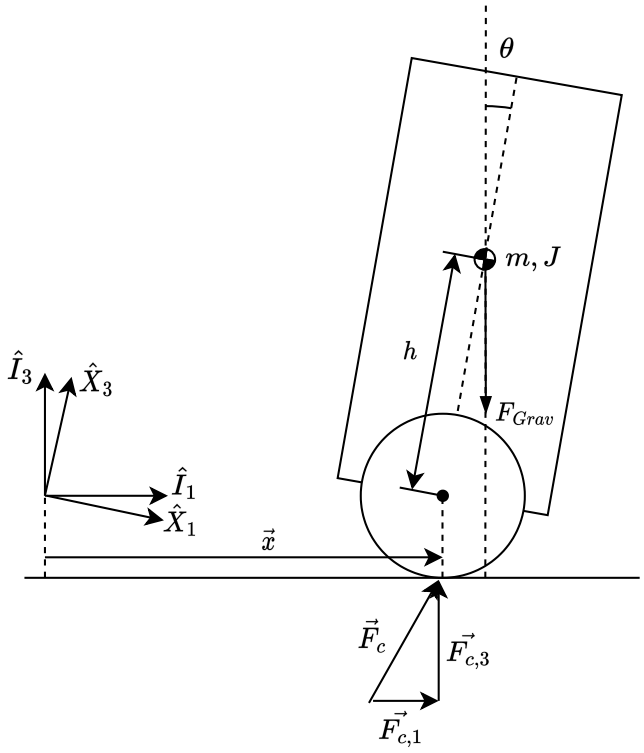


Figure D.1: Reference scheme of the bodies and the different quantities for developing the physical model of the system

To apply the Newton-Euler equations, it is required to compute different quantities such as the position vector, the angular momentum, the external forces and the moment of the forces applied to the robot.

1. Expression of the CG-position vector $\vec{R}$ and it's derivatives. The CG-position vector is the result of the addition of two other vectors:

- $x\hat{I}_1$ the position of the wheels of the robot relative to the $\hat{I}$ reference frame
- $h\hat{X}_3$ the position of CG of the robot relative to the center of the wheels thus relative belong to $\hat{X}_3$

Thus the position vector is written as:

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

The derivatives $\dot{\vec{R}}$ and $\ddot{\vec{R}}$ are found from the vector derivative of a mobile basis formula: $\dot{\vec{u}} = \mathring{\vec{u}} + \vec{\omega} \times \vec{u}$

$$\begin{aligned} \dot{\vec{R}} &= \dot{x}\hat{I}_1 + h\dot{\theta}\hat{X}_1 \\ \ddot{\vec{R}} &= \ddot{x}\hat{I}_1 + h\ddot{\theta}\hat{X}_1 + h\dot{\theta}^2\hat{X}_3 \end{aligned} \tag{D.1}$$

2. Expression of the angular moment relative to the CG $\vec{H^G}$ of the robot and it's derivative $\dot{\vec{H}}^G$:

$$\begin{aligned} \vec{H^G} &= J\dot{\theta}\hat{X}_2 \\ \dot{\vec{H}}^G &= J\ddot{\theta}\hat{X}_2 \end{aligned} \tag{D.2}$$

3. External forces calculation:
   - The weight of the robot : $m\vec{g} = -mg\hat{I}_3$
   - The contact force $\vec{F_c}$ in the $\hat{I}$ reference frame :$\vec{F_r} = -F_{c,1}\hat{I}_1 - F_{c,3}\hat{I}_3$

   The total force applied to the robot is the sum of the weight and the contact force:

$$\begin{aligned} \vec{F} &= m\vec{g} + \vec{F_r} \\ &= -mg\hat{I}_3 - F_{c,1}\hat{I}_1 - F_{c,3}\hat{I}_3 \end{aligned} \tag{D.3}$$

4. Expression of the moment of the forces applied to the robot CG:

$$\begin{aligned} \vec{L^G} &= -h\hat{X}_3 \times \vec{F_r} \\ &= h(F_{c,1}\cos(\theta) - F_{c,3}\sin(\theta))\hat{X}_2 \end{aligned} \tag{D.4}$$

Those equations are injected in the Newton-Euler formulas to find the movement equations of the robot.

1. Newton : $m\ddot{\vec{R}} = \vec{F}$ provide the following equations (after projection on the $\hat{I}_1$ and $\hat{I}_3$ axis) :

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

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

2. Euler : $\dot{\vec{H}}^G = \vec{L}^G$ provide the following equation :

$$\begin{aligned} J\ddot{\theta} &= h(F_{c,1}\cos(\theta) - F_{c,3}\sin(\theta)) \\ \Leftrightarrow \frac{J}{h}\ddot{\theta} &= F_{c,1}\cos(\theta) - F_{c,3}\sin(\theta) \end{aligned} \tag{D.7}$$

By multiplying D.5 by $cos(\theta)$, D.6 by $sin(\theta)$ and summing them with D.7, $F_{c,1}$ and $F_{c,3}$ cancel out and the following equation is obtained :

$$\begin{aligned} m\ddot{x}\cos(\theta) + (mh\ddot{\theta} + \frac{J}{h}\ddot{\theta}) &= mg\sin(\theta) \\ \Leftrightarrow \ddot{x}\cos(\theta) + (h + \frac{J}{mh})\ddot{\theta} &= g\sin(\theta) \end{aligned} \tag{D.8}$$

Appendix E

Software flags

In the following tabular, Table E.1, the interpretation for the input flags, received from the ESP32, is explained for both true and false values.

Flag	Value of 1	Value of 0
Arm_Ready	The arm of the robot is either fully extend or fully retracted	The arm of the robot is in a transition phase
Switch	The angle of the Kalman filter must be used for the stability	The complementary filter must be used instead
Test	Start of a logging sequence	Otherwise
Get_Up	The robot must get up	The robot must get down
Forward	The robot must go forward	No command to go forward
Backward	The robot must go backward	No command to go backwards
Left	The robot must turn left	No command to turn left
Right	The robot must turn right	No command to turn right

Table E.1: Flags description.

A synthesis of all of the state flags and the output flags for the FSM is given in Table E.2. The state flags are represented in an array as [Get_Up, Arm_Ready, Robot_Up] and the output flags are represented in an array as [Power, Freeze, Extend, Robot_Up_Bit]. The flag F_B is not represented because it could have any value in any state.

State	State flags	Output flags
Rest	[0, 1, 0]	[0, 0, 0, 0]
Raising	[1, 0, 0]	[1, 0, 1, 0]
Stand up	[1, 1, 1]	[1, 0, 0, 1]
Prepare arms	[0, 0, 1]	[1, 0, 1, 1]
Free fall	[0, 1, 1]	[1, 1, 1, 1]
Soft fall	[0, 0, 0]	[1, 0, 0, 0]

Table E.2: State flags and output flags in each state.

Appendix F

Maximal delay computation

A simulation has been done to characterise the delay that can be acceptable between the measurement by the sensors and the actuation. The resulting graph, shown in Figure F.2, displays the behaviour of the simulated angle of the robot for different delays. The angle enter a marginal stability around 20 ms and 30 ms of delay. To be safe the upper limit of 20 ms of delay can be imposed to have a small safety factor.

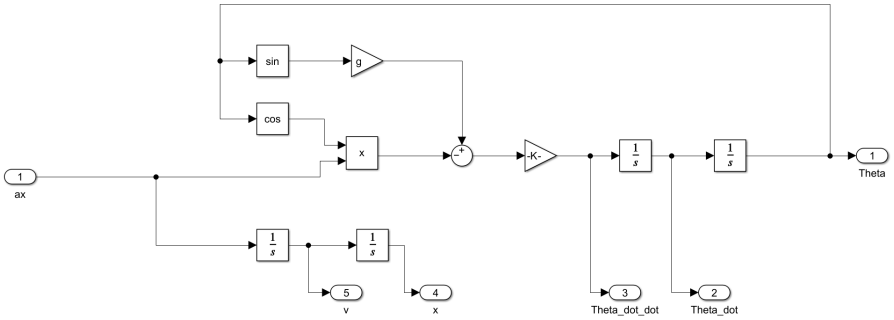


Figure F.1: Simulink model for simulating delays in the system.

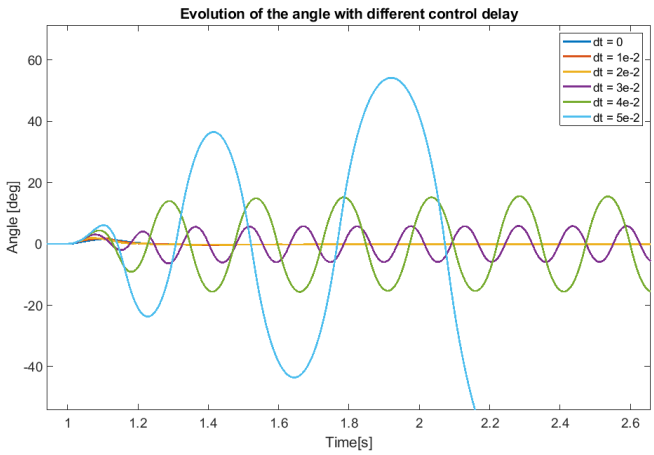


Figure F.2: Graph obtained with simulation in Figure F.1.

Appendix G

GRiSP GPIO Frequency

This appendix presents the generation of high-frequency signals at the GRISP GPIO outputs. This signal is generated independently on its own process, with a fixed frequency of 1 kHz. Figure G.1 shows the test setup with which figure G.2 was taken. The signal tap shows that there are various interruptions in the generation of the signal, making it unusable for stepper control. It's a signal that's reputed to be periodic but that isn't.

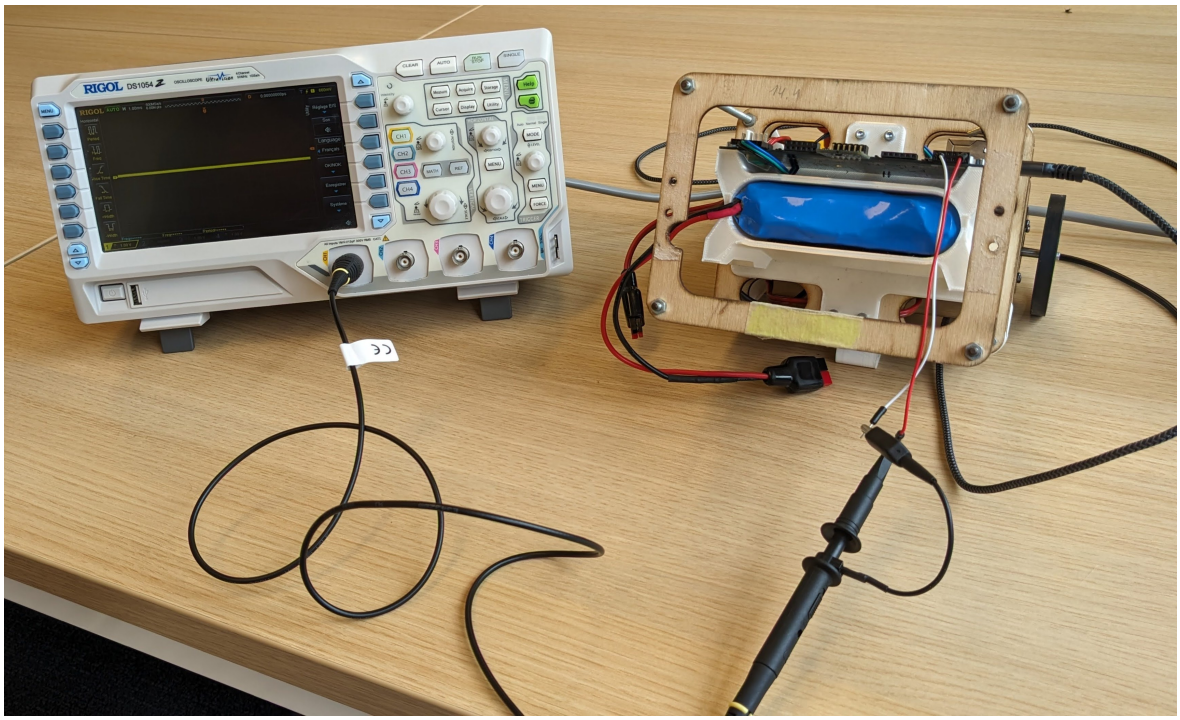


Figure G.1: GRiSP GPIO frequency measurement setup, the oscilloscope taking the measurements is on the left and the GRiSP fixed to the robot is on the right.

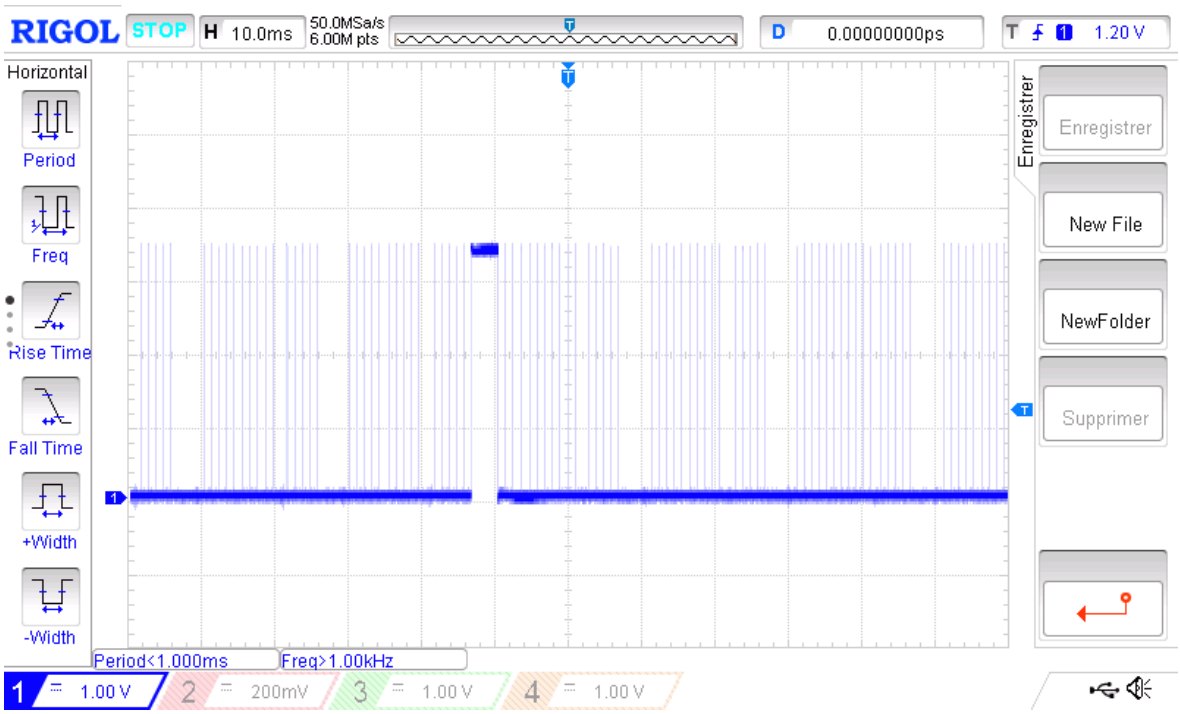


Figure G.2: GPIO tap signal coming from the GRiSP, with interruptions during several periods occurring. The image colours have been inverted to improve visibility.

Appendix H

Experimentation setup

The three following pictures are pictures of the test setup of respectively the mass offset, the slope and the inertia increase tests.

Figure H.1: Picture of the robot during the offset mass test.

Figure H.2: Picture of the robot during the slope test.

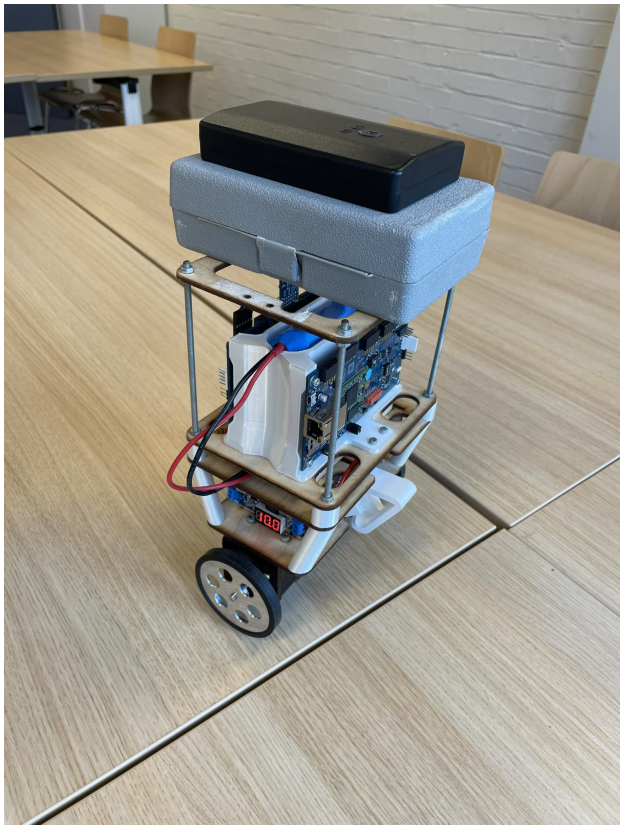


Figure H.3: Picture of the robot during the increased inertia test.

Appendix I

Code

This appendix contains all the code needed to operate the system, in four main sections:

- GRiSP code
- Lilygo LoRa32 code on the robot side
- Lilygo LoRa32 code on the emergency stop side
- Python user interface code

The latest version of the code can be found on our two GitHub repositories:

– Modified Hera: `https://github.com/FrancoisGgg/hera`

– Robot: `https://github.com/FrancoisGgg/balancing_robot`

I.1 GRiSP code

I.1.1 balancing_robot app

```
{application, balancing_robot, [
    {description, "GRiSP application for dynamic balancing of a moving robot"},
    {vsn, "0.1.0"},
    {registered, []},
    {mod, {balancing_robot, []}},
    {applications, [
        kernel,
        stdlib,
        grisp
    ]},
    {env,[]},
    {modules, []},

    {licenses, ["Apache 2.0"]},
    {links, []}
]}.
```

I.1.2 balancing_robot supervisor

```
% @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}, []} }.
```

I.1.3 balancing_robot

```
-module(balancing_robot).

-behavior(application).

-export([start_robot/0]).
-export([start/2, stop/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% API
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start_robot() ->
    timer:sleep(5000), %Waiting for setup of GRiSP application before launching the robot
    hera:start_measure(hera_interface, []),
    ok.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Callbacks
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start(_Type, _Args) ->
    {ok, Supervisor} = balancing_robot_sup:start_link(),
    grisp_led:color(1, {1, 0, 1}),
    grisp_led:color(2, {1, 0, 1}),
  _ = grisp:add_device(spi2, pmod_nav),
    pmod_nav:config(acc, #{odr_g => {hz,238}}),
    numerl:init(),
    Pid = spawn(balancing_robot, start_robot, []),
    {ok, Supervisor}.

stop(_State) -> ok.
```

I.1.4 main_loop

```
-module(main_loop).

-export([robot_init/1, modify_frequency/1]).

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

%Advance constant
-define(ADV_V_MAX, 30.0).

%Turning constant
-define(TURN_V_MAX, 80.0).

%Angle at which robot is considered "down"
-define(MAX_ANGLE, 25.0).

```

```
%Coefficient for the complementary filter
-define(COEF_FILTER, 0.667).

%Duration of a logging sequence
-define(LOG_DURATION, 15000).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Robot
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

robot_init(Hera_pid) ->

    process_flag(priority, max),

    %Starting timestamp
    T0 = erlang:system_time()/1.0e6,

    %Table for global variables
    ets:new(variables, [set, public, named_table]),
    ets:insert(variables, {"Freq_Goal", 300.0}),

    %Calibration
    io:format("[Robot] Calibrating... Do not move the pmod_nav!~n"),
    grisp_led:color(1, {1, 0, 0}),
    grisp_led:color(2, {1, 0, 0}),
    [_Gx0,Gy0,_Gz0] = calibrate(),
    io:format("[Robot] Done calibrating~n"),

    %Kalman matrices
    X0 = mat:matrix([[0], [0]]),
    P0 = mat:matrix([[0.1, 0], [0, 0.1]]),

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

    %PIDs initialisation
    Pid_Speed = spawn(pid_controller, pid_init, [-0.12, -0.07, 0.0, -1, 60.0, 0.0]),
    Pid_Stability = spawn(pid_controller, pid_init, [17.0, 0.0, 4.0, -1, -1, 0.0]),
    io:format("[Robot] Pid of the speed controller: ~p.~n", [Pid_Speed]),
    io:format("[Robot] Pid of the stability controller: ~p.~n", [Pid_Stability]),
  io:format("[Robot] Starting movement of the robot.~n"),

    %Call main loop
    robot_main(T0, Hera_pid, {rest, false}, {T0, X0, P0}, I2Cbus, {0, T0, []}, {Gy0, 0.0, 0.0}, {
     Pid_Speed, Pid_Stability}, {0.0, 0.0}, {0, 0, 200.0, T0}).

robot_main(Start_Time, Hera_pid, {Robot_State, Robot_Up}, {T0, X0, P0}, I2Cbus, {Logging, Log_End,
     Log_List}, {Gy0, Angle_Complem, Angle_Rate}, {Pid_Speed, Pid_Stability}, {Adv_V_Ref, Turn_V_Ref}, {
     N, Freq, Mean_Freq, T_End}) ->

    %Delta time of loop
    T1 = erlang:system_time()/1.0e6, %[ms]
  Dt = (T1- T0)/1000.0,              %[s]

    %%%%%%%%%%%%%%%%%%%%%%%%
    %%% Input from Sensor %%%
    %%%%%%%%%%%%%%%%%%%%%%%%

    %Read data
    [Gy,Ax,Az] = pmod_nav:read(acc, [out_y_g, out_x_xl, out_z_xl], #{g_unit => dps}),

    %%%%%%%%%%%%%%%%%%%%%%%%
    %%% Input from ESP32 %%%
    %%%%%%%%%%%%%%%%%%%%%%%%

    %Receive I2C and conversion
    [<<SL1,SL2,SR1,SR2,CtrlByte>>] = grisp_i2c:transfer(I2Cbus, [{read, 16#40, 1, 5}]),
  [Speed_L,Speed_R] = hera_com:decode_half_float([<<SL1, SL2>>, <<SR1, SR2>>]),
    Speed = (Speed_L + Speed_R)/2,

    %Retrieve flags from ESP32
    [Arm_Ready, Switch, Test, Get_Up, Forward, Backward, Left, Right] = hera_com:get_bits(CtrlByte),
```

```
87
88      %%%%%%%%%%%%%%%%%%%%
89      %%% Command Logic %%%
90      %%%%%%%%%%%%%%%%%%%%
91
92      %Set advance speed from flags
93      Adv_V_Goal = speed_ref(Forward, Backward),
94
95      %Set turning speed from flags
96      Turn_V_Goal = turn_ref(Left, Right),
97
98      %%%%%%%%%%%%%%%%%%%%%%%%%
99      %%% Angle Computations %%%
100     %%%%%%%%%%%%%%%%%%%%%%%%%
101
102     %Angle based directly on the sensors
103     Angle_Accelerometer = math:atan(Az / (-Ax))*?RAD_TO_DEG,
104
105     %Kalman filter computation
106     {X1, P1} = kalman_angle(Dt, Ax, Az, Gy, Gy0, X0, P0),
107   [Th_Kalman, _W_Kalman] = mat:to_array(X1),
108     Angle_Kalman = Th_Kalman*?RAD_TO_DEG,
109
110     %Complementary angle computation
111     K = 1.25/(1.25+(1.0/Mean_Freq)),
112     {Angle_Complem_New, Angle_Rate_New} = complem_angle({Dt, Ax, Az, Gy, Gy0, K, Angle_Complem,
        Angle_Rate}),
113
114     %Select angle between kalman or complementary
115     Angle = select_angle(Switch, Angle_Kalman, Angle_Complem),
116
117
118     %%%%%%%%%%%%%%%%%
119     %%% Controller %%%
120     %%%%%%%%%%%%%%%%%
121
122     %Takes as input:
123     % -Measures
124     % -Process IDs of the two PID controllers
125     % -Advance and turning velocity to reach and loopback velocity of trapezoidal profile
126     {Acc, Adv_V_Ref_New, Turn_V_Ref_New} = stability_engine:controller({Dt, Angle, Speed}, {Pid_Speed,
        Pid_Stability}, {Adv_V_Goal, Adv_V_Ref}, {Turn_V_Goal, Turn_V_Ref}),
127     %Gives as output:
128     % -Acceleration of the motors -> sent to ESP32
129     % -Loopback V_Ref for advance
130     % -Loopback V_Ref for turning -> sent to ESP32
131
132     %%%%%%%%%%%%%%%%%%%%%%
133     %%% Output to ESP32 %%%
134     %%%%%%%%%%%%%%%%%%%%%%
135
136     %State of the robot
137     if
138         Robot_Up and (abs(Angle) > 20) ->
139             Robot_Up_New = false;
140         not Robot_Up and (abs(Angle) < 18) ->
141             Robot_Up_New = true;
142         true ->
143             Robot_Up_New = Robot_Up
144     end,
145     if
146         Angle > 0.0 ->
147             F_B = 1;
148         true ->
149             F_B = 0
150     end,
151
152     %State change
153     case Robot_State of
154         rest ->
155             if
156                 Get_Up -> Next_Robot_State = raising;
157                 true -> Next_Robot_State = rest
```

```
158             end;
159         raising ->
160             if
161                 Robot_Up -> Next_Robot_State = stand_up;
162                 not Get_Up -> Next_Robot_State = soft_fall;
163                 true -> Next_Robot_State = raising
164             end;
165         stand_up ->
166             if
167                 not Get_Up -> Next_Robot_State = wait_for_extend;
168                 not Robot_Up -> Next_Robot_State = rest;
169                 true -> Next_Robot_State = stand_up
170             end;
171         wait_for_extend -> %Buffer state while Next_Robot_State switches state
172             Next_Robot_State = prepare_arms;
173         prepare_arms ->
174             if
175                 Arm_Ready -> Next_Robot_State = free_fall;
176                 Get_Up -> Next_Robot_State = stand_up;
177                 not Robot_Up -> Next_Robot_State = rest;
178                 true -> Next_Robot_State = prepare_arms
179             end;
180         free_fall ->
181             if
182                 abs(Angle) > 10 -> Next_Robot_State = wait_for_retract;
183                 true -> Next_Robot_State = free_fall
184             end;
185         wait_for_retract -> %Buffer state while Next_Robot_State switches state
186             Next_Robot_State = soft_fall;
187         soft_fall ->
188             if
189                 Arm_Ready -> Next_Robot_State = rest;
190                 Get_Up -> Next_Robot_State = raising;
191                 true -> Next_Robot_State = soft_fall
192             end
193     end,
194
195     %State output
196     case Next_Robot_State of
197         rest ->
198             Power    = 0,
199             Freeze   = 0,
200             Extend   = 0,
201             Robot_Up_Bit = 0;
202         raising ->
203             Power    = 1,
204             Freeze   = 0,
205             Extend   = 1,
206             Robot_Up_Bit = 0;
207         stand_up ->
208             Power    = 1,
209             Freeze   = 0,
210             Extend   = 0,
211             Robot_Up_Bit = 1;
212         wait_for_extend ->
213             Power    = 1,
214             Freeze   = 0,
215             Extend   = 1,
216             Robot_Up_Bit = 1;
217         prepare_arms ->
218             Power    = 1,
219             Freeze   = 0,
220             Extend   = 1,
221             Robot_Up_Bit = 1;
222         free_fall ->
223             Power    = 1,
224             Freeze   = 1,
225             Extend   = 1,
226             Robot_Up_Bit = 1;
227         wait_for_retract ->
228             Power    = 1,
229             Freeze   = 0,
230             Extend   = 0,
```

```
231             Robot_Up_Bit = 0;
232         soft_fall ->
233             Power    = 1,
234             Freeze   = 0,
235             Extend   = 0,
236             Robot_Up_Bit = 0
237     end,
238
239     %Send output to ESP32
240     Output_Byte = get_byte([Power, Freeze, Extend, Robot_Up_Bit, F_B, 0, 0, 0]),
241     [HF1, HF2] = hera_com:encode_half_float([Acc, Turn_V_Ref_New]),
242     grisp_i2c:transfer(I2Cbus, [{write, 16#40, 1, [HF1, HF2, <<Output_Byte>>]}]),
243
244     %%%%%%%%%%%%%%%%%%%%%%%%
245     %%% Testing Features %%%
246     %%%%%%%%%%%%%%%%%%%%%%%%
247
248     %Frequency computation
249     {N_New, Freq_New, Mean_Freq_New} = frequency_computation(Dt, N, Freq, Mean_Freq),
250
251     %Logging
252     if
253         Test ->
254             Log_End_New = erlang:system_time()/1.0e6 + ?LOG_DURATION;
255         true ->
256             Log_End_New = Log_End
257     end,
258     Logging_New = erlang:system_time()/1.0e6 < Log_End_New,
259
260     %LED flickering while logging
261     if
262         Logging_New ->
263             if
264                 N rem 9 < 4 ->
265                     grisp_led:color(1, {1, 1, 0}),
266                     grisp_led:color(2, {1, 1, 0});
267                 true->
268                     grisp_led:color(1, {0, 0, 0}),
269                     grisp_led:color(2, {0, 0, 0})
270             end;
271         true ->
272             ok
273     end,
274
275     %Check for start or end of logging sequence
276     if
277         not Logging and Logging_New ->
278             Hera_pid ! {self(), start_log};
279         Logging and not Logging_New ->
280             grisp_led:color(1, {1, 1, 0}),
281             grisp_led:color(2, {1, 1, 0}),
282             Hera_pid ! {self(), stop_log};
283         true ->
284             ok
285     end,
286
287     %Send values to ESP32 if asked, else stack up values in list
288     receive
289         {From, log_values} ->
290             From ! {self(), log, Log_List},
291             Log_List_New = []
292     after 0 ->
293         if
294             Logging_New ->
295                 % Log_List_New = lists:append(Log_List, [[T1-Start_Time, 1/Dt, Gy, Acc, CtrlByte,
     Angle_Accelerometer, Angle_Kalman, Angle_Complem, Adv_V_Ref, Switch]]);
296                 Log_List_New = [[T1-Start_Time, 1/Dt, Gy, Acc, CtrlByte, -Angle_Accelerometer, -
     Angle_Kalman, -Angle_Complem, Adv_V_Ref, Switch, Adv_V_Ref_New, Turn_V_Ref_New, Speed] | Log_List];
297             true ->
298                 Log_List_New = Log_List
299         end
300     end,
301
```

```
302     %Communication with Hera (more messages can be implemented by the user)
303     receive
304         {From1, get_all_data} -> From1 ! {self(), data, [T1-Start_Time, 1/Dt, Gy, Acc, CtrlByte, -
         Angle_Accelerometer, -Angle_Kalman, -Angle_Complem, Adv_V_Ref, Switch, Adv_V_Ref_New,
         Turn_V_Ref_New, Speed]};
305         {From1, freq} -> From1 ! {self(), 1/Dt};
306         {From2, acc} -> From2 ! {self(), Acc};
307         {_, Msg} -> io:format("[Robot] Message [~p] not recognized. ~nPossible querries are: [
         get_all_data, freq, acc]. ~nMore querries can be added.", [Msg])
308     after 0 ->
309         ok
310     end,
311
312
313     %Imposed maximum frequency
314     T2 = erlang:system_time()/1.0e6,
315     [{_,Freq_Goal}] = ets:lookup(variables, "Freq_Goal"),
316     Delay_Goal = 1.0/Freq_Goal * 1000.0,
317     if
318         T2-T_End < Delay_Goal ->
319             wait(Delay_Goal-(T2-T1));
320         true ->
321             ok
322     end,
323     T_End_New = erlang:system_time()/1.0e6,
324
325     %Loop back with updated state
326     robot_main(Start_Time, Hera_pid, {Next_Robot_State, Robot_Up_New}, {T1, X1, P1}, I2Cbus, {
         Logging_New, Log_End_New, Log_List_New}, {Gy0, Angle_Complem_New, Angle_Rate_New}, {Pid_Speed,
         Pid_Stability}, {Adv_V_Ref_New, Turn_V_Ref_New}, {N_New, Freq_New, Mean_Freq_New, T_End_New}).
327
328
329 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
330 %% Internal functions
331 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
332
333 calibrate() ->
334     N = 500,
335     Data = [list_to_tuple(pmod_nav:read(acc, [out_x_g, out_y_g, out_z_g])) || _ <- lists:seq(1,N)],
336     {X, Y, Z} = lists:unzip3(Data),
337     [lists:sum(X)/N, lists:sum(Y)/N, lists:sum(Z)/N]. %[Gx0, Gy0, Gz0]
338
339 kalman_angle(Dt, Ax, Az, Gy, Gy0, X0, P0) ->
340     R = mat:matrix([[3.0, 0.0], [0, 3.0e-6]]),
341     Q = mat:matrix([[3.0e-5, 0.0], [0.0, 10.0]]),
342     F = fun (X) -> [Th, W] = mat:to_array(X),
343         mat:matrix([  [Th+Dt*W],
344                 [W      ] ])
345     end,
346     Jf = fun (X) -> [_Th, _W] = mat:to_array(X),
347         mat:matrix([    [1, Dt],
348                 [0, 1 ] ])
349      end,
350     H = fun (X) -> [Th, W] = mat:to_array(X),
351         mat:matrix([  [Th],
352                 [W ] ])
353     end,
354     Jh = fun (X) -> [_Th, _W] = mat:to_array(X),
355         mat:matrix([    [1, 0],
356                 [0, 1] ])
357      end,
358     Z = mat:matrix([[math:atan(Az / (-Ax))], [(Gy-Gy0)*?DEG_TO_RAD]]),
359     kalman:ekf({X0, P0}, {F, Jf}, {H, Jh}, Q, R, Z).
360
361 complem_angle({Dt, Ax, Az, Gy, Gy0, K, Angle_Complem, Angle_Rate}) ->
362
363     Angle_Rate_New = (Gy - Gy0) * ?COEF_FILTER + Angle_Rate * (1 - ?COEF_FILTER),
364
365     %Angle increment computed from gyroscope
366     Delta_Gyr = Angle_Rate_New * Dt,
367
368     %Absolute angle computed form accelerometer
369     Angle_Acc = math:atan(Az / (-Ax)) * 180 / math:pi(),
```

```

    %Complementary filter combining gyroscope and accelerometer
    Angle_Complem_New = (Angle_Complem + Delta_Gyr) * K + Angle_Acc * (1 - K),

    {Angle_Complem_New, Angle_Rate_New}.

select_angle(Switch, Angle_Kalman, Angle_Complem) ->
    if
        Switch ->
            Angle = Angle_Kalman;
        true ->
            Angle = Angle_Complem
    end,
    Angle.

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}.

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).

%Transforms a list of 8 bits into a byte
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.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Global variables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

modify_frequency(Freq) ->
    ets:insert(variables, {"Freq_Goal", Freq}),
    ok.
```

I.1.5 stability_engine

```
-module(stability_engine).

-export([controller/4]).

-define(ADV_V_MAX, 30.0).
-define(ADV_ACCEL, 75.0).

-define(TURN_V_MAX, 80.0).
-define(TURN_ACCEL, 400.0).


%V_ref_new must be looped to V_ref
controller({Dt, Angle, Speed}, {Pid_Speed, Pid_Stability}, {Adv_V_Goal, Adv_V_Ref}, {Turn_V_Goal,
     Turn_V_Ref}) ->

    %Saturate advance acceleration
    if
        Adv_V_Goal > 0.0 ->
            Adv_V_Ref_New = pid_controller:saturation(Adv_V_Ref+?ADV_ACCEL*Dt, ?ADV_V_MAX);
        Adv_V_Goal < 0.0 ->
            Adv_V_Ref_New = pid_controller:saturation(Adv_V_Ref- ?ADV_ACCEL*Dt, ?ADV_V_MAX);
        true ->
            if
                Adv_V_Ref > 0.5  ->
                    Adv_V_Ref_New = pid_controller:saturation(Adv_V_Ref- ?ADV_ACCEL*Dt, ?ADV_V_MAX);
                Adv_V_Ref < -0.5 ->
                    Adv_V_Ref_New = pid_controller:saturation(Adv_V_Ref+?ADV_ACCEL*Dt, ?ADV_V_MAX);
                true ->
                    Adv_V_Ref_New = 0.0
            end
    end,
    % Adv_V_Ref_New = Adv_V_Goal,

    %Saturate turning acceleration
    if
        Turn_V_Goal > 0.0 ->
            Turn_V_Ref_New = pid_controller:saturation(Turn_V_Ref+?TURN_ACCEL*Dt, ?TURN_V_MAX);
        Turn_V_Goal < 0.0 ->
            Turn_V_Ref_New = pid_controller:saturation(Turn_V_Ref- ?TURN_ACCEL*Dt, ?TURN_V_MAX);
        true ->
            if
                Turn_V_Ref > 0.5  ->
                    Turn_V_Ref_New = pid_controller:saturation(Turn_V_Ref- ?TURN_ACCEL*Dt, ?TURN_V_MAX);

                Turn_V_Ref < -0.5 ->
                    Turn_V_Ref_New = pid_controller:saturation(Turn_V_Ref+?TURN_ACCEL*Dt, ?TURN_V_MAX);
                true ->
                    Turn_V_Ref_New = 0.0
            end
    end,
    % Turn_V_Ref_New = Turn_V_Goal,

    %Speed PI
    Pid_Speed ! {self(), {set_point, Adv_V_Ref_New}},
    Pid_Speed ! {self(), {input, Speed}},
    receive {_, {control, Target_angle}} -> ok end,

    %TODO: send Target_angle to log

    % io:format("~p~n",[Target_angle]),

    %Stability PD
    Pid_Stability ! {self(), {set_point, Target_angle}},
    Pid_Stability ! {self(), {input, Angle}},
    receive {_, {control, Acc}} -> ok end,

    {Acc, Adv_V_Ref_New, Turn_V_Ref_New}.
```

I.2 Robot ESP32 code

I.2.1 ESP32_main

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

#include "motor_engine.h"

#define BAND 868E6
#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


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 n2
```

```
69    engine_init();
70    delay(1000);
71    set_speed(0, 0);
72    set_acceleration(0, 0);
73
74    // time init
75    t_GRiSP = millis();
76    t_LORA = t_GRiSP;
77    t_ESP = t_GRiSP;
78  }
79
80
81  void loop() {
82    unsigned long new_t_ESP = millis();
83    dt_ESP = (new_t_ESP - t_ESP) / 1000.0;
84    t_ESP = new_t_ESP;
85    LoRa_receiver();
86    Event_handle();
87    delay(1);
88  }
89
90
91
92  void GRiSP_receiver(int howMany) {
93    if(howMany == 5){ // check if the packet match the expected lenght
94      unsigned long new_t_GRiSP = millis();
95      dt_GRiSP = (new_t_GRiSP - t_GRiSP) / 1000.0;
96      freq_GRiSP = freq_GRiSP * 0.99 + 1.0 / dt_GRiSP * 0.01;
97      t_GRiSP = new_t_GRiSP;
98
99      byte A;
100     byte B;
101     if (Wire.available()) {
102
103       // read and decode the wheel acceleration
104       A = Wire.read();
105       B = Wire.read();
106       I2C_command[0] = decoder(A, B);
107
108       // read and decode the differential turn speed
109       A = Wire.read();
110       B = Wire.read();
111       I2C_command[1] = decoder(A, B);
112
113       // read the flags
114       GRiSP_flags = Wire.read();
115     }
116
117     //set acceleration
118     if(!disturb && bitRead(GRiSP_flags, 4) && !bitRead(GRiSP_flags, 6)){
119       set_acceleration(I2C_command[0], I2C_command[0]);
120     }
121
122     // Free fall, null wheel speed
123     if(bitRead(GRiSP_flags, 6)){
124       set_acceleration(0, 0);
125       set_speed(0, 0);
126     }
127
128     // extension/retraction of the rising system
129     if(bitRead(GRiSP_flags, 5)){
130       ext_end = stand(-48,30.0);
131     } else {
132       ext_end = stand(0,30.0);
133     }
134
135     // wheel counter rotation activation and direction selection during rise
136     int stand_speed_dir = 0;
137     if(!bitRead(GRiSP_flags, 4)){ // if down
138       stand_speed_dir = (bitRead(GRiSP_flags, 3)) ? -1 : 1;
139     }
140     raise_dir(stand_speed_dir); // -1 back, 0 null, 1 front
141   }
```

```
142
143   // Empty the stack
144   while(Wire.available()){
145     Wire.read();
146   }
147 }
148
149 void GRiSP_sender()
150 {
151   byte v[5];
152   float* speeds = get_speed();
153   encoder(v, speeds[0]);
154   encoder(v+2, speeds[2]);
155
156   // control byte send to GRiSP witth : finsish extension/retraction flag and the command inputs
157   v[4] = (cmd & 127) | (is_ready() * 128);
158
159   //send
160   Wire.write((byte*) v, sizeof(v));
161 }
162
163 double decoder(byte X, byte Y) {
164   // decode half float to double
165
166   byte A = (X & 192);
167   if ((A & 64) == 0) A = A | 63;  // fill the missing exponnent bytes with the right value
168   byte B = ((X << 2) & 252) | ((Y >> 6) & 3);
169   byte C = ((Y << 2) & 252);
170
171   byte vals[] = { 0x00, 0x00, 0x00, 0x00, 0x00, C, B, A };
172   double d = 0;
173   memcpy(&d, vals, 8);
174
175   return d;
176 }
177
178 void encoder(byte* res, double X){
179   // encode double to half float
180   byte vals[8];
181   memcpy(vals, &X,8);
182   byte A = vals[7];
183   byte B = vals[6];
184   byte C = vals[5];
185
186   res[0] = (A&192)|((B>>2)&63);
187   res[1] = ((B<<6)&192)|((C>>2)&63);
188
189   return ;
190 }
191
192 void LoRa_receiver(){
193 // receiption of LoRa packets
194   if (LoRa.parsePacket()) {
195     if(LoRa.available()>=2){
196       byte cmd1 = LoRa.read();
197       byte cmd2 = LoRa.read();
198       if(cmd1 == cmd2){
199         cmd = cmd1;
200         new_cmd = true;
201       }
202     }
203     while (LoRa.available()){
204       LoRa.read();
205     }
206   }
207 }
208
209 void Event_handle(){
210
211   //emergency stop
212   emergency(!bitRead(cmd, 7) || !bitRead(GRiSP_flags, 7));
213   if (!bitRead(cmd, 7) || !bitRead(GRiSP_flags, 7)){
214     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;
}
```

I.2.2 ESP32_motor_engine

```
#pragma once

#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 16
#define steps 400
#define diameter 7 // cm
#define lever_ratio 0.09090909090909090909090909090909 //    12/132
#define lenght_btw_wheels 18.5 // 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();
```

```
#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;

// wheel counter rotation speed during rise, rot/sec
float v_MA_rise = 0;
float v_MC_rise= 0;
int wheel_raise_dir = 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/s
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;
    v_MA_rise = Speed_stand * wheel_raise_dir;
    v_MC_rise = Speed_stand * wheel_raise_dir;
  } else if (total_stepB < target_step){
    v_MB = -Speed_stand / lever_ratio;
    v_MA_rise = -Speed_stand * wheel_raise_dir;
    v_MC_rise = -Speed_stand  * wheel_raise_dir;
  } else {
    v_MB = 0;
    v_MA_rise = 0;
```

```
206     v_MC_rise = 0;
207   }
208
209   //acceleration calculation of motor A with limit to +/- vmax
210   v_MC += a_MC * dt_loop * 1e-6;
211   if (v_MC > v_max_ang) {
212     v_MC = v_max_ang;
213   } else if (v_MC < -v_max_ang) {
214     v_MC = -v_max_ang;
215   }
216
217   //total speeds
218   v_tot_A = v_MA + v_diff + v_MA_rise;
219   v_tot_B = v_MB;
220   v_tot_C = v_MC - v_diff + v_MC_rise;
221
222
223   //compute freq from speed
224   f_MA = v_tot_A * steps * microsteps;
225   f_MB = v_tot_B * steps * microsteps;
226   f_MC = v_tot_C * steps * microsteps;
227
228   //compute the increment period from freq and set the direction
229   if (f_MA < 0) {
230     dt_MA = 1e6 / abs(f_MA);
231     digitalWrite(MOTOR_A_DIR_PIN, LOW);
232     dirA = 1;
233   } else if (f_MA > 0) {
234     dt_MA = 1e6 / abs(f_MA);
235     digitalWrite(MOTOR_A_DIR_PIN, HIGH);
236     dirA = -1;
237   } else {
238     dirA = 0;
239   }
240
241
242   if (f_MB > 0) {
243     dt_MB = 1e6 / abs(f_MB);
244     digitalWrite(MOTOR_B_DIR_PIN, LOW);
245     dirB = -1;
246   } else if (f_MB < 0) {
247     dt_MB = 1e6 / abs(f_MB);
248     digitalWrite(MOTOR_B_DIR_PIN, HIGH);
249     dirB = 1;
250   } else {
251     dirB = 0;
252   }
253
254   if (f_MC > 0) {
255     dt_MC = 1e6 / abs(f_MC);
256     digitalWrite(MOTOR_C_DIR_PIN, LOW);
257     dirC = -1;
258   } else if (f_MC < 0) {
259     dt_MC = 1e6 / abs(f_MC);
260     digitalWrite(MOTOR_C_DIR_PIN, HIGH);
261     dirC = 1;
262   } else {
263     dirC = 0;
264   }
265 }
266
267 void reset_dist() { //reset function
268   total_stepA = 0;
269   total_stepB = 0;
270   total_stepC = 0;
271 }
272
273 void set_turn(float angular_speed){ // apply a differential speed on the wheel to turn
274   v_diff = (lenght_btw_wheels*PI*angular_speed / (180.0*2))/(PI * diameter) ;
275 }
276
277 void emergency(bool stop){ //shut off the power
278   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;
}

void raise_dir(int dir){ // Use to specify the wheel counter rotation direction during raise/fall
  wheel_raise_dir = dir;
}

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

I.3 Emergency stop ESP32 code

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


#define Pin 15
#define Buzz 13

#define LORA_PERIOD 868
#define BAND 868E6


#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();
  }
}
```

I.4 User interface

```
import pygame
import sys
import numpy as np
import serial
import time


#Pygame init
pygame.init()
ser = serial.Serial(port="COM4", baudrate=115200)
width, height = 800, 600
screen = pygame.display.set_mode((width, height), pygame.RESIZABLE)
```

```
pygame.display.set_caption("Rotation de la flche")

# Colors
white = (255, 255, 255)

#load figures
arrow_img = pygame.image.load('arrow.png')
arrow_img = pygame.transform.scale(arrow_img, (arrow_img.get_width() // 2, arrow_img.get_height() // 2))
arrow_rect = arrow_img.get_rect(center=(width // 2, height // 2))
circle_img = pygame.image.load('point.png')
circle_img = pygame.transform.scale(circle_img, (circle_img.get_width() // 2, circle_img.get_height() // 2))
stop_img = pygame.image.load('Stop_sign.png')
stop_img = pygame.transform.scale(stop_img, (stop_img.get_width() // 5, stop_img.get_height() // 5))

font = pygame.font.Font(None, 36)



#state variables
running = True
message = 0
run = True
stand = False
kalman = True
release_space = True
release_enter = True
release_t = True
release_tab = True


test_time = 0

# main loop
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
    #get window size
    width, height = screen.get_size()
    arrow_rect = arrow_img.get_rect(center=(width // 2, height // 2))

    #Get the keys pressed
    keys = pygame.key.get_pressed()
    x = 0
    string = ""

    if keys[pygame.K_z] or keys[pygame.K_UP]:
        x += 1
    if keys[pygame.K_s] or keys[pygame.K_DOWN]:
        x += -1
    if keys[pygame.K_q] or keys[pygame.K_LEFT]:
        x += 1j
    if keys[pygame.K_d] or keys[pygame.K_RIGHT]:
        x += -1j
    if keys[pygame.K_ESCAPE]:
        running = False
    if keys[pygame.K_SPACE]:
        if release_space:
            release_space = False
            if message < 10000000:
                run = True
            else:
                run = False
    else:
        release_space = True

    if keys[pygame.K_TAB]:
        if release_tab:
            release_tab = False
            kalman = not kalman
    else:
```

```
        release_tab = True

    if keys[pygame.K_k]:
        kalman = True

    if keys[pygame.K_c]:
        kalman = False

    if kalman :
        string += "Kalman filter\n"
    else:
        string += "Complementary filter\n"

    test = False
    if keys[pygame.K_t]:
        if release_t:
            release_t = False
            test = True
    else:
        release_t = True

    if keys[pygame.K_RETURN]:
        if release_enter:
            release_enter = False
            stand = not stand
    else:
        release_enter = True

    if not stand:
        string += "DOWN \n"
    else:
        string += "UP \n"


    if test:
        test_time = time.time()

    #if (1<= time.time()-test_time < 10) : stand=True
    #if (10<= time.time()-test_time < 25) : stand=False


    # Clear the screen
    screen.fill(white)

    # Draw the arrow
    if message < 10000000:
        screen.blit(stop_img, stop_img.get_rect(center=arrow_rect.center))
    elif abs(x) == 0:
        screen.blit(circle_img, circle_img.get_rect(center=arrow_rect.center))
    else:
        angle = np.angle(x, deg=True)
        rotated_arrow = pygame.transform.rotate(arrow_img, angle)
        rotated_rect = rotated_arrow.get_rect(center=arrow_rect.center)
        screen.blit(rotated_arrow, rotated_rect.topleft)

    # Draw the text

    if run:
        string += "Running\n"
    else:
        string += "Stopped\n"

    string += "Message: " + str(message) + "\n"

    for i, line in enumerate(string.split("\n")):
        text = font.render(line, True, (0, 128, 0))
        screen.blit(text, (10, 10 + i * 30))




    pygame.display.flip()

```

```
157     # Limit the frame rate
158     pygame.time.Clock().tick(200)
159 
160     data = run << 7 | kalman << 6 | test << 5 | stand << 4 | (x.real == 1) << 3 | (x.real == -1) << 2 |
          (
161                 x.imag == 1) << 1 | (x.imag == -1)
162     ser.write(bytes([data]))
163     # print data as binary
164     # print(bin(data))
165 
166     # read data until \n is received
167     Content = ser.readline()
168     # remove the \r and \n from the string
169     Content = Content.decode().replace("\r\n", "")
170     # print(Content)
171     message = int(Content)
172     print(message)
173 
174 # Quit
175 pygame.quit()
176 sys.exit()
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**