UCLouvain

epl

École polytechnique de Louvain

Controlling everyday life objects using the Hera platform on GRiSP 2.0

The *movement_detection* application

Author: **Nicolas ISENGUERRE**
Supervisor: **Peter VAN ROY**
Readers: **Benoît HERMAN, Peer STRITZINGER**
Academic year 2023–2024
Master [120] in Electro-mechanical Engineering

Abstract

In today's world, applications based on the *Internet-of-Things* (IoT) are becoming more and more important. However, not all IoT devices are able to process the data they collect. This can become problematic in the context of real-time applications, as the latency introduced by the usage of the cloud for data processing can be unacceptable. This is limiting the potential that IoT devices could have in some fields, like in domotics or *home automation* applications where the real-time requirement is of the utmost importance. To solve this latency problem, IoT devices with more powerful embedded systems rely more and more on edge computing.

In this context, the prototype of a moving robot using the GRiSP2 board for IoT and edge computing has been developed, as well as a dedicated Erlang/OTP application, named *movement_detection*. The robot is remotely controlled using dedicated gestures, which are detected by the accelerometer of an Inertial-Measurement-Unit plugged on a second GRiSP2 board. A library of gestures has been created for this application and the resulting robot behaviour for each gesture allows the user to move the prototype around as he wishes. Furthermore, many safety measures have been added to this prototype, as this application, based on "off-the-shelf" and cheap components, intends to show the potential of the GRiSP2 board in the context of domotics and Human-Robot interactions.

In the first chapters of this thesis, the whole application, from the prototype to the software behind it, is fully described. Moreover, the performance of the system is detailed in the penultimate chapter, which also covers the current limitations the application faces. By going over the full process of thought, this thesis intends to inspire future generations of GRiSP2 users to improve the current system, as this application already shows promising results. Ideas for improvement and future works can be found in the final chapter of this work.

Acknowledgements

This work was only possible thanks to the support I have received throughout the semester. I would like to take some time to thank various people.

First, I would like to thank the GRiSP2 team and contributors, namely Igor K., Gwendal Laurent, Luca Succi, an unknown user that goes by the nickname Hoss and Peer Stritzinger, for their reactivity on the GRiSP Slack and all the help they have provided regarding the GRiSP2 board, the network problems and the Erlang language. Special thanks go to Peer Stritzinger who has always answered rapidly to all my questions, and who proposed the I2C communication `gen_server` idea.

Next, I would like to thank some staff and academic members: Thierry Daras for the provided freebear wheel and battery, Souley Djadjandi for the help with the battery connectors, the provision of the perfboard and some cables, Simon De Jaeger for all his help with the mechanical aspects of the robot and especially with the hinge, Vanessa Maons for her availability and for granting me access to a room with a locker to be able to work and store all the equipment safely and Benoît Herman for granting me access to the *Making Pro* room to allow me to build the robot.

I would also like to thank Simon Alaerts and Martin Brans for the interesting conversations about the ESP32 problems and Guillaume Deceuninck for the 3D-printed battery support. A special thank you goes to François Goens for his help with the GRiSP environment and Cédric Ponsard for his help with the 3D printed parts, the I2C communication, setup of the thesis' LaTeX, suggesting the Raspberry Pi Pico W in place of the ESP32 and overall experience in the field of robotics. To François Goens and Cédric Ponsard again, but also all the other students who were working on their master thesis in the same room as I was, a big thanks for all the laughter and time spent together. It was a great work environment. Lastly, I would like to thank all the previous contributors to the Hera platform and the *sensor_fusion* application and especially Sébastien Kalbusch, Lucas Nélis and Sébastien Gios for their explanations regarding their parts and help with some problems in Hera.

On a more personal note, I would like to thank my friends and family for their support, but most importantly my girlfriend, Alix, for her unwavering support, my two maternal grandparents for all the time they spent correcting and proof-reading this work, and a very special thank you goes to my father, for all these years of support throughout my engineering studies and all he has done for me in this thesis, including some help debugging the encoder's and the controller's code, his help with the logger `gen_server` and overall valuable outside perspective.

And last but not least, I would like to thank my supervisor, Peter Van Roy, for his guidance, constantly renewed support, ideas and for allowing me to work on this project which has been very rewarding. I am thankful for the opportunity to work with him and this thesis has overall been a great experience for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Internet-of-Things and domotics

The *Internet-of-Things*, or IoT, is a booming field in engineering. It describes a network of physical devices, vehicles and other physical objects that are embedded with sensors, software, and network connectivity, allowing them to collect and share data[1]. It is an ever growing part of the Internet infrastructure at its very logical edge.

Thanks to the embedded sensors, IoT devices collect lots of data that need to be processed. Generally speaking, cloud computing, which describes the use of remote computer servers hosted in data centres connected to the Internet to store, manage and process data[2], is a reliable choice to process this data. But it has one big drawback: sending the data to the cloud, then waiting for it to be processed before retrieving it can lead to some latency which, in real-time applications, can become unacceptable. This is where edge computing becomes interesting: it aims at processing the data as close as possible to its source, without relying on the cloud or centralised data networks, which in turn can improve response times, reduce latency and reduce the amount of data that needs to be transferred over IoT networks[1].

With the increasingly powerful embedded systems, edge computing is becoming more and more important for IoT to get fast, reliable and real-time applications. This ability to create networks of powerful and small computing units opens the gates to new opportunities in a vast field of robotics: domotics.

If it did strike fear in the hearts of men in the past, domotics, or more generally speaking *smart houses*, are becoming more and more essential[3] and the next logical development step in the field of robotics and IoT. Many devices already exist: from smart dishwashers, fridges and washing machines to automatic vacuum cleaners or thermostats, the list goes on. A modern house is filled with robots and automatic devices[4]. However, this idea of "home automation" has not yet been fully exploited: the human user still needs to open the fridge, or the windows, unload the dishwasher, etc. This is where *assisted living* comes into place: the house helps its occupants age in place or cope with injuries[3], which could be very beneficial for the independence of the elderly or sick patients. The house could become a very large health monitoring app with lots of sensors[3] on many devices placed everywhere in the house and assistance could be given to the users who wish, for example, by using tiny yet powerful DC motors that could be used to automate the opening of fridges or windows. Moreover, from a healthcare perspective, such a house could not only be designed for increasing comfort, security, and autonomy of patients, but can also be a rich source of continuous data, for example in the monitoring of the evolution of Parkinson's disease patients[5].

In particular, one branch of domotics is still in its infancy: robots that would immediately interact with humans. Whether it would be to help a patient get out of bed or bring him some medicines, due to the challenging environment of houses and Human-Robot-Interactions (HRI) and for safety considerations, such robots are yet to be globally commercialised. And this is where IoT can become a major breakthrough. By providing small and powerful computing units with integrated sensors throughout the house, one could monitor moving robots in a continuous manner, or register patients' gestures to send orders to the robots in real-time, providing permanent control over the robots and more safety for the user, whilst rendering them increasingly autonomous. But before this becomes reality, many more questions, like security and privacy implications of IoT devices, need to be assessed[6].

1.2 Contributions

In this context, the prototype of a moving robot supported by two 6V DC-motors with integrated gearbox that actuate two wheels and one freebear wheel for stability has been developed from an existing wine-crate. This robot is controlled remotely using the GRiSP2 board for IoT and edge computing by detecting gestures, which is done by processing the data coming from an Inertial-Measurement-Unit (IMU) provided by the Pmod™ NAV, a small but powerful sensor composed of a 3-axis accelerometer, gyroscope and magnetometer, and comparing the received data to a library of known gestures using pattern matching.

Using the Hera platform for sensor fusion, a fault-tolerant framework developed in the Erlang programming language to work with the GRiSP environment, an application named *movement_detection* and which extends the already existing *sensor_fusion* application has been developed to control remotely the robot. It uses the principles of distributed network, a powerful feature of the Erlang programming language, to send the gestures detected by a GRiSP2 board named the *detector* to a GRiSP2 board named the *receiver*, which then communicates using the I²C communication protocol with a Raspberry Pi Pico W board that acts as the low-level controller of the robot.

The following list summarises the contributions of this master thesis:

- A first prototype of a remotely controllable robot using the GRiSP environment, intended to show the potential of these boards for such applications.
- The integration of both 2023's theses: the `numerl` NIF (Native Implemented Functions) library and the gesture recognition algorithm, into one common application[7][8].
- The *movement_detection* application, which includes:
  - A gesture library to get the appropriate behaviour out of the robot;
  - An improved gesture recognition algorithm to fit the need of the application;
  - Two `gen_server` related to the application: the first to transform the received gesture into the appropriate command and the second to perform safe serial communication between a GRiSP2 board and any other device using the I²C protocol. Both of these are generalisable to any future application;
  - A complete low-level velocity controller in C++ with routines for precise movements;
  - Several software guard-rails for the safety of the device.
- An extension to the Hera platform to correctly log data to a .csv file using another `gen_server`.
- An updated user manual covering the installation of Erlang, Hera, the application, etc. to ease the handover process.

1.3 Roadmap

The thesis is divided into 7 chapters, excluding the appendices. After this introduction, the next chapter will describe control protocols and some of the hardware used in this project, as well as presenting the work that has been done in the previous years and on which this master thesis builds.

Next, chapters 3, 4 and 5 will explain in thorough detail what has been done during this thesis:

- In the third chapter, the prototype is fully described, from a mechanical and electrical standpoint. The chapter will describe the choices that were made, the hardware used, the connections between the different pieces of hardware and the overall structure of the robot.
- The fourth chapter dives deeper into the low-level controller by explaining how the velocity is computed and controlled using a PI controller and the problems that were faced. The PID controllers and encoders inner workings will be explained in this chapter.
- The fifth chapter details the application developed for this project, named *movement_detection*. It goes over the application's specifications and the limitations of the previously developed algorithm for gesture recognition before explaining the whole application's software and the gesture library, as well as the several protections that were added to make the system safer.

These chapters are followed by chapter 6, where the performance of the system is analysed and criticised. The limitations of the different parts are detailed and the extension made to Hera to log the data coming from the IMU is explained.

Finally, this document concludes by a few words about the work done in this thesis before going in detail over the possible future works and improvements to the system.

Any complementary information, from the source code to the updated user manual or more detailed figures explaining the gestures, for example, can be found in the appendices, after the bibliography.

Chapter 2

Resources and background

2.1 The GRiSP2 board

Developed by *Peer Stritzinger GmbH*, this second version of the GRiSP board, visible in Figure 2.1, aims at offering the best prototyping solution for Erlang and Elixir developers[9]. Its main goal is to provide a micro-computer, like a Raspberry Pi, to create Internet-of-Things (IoT) projects without having to drop down to C.

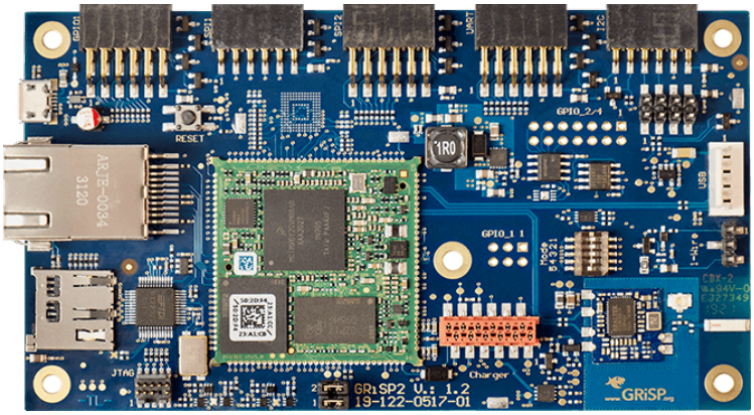


Figure 2.1: The GRiSP2 board

This board provides a complete Erlang Virtual Machine (VM) running on the CPU. It is compiled together with the real-time abstraction layer Real-Time Executive for Multiprocessor Systems (RTEMS) Real Time Operating System (RTOS), which allows Erlang to run directly on the hardware with no layer in between. The users can interact with the board using an Erlang shell that runs on the board. It can be accessed via serial (cabled) connection or by launching a remote shell using the Wi-Fi (or Ethernet). Finally, the board also supports the Elixir language via Nerves and Linux.

Furthermore, the board is equipped with five built-in sockets[1] which can be used to plug-in Digilent Pmod sensors to the board, as these sockets adhere to the Digilent Pmod Interface Specification (for more detail on this, refer to Section 2.3). This is advantageous for IoT applications, as it allows the user to work with sensor data directly on the board and perform edge computing. The data can be easily retrieved thanks to dedicated Erlang drivers which have been developed for the GRiSP2 board.

[1]Note that a sixth one can be added by the user to have access to one more Type 1 (GPIO) Pmod interface. More on Pmod interfaces later.

The hardware specifications of the GRiSP2 board, taken from its Kickstarter page[9], can be found in Appendix B.1

2.2 The Raspberry Pi Pico W board

Designed by Raspberry Pi and based on the RP2040 microcontroller chip[10], the Raspberry Pi Pico W board is a micro-controller board, unlike the other Raspberry Pi products which are single-board computers that can run an operating system. The "W" in the name stands for Wireless, as it includes Wi-Fi support (802.11n)[11]. It has almost no RAM or storage and is powered by a tiny processor operating at 133 MHz. This board is designed to run a very specific task and can interact with small devices like sensors, motors or other boards via its pins. In the landscape of Raspberry Pi products, it positions itself more as a direct concurrent to the Arduino boards than as a relative of the Raspberry Pi micro-computers, as it has no micro-processor. It can typically be used to fetch data from sensors, process them and then perform an action (like controlling a motor).

The Raspberry Pi Pico W board is shown in Figure 2.2.

Figure 2.2: Illustration: the Raspberry Pi Pico W board

Note that the complete specifications of the board can be found in Appendix B.2.

2.3 The Digilent Pmod™

From their own words: "Digilent Pmods are small, low-cost peripheral modules designed to be used with a wide range of embedded systems and development boards. Pmod is short for 'peripheral module' and these modules are specifically designed to extend the capabilities of embedded systems and development boards by adding new features and functionality."[12] Many Pmods exist, with either 6- or 12-pin connectors, and several interface types, offering a support for different communication protocols (e.g.: SPI uses the Pmod interface type 2). It is a standard for building small peripheral modules, whose full interface specification can be found here[13].

As mentioned in Section 2.1, the GRiSP2 board has the ability to host Pmods as extensions to the board and offers specific drivers to work with them. In the context of this thesis, two Pmods will be used, the Pmod NAV and the Pmod HB5, which will be referred to for the rest of this thesis as the PNAV and the HB5, respectively.

2.3.1 Pmod™ NAV

This Pmod[14], which is visible in Figure 2.3a, contains two sub-units: the LSM9DS1 which comes with a 3-axis accelerometer (±2/±4/±8/±16 g linear acceleration full scale), a 3-axis gyroscope (±245/±500/±2000 degrees-per-second angular rate full scale) and 3-axis magnetometer

(±4/±8/±12/±16 gauss magnetic full scale), and the LPS25HB digital barometer (260-1260 hPa piezoresistive pressure sensor), giving the user 10 degrees of freedom to work with. This Pmod comes with a 12-pin connector with SPI interface, following the Digilent Interface specification type 2A. The PNAV is an excellent sensor to determine the exact position and heading of the module, as it offers a small IMU (Inertial-Measurement-Unit), and has been used in a previous thesis to build an attitude and heading reference system (AHRS)[15].

(a) The Pmod NAV (b) The Pmod HB5

Figure 2.3: Illustration of the two Pmods used in this thesis

2.3.2 Pmod™ HB5

This Pmod[16], visible in Figure 2.3b, comes with a 2A H-bridge to drive small to medium sized DC motors. The particularity of this Pmod is that it has been designed to work with the Digilent gearbox motor, which is the motor used in this thesis as explained in Section 3.2.3. Indeed, the HB5 provides two sensor feedback pins which are designed to work with the encoder of the aforementioned motors and it allows to drive them using PWM (for more explanation on PWM, refer to Section 2.7).

The HB5 can drive DC-motors with operating voltage up to 12 V and follows the Digilent Pmod Interface Specification Type 5. To drive the motors, it uses an H-bridge. An H-bridge is an electrical circuit composed of four switches, as visible in Figure 2.4[2], which allows the controlled motor to go both forward and backward by switching the polarity of the voltage applied to it[17].

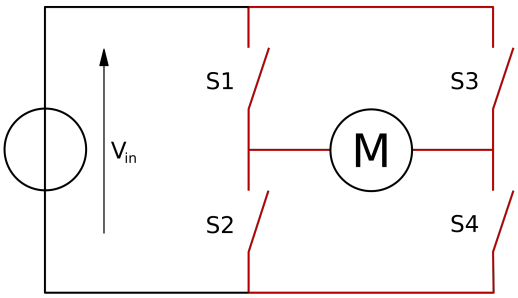


Figure 2.4: Typical circuit using an H-bridge, highlighted in red

Indeed, opening switches S2 and S3 but closing switches S1 and S4 applies V_{in} to the motor terminals, but opening switches S1 and S4 and closing switches S2 and S3 applies $-V_{in}$ to the motor terminals. It should however be noted that special care should be taken when using such a circuit to avoid causing a short on the power supply. To this end, one should never change the state

[2]Illustration courtesy of Cyril BUTTAY - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=854051.

of the Direction pin, which informs the HB5 which polarity the output voltage must have, when the Enable pin, containing the digital PWM command, is at high voltage state. This is because as one pair of switches closes, the other will open, and there may be a short window within which two switches from the same branch are closed at the same time, effectively shorting the power supply.

2.4 Erlang/OTP

As mentioned earlier, the GRiSP2 board runs with the Erlang programming language. This functional high-level programming language has built-in support for concurrency, distribution and fault tolerance[18]. It thus supports highly scalable, distributed and concurrent applications. It is particularly well-suited for distributed, reliable and soft real-time concurrent systems, like a telecommunication system, or servers for internet application[19].

On the other hand, OTP is a large collection of Erlang libraries, ready-to-use tools and design principles that provide middleware to develop Erlang applications and systems[18].

Unlike many other languages, Erlang does not use threads to work in parallel, but instead uses processes: Erlang processes are lightweight and they grow and shrink dynamically with small memory footprint. They are fast to create and terminate[20]. Processes can be uniquely identified by a PID, or process identifier. PIDs are unique among processes that are alive on connected nodes[21]. The concept of distributed network and Erlang node are explained in more detail in Section 2.4.1. Concurrent processes communicate between one another through message passing.

Finally, Erlang is a functional language, meaning that unlike languages such as C or Python, Erlang variables are immutable. This means that once it has been attributed a certain value, a variable cannot be changed. This is a security measure in the landscape of distributed applications, as it ensures that all variables are not changed during the execution, despite the interactions with other processes. Moreover, because it is a functional language, there is a programming paradigm in which functions are treated like any other data-type[22], which allows to use them as arguments to other functions, for example. It however means that iteration in Erlang is not done in the same way as other languages like Python or C: instead of using loops like *for* or *while*, iteration is accomplished using recursion.

2.4.1 Distributed network and nodes

As explained previously in Section 2.4, Erlang is well-suited to support distributed and concurrent applications. As will be explained in Section 2.5, it is precisely thanks to this that the Hera platform for sensor fusion came to be, and the principle of distributed network has been re-used in this thesis.

A distributed Erlang network consists of a number of Erlang nodes, which are Erlang runtime systems, that are communicating with each other[23]. A node is typically, in this case, a GRiSP2 board, which has been given a name when deploying the application on the board. The node's name format is an atom "*name@host*". In this thesis, *name* is the name of the application running on the GRiSP2 board, while *host* is the name attributed to the board.

The advantage of such a configuration is the ability to pass messages from one to another, but most importantly, the ability to call a module's function on a node from another one. When two nodes, e.g. *node1* and *node2*, are connected to one another on a network, using the node's name, it is possible to use the RPC module, which stands for Remote Procedure Call and offers services similar to this, to make a call to a function from *node1* on *node2*, for example. This is achieved using the `rpc:call(Node, Module, Function, Args)` function, where `Node` is the name of the

node on which the desired `Function` from the desired `Module` has to be launched with arguments `Args`[3][24].

Beforehand, the user has to make sure that the nodes are connected together: this is achieved using the function `ping/1` from the `net_adm` module and specifying as argument the node with which the current node should try to connect, as such: `net_adm:ping(node_name)`. Finally, note that in order to be able to connect together, two Erlang nodes need to have the same "magic cookie": indeed, when attempting a connection, after node names have been exchanged, the magic cookies the nodes present to each other are compared. If they do not match, the connection is rejected[23].

2.5 The Hera platform: a sensor fusion framework

In the words of the creators of Hera2.0, Hera is an Erlang/OTP framework for handling asynchronous and dynamic measurements[15]. It is a fault-tolerant and distributed framework that can work on multiple connected nodes. It allows data sharing between the nodes, data which can be used locally to accomplish sensor fusion. Sensor fusion, a fast developing area of research in recent years thanks to the increase of availability in number and sensor types[25], is the process of combining data coming from multiple sources, like sensors, in order to obtain an output that is less uncertain than if only one of these sources was used to obtain that output[26]. The framework provides the means to do so by providing Kalman filtering, which is described in more detail in Section C.1.1.

The way the framework is intended to be used is by connecting multiple nodes, typically GRiSP2 boards with one sensor plugged on each. Each board can fetch independently its sensor data by using an appropriate measuring function and then share it with the other boards, using the Hera framework. Then, with all the data, sensor fusion can be done locally on each node using the provided Kalman filters to estimate with a higher reliability a state variable, for example the angular velocity of a toy-train.

It is to be noted that the framework can also work with only one node in the network. Typically, its creators have shown how a single board on which a PNAV is plugged can be used to create a reliable AHRS[15], as already mentioned in Section 2.3.1.

2.6 Gesture recognition

2.6.1 State-of-the-art

Gesture recognition has been an important topic of research for many years. It has many applications, from gaming to monitoring, and is a booming field in engineering thanks to the development of AI and the ever increasing computing power of machines. In the context of this thesis, movements are detected using an IMU. Many applications exist using this technology.

For example, in a study[27], ten volunteers wearing a pair of 3-axis accelerometers and gyroscopes were asked to perform 12 exercises. From the data acquired on the sensors during these exercises, features were extracted and have been used to classify human activities based on novel data from the pair of sensors. The same idea was studied in another study[28], where a 3-axis magnetometer was added to the mix. This study uses the fusion of both statistical and non-statistical features to

[3]Note: for the rest of this thesis, an Erlang function will be of the format `nameOfFunction/numberOfArgs` when the arguments are not specified. Functions in other programming languages will be formatted as `nameOfFunction()`, without specifying the number of arguments.

detect activity patterns based on these three sensors. Using a generic algorithm which processes the features, they achieve selection and classification of these inertial signals to recognise abnormal human movements. Both these studies aim to provide a means of recognising movements in order to give information to the user about what could be wrong, improved upon, or simply for statistical purposes on the physical activity during the day.

Finally, as a last example, in a third study[29], a system using an accelerometer and gyroscope has been implemented to supervise and detect falling movement in real-time, allowing to warn a family member or anyone capable of helping the fallen user. Indeed, elderly people tend to experience a decrease in their core strength and physical quality, especially in the legs, leading sometimes to falls which can have grave consequences if no help is rapidly provided. This accurate system can distinguish between a normal bending motion and an unexpected fall, while also rapidly contacting someone for help.

2.6.2 Implemented algorithm

Last year, a gesture detection algorithm that uses the Hera platform, a GRiSP2 board and a PNAV has been implemented. Its working principle is summarised in Figure 2.5. Note that only the version of the algorithm that allows continuous gesture recognition is detailed here. The implemented algorithm also allows to perform a single gesture recognition over a time period defined by the user, which is useful for debugging purposes. Refer either to the thesis' text[8] or to the application's source code[4] for more detail about this.

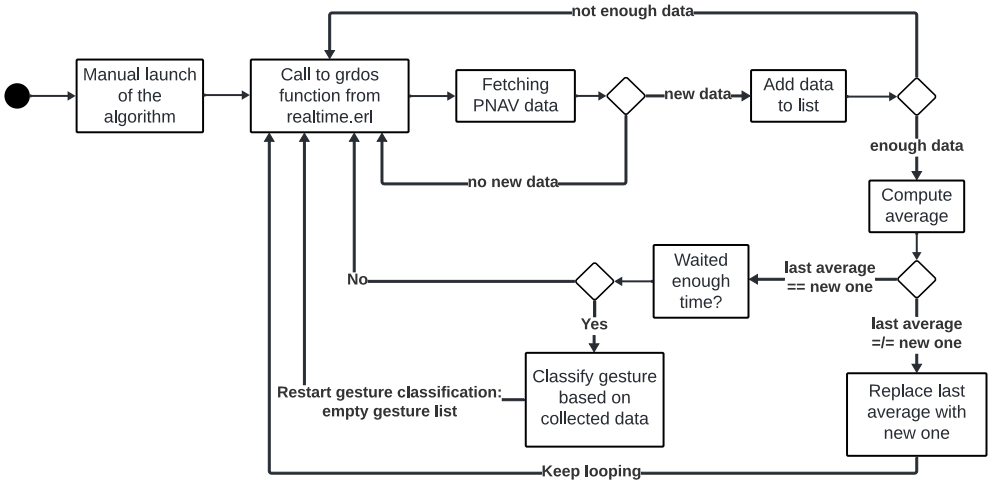


Figure 2.5: Implemented algorithm for continuous gesture recognition. Note that the black dot indicates the start of the algorithm and a rhombus indicates a conditional statement

1. The algorithm starts when the user calls `sensor_fusion:realtime(Time, Period)` in the Erlang VM. The `Time` parameter dictates the minimum time the user has to wait for his gesture to be classified when he stops moving, while `Period` defines for how long the algorithm will run. Any negative number will make the algorithm loop indefinitely. Note that Figure 2.5 illustrates the algorithm working for an infinite amount of time, thus when `Period` is a negative number.
2. The aforementioned function will call the `grdos/11`[5] function from the `realtime` module.
3. This function will fetch data from the PNAV's 3-axis accelerometer and will store it in three different lists, until it has enough data (in this case, 50 measures).

[4] https://github.com/Neackow/movement_detection/tree/main/src
[5] `grdos` stands for `gesture_recognition_division_over_stop`.

4. When it has enough data, an average is performed on the collected data for each axis. This will output an atom for each axis, which can be one of the following five: `nn`, `n`, `o`, `p` and `pp`, which stand respectively for very negative, negative, zero, positive and very positive. This translates the average acceleration that has been detected on the axis over the past 50 fetched data by converting numeric data into a simple atom.
5. From there, two possibilities exist:
   - either all the detected averages on the three axes are the same as the previously detected ones, which means that the board on which the PNAV is plugged has not moved (or barely), in which case the algorithm checks if the elapsed time since the last time a movement was detected is superior to `Time`. If it is not, it performs another average until it is the case. If it is, it classifies the gesture by comparing the detected gesture to a set of pre-existing gestures stored in a file by using pattern matching, outputs the result, empties the list of data and starts again;
   - or any of the detected averages on one of the three axes is not the same as the previously detected one, which means that the board on which the PNAV is plugged is moving, in which case the old averages for those axes are replaced by the new ones and the algorithm keeps looping to collect data.

An updated version of Figure 2.5, which takes into account the changes made to the algorithm in order to adapt it to this project, will be detailed in Section 5.3.6.

2.7 Pulse-Width-Modulation

Pulse-Width-Modulation (PWM) is a technique to get an analog output from a digital mean[30]. A square wave signal is generated by digital control, with the wave switching from high voltage state (typically V_{cc}, the supply voltage) to low voltage state (typically ground, or 0 V) at a given rate. This on-off pattern allows to get a simulated voltage that lies between the supply voltage and the ground. This allows to control the average power delivered to the load[31].

The duration of "on-time", during which the square wave is in high voltage state, is called the "pulse width", thus the name Pulse-Width-Modulation. The shorter it is, the smaller the average output value is, and vice-versa. The duty-cycle, in %, defines the relative amount of time the signal will be on during a period of the square wave[32]. A duty-cycle of 0% outputs a constant voltage of 0 V, while a duty-cycle of 100% outputs a constant voltage of V_{cc}. This is illustrated in Figure 2.6[6].

This method is well suited to control motors, typically, as a motor does not have the time to significantly change its speed between two PWM periods due to its spinning inertia. Beware that at too low frequency, there is a risk that the motor starts surging[32], so it is important to use an appropriate frequency for the given application. In fact, the required switching frequency will highly depend on the load and the application[31]. Generally speaking, the higher the frequency at which PWM is performed, the smoother the control will be.

In this thesis, the Raspberry Pi Pico W, which will be used to control the velocity, will send two digital items of information to the HB5: the direction and the PWM command, coded on 8 bits, and thus comprised between 0 and 255. 0 will lead to a duty-cycle of 0%, while 255 leads to a duty-cycle of 100%. The direction information will select which switches will be opened or closed on the HB5's H-bridge, while the second determines how long the appropriate switches have to be

[6] Adapted from https://mintwithraspberry.blogspot.com/2019/02/raspberry-pi-pwm-duty-cycle.html

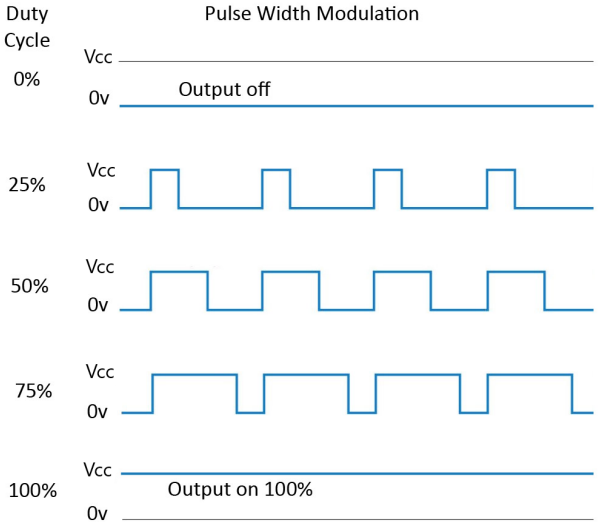


Figure 2.6: Effect of the duty-cycle in PWM

closed within a period, effectively generating the PWM signal and the correct average to be sent to the motor as the supply voltage.

2.8 I2C communication

Standing for *Inter-Integrated Circuit*[7], this communication protocol uses a bus on which one master can control multiple slaves or multiple masters can control one or multiple slaves[33]. Information exchange always happens between one master and one or many slave devices, on the sole initiative of the master[34]. It is a serial communication protocol where data are transmitted bit by bit between devices using two wires: *SDA*, the line to send and receive data, and *SCL*, the line carrying the clock signal, which is always controlled by the master. Furthermore, I2C is synchronous[33].

Data are sent in *messages* that are a series of data frames, like the address frame that allows a slave to know that the master wishes to communicate with it. This is called *addressing*. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame[33], as illustrated in Figure 2.7.

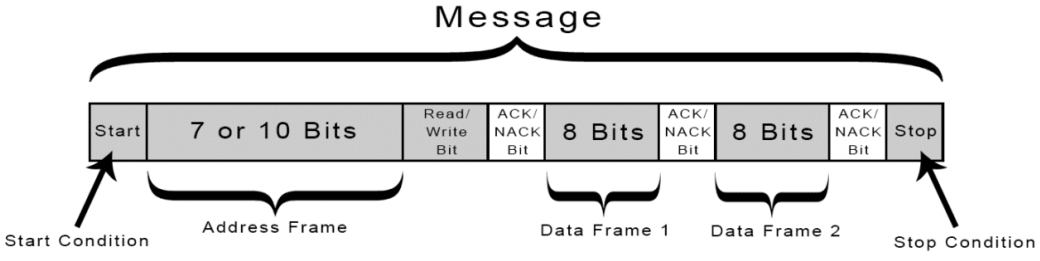


Figure 2.7: Messages in the I2C communication protocol

A master can communicate in two ways with the slave: either the master is requesting data from the slave, labelled the *read* operation, which corresponds to a high voltage level on the Read/Write Bit in Figure 2.7, or the master is sending data to the slave, labelled the *write* operation, which corresponds to a low voltage level on the same bit[33].

[7]Note that the official abbreviation is I²C, but it was simplified to I2C for this thesis.

Chapter 3

The controllable object: mechanical and electrical description of the robot

This chapter and the following two will explain in more detail what has been done during this thesis:

- Chapter 3 focuses on the mechanical and electrical aspects of the robot, from its hardware to the design choices. It starts by explaining the specifications that the prototype has to fulfil before detailing the mechanical and the electrical choices that have been made to meet these specifications.
- Chapter 4 goes into the low-level controller's code, starting by explaining how encoders work and the problems that were faced, then explaining the theory of PID controllers and the implemented PI controller, followed by how the velocity is computed. The chapter ends by explaining the code of the low-level velocity controller.
- Chapter 5 details the GRiSP application developed during this thesis, which has been named *movement_detection*. The chapter starts by explaining the limitation of the previously developed gesture recognition algorithm, then details the whole code flow and the application's modules before going over the specifics of the robot's control routines, a full description of the gestures and associated command and the security measures that have been added to the system.

About this chapter: it is organised as follows:

- Section 3.1 reminds the reader of the goal the robot tries to fulfil and details the specifications of the prototype that will be the guidelines to build the robot.
- Section 3.2 fully details the mechanical aspects of the robot, from the wheels to the different components that it is based on, and justifies the components' placement.
- Finally, Section 3.3 explains the electrical aspects of the robot by going over the circuitry of the robot, explaining the hardware used and how it is incorporated into the prototype and showing the way the robot deals with power management.

3.1 Goal and specifications

As the primary goal of this thesis is to control an object remotely using gestures, it is first necessary to have a controllable solution at hand. Several options came to the drawing-board, e.g.:

- a window that could be opened or closed at will by pointing at it;

- a rolling robot whose behaviour would be decided by gestures;
- or a modified toy, like a remotely controlled race-car, acting just like the aforementioned example.

The window idea was deemed too limited, both in design perspectives and applications. Moreover, in order to have full control over the design process, it has been decided to build a prototype from scratch, instead of using an already existing toy-train, for example. Finally, instead of just being controllable, it was decided to make it possible for the robot to carry objects around.

Considering all this, it has been chosen to build a robot on wheels, from scratch, which could carry objects around using a plate.

3.1.1 Specifications of the robot

Before making the prototype, it is essential to define its specifications. The following section details them. The methodology used is inspired by the one taught in the *LMECA2801 - Machine Design* course[35]. One can define its Main Functions, Constraint Functions and their Requirements. For each function or constraint, their requirements are defined below them. A requirement specifies what is needed to achieve the function or constraint.

Main Functions

These describe the actions that the controllable object will be required to perform:

1. Carry objects around:
   - 1.1 have a storage plate;
   - 1.2 be stable;
2. Move in a 2D environment:
   - 2.1 move freely on flat ground with no obstacles;
   - 2.2 actuation using motors;
   - 2.3 a minimum of 2 motorised standard wheels, or 1 motorised omnidirectional wheel;
   - 2.4 depending on the number of motorised wheels, 1 or 2 freebear wheels for stability;
   - 2.5 avoid hyperstaticity.

Constraint Functions

These describe the constraints with which the design of the controllable object has to deal.

1. Usage of a GRiSP2 board:
   - 1.1 need of a 5 V power source;
   - 1.2 need of space for the board (13×7.1×1.5 cm at minimum) ;
   - 1.3 any hardware communicating with the GRiSP2 board should be able to interface it ;
   - 1.4 no cabled connections to something outside of the robot;
2. Easily reproducible “off-the-shelf” prototype:
   - 2.1 use as little extra hardware as possible;

2.2 low-cost components;

2.3 use 3D printing for special parts;

3. Usage in a limited space:

3.1 easily manoeuvrable;

3.2 slow motors for accurate control;

4. Be user-friendly:

4.1 manual emergency-stop button;

4.2 lightweight, to be able to carry the prototype (< 5 kg);

4.3 intuitive design;

4.4 low voltages (< 20 V).

With all this in mind, it is time to start building the prototype.

3.2 Zoom-in on the mechanical aspects of the robot

The first step was to think of a structure that fulfils the main functions and constraint functions. To shorten the process, it was decided to start with an already existing structure, namely, an empty wine crate. Weighing initially 1.675 kg and having dimensions 33×26.1×16.7 cm, this crate offered a nice first skeleton. Indeed, the inside of the crate could be used to place all the hardware and thus hide it from the outside, while the top of the crate could be used as a support to carry objects around. It fulfils the constraints of having an intuitive design and that it is lightweight.

Concerning the top of the crate, its initial lid was deemed too complicated to work with. A new lid was thus cut from a wooden plank at the Makilab, and added to the crate using a hinge, giving easy access to the inside of the crate while fulfilling the need of having a plate to carry objects.

Starting from such a structure, despite having some advantages, comes with big challenges:

- The main challenge is that the structure is fixed: It cannot be made bigger, nor can its shape be changed or adapted. This can increase the complexity of the design process, since the wine crate has not been made to fulfil such a purpose. It however turned out to not be a problem, since the crate had a lot of available space inside and that there was not a lot of hardware required for this prototype, which made it easy to place everything on the bottom plate of the crate.
- One mistake would be catastrophic: place the motor at the wrong place and the design may be doomed, since it is required to drill a hole through the wood to fix it. Since the structure is pre-existent, the placement of the motors, the different pieces of hardware, the freebear wheel, and so on cannot be carefully planned in advance, so a lot of care needs to be taken in correctly placing everything on the first try.

3.2.1 Choosing the wheels

First, it was necessary to choose which type of wheel to use. There are two main types of wheels for such an application: the standard wheel and the omnidirectional wheel, as visible respectively in Figure 3.1a[1] and Figure 3.1b[2].

[1]Image from https://www.pololu.com/product/1420

[2]Image from https://en.wikipedia.org/wiki/Omni_wheel

(a) Illustration: Standard wheel

(b) Illustration: Omnidirectional wheel (omniwheel)

Figure 3.1: Illustration of the main possible types of motorised wheels for the prototype

Figure 3.2: Illustration: the freebear wheel

The first type, the standard wheel, is typically made of plastic or hard rubber, and it offers stability and durability[36]. It is a solid choice to manoeuvre in rough terrains or carry heavy loads. The second type, the omniwheel, is designed to allow the robot to move in any direction in a 2D plane by using roller wheels added to the main one. These wheels remove the need of steering mechanisms, as they allow lateral movements. They allow easier navigation in tight spaces and can maintain precise positioning.

Many possible wheel combinations exist, but the possibilities will be limited by the fact that it is of paramount importance to avoid being hyperstatic. Hyperstatism happens when a mobile robot is more constrained than is strictly necessary for its balance, which means that at least one degree of mobility is suppressed[37]. If rigid models are used, this leads to the appearance of non-controllable (possibly infinite) internal forces[38] and the impossibility of having a correct motion. Three wheels are sufficient for static stability[39], and any additional wheels need to be synchronised. For example, it would be possible to add a freebear wheel (visible in Figure 3.2[3]) to the three others, but to avoid hyperstaticity, it would be required to put it on dampers. To ease the design, only designs with three wheels will be considered.

Two possibilities came to mind: the differential drive robot (two standard wheels and one freebear wheel) and the omnidirectional robot (three omnidirectional wheels) designs, illustrated respectively in Figure 3.3a and Figure 3.3b. Despite the omnidirectional robot fulfilling perfectly the constraints of manoeuvrability, in the end it was decided to use the differential drive for the following reasons:

- For a first design, it was deemed unnecessarily complicated and counter-intuitive for the user to use omnidirectional wheels.
- The design and control of a differential drive is easier than that of the omnidirectional robot.
- There is no need of a holonomic robot[4].

[3]Image from https://jswallong.en.made-in-china.com/productimage/pmJUaSzBoRYE-2f1j00sGIkjRcPbFuL/China-Freebear-Unit-Ball-Carrier-Freebear-Conveyor-Ball-Rollers-Suitable-for-The-Labor-Saving-of-Conveying-Lines.html

[4]Holonomic robot means that the controllable degrees of freedom at any instant match the total degrees of freedom for the robot. An advantage of holonomic robots is that the holonomic kinematic constraints can be expressed as an explicit function of position variable only, making it easy to use, whereas non-holonomic constraint requires a different relationship, such as the derivative of a position variable. In the present case, such considerations are beyond the scope of the prototype, and thus only the ease of prototyping has been considered when choosing between holonomic

(a) Differential drive robot (b) Omnidirectional robot

Figure 3.3: Illustration of the two possible wheel placement configurations

So overall, manoeuvrability was partially sacrificed for the ease of control and intuitive design. The differential drive robot has two standard separately driven wheels placed on either side of the robot's body. There will be no need of a steering mechanism, as turning around can be accomplished by varying the relative rate of rotation of its wheels[40]. Moreover, this design offers two degrees of manoeuvrability $\delta_M = 2$, which relates to the degrees of freedom[5]. This design is thus able to move in a 2D plane, as required.

3.2.2 Placing the wheels

There are thus three wheels to be placed: two motorised standard wheels and one freebear wheel. It has been decided to follow the schematic visible in Figure 3.3a, with the two motorised wheels at the back and the freebear wheel in front, in the middle. This ensures a good stability of the robot. It is important to notice that by placing the wheels on each side of the crate as visible in Figure 3.4, the manoeuvrability of the robot is reduced, as it will take more time and distance in order for it to turn. Stability has been here chosen over manoeuvrability, as putting the wheels closer to the centre would have meant a smaller supporting surface and more imbalance.

Figure 3.4: Placement of the motorised wheels, at the far back of the robot. The freebear wheel is placed in the middle-front of the robot

The wheels used are the Pololu Multi-Hub Wheels, 80×10 mm[6]. It must be noted that the bigger the wheel, the harder it is to manoeuvre the robot, as the distance travelled per output

or non-holonomic robot designs.

[5]The degree of manoeuvrability is defined as $\delta_M = \delta_m + \delta_s$ where δ_m is the degree of mobility and δ_s is the degree of steerability. As there are no steering mechanisms, $\delta_s = 0$ and due to the placement of the standard wheels on opposite sides of the crate, $\delta_m = 2$ instead of 3, because there is a constraint, called no-slipping constraint, which means that it cannot go left/right but can go forward/backward and turn on itself with ease.

[6]https://www.pololu.com/product/3690

shaft rotation is bigger. These wheels were chosen as they were perfectly adapted to the chosen motors, which are detailed in Section 3.2.3. Bigger wheels were also needed to avoid too much tilting of the robot, due to the freebear wheel in front. Despite this, there is still a tilting of 0.83°, but that is negligible.

3.2.3 The motors

The motors used in this prototype are the Digilent gearbox DC-motors IG220053X00085R, visible in Figure 3.5. These motors come with an integrated gearbox of 1:53 and can be supplied by a constant 6 V power source at most. They also come with integrated quadrature encoders, which use a 3.3 V supply and are described in more detail in Section 4.1. The motor rated velocity, while loaded, is 125 RPM (Rotations Per Minute) ± 15%[41].

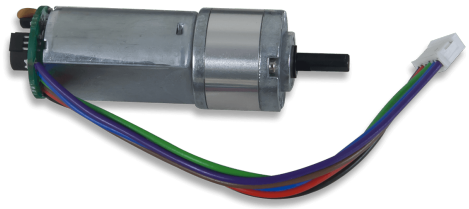


Figure 3.5: Digilent DC-motor used for the prototype

These motors were chosen for their simplicity, low-cost, low-power, integrated quadrature encoders which allow monitoring of the robot's velocity to control the motors appropriately, and also due to their immediate availability.

3.2.4 Other component placements

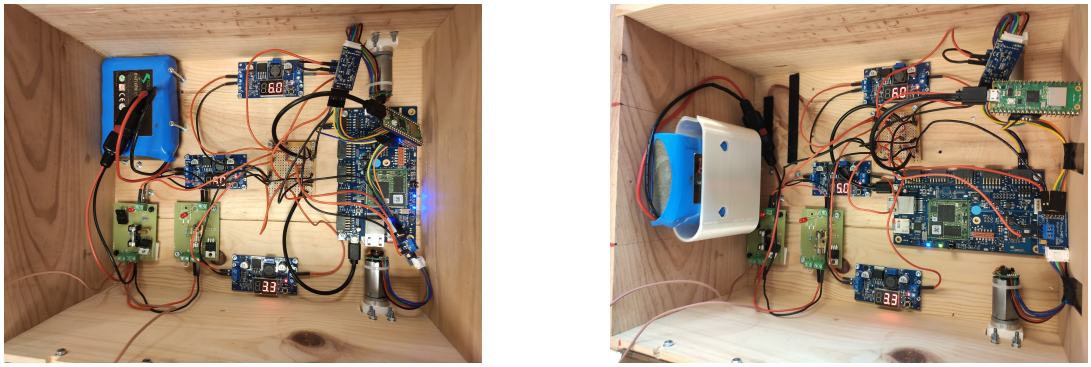
Many other components were added to the crate during its development. As each piece of hardware was lightweight, no special consideration regarding its placement was taken into account. However, two components needed to be carefully placed in order to reduce the weight carried by the standard wheels: the emergency-stop button and the battery.

Emergency button

This is only required in case of emergency, where the user presses the button to cut the power supply to the motor. In order to deal with the sudden weight of the hand pressing the button, it will be placed right on top of the freebear wheel, on the crate's lid, so that the freebear wheel absorbs as much of the supplementary weight as possible.

Battery

It is the heaviest of all the components, weighing 0.387 kg. Initially, the battery was placed on the front right side, flat on the bottom of the crate, as visible in Figure 3.6a. The idea was again to use the freebear wheel to carry as much of its weight as possible. It turned out however to be a bad idea: upon analysing the PWM command sent to the motor, it was discovered that for the same desired velocity, the controller was outputting significantly different commands, with a difference of around 30 PWM between the two wheels, the right one receiving the highest command. In order to reduce this value, the battery was moved towards the center, carried by a 3D-printed lightweight support which is fixed to the wall of the crate as visible in Figure 3.6b. This did not solve completely the aforementioned problem, but reduced it by around 10 PWM on average.



(a) Version 1 of battery placement: in the front-right

(b) Version 2 of battery placement: in a dedicated support placed in the front center

Figure 3.6: Illustration of the different placements of the battery inside the crate

Due to the symmetry of the crate, it was hypothesised that the remaining difference was due to the wheel placement: upon looking at the distance between the contact point of the wheels on the ground and the bottom of the crate, a small difference of 2 mm was found, with the right wheel protruding more from the crate. It is assumed that because of this discrepancy, the right wheel is carrying more weight than the left one, and thus, to reach the same velocity, needs a higher PWM command.

The final result, with all its hardware and mechanical components, weighs 2.75 kg, which is far below the fixed limit of 5 kg. Pictures of the robot can be found in Appendix C.2.

3.3 Zoom-in on the electrical aspects of the robot

In this section, the electrical aspects of the robot are explained in detail. Before considering in depth the robot's electronics, it is interesting to first look at the complete system from an electrical standpoint.

3.3.1 Electrical schemes

Overview of the whole electrical system

The global electrical scheme of the whole system can be found in Figure 3.7.

Note that this electrical diagram has been simplified by not considering the voltage management board and that, to ease the representation, the outputs of the buck converters were directly connected to what they power-up, which is *not the case in reality*. For a complete description of how the voltage is managed and distributed across the robot, refer to Section 3.3.3.

The system is composed of three parts:

1. A GRiSP2 board on which is plugged a PNAV, used to detect the gesture made by the user. For the rest of this thesis, this GRiSP2 will be referred to as the *detector*.
2. A monitoring computer to which the aforementioned GRiSP2 board is connected, both for supply and monitoring purposes. It is used to launch the commands and to check the state of the program during execution.
3. The controllable object, whose mechanical aspects have been described in Section 3.2, and whose entire hardware is shown in the schematic, except for the voltage management board

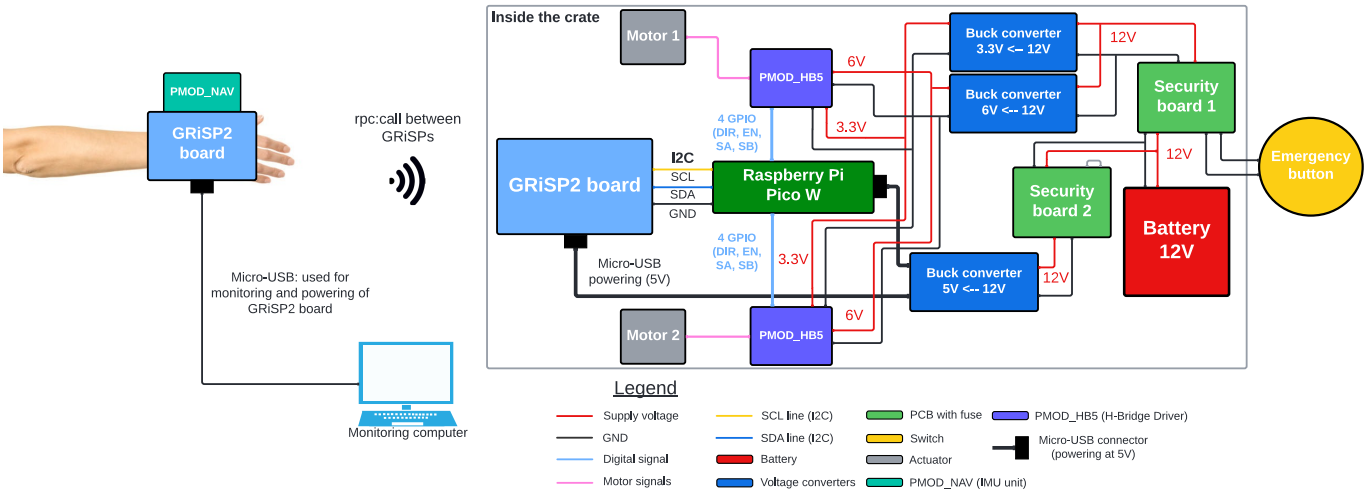


Figure 3.7: Global electrical scheme of the system

as previously explained. It contains the second GRiSP2 board, which will be referred to as the *receiver* for the rest of this thesis.

The monitoring computer powers up and communicates with the detector via a micro-USB cable, whilst the detector communicates with the receiver using the `call` function of the `RPC` module of the Erlang/OTP framework, which itself uses the Wi-Fi network to which both GRiSP2 boards are connected to transfer the data.

The robot's electrical scheme

An enlarged version of Figure 3.7 which focuses on the robot's hardware can be found in Figure 3.8.

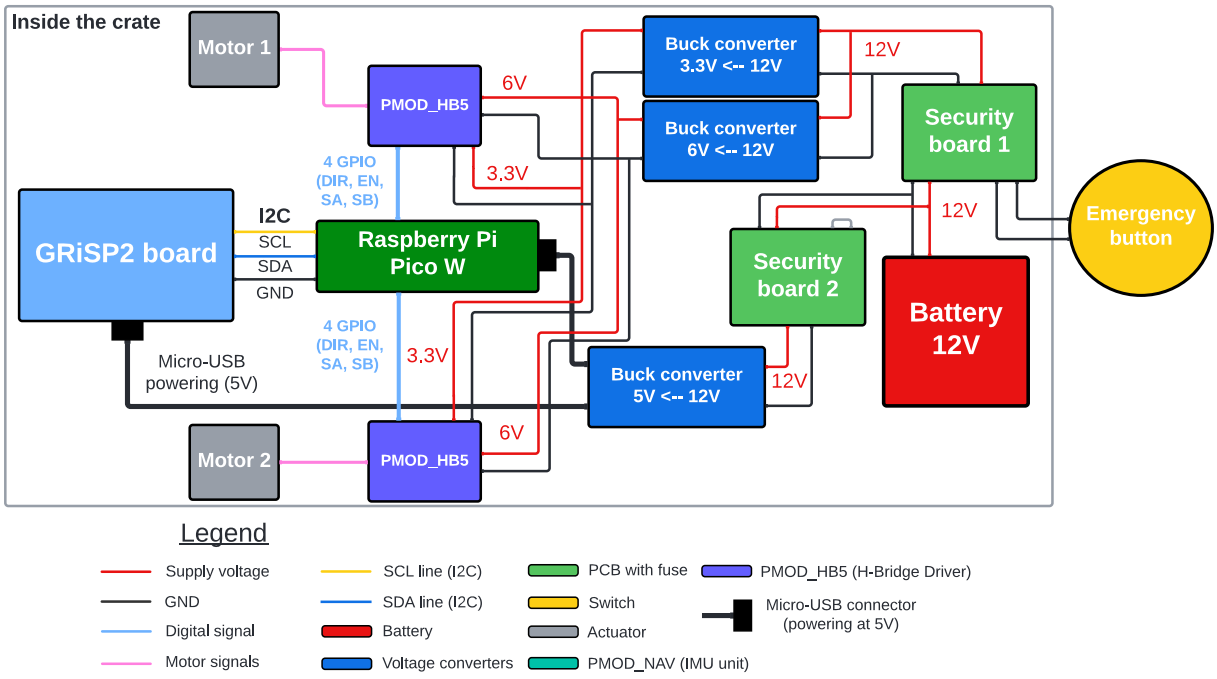


Figure 3.8: Focus on the electrical scheme of the robot

All connections are made using AWG24 sized wires using Dupont connectors, except for the battery cables and the connectors to these cables which use AWG18 sized wires.

The rest of the hardware, which is described in more detail in Section 3.3.2, consists of:

- One GRiSP2 board, also known as the receiver;
- Two motors with integrated encoders, presented in Section 3.2.3. The encoder part will be described in more detail in Section 4.1;
- Two HB5, which have been described in Section 2.3.2;
- One Raspberry Pi Pico W board;
- Three buck converters;
- Two home-made security boards;
- One battery;

3.3.2 More detail about the hardware and connections

In this section, the hardware, its logic level, supply voltages and what it is used for, are explained in more detail.

The GRiSP2 board: the receiver

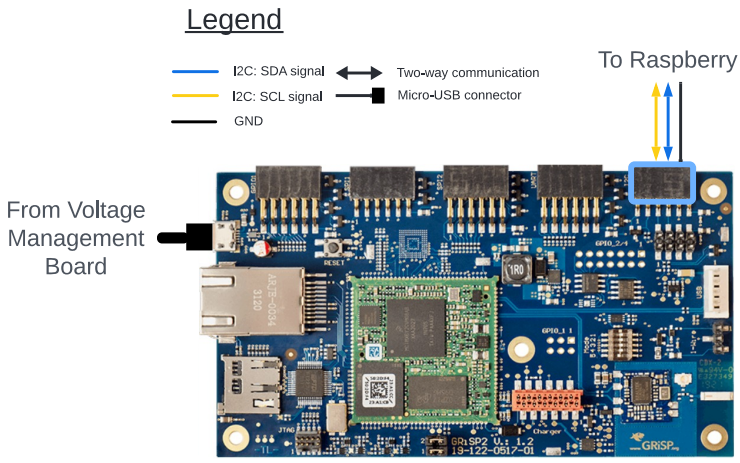


Figure 3.9: Pins used on the GRiSP2 board

The performance of this component has already been detailed in Section 2.1. It is powered via a micro-USB cable at 5 V voltage. Its purpose is to receive the recognised gesture detected by the detector and convert it into the appropriate command to be sent to the Raspberry board. It communicates with the latter using I2C communication, which has been explained in Section 2.8. There are three wires: SCL, SDA and a ground wire, used to have a common ground on both boards in all circumstances to avoid any problems with the I2C communication due to varying grounds. The logic level for the SCL and SDA signals is 3.3 V. These connections are made using the I2C port of the receiver, as highlighted in Figure 3.9.

It is important to note that in this I2C communication, the receiver acts as an I2C master, while the Raspberry Pi Pico W is the slave.

The motors and encoders

They are used for the propulsion of the robot and to get a feedback on the behaviour of the wheels, respectively. As already explained in Section 3.2.3, the motors need to be supplied by a 6 V voltage, at most. In fact, they will be actuated by receiving PWM commands, which are converted by the HB5s into the appropriate square wave. The motors and encoders receive four signals: MOTOR+ and MOTOR-, which are respectively the motor analog input and ground pin, and $V_{cc} = 3.3$ V and a ground signal, fed to the encoders. Finally, the encoders output two square

waves, A and B, labelled SA and SB respectively, for a total of six wires between a motor and a HB5.

The HB5s

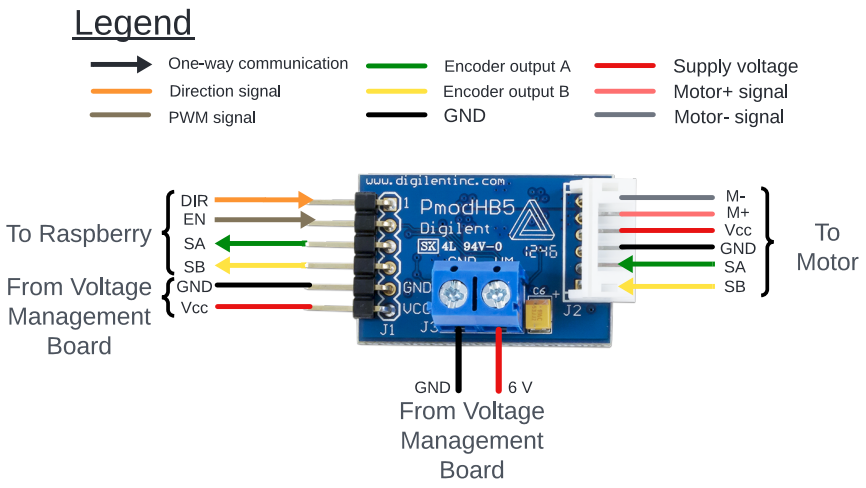


Figure 3.10: Summary of all the connections on the HB5. Note that $V_{cc} = 3.3$ V

They are used as a relay between the motors and the Raspberry board for the encoders, as well as acting as a driver for the motors. They receive a constant 6 V voltage and ground signal via two screw terminals, which act as the bounds for the H-bridge, and they receive a 3.3 V voltage (V_{cc}) and ground signal on their left pins to supply the encoders. From the Raspberry board, they receive two signals : DIR, which is used to determine in which direction the motor has to spin and EN, which is the PWM command which will be converted by the H-bridge into MOTOR+ and MOTOR-, the appropriate voltage levels to be sent to the motor. The HB5s themselves send the output square waves from the encoders to the Raspberry board, as signals SA and SB.

A summary of the connections on a HB5 can be found in Figure 3.10.

The Raspberry Pi Pico W

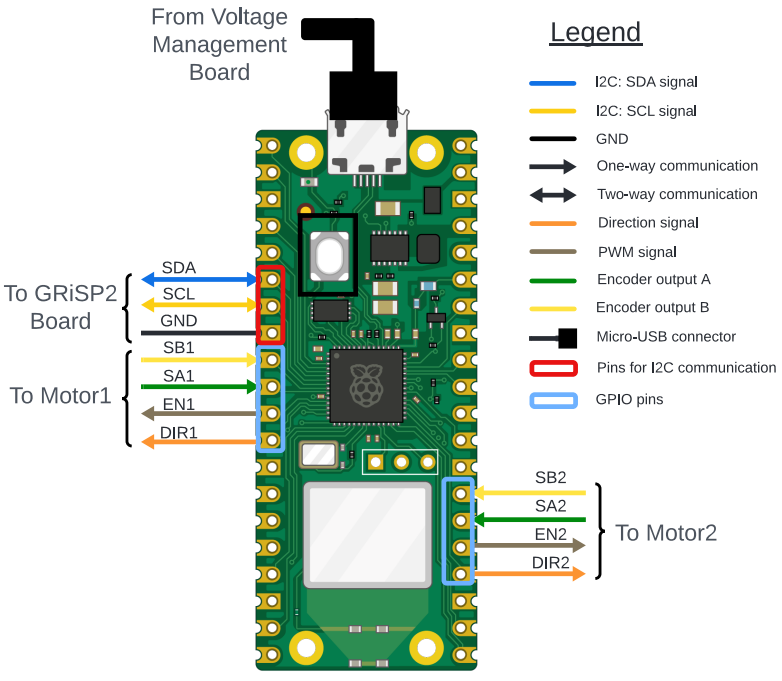


Figure 3.11: Pins used on the Raspberry board

This micro-controller is the main "brain" of the robot. Note that, in the first design version, an ESP32 NodeMCU-32S was used as the main micro-controller, but it was later replaced by the current board. An explanation of the reason behind this change and a description of the Raspberry Pi Pico W hardware can be found respectively in Section 4.2 and Appendix A.

This board is used to control the velocity of the robot. The velocity controller will be detailed in Section 4.5. Moreover, as will be detailed in Section 5.4, beyond controlling the robot's velocity, the Raspberry board's code also contains some routines for some special movements.

A summary of all its connections to other boards is detailed in Figure 3.11. All signals have been previously defined.

The pins used are the following:

- Pins for motor 1 (left motor): 12, 11, 10 and 9 (or GPIO 9, 8, 7 and 6) respectively for signals DIR1, EN1, SA1 and SB1.
- Pins for motor 2 (right motor): 24, 25, 26 and 27 (or GPIO 18, 19, 20 and 21) respectively for signals DIR2, EN2, SA2 and SB2.
- Pins used by the I2C communication: 6 and 7, respectively for SDA and SCL. Note: these pins are multi-usage, but are the default pins for I2C communication, thus they are used for it.

The board is supplied by a micro-USB cable at 5 V. The board itself does not power anything. Finally, note that the Raspberry board acts as the slave for the I2C communication.

The buck converters

Three buck converters are used to convert the 12 V supply voltage from the battery to the appropriate supply voltages of each component.

Figure 3.12: Buck converter used

It is important to note that *no immediate connections* exist between the output of the bucks and the devices to power-up, as each output is connected to a supply voltage management board. A complete explanation of the supply voltage management of the robot can be found in Section 3.3.3.

Three bucks were used to obtain three different voltage levels, as:

- the encoders require a 3.3 V supply voltage;
- the HB5s' H-bridges require a 6 V supply voltage;
- the Raspberry and GRiSP2 boards require a 5 V supply voltage.

The home-made security boards

These boards are used as a security measure. Placed between the battery output and the rest of the circuit, they ensure that any short circuit inside the robot's circuitry does not lead to a battery meltdown, and protect the circuit against battery failures by making the current flow through a fuse, which will explode the moment that a problem happens.

These boards come with a unique characteristic: two screw terminals need to be short-circuited in order for the board to transmit the power from its input to its output. This means that in order to shut down the power supply, all one needs to do is disconnect these two screw terminals. This is what is used by the emergency-stop button: the button is in a normally closed (NC) position at rest but when pressed, the circuit becomes open and the two screw terminals are disconnected from one another.

However, this is not ideal for the supply of the GRiSP2 and the Raspberry board, as both have a booting period. Especially the GRiSP2, which takes on average 22.14 s to boot[7], launch the application and connect to the Wi-Fi. This is why a second security board was added. This second board is constantly short-circuited and deals with the supply of the GRiSP2 and Raspberry, allowing to avoid having to reboot every time someone presses on the emergency-stop button.

The battery

A single $LiFePO_4$ battery from Solise is used to supply the entirety of the robot. It is a 12 V, 3200 mAh battery whose voltage is an input to the security boards. It is thus not in direct connection with the rest of the circuit, as it is separated from it by these security boards.

3.3.3 Supply voltage management of the robot

The robot's supply voltage management is done entirely by a single perfboard. As can be seen in Figure 3.13, a perfboard is like a PCB made of many holes, which are electrically conductive[8]. The holes can be connected by soldering.

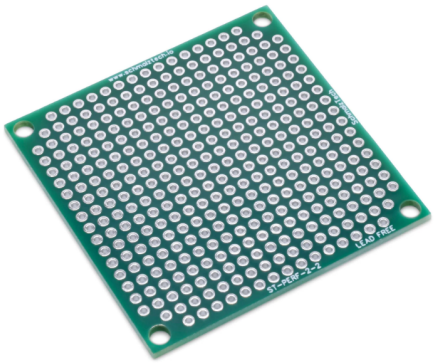


Figure 3.13: Illustration: the perfboard

It was decided to use this board as the main supply voltage management unit because of how easy it made it to multiply one particular voltage. Instead of soldering many wires together and having trident like wires, it was deemed cleaner to use this board. It also allowed to centralise everything to one single unit.

The complete voltage distribution of the robot is detailed in Figure 3.14. Since the motors and encoders supply voltages and the GRiSP2 and Raspberry board supply voltages are separated at the battery, it could have been possible to split this board in two, but it was not deemed necessary.

First, let us look at the motors and encoders part:

- The 12 V battery output goes through the first security board. When the emergency-stop button is not pressed, the voltage passes through it and reaches the perfboard. This is shown in the lower right corner of the schematic.
- From there, two wires leave the perfboard, each at 12 V voltage. They enter respectively the 12 V $\longrightarrow$ 3.3 V and the 12 V $\longrightarrow$ 6 V buck converters.
- The outputs of these two buck converters are then fed back as wires with the new voltage levels to the perfboard and soldered to it, creating two new "supply voltage blocks". This is shown in the middle-right of the schematic, where the 3.3 V block is visible on top and the 6 V block is visible on the bottom.

[7]This value was found by powering-up the board five times and computing the average time it took for everything to launch.

[8]Image from https://www.schmalztech.com/products/2-x-2-perfboard

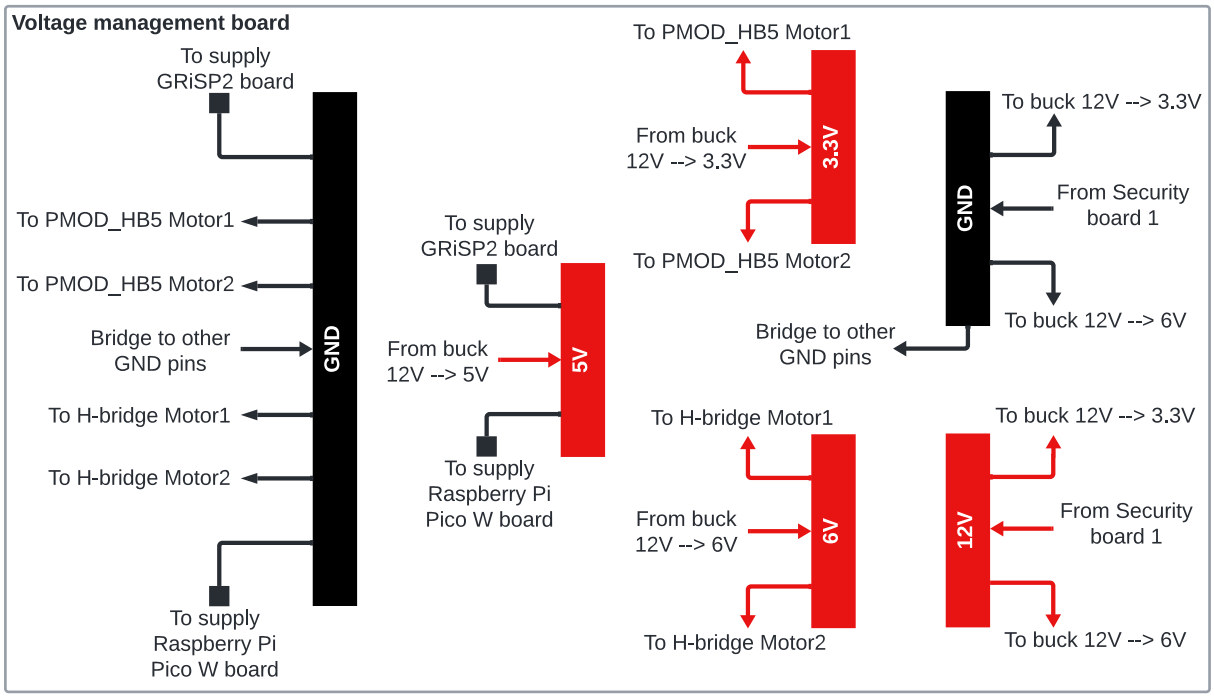


Figure 3.14: Supply voltage management schematic

- From the 3.3 V supply voltage block, two wires exit to supply the two encoders.
- From the 6 V supply voltage block, two wires exit to supply the HB5s' H-bridges.

This whole part can be fully disconnected from the battery by pressing the emergency-stop button: both the encoders and the motors will stop working.

Next, there is the GRiSP2 and Raspberry boards part:

- The battery input has been doubled by soldering two wires into one just before the first security board. This second battery input is plugged to the input of the second security board which, as can be seen in Figure 3.8, is constantly short-circuited.
- The output of the security board is directly plugged into the 12 V $\longrightarrow$ 5 V buck converter, whose output is then wired and soldered to the perfboard. This is shown in the middle-left of the schematic.
- From this block, two special wires, with a micro-USB plug at the other end, leave the board to supply the GRiSP2 and Raspberry board.

Finally, the ground (GND) signals are all the same and are coming from the battery ground. To make it easier to have enough ground wires, two ground blocks were created. On the top right corner of the schematic, as one can see, the ground output of the first security board is connected to the perfboard. To create a second zone, a wire has been used to make a "bridge" between this block and another bigger one, shown in the schematic on the left side. All the required ground signals leave the perfboard from these two blocks.

Chapter 4

The low-level control of the robot: a software solution

This chapter presents the low-level of the application's software, which is used to control the robot by directly interacting with the motors. The encoders and the theory behind the controller are presented and the implemented controller is detailed.

The chapter is organised as follows:

- Section 4.1 details how the encoders that come with the DC-motors work and how they can be used to monitor the spinning direction and speed of the motor.
- Section 4.2 explains the problems that were faced when trying to use an ESP32 NodeMCU-32S to fetch the encoder counts and the reason why a Raspberry Pi Pico W is used instead.
- Section 4.3 goes over the theory of PID controllers and then describes the PI controller that has been implemented in this thesis.
- Section 4.4 specifies how the velocity of the wheels is computed and processed to be used as input to the low-level controller.
- Finally, Section 4.5 shows how the code of the implemented low-level velocity controller works and explains some design choices.

4.1 The encoders: working principle and resolution

As mentioned previously, the motors used come with built-in rotary encoders which are quadrature encoders, a type of position sensor. By observing changes to the magnetic field created by a magnet attached to the motor shaft, these encoders are generating two signals: signal A and signal B, generated by two Hall-sensors which are 90° apart[42]. The two signals are thus 90° out of phase[43]. These outputs trigger periodically, generating a square wave as output. Depending on the rotation direction, output A will trigger before or after output B, when the motor respectively spins in clockwise or in counterclockwise direction. This is illustrated in Figure 4.1.

These two signals can be used together to measure the speed and direction in which the motor spins[44]. First, the direction is found by looking at the value that the B signal has each time the A signal transitions. If, upon rising and falling, A sees B respectively as low and high, this means that the motor shaft spins in clockwise direction. On the other hand, if, upon rising and falling, A sees B respectively as high and low, the shaft is rotating in counterclockwise direction.

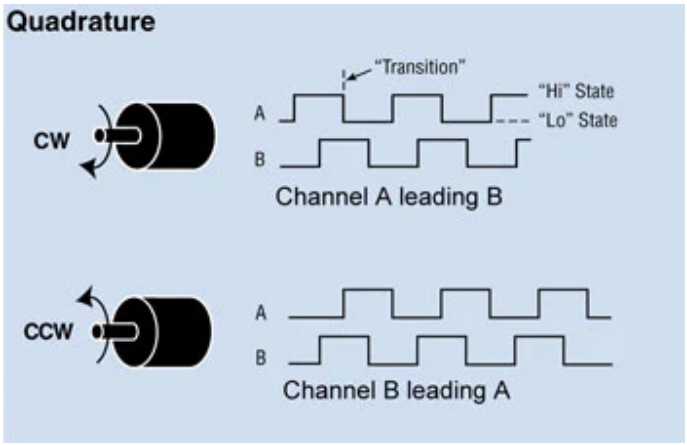


Figure 4.1: Working principle of the rotary encoders

To correctly take into account the spinning direction, within the controller code, a counter, hereafter referred to as the encoder count, is incremented when the motor rotates in clockwise direction, and decremented when it rotates in counterclockwise direction.

From the datasheet of the supplied encoders[45], it is found that the encoders have a low resolution of 3 PPR, or Pulse Per Rotation. This means that for every full magnet rotation, the signals will only trigger three times[46]. However, as the motors come with a gearbox of 1:53, the PPR of the output shaft is actually $3 \times 53 = 159$ PPR. To further increase the PPR, one can compare the two signals both when A rises and falls. This is what is done in the code, as one can see in Appendix F.10, leading to a resolution of 318 PPR for the output shaft.

Finally, it should be mentioned that rotary encoders can suffer from a phenomenon known as bouncing. This happens when the output A or B changes abruptly for a short period of time before returning to its correct value, for example, a sudden drop from high to low state followed by an almost immediate rise back to high state. This can be problematic, as the counter would be updated twice if such a glitch on the signal happened. To deal with such a problem, solutions exist, such as a Finite-State-Machine to correctly switch between the states or other software solutions[47]. Luckily, it was observed that this phenomenon was not happening on the given encoders. A test where approximately three wheel turns were manually done led to an encoder count of 951, which is very close to the expected $3 \times 318 = 954$ ticks.

4.2 The struggles of the micro-controller: switching from an ESP32 to the Raspberry Pi Pico W

The first version of the robot's hardware was using an ESP32 NodeMCU-32S. However, for some unforeseen reason, this led to some struggles with the encoder count. Using a basic code to update the encoder count, which is based on attaching an interrupt on the pin connected to channel A that triggers whenever a change in value is detected and assigning an Interrupt Service Routine function to that interrupt which would then, depending on the value of channel B, decide whether to increment or decrement the counter, the encoder count would only increase by blocks, as can be seen in Figure 4.2a.

Figure 4.2a shows that the value of the counter is sometimes correctly incremented, whilst at some other times it is not and evolves by blocks. Note that the values shown are those of a variable named *pos*[2], which is a list of two integers updated at each loop iteration using a special command to block the interrupts when the value is being fetched and then immediately re-activating them

(a) With the ESP32: the encoder count evolves by blocks

(b) With the Raspberry Pi Pico W: the encoder count evolves as wanted, by steps of one

Figure 4.2: Results of a test made on the encoders

afterwards. This is a safe way of accessing the encoder count value. Indeed, not using such a block may result in unforeseen events where another interrupt may happen just when the value of the variable is being read. It is preferred to eventually miss some increments than to have a potential misread problem.

This evolution of the encoder count was surprising. It was quickly discovered that the effect was worse the faster the motor spun, as the steps would progressively increase depending on the speed. After a thorough research on the Internet, no valid culprit or solution was found.

One possible explanation is that the ESP32 was struggling with the speed at which the interrupts were coming in and that it led to the value evolving by blocks. This is however hard to believe, as many projects encountered on the Internet while searching for an answer to this problem had interrupts triggering at frequencies above 200 kHz and using an ESP32.

It is said in this manual[48] that the stack in memory stores local variables and information from interrupts and functions. So one hypothesis for why the encoder count incrementation is not working could be the following: the stack sees interrupt information coming in continuously while the code is being run on the ESP32 which could lead to some tasks being executed in priority before all the registered interrupts were taken into account. This, again, would be strange, as the very definition of an interrupt is that it is a signal emitted when a process or event needs immediate attention, so it alerts the processor to a high-priority process requiring interruption of the current working process[49]. So this hypothesis of the interrupt calls being stored in the stack and treated all in one block seems hard to believe.

Whatever the reason, after discussing with some peers, one proposed to try out the same code on a Raspberry Pi Pico W, which led to the desired behaviour: the encoder count was incremented by steps of one, as visible in Figure 4.2b. Note that the same value can be printed several times, as can be seen in Figure 4.2a as well. This is due to the fact that the `loop()` function from the Arduino IDE is too fast compared to the speed at which the encoder count evolves. This is not a problem, since the velocity will only be computed every 18 ms, as explained in Section 4.4.

Besides this problem, other strange behaviours that motivated the change were observed on the ESP32 and not on the Raspberry Pi Pico W board:

- For some unknown reason, on the ESP32, the encoder count would not always be incremented but sometimes decremented, from one test to another, despite using the exact same code and

being asked to always spin in the same direction. This effect was not observed on the Raspberry board.

- As a security measure, to avoid being at risk of short-circuiting the H-bridge whenever there is a change in direction from one loop iteration to the other, the function which sends the command to the HB5 starts by briefly putting the PWM command to zero before applying the correct direction and PWM command. This, however, turned out to be impossible to implement using an ESP32 without adding a delay before and after that command.
- And finally, upon removing this protection and the delays, the motor stopped spinning. However, printing some random data to the serial monitor made the code work again. This means that on the ESP32, a delay is required in order for the command to be correctly sent to the HB5, which makes it not practical to use to drive the motors.

For all these reasons, it was decided to change the micro-controller to the Raspberry Pi Pico W.

4.3 PID control

4.3.1 Theory of the PID control

A popular and simple choice to control processes ever since its creation, PID control (which stands for Proportional-Integral-Derivative control) is a control mechanism which allows a *process value* to reach a fixed target, called the *setpoint*, by measuring the state of the process and comparing it to the setpoint to obtain the *error*, which is then sent to the controller whose output, or *control effort*, is finally applied to the process, and the loop repeats, as illustrated in Figure 4.3[1][50]. In more formal words, it is a control loop mechanism using feedback in applications which require continuously modulated control[51].

A PID controller is typically an example of negative feedback, a very popular way of controlling continuous processes[52]. It is thanks to this feedback, iteration process and use of the PID controller that the process manages to reach the desired setpoint.

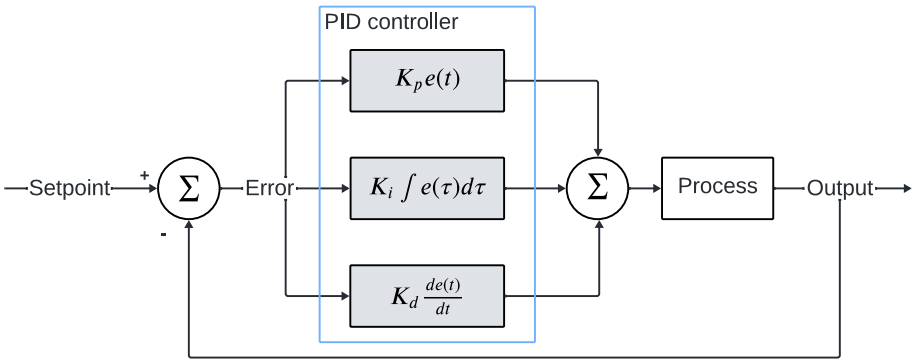


Figure 4.3: PID controller in a negative feedback loop

In itself, a PID controller is composed of three control terms: a proportional term, an integral term and a derivative term. Each of them works with the error:

- The *proportional term* multiplies the error by a coefficient, the *proportional gain* K_p. Its effect is to reduce quickly the error between the setpoint and the output value of the process. However, a P controller on its own is not able to fully reach the setpoint, as a small but non-zero steady-state error will remain[53]: indeed, if it reached the setpoint, the control effort would be zero.

[1]Reproduced from https://maldus512.medium.com/pid-control-explained-45b671f10bc7

- The *integral term* multiplies the integral of the error over time by the *integral gain* K_i. This term is often used to cancel the steady-state error of the proportional controller, as its value will increase over time until the error reaches zero, by which point the proportional term will be zero, but the integral term will have reached a non-zero value that will keep the process running. It is to be noted, however, that a PI controller can cause closed-loop instability if the integral action is too aggressive. When a PI controller is not well-tuned, it may so happen that the output gets driven back and forth between fully on and fully off, leading to a phenomenon known as hunting, due to over-correction of the error[52].
- The *derivative term* multiplies the derivative of the error by the *derivative gain* K_d. This term is often used in combination to a P or PI controller in order to limit the output in case the process value is approaching the setpoint too fast, in the hopes of preventing the process value from overshooting the setpoint[50]. It however needs to be carefully used, as if the derivative action is too strong, it can counter the action of the P/PI controller so hard that it causes hunting by itself, an effect particularly pronounced in processes that react quickly to the controller efforts, such as motors[53].

Note that all coefficients, K_p, K_i and K_d, are always positive.

To get the appropriate response out of a PID controller, a tuning phase is required. The simplest way of tuning such a controller is the following:

1. Start by looking at the process' response to a simple P controller, so $K_i = K_d = 0$. The ideal integral gain K_p is found by tuning it so that it allows the output value to barely reach the setpoint, without causing an overshoot.
2. Then, tune the integral term, or the derivative term, or both, depending on your application and requirements, to obtain the best possible behaviour of the controller, that is, quickly reaching the setpoint without overshooting and avoiding oscillations of the process value around the setpoint.

4.3.2 Implemented PI controller

For this project, it was chosen to use a velocity controller to control the robot. A simple PI controller has been implemented to fulfil this need. Indeed, as noted earlier, a P controller by itself is not able to drive the output value to the correct setpoint, so an integral term was added to reach this. On the other hand, it was quickly found that the derivative term was hard to tune and would most of the time lead to hunting, or surging of the motors' output, due to the reactivity of the motors, without bringing any significant contribution when it was not affecting negatively the controller's output. It was thus decided to only use a proportional-integral controller. Using the aforementioned tuning methodology, a proportional gain $K_p = 8$ and an integral term $K_i = 8$ have been found.

As can be seen in Figure 4.4, these two coefficients lead to some overshooting and some oscillations before reaching the desired steady-state value. This final K_i coefficient was chosen based on a trade-off. Indeed, a higher proportional gain K_i would make the controller overshoot more and then oscillate more around the steady-state value. But a too small K_i gain, despite reducing the overshoot, would render the controller less efficient, as it would take a lot of time for the controller to correct its steady-state error. Thus, a value $K_i = 8$ was chosen, as it was found to give a reasonable overshoot and oscillations but still allow the controller to quickly reach its steady-state value.

It is important to note that the K_p gain was found using the lowest possible steady-state velocity of the system, which is 80 RPM. For more detail on the velocities and commands of the robot,

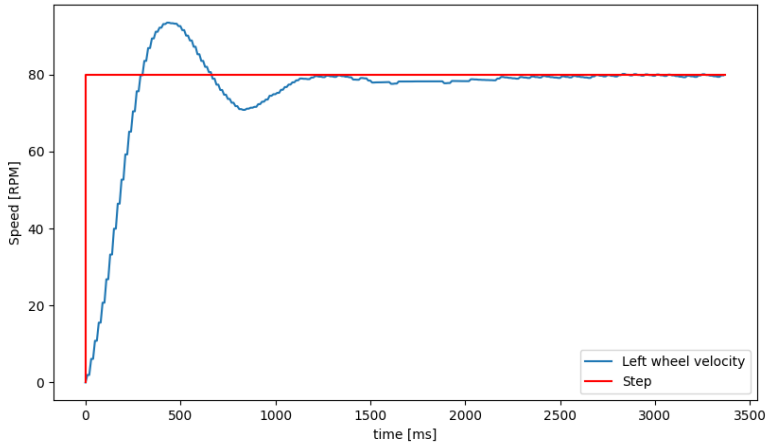


Figure 4.4: Step response of the left wheel to a setpoint of 80 RPM. The velocity is controlled using the implemented PI controller. The data have been obtained by forcing a delay of 10 ms between two measures, leading to this "staircase" effect

refer to Section 5.5. Also note that despite the small overshoot during a step response, due to the continuous velocity profile, this overshooting problem was found to be irrelevant. Refer to Section 6.1.1 for more detail on this.

4.4 Velocity computation

As has previously been explained in Section 4.1, the speed and spinning direction of the output motor shaft are found by looking at the encoders:

- the direction in which the motor spins is found by looking at the value of the encoder output B each time the output A changes and
- to compute the speed of the output shaft, the controller counts how many times the encoder output A has changed within a fixed time interval. In the controller's code, this fixed time interval has been chosen as 18 ms. This acts as a first "filter" by reducing the update frequency of the velocity.

This is an arbitrary value based on empirical observation of the behaviour of the robot. This value will however have an impact on the reactivity of the controller and the computed wheel velocity:

- On the reactivity of the controller: the shorter this interval, the more the wheel velocity will be updated, meaning the controller corrects more often the detected errors to the setpoint. No particular benefits have been observed when choosing a smaller or larger interval than 18 ms.
- On the computed wheel velocity: it is computed as the finite difference of the motor's encoder count divided by the time elapsed since the last velocity computation, as visible in Equation 4.1:

$$velocity = \frac{\Delta pos}{\Delta t} \tag{4.1}$$

where the variable *pos* is the encoder count[2]. It is to be noted that this finite difference (Δpos) will not have, at each loop iteration, the same exact value, meaning that even at constant velocity, the measured one will not be constant and will fluctuate around the correct value.

Let us look at an example, assuming the wheel turns at 100 RPM. Since the encoders are on the motor shaft and the wheel is on the output shaft, the gearbox (1:53) which is present within the motor between the two aforementioned shafts has to be taken into account. As already mentioned in Section 4.2, since the encoders have a 3 PPR, doubled to 6 PPR upon updating the encoder count both when output signal A rises and falls, and taking the gearbox into consideration, a resolution of 318 PPR of the output shaft is obtained. This means that, at 100 RPM, it follows that the encoder evolves at:

$$6 \times 53 \times \left(\frac{100}{60}\right) = 530 \ \frac{ticks}{s} \tag{4.2}$$

a *tick* being defined as an update of the encoder count. This yields an update frequency of 530 Hz, which means that between two updates, around 1.9 ms elapses. Thus, during the first 18 ms, nine ticks will have been registered, the same as during the second interval. But on the third 18 ms interval, ten ticks will have been registered, as illustrated in Figure 4.5. This leads to a non-negligible 11.11% variation between these two successive Δpos, thus changing the velocity estimation.

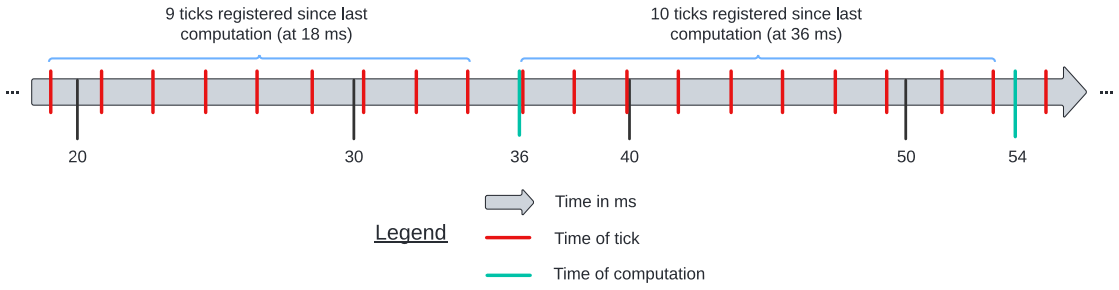


Figure 4.5: Illustration of the encoder ticks counting. Only the second and third time intervals are shown. It is assumed that the counter started at 0 ms, with the motor already at 100 RPM by then

A smaller interval means that the value of Δpos is smaller, leading to a big impact on the velocity estimation between two consecutive intervals since the variation of Δpos will be proportionally bigger, but the velocity is updated more frequently, whilst on the other hand, a longer interval means that the variations between two successive Δpos will be smaller proportionally, but whenever there is an over/under-estimation of the velocity, it will have more time to impact the filtered one.

4.4.1 Filtering the velocity

In order to attenuate the impact of two different consecutive Δpos, a low-pass filter on the velocity has been added to the code of the controller. The filter has been created using the following code[3]. It requires to fix a sampling frequency and a cut-off frequency. A low-pass frequency of 1 Hz has been chosen, with a sampling frequency of 56 Hz, since the velocity is computed every 18 ms, leading to an update frequency of 55.56 Hz which was rounded up to 56 Hz. This all leads to the filter shown in Equation 4.3:

[2]This variable is named *pos* as the evolution of the encoder count basically transcribes the position of the motor shaft.

[3]https://github.com/curiores/ArduinoTutorials/blob/main/BasicFilters/Design/LowPass/LowPassFilter.ipynb

$$vFiltNew = 0.894 \times vFilt + 0.053 \times vComputed + 0.053 \times vPrev \tag{4.3}$$

where $vFilt$ is the previous value of the filtered velocity and $vFiltNew$ the newly computed one, $vComputed$ is the velocity computed at the current loop iteration and $vPrev$ is the previous value of the computed velocity. As one can observe, most of the final filtered velocity will come from the previous value of the filtered velocity itself. The computed velocity has little immediate impact on the filtered one, thus reducing heavily the impact of the aforementioned problems with different successive Δpos.

Several cutoff frequencies were tested. It should come as no surprise that a smaller cutoff frequency leads to a filtered velocity which will barely evolve, thus leading to a very good estimation of the velocity at steady-state. However, this also means that the filter is slow: when the actual velocity evolves, since more weight is given to the previous filtered value, the filtered velocity will change slowly, progressively evolving towards the actual value.

This has several implications:

1. Since the controller uses the filtered velocity to compare it to the target one and compute the error, a slower evolution of $vFilt$ means that the error reduces more slowly, thus the controller will overshoot by sending a high command to the motors for more time than necessary.
2. When the robot stops, the velocity goes down slowly. If the robot restarts all of a sudden with a desired velocity target which is below the filtered value, due to the way the controller works, the robot will start going backward because the error is negative, even if the goal was for it to move forward.
3. Later in the code implementation, a slowing down routine has been added. Again, a slower evolution of the filtered velocity means that the robot will take more time to slow down than necessary, which may lead to unwanted situations (bumping into a wall, falling down some stairs, or any situations in which the user would have wanted the robot to stop quickly to avoid a problem).

These behaviours are of course unacceptable. On the other hand, a high cutoff frequency will give more weight to the computed velocity, leading to more variations in the filtered velocity. This implies that the command sent to the motors will vary more, leading to a more "shaky" behaviour of the wheels.

There is thus a trade-off between filter performance and controller reactivity. It has been found empirically that a cutoff frequency of 1 Hz, in this project, leads to good results. This is something that could certainly be improved upon.

4.5 The low-level velocity controller loop

A low-level controller is the controller which is the closest to the hardware, so here, the motors. Its role in this project is to control the velocity of the robot, as explained in Section 4.3. The low-level velocity controller's code[4] is directly implemented within the Raspberry Pi Pico W microcontroller. The low-level controller is composed of one main loop, as illustrated in Figure 4.6, which works in the following way[5]:

[4]Which can be found here: https://github.com/Neackow/ESP32

[5]To avoid overloading the schematic, it appears that there is only one control loop, but in reality, there is one loop for each motor.

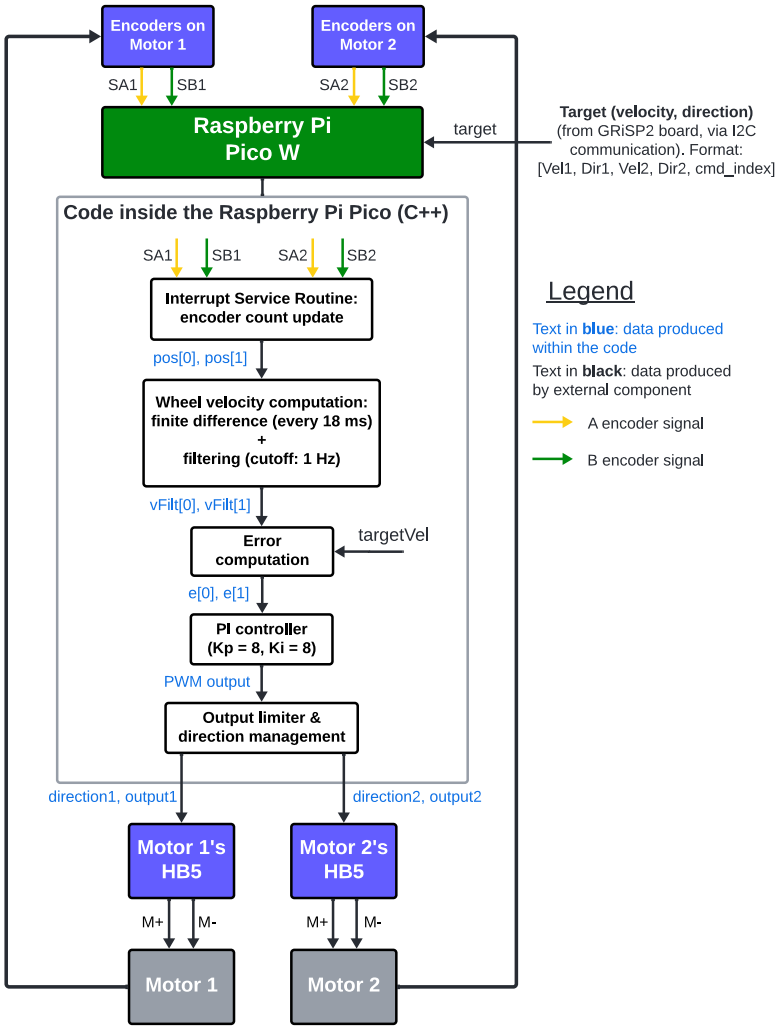


Figure 4.6: Low-level controller loop schematic

1. Coming from the encoders that are included on the motors, the two square wave signals SA and SB enter as inputs in the Raspberry board;

2. A special function, named an Interrupt Service Routine (ISR), is called whenever there is a transition on signal SA (either a rising or falling edge). It checks whether the counter needs to be incremented or decremented by looking at the value of signal SB, implementing what has been explained in Section 4.1. This will update the encoder count, $pos[2]$. $pos[0]$ and $pos[1]$ are respectively motor 1's and motor 2's encoder counts;

3. These encoder counts go into the velocity computation block, which contains the two filtering steps explained in section 4.4. This block outputs the filtered velocity of the motors;

4. Using the filtered velocities and the target ones, which have been received alongside the spinning direction to be sent on the Direction pin of the HB5s and a fifth element, labelled *command_index*, which is used by another part of the code to choose the right routine (refer to Section 5.5 for details about the format of the received command), the error is computed as such:

$$error = targetVel - vFilt \tag{4.4}$$

   This error is then used as input to the main controller block. Note that the Raspberry board receives the absolute value of the desired velocity from the GRiSP2 board. Depending on the direction required, this absolute value is corrected to be negative if necessary;

5. The controller block implements a PI controller to compute the output (a digital PWM

command) required to reach the setpoint (target velocity, in this case). As the continuous version of the PI algorithm, which has been explained in Section 4.3, cannot be achieved in the controller's code, a discrete version has been implemented and is visible in Equation 4.5:

$$output = K_p e + K_i \sum_k e_k dt \tag{4.5}$$

where e is the error, K_p and K_i are respectively the proportional and integral gain and dt is the time one loop iteration takes.

6. The final block is there to limit the maximum output at 255, since the value of PWM is coded on eight bits, and to make sure to send the correct spinning direction to the HB5s. Note that the spinning direction is not the same for both motors when asked to go in a straight line, as they need to spin in opposite directions due to the way they are placed on the robot in order to do so;
7. Finally, the loop ends by sending the output command and spinning direction to the HB5s, which will then create the PWM square wave and send the appropriate MOTOR+ and MOTOR- signals to the motors to drive them, and the resulting velocity will be measured by counting the encoder pulses and looping once more.

A note about the limiting block. Prior to blocking the maximum output at 255, this block first implements a limiter on the integral term. The first implementation of the controller did not have this, as it was deemed unnecessary at first. However, it quickly became clear that this limiter would be useful. Indeed, with no limitations on it, the integral term was climbing very rapidly to high values at the start. This led to overshoots, as the command would still be at maximum despite reaching beyond the setpoint, as the proportional term (negative because beyond the target) could not compete with the integral term (highly positive and progressively decreasing).

To limit this unwanted effect, a limiter has been added. This limits the maximum value that the integral term can have, in order for the controller to be more reactive. This limit has been empirically set as:

$$K_i \sum_k e_k dt \leq K_i \times 70 \tag{4.6}$$

This effect was especially bad with slower filters. With the filter having a cutoff frequency at 1 Hz, the integral term has less time to increase too much, but the limiter still helps to make the controller more reactive.

Chapter 5

Gesture recognition and remote control: the *movement_detection* application

This chapter details the *movement_detection* GRiSP application. The specifications of the application will be mentioned and the limitations of the existing gesture recognition algorithm, presented in Section 2.6.2, are explored. The chapter then goes over the whole code flow and the application's modules, the routines and the gestures and associated commands. It ends by explaining the implemented software guard-rails.

The chapter is organised as follows:

- Section 5.1 details the specifications of the application.
- Section 5.2 explains the current limitations of the implemented algorithm for gesture recognition.
- Section 5.3 details the whole application's code flow and gives more information about the modules of the application.
- Section 5.4 describes the control routines for the robot and their uses.
- Section 5.5 details the gesture library that was implemented in this thesis and the associated commands. It also explains the iterative process that the application went through.
- Finally, Section 5.6 shows the software guard-rails implemented to protect the system from hardware failures and block forbidden gesture combinations.

5.1 Specifications of the application

Hereafter are detailed the performance levels that are expected from the application:

- Wireless communication between the robot and the remote controller;
- Gestures detected reliably with high accuracy ($\geq 70\%$);
- New orders to be sent in less than five seconds;
- Detection of the gesture based on the PNAV data;
- Ability to detect multiple gestures one after the other with high accuracy;
- Convert gestures into the appropriate robot behaviour;

- Smooth movements of the robot;
- Detect and deal with hardware failures and users' misuses of the algorithm by stopping the robot as a security measure.

5.2 Problems and limitations of the gesture recognition algorithm

In order to detect gestures, the algorithm developed last year in this thesis[8] and whose working principle has been described in Section 2.6.2 is used. It however soon became apparent that the algorithm presented some inadequacies:

- The main problem is the speed at which the gestures are detected. Indeed, as long as the algorithm detects a change in the acceleration on one of the three axes, data are collected, and only once no movement has been detected on all three axes for a certain time can the classification occur. To have a clean detected evolution of the acceleration on an axis[1], the gestures need to be slow.

  It was discovered that despite not using the values returned by the Kalman filter implemented in `e11`, calls to its measure function were not removed from the main loop, which was slowing down the process. By printing the timestamp at each newly fetched value of the accelerometer, it was discovered that the process was running at 90.15 Hz. When removing the calls to `e11`'s measure function, the algorithm was faster, running at 115.92 Hz.

  This means that to properly detect a movement in which one axis goes through the whole spectrum of possible values, like in the aforementioned example, a strict minimum of $5 \times 0.4313 = 2.157$ s is required[2], plus the time to wait that the algorithm detects no new movement. Experimentally, it was discovered that this minimum of 2.157 s was very optimistic and that, to correctly detect the whole spectrum of values on one axis consistently, at least 5 s were necessary, without taking into account the data processing and classification time. This is far too slow in the context of a robot on which the user needs to keep a good control.
- The algorithm uses only the accelerometer. This means that the data are continuously affected by gravity, pointing downwards with a constant value of 9.81 m/s^2. In fact, the algorithm works only with gravity. This means that only rotations can really be detected, which limits heavily the number of possible gestures. It is practically impossible to detected linear movements.
- When launching the continuous gesture detection algorithm for an infinite amount of time, there is no other way of stopping it than quitting the program manually or disconnecting the board. This is not very practical nor user-friendly.
- Late into the thesis, the author realised that the algorithm was doing everything twice. Indeed, inside the main function for gesture recognition (`grdos/11`, as explained in Section 2.6.2), data treatment is done to be able to detect two successive equal means on an axis. It was believed by the author that this list of treated data was then the one sent to the classification algorithm, but it is not the case, as the complete list of collected data is sent and treated once more in the exact same manner. This is a waste of computational power.

Despite these limitations, mostly due to a lack of time, it was decided to keep the existing algorithm to detect the gestures. Though, as will be explained in the next sections, some problems have been removed, whilst the impact of some others were attenuated. Despite these modifications,

[1] E.g.: from `nn` to `pp`, a clean list would be `[nn, n, o, p, pp]`.

[2] 0.4313 s is the time it takes to fetch 50 values, since the averages are done on 50 values and that the algorithm runs at 115.92 Hz. Note: the fact that it is necessary to wait for 50 values to be fetched to compute an average also slows down the process of recognising gestures.

the implementation still presents limitations, which are discussed in Section 6.5. Finally, possible improvements will be discussed in Section 7.1.

5.3 The application's code flow and modules

Built upon the *sensor_fusion* application which was extended to allow the remote control of a robot using a GRiSP2 board on which a PNAV is plugged, this application is based on a module, named `sendOrder`, which is a `gen_server` (for more explanations on this, see Section 5.3.3) that can be re-used and adapted to any controllable object. This module runs on the GRiSP2 board that is inside the robot (the receiver) and is used to convert the detected gestures which are sent from the detector into commands to be sent to the micro-controller. This `gen_server` has its own supervisor, named `sendOrder_sup`, which is launched by the *movement_detection* application at the initialisation, when the receiver is first supplied. This supervisor also initiates the I2C communication between the GRiSP2 and Raspberry boards by opening the bus and then waiting for read or write order. The I2C communication itself is done using another `gen_server` module, named `i2c_communication`, for reasons that will be explained in Section 5.3.4.

5.3.1 The different *behaviours*

Specific "behaviours" of the *movement_detection* application exist. What is meant by *behaviour* is that, depending on the name attributed to the board during the deployment of the application, the board can select which part of the application's code it will run.

On one hand, to specify that the GRiSP2 board will be used as a detector, which uses mainly the `grdos/12` and gesture classification functions of the application, it is necessary to deploy the application by naming the board as *navX*, where *X* can be anything, as long as the name starts with *nav*. For historical reasons explained in the updated user manual[3], the detector is deployed with the name *nav_1*.

On the other hand, to specify that the GRiSP2 board will act as a receiver, which converts the received gestures into the appropriate command before sending them to the Raspberry by using the `sendOrder` module, it is necessary to name it as *orderX*, where again, *X* can be anything, as long as the name starts with *order*. In this project, the receiver is named *orderCrate*, to fit with the wine-crate origin of the prototype.

5.3.2 Explanation of the whole application's code flow

The overview of the code's working and the different communications between the boards are illustrated in Figure 5.1. The rest of this section gives more detail on how the code globally works.

- The control of the robot starts when the user calls the `realtime/2` function from the main module of the detector, namely, `movement_detection`.
- This function will call the start function of the `realtime` module, which then calls `grdos/2` that in turn calls `grdos/12`. This function is explained in more detail later in this chapter.
- If the detected average on the z-axis at any point in time is `nn`, a `stopCrate` order is sent directly by the `grdos/12` function.
- The gesture is classified when no changes in the axes' averages have been detected for a certain amount of time by calling `classify_new_gesture/1` from the `classify` module. This function call happens within the `grdos/12` function. `classify_new_gesture/1` compares the

[3]Refer to the application's GitHub repository or to Appendix E.2.

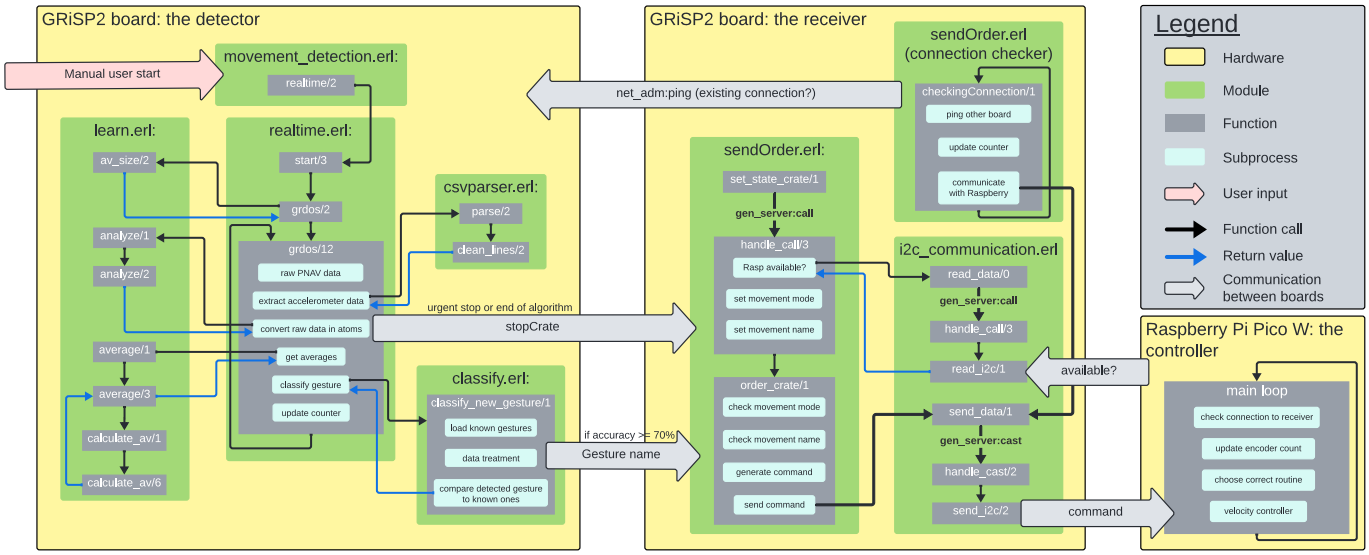


Figure 5.1: Overview of the application's code and of the links between the boards

detected gesture[4] to a list of known gestures using pattern matching and, if the accuracy[5] is above 70%, the name of the detected gesture is sent to the receiver using the `rpc:call/4` function which has been explained in Section 2.4.1. If the accuracy is too low, the `Name` sent is `stopCrate`, to order the robot to stop. The complete call is of the following form:

```
rpc:call(movement_detection@orderCrate, sendOrder, set_state_crate, [Name])
```

Note that when the algorithm is terminated, `Name` will also be `stopCrate`, and that the algorithm termination is directly dealt with within `grdos/12` for a rapid stop of the robot when necessary. More detail about the commands, movement modes, the associated `gen_server`'s state and gestures can be found in Section 5.5.

- The classified gesture's name is now sent over to the receiver. It is used as an argument for the function `set_state_crate/1`, which will proceed to make a call on the `gen_server`.
- In response to this call, the callback function `handle_call/3` is launched. It will proceed to the first classification steps:
  - it first checks if the Raspberry board is not busy with a previous command by checking the value of the *available* variable of the micro-controller. The receiver will ask for and receive this value via I2C communication. If the controller is available, the procedure continues. If not, nothing happens;
  - depending on the gesture name, some special *movement modes* can be set;
  - and finally, the appropriate *movement name* is stored in the `gen_server` state;
- Next, the `order_crate/1` function is called. It converts the movement modes and names into the appropriate command and sends it to the Raspberry board using I2C.
- Finally, the command is received by the Raspberry board and the controller will ensure that the desired behaviour is achieved.

[4]Note that in Figure 5.1, a subprocess labelled *data treatment* is shown. The fact is that, as mentioned previously, the gesture detection algorithm does the whole job twice. In order to avoid an overload of the schematic, the whole process of filtering the data, converting it into atoms, etc. is summarised by this simple designation. Furthermore, another function not shown in the schematic is used, named `regroup/2`, from the `learn` module. It is used to get the final cleaned list of averages.

[5]The accuracy is a measure of how confident the code is that the detected gesture is actually the one that the code has classified. For more detail on how it is computed, refer to this thesis[8].

- Gesture recognition continues until three consecutive `stopCrate` gestures have been registered. At each detected gesture, a command is sent to the micro-controller, unless it is not available.

Something that was not mentioned up until now is the looping function visible in the top right corner of the receiver module in Figure 5.1. This implements two of the guard-rail functions that will be explained in Section 5.6.

5.3.3 The `sendOrder` module

The module uses the `gen_server` behaviour, which is an OTP behaviour that gives a few useful functions, or interface functions, to use in exchange for which the user needs to provide some functions, called *callback* functions[54]. More generally speaking, it provides the server of a client-server relation[55]. Two of the callback functions are `handle_call/3` and `handle_cast/2`, which work respectively with synchronous and asynchronous messages. The difference between the two is that a process working with synchronous messages expects an answer, meaning a return value, and waits for it, while working with asynchronous messages does not require any return value.

The main advantage of this behaviour is that it can work with a *state* that can be updated, which is very practical since variables in Erlang are immutable. The state is updated every time a new gesture is received from the detector. In fact, the `order_crate/1` function mentioned previously returns the updated state at the end. The state is stored within an Erlang *record*, which is a data structure used for storing a fixed number of elements, similar to a *struct* in C[56]. In this thesis, the `gen_server` state, named *movState*, is of the following format:

$$\{currentVelocity, prevName, movName, movMode\}$$

where:

- *currentVelocity* keeps track of the velocity at which the wheels should spin. This variable is used to be able to change the desired wheel velocity and to store it. It is often used to build the command tuples to be sent via I2C and is updated when in `changeVelocity` mode;
- *prevName* stores the previous movement name. It is only used by two guard-rails that will be explained later in Section 5.6 and can take one of the following values:
  - `forward`
  - `backward`
  - `stopCrate`
  - `turnAround`
  - `changeVelocity`
  - `exitChangeVelocity`
- *movName* is used to store the correct movement name, depending on the received gesture and movement mode and
- *movMode* stores the movement mode, which is either `normal` or `changeVelocity`.

At the initialisation, the state is set as:

$$\{currentVelocity = 100, prevName = stopCrate, movName = stopCrate, movMode = normal\}$$

which puts everything at rest and sets a baseline velocity of 100 RPM.

This module is generalisable to any application. All one has to do is to set the correct state at initialisation and to modify the callback functions to change the functions they call.

5.3.4 The `i2c_communication` module

A second `gen_server` module is used in the receiver. This module is only used to deal with the I2C communication, whose protocol has been explained in Section 2.8. Originally, the bus management was directly done within the `sendOrder` module, but this could have led to some unexpected behaviour. Indeed, as mentioned earlier, there is a function, called `checkingConnection/1`, that sends data on the I2C bus every second or so. This process is independent from the rest of the code, which also uses the bus, both in `handle_call/3` to check the availability of the Raspberry (read operation) and in `order_crate/1` where it sends the appropriate command to the Raspberry (write operation).

In the absence of a dedicated process and with some bad luck, it could have been possible for two operations to happen at the exact same time, and the resulting behaviour could have led to potentially catastrophic consequences. This is why a dedicated `gen_server` was created, in order to prevent this from ever happening. Indeed, the `gen_server`, upon receiving two access requests to the bus at the same time, will carry them out one after the other.

The read operation has been chosen to be a `call`, while the write operation is achieved by a `cast`. The reason behind this choice is that when the receiver asks for the availability of the Raspberry, it is necessary to wait for an answer before proceeding with the rest of the code, whilst on the other hand, once the command is sent to the micro-controller, the GRiSP2 board does not expect a feedback and can move on to do more work.

At the module's initialisation, an I2C bus is opened between the GRiSP2 and the Raspberry board using the GRiSP2 driver for I2C communication, and the bus' PID is then stored in a record as the `gen_server`'s only state variable, to be accessible everywhere in the code. Then, the module can be used as such:

- to read, one needs to call `read_data/0`, which then makes a `gen_server:call` that will, in turn, use the `read_i2c/1` function which takes the bus' PID as argument;
- to write, one needs to call `send_data/1` with the message to be sent as argument, which then makes a `gen_server:cast` that will, in turn, use the `send_i2c/2` function which takes the message and the bus' PID as arguments.

Again, this module can be adapted to accomplish any desired behaviour.

5.3.5 The other side of the I2C communication

If the GRiSP2 board is the master responsible for initiating any communication, the Raspberry board is the slave answering the calls. It has a fixed address, arbitrarily fixed as *0x40*. This is set in the board's `setup()` function, when the Raspberry is booted. It also defines two functions: `receiveMessage()`, which is called whenever the master wants to write to the slave, and `sendMessage()`, which is called whenever the master wants to read something from the slave.

The `receiveMessage()` function is able to read, byte after byte, what has been sent on the bus. The number of bytes it receives is stored into a variable, which is then used to decide what to do with the received data. In this thesis, there are only two possible message lengths that the Raspberry can receive:

- if it received only one byte, it means this has to do with one of the guard-rails. More on that in Section 5.6;
- if it received 5 bytes, it means that it received the new command which the controller has to deal with.

Finally, the `sendMessage()` function will only fetch the value of the *available* variable and send it to the master upon receiving a read request.

5.3.6 The updated `grdos/12` function

As mentioned earlier, an updated version of the `grdos/11` function, called `grdos/12`, has been developed to fit the needs of the current application. A schematic explaining the updated version of this function is visible in Figure 5.2.

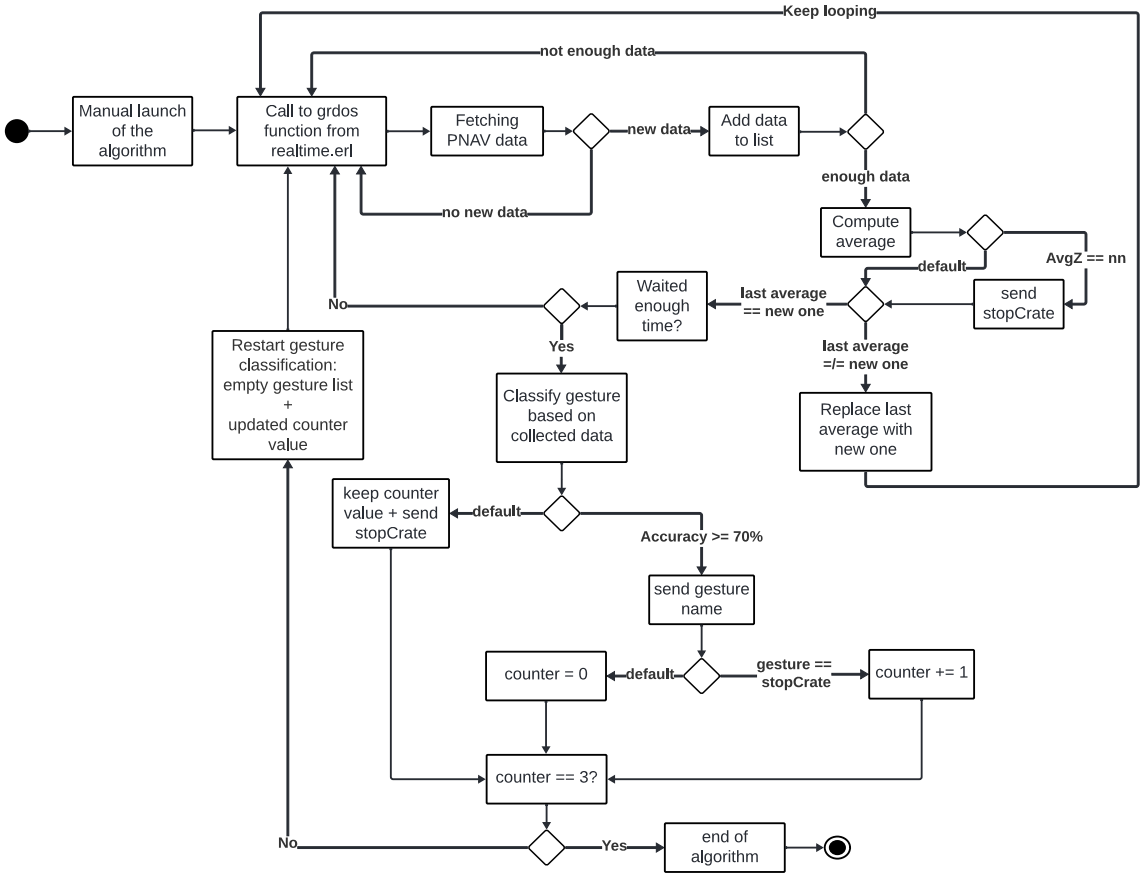


Figure 5.2: Updated `grdos/12` function. The black dot indicates the start of the algorithm, a rhombus indicates a conditional statement and the black and white dot indicates the end of the algorithm. `default` means "all other possibilities than the one specified in the other path"

The algorithm remains globally the same, but some additions have been made to fit the application:

1. In the final code's implementation, a rapid stop of the robot procedure was added. This is visible after the "*Compute average*" block. As already explained in Section 5.3.2, if the computed average value over 50 data for the z-axis is `nn`, then a `stopCrate` command is sent to the receiver using `rpc:call/4` and the algorithm keeps going.

2. Initially, the name of the detected gesture was simply printed by the algorithm in the console before moving on. Now, if the accuracy of the classification is above 70%, the gesture's name is sent to the receiver. Else, it is deemed not enough to do anything with that classification result: the name `stopCrate` is sent to the receiver and the counter's value is kept.

3. A problem of the original algorithm was the absence of a stop condition. To fix this, an Erlang *record* simply named `counter` and storing a single element has been added. On the first call to the `grdos/12` function, the counter is initialised at 0 and is passed as the twelfth argument

of the updated function. The counter is then updated at each accurately classified gesture: if the detected gesture implies a stoppage of the robot, then the counter is incremented by one. If not, the counter is reset.

4. In any case, the counter value is evaluated. If three consecutive stops have been registered, then the algorithm will stop by calling the `grdos/12` function with a `Period` equal to 0. In that case, during the next iteration, `grdos/12` does not even try to fetch the PNAV data and proceeds to stop calling itself, effectively ending the loop. However, if the counter value is ≤ 3, then the gesture recognition starts again with the updated counter value by emptying the gesture list.

The fact that the algorithm stops after three `stopCrate` commands is a design choice. In future versions of the algorithm, this value could be made a user input to offer as much comfort as possible to the user when using the algorithm. Indeed, a user who is very confident in his decisions could decide to stop the algorithm after less stops[6], but hesitant ones could ask for more time to decide which gesture they want to perform next.

5.4 The robot's control routine

As has been mentioned previously several times, beyond the low-level velocity controller, the code within the Raspberry board implements several *routines*. These routines have been implemented in order to achieve specific behaviours. The need for such routines came from the fact that the micro-controller's code is based on the `loop()` function from the Arduino IDE. As this function allows to actively control the board by looping continuously[57], if one wants to achieve a precise and finite in time behaviour, a special block that achieves this behaviour is required. Once the behaviour is done, the code can go back into the "normal" mode and continue looping.

In the micro-controller's code, then, this is achieved by the routines. To know which routine should be running, the receiver sends, on the fifth byte of the command tuple (the command tuple is described in Section 5.5), a value which is known as the *command_index*. Moreoever, the received commands are stored in a special variable, named `storeTarget[5]`, that is a list of five integers, which is only updated by the commands received by I2C communication or at the end of the routines. The controller, on the other hand, uses another variable to know which velocity the wheels should achieve, named `target[5]`, which is a list of five floats.

To better understand the relationship between these variables, let us look at an example. In the "normal" mode, associated with *command_index* = 0, the loop will call the `evolveVelocity()` function. This function is used to implement the continuous velocity profile, which will be explained in Section 5.5. At first, everything is initialised to zero, and the robot does not move. Upon receiving, e.g., the result of a `forward` gesture at 100 RPM, the variable `storeTarget[5]` is updated and now contains the received order. Now, the `evolveVelocity()` function will compare periodically `target[0]` and `target[2]`, respectively containing the velocity for the left and right wheels which the controller needs to achieve, with `storeTarget[0]` and `storeTarget[2]` respectively:

- if `target[i]` $<$ `storeTarget[i]`, then the `target[i]` variable is incremented by 0.5;
- if `target[i]` $>$ `storeTarget[i]`, then the `target[i]` variable is decremented by 0.5;
- else, the `target[i]` variable is kept at its current value;

[6]Note that stopping the algorithm after only one detected `stopCrate` would not be practical: whenever the user wants to stop the crate to proceed to some more complicated gestures, the algorithm would end.

where `i` is either 0 or 2. Thus, starting from 0 RPM, the `target` will progressively evolve toward the desired velocity for both wheels, and the controller follows this evolution.

This relationship between the two variables allows to stop the robot after some routines. By setting every element of `storeTarget[5]` to 0 at the end of the routine, at the next loop iteration, `evolveVelocity()` will either make the robot slow down, or if `target` was already at zero, nothing will happen.

The simplified general flow of the micro-controller's algorithm is illustrated in Figure 5.3.

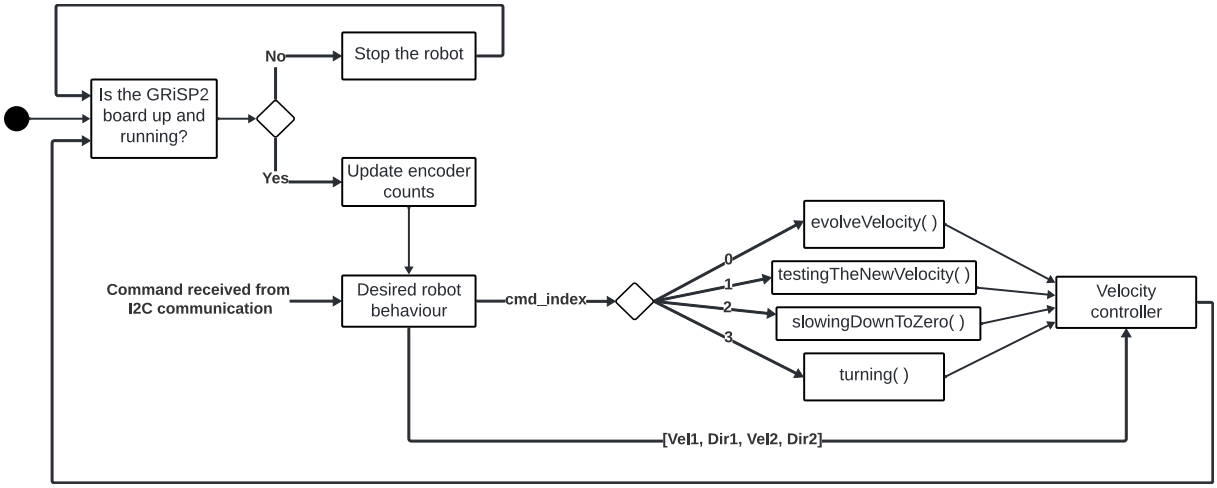


Figure 5.3: Simplified general flow of the micro-controller's algorithm. The black dot indicates the start of the algorithm, a rhombus indicates a conditional statement

The first block represents one of the four implemented guard-rails, which are described in Section 5.6. As one can see in the schematic, there are four possible routines, all attributed to one value of *command_index*:

- *command_index* = 0: this is the "normal" mode, in which nothing in particular happens. It simply implements a continuous velocity profile. This is referred to as the `normal` mode;
- *command_index* = 1: this routine is used to test the new velocity while in the `changeVelocity` movement mode by calling the function `testingTheNewVelocity()` in the Raspberry board.

  Indeed, one of the possibilities offered with the robot is that the user can change the velocity at which the wheels turn. To visually see if the new velocity is better, the user can test it by performing a specific gesture which will launch a routine that consists in two steps:

  1. Travel 2 m forward before stopping: the robot accelerates, then upon reaching the 2 m mark, it slows down to zero;
  2. Travel back 2 m before stopping: again, the robot accelerates, going backward this time, and upon reaching the 2 m mark, it slows down again to zero and finishes the routine;

  The robot knows it has travelled 2 m by looking at the encoder count, using the same idea explained in the `turnAround` routine.
- *command_index* = 2: this routine calls `slowingDownToZero()` which, as its name suggests, is a function which is used to stop the robot. It is primarily used by `testingTheNewVelocity()`, as special variable changes and conditions need to be made in order to get the desired back and forth movement of this routine.

- *command_index* = 3: this routine is used to make the robot spin on itself, by calling the function `turning()` in the Raspberry board, to rapidly change the heading direction.

  Since the distance separating the centre of the two wheels is equal to $d = 28.5$ cm, turning on itself makes the robot's wheels follow a circle of diameter d. Thus, both wheels need to travel half that circle's perimeter, meaning the distance to go is $\frac{\pi d}{2}$ which is equal to 44.77 cm. Since the wheels have a diameter of 8 cm, one wheel turn is equal to 25.13 cm, so the wheels need to do 1.78 turns to travel this distance. Since one wheel turn is equal to 6×53 ticks, the robot will have spun on itself when one wheel has its encoder count incremented by 566.44 ticks, rounded up to 567 ticks. After having registered enough ticks, the robot is stopped.

  In reality, some overshooting was observed when asked to stop after 567 registered ticks. To compensate for this, the robot now stops after 445.2 ticks, or 445 ticks when rounded, which is equal to 1.4 wheel turns.

The reliability and repeatability of the routine accomplished by `testingTheNewVelocity()`, as well as the one accomplished by `turning()`, will be analysed in Sections 6.2.1 and 6.2.2, respectively.

Finally, it is important to note that, in order not to disturb a routine, a special variable named *available* has been added in the Raspberry board's code. This variable is *1* by default but is put to *0* whenever a routine is started and is changed back to *1* when the routine is over. Indeed, as was mentioned in Section 5.3, whenever a new gesture is detected and classified, it is first checked by the receiver if the value of the *available* variable is *1* by performing a read. If a routine is currently being performed, *available* = 0, and the detected gesture is discarded.

5.5 The gestures and associated commands

To control remotely the robot, specific gestures have to be performed. Arguably the most important part of this work, the gestures and associated robot behaviour went through two iterations.

The first version of the gesture was following the "absolute control of the robot" mindset: before each change in direction, a stop had to be registered, and then special routines would perform predetermined movements before once again stopping the robot and waiting for new orders. For example, turning left or right was fixed at a 90° turn, with only one wheel spinning, after which the robot would stop. This choice was motivated by the limitations of the gesture recognition algorithm: due to how slow it was to detect correctly every gesture, it was preferred to keep an absolute control of the robot by stopping often.

This version was not satisfactory, as it limited the control potential that the user could have on the robot. Moreover, the robot's velocity profile was discontinuous: this was leading to shocks on the robot and abrupt wheel velocity changes.

To smooth out the position profile of the robot and avoid shocks, a second version using a continuous velocity profile was implemented by creating the `evolveVelocity()` function. Moreover, instead of having a special routine to turn by 90°, a turn is now performed in a continuous manner without stopping the robot first: the robot turns because there is a difference of 30 RPM in velocity between the two wheels.

The implemented continuous velocity profile is a trapezoidal one, drawing inspiration from what is taught in the *LELME2732 - Robot Modelling And Control* class[58]. This is illustrated in Figure 5.4. This leads to the same acceleration and deceleration duration, so the trajectory is symmetric.

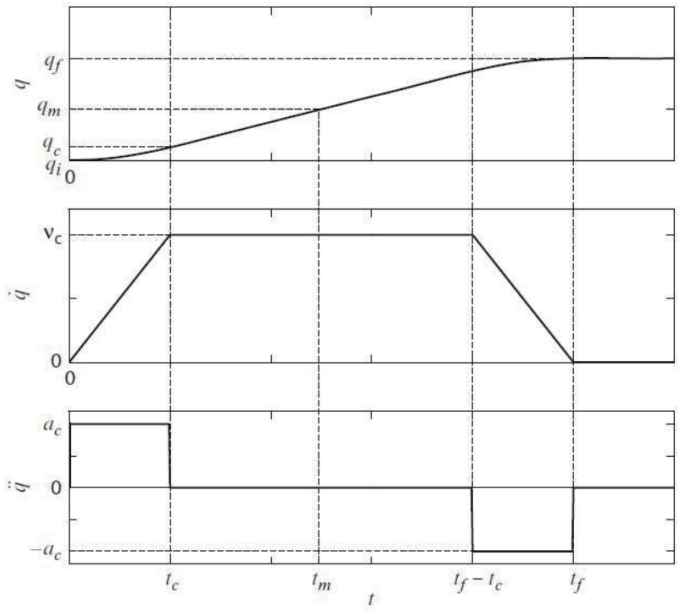


Figure 5.4: Illustration: the trapezoidal velocity profile. q is the position, $\dot{q}$ is the velocity and $\ddot{q}$ is the acceleration

As of today, there are 47 registered gestures for 13 gesture names. All these gestures are stored in a file, simply named *gesture*, whose content is visible in Appendix F.11[7]. A complete summary of the gestures' names, the number of associated gestures and what effect they have on the robot can be found in Table 5.1. When a gesture calls a specific routine on the Raspberry, the associated *command_index* value is specified. If not specified, by default, *command_index* is zero.

The command sent from the receiver to the Raspberry board has the following format:

`[Velocity1, Direction1, Velocity2, Direction2, command_index]`

It is to be noted that on the left motor, a direction of 1 will make it spin in forward direction while a direction of 0 will make it spin in backward direction. The opposite goes for the right wheel, as previously explained in Section 4.5. This command is sent in 5 bytes on the I2C bus and stored in the `storeTarget[5]` variable.

The precise values sent to the micro-controller depend on the detected gesture and the internal state of the robot, stored in the record *movState* inside the `sendOrder` module. The following details more what each command does internally[8]:

- `forward` and `backward` will send as velocity values the one stored in the state along with the appropriate directions. Typically, for the `forward` command:

`[currentVelocity,1,currentVelocity,0,1]`

[7]Note that the file ends with a gesture labelled `templateMove`. This is one of the gestures that was used by the author of the gesture recognition algorithm to test it. It was kept for debugging purposes.

[8]The complete illustration of the gestures and the associated commands and internal states can be found in Appendix C.3.

Table 5.1: The gestures and their effect on the robot. A gesture name can have multiple gestures associated to it, as illustrated in Appendix C.3

Gesture name	Number of associated gestures	Effect on the robot
`stopCrate`	9	Stops the robot by using `slowingDownToZero()` (*command_index* = 2)
`forward`	7	The robot moves forward
`backward`	7	The robot moves backward
`forwardTurnRight`	5	Makes the robot turn right, while going forward
`forwardTurnLeft`	5	Makes the robot turn left, while going forward
`backwardTurnRight`	4	Makes the robot turn right, while going backward
`backwardTurnleft`	4	Makes the robot turn left, while going backward
`turnAround`	1	Makes the robot spin on itself by calling `turning()`. *command_index* = 3
`changeVelocity`	1	The robot enters in `changeVelocity` mode. Stops the robot by using `slowingDownToZero()`, as a security measure. *command_index* = 2
`accelerate`	1	The robot changes its internal state by increasing the velocity by steps of 10 RPM, up to 120 RPM maximum. No movement
`decelerate`	1	The robot changes its internal state by decreasing the velocity by steps of 10 RPM, down to 100 RPM minimum. No movement
`testingVelocity`	1	Launches the routine using `testingTheNewVelocity()`. *command_index* = 1
`exitChangeVelocity`	1	Robot goes back in normal control mode. Stops the robot by using `slowingDownToZero()`, as a security measure. *command_index* = 2

- Any `turn` movement is accomplished by sending the appropriate directions to either make the wheels spin in forward or backward direction, in accordance with the gesture detected, and fixed velocity values for the wheels. These values are 80 RPM and 110 RPM. The 80 RPM value was chosen based on the motors' datasheet[41]: indeed, it is said that when receiving a PWM command with duty-cycle $\leq 60\%$ (so a digital PWM value ≤ 153), the motors' graphite brush wears down faster. Thus, to avoid damaging the motors too much, the minimum value of 80 RPM was chosen. It still leads to a duty-cycle of $\leq 60\%$ (on average, it turns around 45-50%), but since it is only for a determined amount of time, it was deemed acceptable to do so. Finally, the 110 RPM was chosen arbitrarily and could be increased or decreased. The smaller this velocity, the larger the turn the robot makes, and vice-versa.
- `turnAround` will use a default velocity of 100 RPM. This choice of value is arbitrary. Here, as the wheels need to spin in opposite directions, both `Direction` values are the same and equal to 1.
- `accelerate` and `decelerate` both change the *currentVelocity* variable of the state, but send a command of 0 RPM to the motors. The *currentVelocity* variable can evolve between 100 RPM and 120 RPM, by steps of 10 RPM. A maximum of 120 RPM was chosen because the motors' rated velocity is 125 RPM $\pm$ 15% as mentioned in Section 3.2.3, and because depending on the floor on which the robot is moving, to reach this maximum value, a different PWM command will be necessary: indeed, if the floor is perfectly flat and hard, like tiles, it

will be easier for the robot to move than on a carpet floor because there is less friction. So by choosing this lower maximum value, it helps the robot to deal with more floor types.

- `changeVelocity` and `exitChangeVelocity` change the *movMode* state variable, and send a complete stop command to the Raspberry board: `[0,1,0,0,2]`. This is because upon receiving this gesture name, the *movName* variable is changed to `stopCrate` in order to stop the robot, which sends the aforementioned command.

  Note that it was decided that, by design, whenever the user enters the `changeVelocity` mode, the velocity is reset at its base value of 100 RPM. This allows the user to always know with certainty which velocity is currently stored in the state when he wants to modify it.

- Finally, `testingVelocity` sends the following command:

  `[currentVelocity,1,currentVelocity,0,1]`

Lastly, it is to be noted that this second implementation allows for slow and fast gestures to be detected. What is meant by this is that, in order to keep as much control over the robot as possible, the minimum time it takes for a gesture classification to be launched has been set to 1.5 s since last detected movement[9] and to get even faster gesture detection, lists with only the starting and ending values of the gesture were added. Combining these two means that a gesture can be detected reliably in around 2.588 s as it takes 0.863 s to fetch the necessary 100 PNAV data to get the two averaged values and four more averages, so 1.725 s, to meet the 1.5 s timing condition and launch classification. The precise time taken between two gestures will be analysed in Section 6.4.

A "slow" gesture is a gesture in which, for every axis, all the possible values between the first and last averages are present: e.g., if the first average value for an axis is `nn` and the last is `pp`, a slow movement is described by `[nn,n,o,p,pp]` and a fast movement by `[nn,pp]`. It is to be noted that these two lists, despite representing the same movement, are seen as belonging to two separate gestures by the algorithm.

In the *gesture* file, the following movements were given "fast" gesture versions: `stopCrate`, `forward`, `backward` and every turn movement except `turnAround`. Indeed, it is expected that the user stops the crate before making the robot spin on itself. A special guard-rail has been implemented to make sure this is the case and avoid unexpected behaviours. Also, it was decided not to give "fast" gesture versions to the last five gestures of Table 5.1 as the robot is always put at rest by these gestures.

5.6 The guard-rails

These act as protective measures for the application. To protect the system from hardware failures or unexpected problems, two guard-rails have been implemented:

1. The first one monitors the connection between the two GRiSP2 boards. In the eventuality where the detector gets disconnected from its power supply, or if the Wi-Fi network goes down, for example, the two boards will not be able to communicate with one another, meaning the robot is no longer controllable. In order to limit the risks if this happens, the receiver attempts to connect to the detector every second using `net_adm:ping/1`. If the connection returns `pang`, it means the receiver was not able to connect to the detector, which means the latter is most likely down. It then immediately sends an order to the Raspberry board to stop the robot.

[9] This minimum value was found empirically. A lower value was found to lead to more unpredictable behaviours.

2. The second one monitors the connection between the receiver and the Raspberry board. Indeed if for some reason the receiver goes down, the micro-controller will no longer receive any commands and the robot is no longer controllable. To avoid this, every second, the receiver sends a "*1*" on the I2C bus, which is received by the Raspberry and added to an internal counter. The value of this counter is checked every second and if the new counter value is the same as the previous one, it means that nothing has been sent by the receiver for at least 1.999 seconds[10], meaning that the communication between the two boards is down.

Note that, in the receiver, both these guard-rails are implemented in the `sendOrder` module by the function `checkingConnection/1`, while the counter incrementation and the monitoring of its value are done within the Raspberry board. Moreover, it is to be noted that if the detector sends messages to the receiver, the other way around is not true. This is a design choice and it means that whenever there is a hardware failure in the robot, the detector will never notice anything and keep trying to send the detected gestures, to no avail.

A third guard-rail is implemented within the `handle_call/3` function of the `sendOrder` module: if for some reason the user goes from one gesture labelled as *forward* to one labelled as *backward* without stopping first, the guard-rail will forbid this transition and stop the robot to avoid any shock due to the rapid switch in direction of the wheels. This is simply done by comparing the seven first letters of each newly received gesture to the seven first letters of the previously registered gesture. If either *forward* and *backwar* or vice-versa are detected, the stop occurs. To keep track of the movement direction, each time a movement starting by *forward* is detected, the *prevName* state variable is updated to that. The same goes with *backward*.

Finally, in order to make sure that the user has stopped the robot before performing the `turnAround` routine, a fourth guard-rail uses the same aforementioned principle: if the previous move was not `stopCrate`, then the crate is stopped and the `turnAround` command is discarded.

[10]This is due to the fact that if the last sent value is received right after the checking of the condition, say 0.001 s after, then the Raspberry board will not detect any problem the next time it checks the condition, despite the receiver being down: it will only detect it at the following check.

Chapter 6

Evaluation: system performance and limitations

In this chapter, the system is fully characterised. The tests performed will be explained and results will be shown, and limitations of specific parts will be discussed. Finally, the chapter ends with a critical look at the system in general and a discussion of its global limitations.

The chapter is organised as follows:

- Section 6.1 goes over the controller's performance. The performance of the velocity profile, the outputted PWM values, travelled distance and performance of the robot while turning are analysed before the section ends with critical comments about the controller.
- Section 6.2 analyses the performance of the control routines by showing how well they perform what they are meant to do.
- Section 6.3 shows the performance and timings of the hardware guard-rails between the two GRiSP2 boards and between the receiver and the Raspberry board.
- Section 6.4 shows the time it takes on average to detect gestures and the timing performance of the gesture recognition algorithm.
- Finally, Section 6.5 describes the global system's limitations and also explains the *buffered_logger* added to Hera to overcome the platform's limitations.

Illustrations and the complete data tables can be found in Appendix D.

6.1 Controller performance

To analyse the velocity controller's performance, a first experiment where the robot was tasked to go forward for eight seconds before being ordered to stop was performed. From this experiment, several metrics will be analysed:

1. the accuracy of the velocity controller, which was established by comparing the measured wheel velocity evolution to the ideal one;
2. the value of the PWM commands sent to the motors at steady-state;
3. how the distances travelled by the robot during these tests compare with the theoretical ones;

This experiment is illustrated in Figure D.1 and Figure D.2.

6.1.1 The continuous velocity profile

To analyse how well the controller follows the trapezoidal velocity profile presented in Section 5.5, this experiment was performed for all the velocities at which the wheels could be spinning, namely, 80[1], 100, 110 and 120 RPM. The results of these tests can be seen respectively in Figure 6.1, Figure 6.2, Figure 6.3 and Figure 6.4. Only the velocity of the left wheel is shown in these figures.

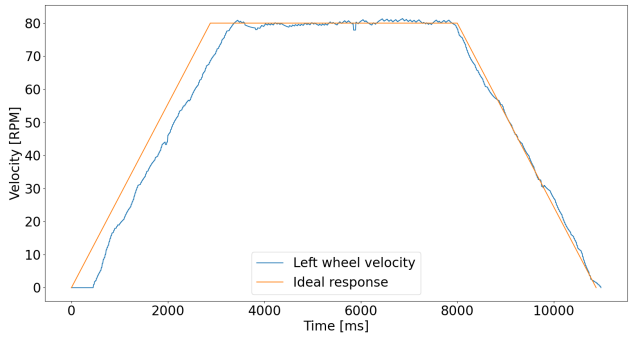


Figure 6.1: Velocity profile at 80 RPM

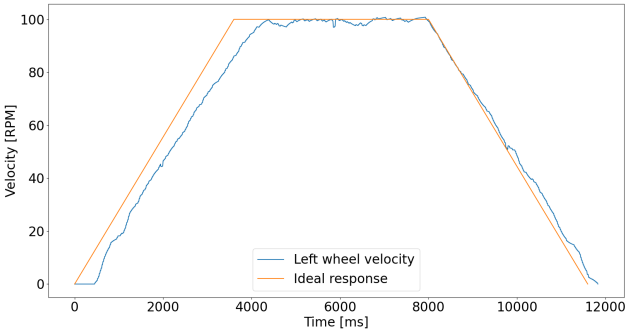


Figure 6.2: Velocity profile at 100 RPM

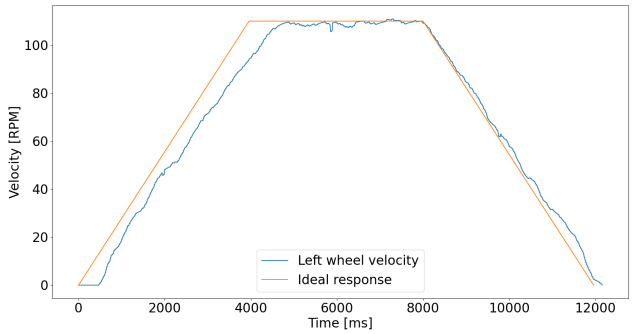


Figure 6.3: Velocity profile at 110 RPM

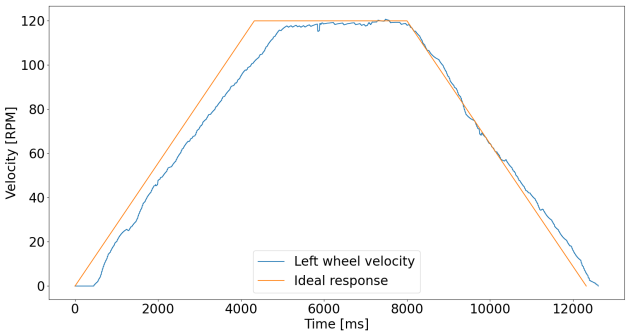


Figure 6.4: Velocity profile at 120 RPM

From these figures[2], one can observe:

- There is a very small overshoot in the velocity profile at 80 RPM, at the end of the acceleration period. However, no overshoot can be observed in the other tests. In fact, it can be seen that for 120 RPM, the controller does not reach immediately the correct velocity and relies on the integral term to correct the steady-state error.
- For each velocity, there is a delay at the start between the ideal and the measured velocity. The reason is that, for the motors to be able to move the weight of the robot, it is necessary to have a certain PWM value sent to them, so that the voltage applied is strong enough to produce the required torque. As one can observe, for every velocity tested, there is a "bump" at start in the measured velocity, which translates the fact that the controller knows it is behind the desired command and outputs a strong PWM value to the motors. But as the wheels approach the ideal velocity value, the command decreases, and so does the PWM value, which explains that the measured velocity seems to slow down slightly, before increasing again since it remains behind the desired one. This "bouncing" effect can be seen multiple times in the figures.
- When slowing down, the measured velocity is very close to the ideal velocity profile. However, for low velocities, it tends to deviate from the ideal curve.

[1]This happens when the robot turns. If it starts turning from zero, one wheel will have to reach 80 RPM.
[2]Enlarged versions of these figures can be found in Appendix D.1

At steady-state, the velocity oscillates a little around the correct value, which was to be expected as has been explained in Section 4.4. These variations are summarised in Table 6.1. Note that there are small peaks away from the steady-state value, which could be due to the nature of the ground, as there are joints between the tiles on which the robot was moving.

Table 6.1: Velocity variation around the steady-state value for each velocity

Tested velocity [RPM]	Minimum value [RPM]	Maximum value [RPM]	Max deviation from setpoint [%]
80	77.89	81.36	2.71
100	97.01	100.8	3.08
110	105.63	110.94	4.14
120	115.27	120.72	4.1

Despite these erroneous peaks, the velocity does not exceed a variation of 4.2% around the setpoint.

6.1.2 PWM values at steady-state

It was noted earlier in Section 3.2.4 that there were big differences between the PWM values sent to the two motors. However, since then, the velocity filter has been reworked, which significantly improved the outputs[3]. This is visible in Table 6.2. Note that, in a forward motion, the two wheels can only go at 100, 110 and 120 RPM, which is why no values are presented for 80 RPM[4].

Table 6.2: PWM value observed on the wheels for a given velocity

Tested velocity [RPM]	Wheel	Average PWM value [/]
100	Left	154.80
	Right	151.43
110	Left	181.57
	Right	177.13
120	Left	212.45
	Right	216.04

These values were obtained by taking the average of three tests. At each test, an average was made by taking a sample of ten PWM values at three different times : 7 s, 7.2 s and 7.4 s after the robot was ordered to move forward, in order to be certain that the robot was at its steady-state velocity. So overall, one value shown in the table corresponds to an average made over 90 values, one test providing 30 values.

Compared to previously, the PWM values are now much closer on average to one another, as the maximum difference between the two wheels is 4.44 PWM, registered at 110 RPM. As one can observe, the right wheel is no longer always the one which receives the higher command, as for both 100 and 110 RPM, the left wheel was the one receiving, on average, the biggest PWM command. Also, the values are almost always above the recommended limit of 153 PWM, to have a duty-cycle of $\geq 60\%$ on the motors.

[3]However, it seems hard to believe that a small change in the velocity filter would have this much of an impact. The author believes that, since these data were obtained using a different starting point than the one used to measure the PWM value when testing different battery placements, the difference in the floor structure probably impacted the results. In fact, by making the robot start from another point, an average difference of $\simeq 12$ PWM between the two wheels was observed.

[4]Note: [/] means that the value has no unit.

Note: it was mentioned earlier that a value of 120 RPM had been chosen as the maximum possible value achievable by the robot to help it deal with different types of floor. As it turns out, the robot can only reliably reach 100 RPM on a carpet floor (as illustrated in Figure D.8), as on the left and right wheels, an average PWM value of 244.17 and 242.44 respectively were observed[5].

6.1.3 Travelled distance during the velocity profile experiment

Since the velocity theoretically follows a trapezoidal profile, it is easy to compute the theoretical distance that the robot would be travelling during these tests. Indeed, since the acceleration and deceleration phases are symmetric, the same travelled distance is to be expected during these phases and is equal to:

$$distance = \frac{1}{2} a_c t_c^2 \tag{6.1}$$

where a_c is the acceleration/deceleration on the wheels and t_c is the total time it takes to accelerate/decelerate.

In the controller's code, 0.5 RPM are added every 18 ms to the target velocity. This is a design choice: one could have desired the velocity to reach the setpoint within a fixed amount of time, leading to different acceleration values, but it was here decided to have the same acceleration for every desired setpoint. Table 6.3 summarises the time it takes for the controller to reach the setpoint and the distance travelled during that time.

Table 6.3: Summary of the time it takes for the `target` variable within the controller's code to reach the desired setpoint and the travelled distance during that time

Tested velocity [RPM]	Time [s]	Theoretical distance [m]
80	2.88	0.483
100	3.6	0.754
110	3.96	0.912
120	4.32	1.086

Indeed, since:

$$v_c = a_c t_c \tag{6.2}$$

where v_c is the setpoint value of the robot's speed in m/s[6], one can quickly find that the robot experiences a constant acceleration $a_c = 0.116$ m/s^2 during the acceleration and deceleration phases and the travelled distance is computed using Equation 6.1.

Finally, since the robot is ordered to move forward for eight seconds before stopping, the total theoretical distance travelled is given by:

$$totalDistance = 2 \times dist_{acc} + (t_{tot} - t_c) \times speed_{robot} \tag{6.3}$$

where $speed_{robot}$ is the setpoint velocity converted in m/s, t_{tot} is 8 s, the time during which the robot is ordered to move forward and $dist_{acc}$ is the distance travelled during the acceleration and deceleration phases. The final theoretical values are summarised in Table 6.4. In the same table, the distances actually travelled by the robot are shown. For each velocity, these were obtained by measuring the distance travelled in three separate tests and averaging them.

[5]These values were obtained by using the same principle explained earlier.

[6]Which is found by converting the RPM into RPS by dividing by 60, then knowing that since the wheels have a diameter of 8 cm, one wheel turn is equal to 0.2513 m.

Table 6.4: Summary of the theoretically travelled distance during a test for each velocity and the distance travelled in reality

Tested velocity [RPM]	Theoretical distance [m]	Travelled distance [m]
100	3.355	3.347
110	3.686	3.630
120	4.02	3.967

Overall, the measured travelled distance is very close to the theoretical one[7].

6.1.4 Deviation from the straight path

It was observed, during all these tests, that the robot would not always go in a straight line. Most of the time, it would divert from the ideal path. This effect was found to be extremely random as no link between the observed deviations and the velocity of the robot could be made. The curious reader can refer to Appendix D.1.4 to get more details about this.

6.1.5 Performance of a turning motion

In a second experiment, illustrated in Figure D.9, the reliability of a turning motion was analysed.

Theoretically, it was found that the outer wheel of the robot, considering the velocities sent to the controller (80 and 110 RPM), would describe a circle whose radius is equal to 1.044 m. Indeed, in a circular motion, it is found that:

$$d = \frac{2\pi R\alpha}{360} \tag{6.4}$$

where d is the length of the travelled arc in m, R is the radius of the circle in m, and α is the opening angle, in degrees. Since both wheels travel the same opening angle, it is found that:

$$\frac{r_1}{v_1} = \frac{r_2}{v_2} \tag{6.5}$$

where r_1, r_2 are the radii, in m, respectively of the inner and outer wheels[8] and v_1, v_2 are the travelling speed of the wheels in m/s. Finally, knowing that the wheels are separated by a distance $L = r_2 - r_1 = 28.5$ cm, the theoretical radius of 1.044 m is found. This reasoning is illustrated in Figure D.10.

In practice, an average $r_2 = 1.083$ m was found, which is close to the expected value. This average is found using the measured radii in three different tests.

6.1.6 Criticism and limitations

Overall, the velocity controller shows excellent results, with travelled distances close to the expected ones, very similar average PWM values for both motors and a velocity profile close to the ideal trapezoidal one.

[7] It is to be noted that some tests were affected by a strange behaviour of the motors: when the freebear wheel became stuck in a joint, the motors would slow down slightly, seemingly for no reason. This happened in the first two of the three tests at 120 RPM, and in one test at 110 RPM. This is visible in Table D.3.

[8] It is considered that the robot turns to the left in a forward motion.

However, it is to be noted that the implemented velocity controller is the simplest form of control possible, as:

- no constraints (on the movement, on the wheels, etc.) are taken into account;
- there is no way to keep track of the robot's trajectory. In fact, nothing is implemented to correct it in case it diverges from its ideal trajectory;
- in cases where the wheels slip, or if the robot is stuck, the controller has no way of knowing that the robot is not moving. This could be dramatic, as the robot could fail a routine: when the wheels slip, the robot thinks it is moving, but it is not. The only feedback that is currently available is the human eye;

6.2 Performance of the control routines

6.2.1 Performance of `testingTheNewVelocity()`

As explained in Section 5.4, this routine allows the user to test the newly set velocity. In theory, the robot should travel, in m, the distance given by Equation 6.6:

$$distance = dist_{acc} + 2 \tag{6.6}$$

where $dist_{acc}$ can be found in Table 6.3, corresponding to the distance travelled by the robot when it slows down. The actual distance travelled, for each velocity, was found by performing the routine three times and averaging the results. The values for the theoretical and measured travelled distances are shown in Table 6.5.

Table 6.5: Summary of the theoretical and measured travelled distance for the `testingTheNewVelocity()` routine. Note: the distance shown is taken between the front of the robot at the end of phase 1 and the front of the robot at the end of phase 2. To get the total distance travelled, one can simply multiply by 2, as the robot performs a back and forth motion

Tested velocity [RPM]	Theoretical distance [m]	Travelled distance [m]
100	2.754	2.753
110	2.912	2.926
120	3.086	3.113

Once again, the measured distances are very close to the expected ones. Note that the system overshoots a little for both 110 and 120 RPM. This experiment is illustrated in Figure D.11.

6.2.2 Performance of the `turnAround` routine

This routine performs a 180° turn by making the wheels rotate in opposite directions at a target velocity of 100 RPM. As explained in Section 5.4, the routine is composed of two steps:

1. It starts by accelerating to the target velocity, then;
2. When it reaches a predetermined number of wheel turns, it slows down, counting on its inertia to reach the 180° turn.

An experiment was performed to see how reliable the turn is. By measuring the distance separating the back and the front of the crate from the line next to which the robot was positioned at the start, it is possible to evaluate how much it diverts from a perfectly parallel position. The result can be found in Table 6.6. Note that these values have been obtained by averaging the result

of five tests, and that the angular deviation is found by using trigonometry, since it is known that the distance separating the back and the front of the robot is 33 cm. The experiment and the way the angular deviation are computed are illustrated in Figure D.12 and Figure D.13 respectively.

Table 6.6: Average distance between the back and the front of the robot from the starting line and angular deviation to a final position perfectly parallel to that line

Distance back [cm]	Distance front [cm]	Angular deviation [°]
2.56	2.68	0.21

The routine is very accurate on average, but as one can see in Table D.8, it can overshoot or undershoot on some occasions.

6.3 Guard-rails performance

As explained in Section 5.6, there are two guard-rails which are meant to stop the robot whenever there is a hardware problem, whether the fault occurs on the detector or the receiver.

6.3.1 Guard-rail between the two GRiSP2 boards

This guard-rail stops the robot whenever the receiver detects that the detector is down, whether this is due to a Wi-Fi problem or because the detector is no longer powered. To see how quickly the robot stops when the detector goes down, five tests were made, in which the detector was unplugged from its power source and the time it took for the robot to slow down to a complete stop, starting from 100 RPM, was measured.

In theory, the receiver pings the detector every second and if it is not able to connect to the board, it immediately sends a stop order to the robot. Since it takes 3.6 seconds for the robot to stop when starting from 100 RPM, the system should take at most 4.6 s to slow down. The result of this experiment is shown in Table 6.7.

Table 6.7: Time it takes for the robot to fully stop when the detector is unplugged

Max theoretical time [s]	Average time [s]
4.6	4.3762

The measured average time to stop is below the theoretical maximum, which is what was expected. On average, the guard-rail takes 0.776 s to detect the problem.

6.3.2 Guard-rail between the receiver and the Raspberry board

The second hardware guard-rail allows the Raspberry Pi Pico W board to detect if the connection between it and the receiver is not working, whether this is because the receiver or the I2C communication between the two is down. Every second, the receiver sends a value of 1, using I2C, to the Raspberry, which adds it to an internal counter. The Raspberry itself checks every second if the counter has changed. If it did not, it means that the GRiSP2 board did not send anything for a maximum of 1.999 s, as explained in Section 5.6.

To measure how quickly the guard-rail acts, a test was performed, in which the receiver was unplugged from its power source and the average time it takes for the robot to fully stop, starting from 100 RPM, was measured. Five trials were performed, and the resulting average time can be seen in Table 6.8.

Table 6.8: Time it takes for the robot to fully stop when the receiver is unplugged

Max theoretical time [s]	Average time [s]
5.599	4.123

Again, the guard-rail quickly reacts to the problem and the average time is below the maximum theoretical reaction time. On average, the guard-rail takes 0.523 s to detect the problem.

6.4 Gesture recognition timings

To evaluate the speed at which the gesture recognition algorithm can perform classifications, two experiments were performed:

1. In a first one, the minimum time between gestures was evaluated by launching the algorithm and observing how quickly three consecutive `stopCrate` gestures were detected. The board did not move during this experiment, and five tests were performed.
2. In a second one, the board was moved according to the following pattern: `forward` from a stop position, then `forward` by keeping the board up, then back to `stopCrate` from the upward board position and next, two `stopCrate` by keeping the board flat on the table. Again, five tests were performed.

Both these experiments are illustrated in Figure D.14. To measure the time, the shell would be used to visually see when the algorithm had classified a gesture, and an online stopwatch was used to record the time.

The results for both these tests are shown in Table 6.9. Theoretically, a minimum of 1.725 s is to be expected between two gestures leading to the same name, since it takes 0.4313 s to fetch 50 data from the PNAV and that since the averages remain the same, it will take three more averages and thus a total of $4 \times 0.4313 = 1.725$ s for the algorithm to meet the timing condition and thus perform the classification[9]. Between `forward` and `stopCrate`, since the gesture list will contain two averages, the minimum theoretical time that will elapse is 2.157 s, following the same reasoning[10].

Table 6.9: Results for the two aforementioned experiments. The values shown correspond to the time it takes for a new gesture to be detected, since the previous one

Gesture number [/]	Time taken [s]: experiment 1	Time taken [s]: experiment 2
1	2.894	3.612
2	1.9092	1.93
3	1.77	3.516
4	/	1.941
5	/	1.76

[9]It is to be noted that due to the way the algorithm works, for the first gesture, a minimum of 2.157 s is to be expected, as it takes 0.4313 to get the first average and the *Time-Since-Movement*, used to verify the timing condition, is set to the last time at which a data was fetched, and so the algorithm has now to wait for 1.725 s to meet the timing condition.

[10]Theoretically, for a gesture which only needs two averages, it would take at most 3.45 s to be detected. Indeed, after the first average has been computed, which takes 0.4313 s, if the user decides to move during the last possible period before the classification, meaning during the fourth interval, the algorithm will detect a new average and set the new *Time-Since-Movement* at the latest value recorded, so at 1.725 s. Then, the algorithm will keep going and to pass the timing condition, four averages are needed to exceed the 1.5 s limit. So in total, eight periods of 0.4313 s are required, theoretically.

The values shown in Table 6.9 can be considered to be the time it takes for the robot to receive new commands, as the delay between the `rpc:call/4` and the robot reaction to this new command is negligible.

One can see, in the first experiment, every gestures are above the expected theoretical minimum. However, for the second experiment, both the first and second gestures are above the theoretical maximum: this is probably due to human error which leads to an over-estimation of the time.

Finally, one can see that the first gesture, in both experiments, takes more time to be detected, especially in the first experiment where there is a difference of almost one second between the time it takes to detect the first and second gestures, despite the board not moving. Besides the human-error factor, this longer time to detect the first gesture, besides the way the algorithm works, is probably due to the `grdos/12` function launching, which creates many variables within the GRiSP2 board.

Despite these times being within the expected range, the robot proves hard to manoeuvre on the ground, as will be discussed in the next section.

6.5 System criticisms and limitations

Overall, the robot's performance is really good and it achieves what was expected from it: the velocity controller makes the robot follow a trapezoidal velocity profile reliably, the routines give consistently good results, the PWM values are above the recommended 153 PWM limit, etc. Moreover, the robot's design is cheap, intuitive, and meets all the requirements.

However, there are many aspects which need to be factored in when evaluating the robot's performance.

6.5.1 The robot's controllability

The robot proves to be sometimes difficult to control with the remote controller. The controllability of the robot is good, but may not good enough to go over a "challenging" environment. What is meant by *challenging environment* is anything that is not perfectly flat ground made of hard tiles with no obstacles in its path.

Moreover, it was observed that in practice, the gesture recognition algorithm could have some difficulty correctly detecting two consecutive gestures. This makes the continuous control of the robot harder to achieve, as a `stopCrate` command is sent to the robot whenever a gesture is not classified with enough accuracy.

Most of the time, a gesture is wrongly detected for two reasons:

1. Due to the "fast" gestures, which only contain at most two averages per axis, and due to how the accuracy is computed, whenever the user makes a gesture in a slightly different way than expected, an extra average or a bad value can be detected and the accuracy will most certainly end up below 70%;
2. If the user starts a new gesture not immediately after the previous one has been classified, it has been observed experimentally that it would lead to more badly classified gestures;

The limitation is here clearly due to the gesture recognition algorithm. When used "correctly", that is, when any gesture which changes the board's orientation from its previous one is started right after the previous classification, then the application performs well.

Beyond the limitations linked to the algorithm, the hardware used also limits the controllability. Indeed, the motors are not as well suited for this application as it was initially believed. Their biggest problem is the 60% PWM lower-bound, which means that they cannot be used at low velocity, which lowers the controllability of the robot as it is moving fast (minimum 100 RPM in a straight motion, which is here equivalent to 41.89 cm/s). Also due to this limit, a turning motion describes a wide circle, which makes it harder to go around potential obstacles, while the freebear wheel is a limitation in itself, as it can get stuck in joints or be affected by them.

6.5.2 The gesture recognition

Beyond affecting the robot's controllability, the gesture recognition algorithm has other limitations:

- The list of possible known gestures is very limited, as the algorithm can only correctly detect rotations. Moreover, due to this limitation, two gestures that are completely different can be detected as the same. This is the case with the gesture in which the board goes from a `forwardTurnLeft` to a `forwardTurnRight` position and vice-versa, as illustrated in Figure C.10. The fast version of the gesture is exactly the same as its counterpart in the `backward` direction, but the algorithm will only return the first gesture from the list which fulfils the accuracy requirement. This is why the careful reader will have noticed that in Table 5.1, there are only four gestures for turning in the backward direction, while there are five for the forward direction: the fast version of the aforementioned movement was simply deleted in the backward direction, as it would never be outputted by the algorithm.
- To get a faster gesture recognition, fetching the data more quickly would be a step in the right direction. Currently, the system is limited to a fetching frequency of 115.92 Hz. This problem is due to the Hera platform, which is slowing down the fetching process. Another way of getting faster classification would be to have less noisy data to begin with and perform an average on less data.

6.5.3 Other system limitations

This section mentions the general limitations of the current system.

- The detector is connected via a cable to the computer. Previously, Hera users could launch the application on their computer in "default" mode and use the command `launch_all/0` to send an `rpc:call/4` order to all the nodes connected to the network. But this is not possible in this case, for two reasons:

  1. When connecting remotely from a computer to a distant node, it creates a distributed Erlang node solely for the purpose of doing a remote shell, effectively masking the node from every other node on the network, which means that the receiver cannot connect to the detector and vice-versa, effectively preventing the system from working.
  2. Due to the `numerl` NIF, it is impossible[11] to launch the application on the computer, as there are files missing to get a correct build of the application.

  A simple solution to this problem would be to go back to an outdated version of Hera to launch the application on the computer and launch the measurements remotely. However, for the sake of monitoring, it was decided to keep the cabled connection.

- The gesture detection never stops: when the algorithm is launched, the user needs to be constantly focused on what is happening to avoid losing control over the robot. The only way to stop it, currently, is to put the board in a `stopCrate` position and wait until the algorithm detects three of them in a row.

[11] Or at least, the author did not find any solution to this problem.

6.5.4 A contribution to Hera: storing the data

Hera has the ability to store the data that the sensors have fetched into a .csv file. This is included at the end of the `hera_data` module, using a logger that could be activated or not by setting an environment variable of the Hera application respectively to *true* or *false*. Furthermore, this logging of the data was happening directly in the measurement loop, right at the end. It however turned out that this method of logging the data was problematic. Indeed, when trying to save the data of the `nav3` and `e11` measurements, Hera crashed. It quickly turned out that the problem was due to an overloading of the system that could not handle the speed at which it had to log and write the data to the .csv file.

The main culprit has been found to be the logging of the `e11` data. The time evolution between two stored value is shown in Figure 6.5.

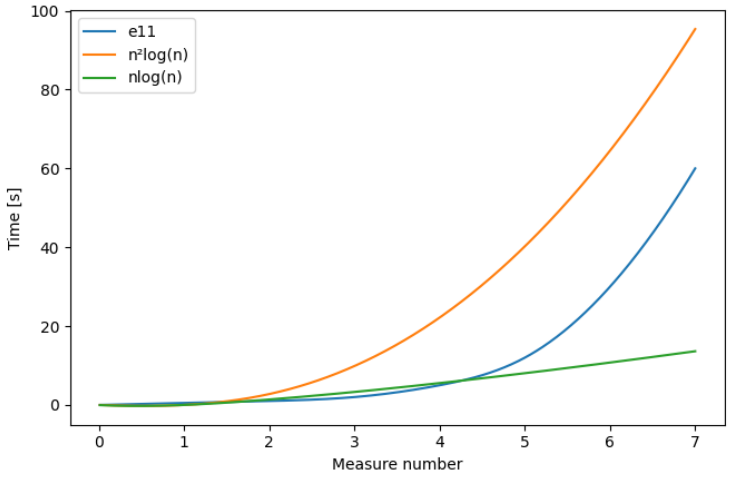


Figure 6.5: Logging of `e11` time evolution between two measures. The value for a measure number is the time taken since the previous measure to log it

This shows that the time it takes between two measures to log the data increases globally like a $n^2 \log(n)$ function delayed in time, for the first few data points. To prove that this was the problem, the logger was turned off, and Hera started working again without timing-out.

This is a limitation of Hera: this feature of storing the measurement data into a .csv file is a valuable debugging or analysing tool and a nice feature of the Hera platform. This is why a new logger was implemented, a `gen_server` which has been named *buffered_logger* and added to Hera as a process of its own to the Hera supervision tree. The platform still uses the `hera_data` `gen_server` to fetch the data, but now the data are stored in an synchronous way using this new `gen_server`. The curious reader can find more explanation on how the logger works in Appendix C.4.

Chapter 7

Conclusion

In this master thesis, an application for remotely controlling an object using the GRiSP2 board and the Hera platform was developed. Named *movement_detection*, this application is based on several modules, including `gen_server`s, that are completely reusable and modifiable to fit future applications. To show the potential of such an application, a robot on wheels was built, starting from a wine-crate, using two rubber wheels, actuated by 6 V DC-motors with integrated gearbox, to move around and one freebear wheel for stability.

The robot is controlled remotely using a GRiSP2 board, labelled the detector, on which a PNAV is plugged. By using the detected accelerations, filtering the data and comparing it to a library of known gestures that was created for the occasion, a gesture is classified periodically by the algorithm and if the accuracy of the classification is good enough, the gesture's name is sent over to the robot which then converts it into the appropriate command. Within the robot, a micro-controller board, the Raspberry Pi Pico W, is used to receive the orders from the receiver through I2C and control the robot's velocity.

The robot is described fully, from its mechanical aspects to the whole hardware used, in Chapter 3, while the controller and the application have been described respectively in Chapters 4 and 5. The prototype fulfils all its requirements and the application has been made safe by adding four guard-rails and safety measures in order to deal with wrongly detected gestures, hardware failures or any misuses of the application. Moreover, the application has had two versions: initially, to reduce the impact of the gesture recognition algorithm, the robot would be stopped often to keep as much control over it a possible, and precise gestures for turning, etc. were implemented, but this led to shocks in the robot's movements. This version was however not deemed satisfactory or user-friendly, so a second version of the application was designed, now implementing a continuous control of the robot which would follow a continuous velocity profile.

The built prototype proved to perform very well: through numerous experiments, the performance of the robot was analysed and it was shown in Chapter 6 that the theoretical expectations and the measured results were always close, whether it is regarding the travelled distance, following the appropriate velocity profile or performing precise routines. The gesture recognition algorithm proved to be reliable as well, with timings within expectations. Overall, the implemented system is really good at performing isolated movements.

However, due to how the gesture recognition algorithm works, gestures can be classified with a low accuracy, meaning that some gestures are "wasted". To avoid any risk of losing control, if this happens, the robot is stopped. This diminishes the overall controllability of the robot, but makes

it safer. Also, the motors used to build the robot proved to be less suited than expected, which further reduces the controllability of the robot.

In this final chapter, the author suggests future works and ways to improve the current system.

7.1 Future work and improvements to the current system

7.1.1 Put everything on the GRiSP2 board

This whole work has used the GRiSP2 board only to detect the gestures and receive them through the `rpc:call/4` function. But, technically, there is "one piece of hardware too many" in the robot: the Raspberry board. Indeed, it should theoretically be possible to do everything that the micro-controller does but on the GRiSP2 board. However, a number of obstacles stand in the way:

- Probably the biggest one is that the Erlang language "is not made for that". Erlang is a general-purpose, concurrent, functional high-level programming language[59]. However, to control motors, a low-level programming language like C, which is by design made to work with hardware, is much better suited.

  In fact, it was found by creating a simple application containing a loop that just prints the timestamp in the shell that the maximum frequency at which this simple application is running[1] is 1958.86 Hz. This is due to how Erlang works, by design, and the use of the Virtual Machine that is basically a scheduler which decides which process runs when. In this simple application, only one process was being run. An even slower frequency is to be expected when many modules work in parallel, as is the case with Hera. This frequency limitation could pose some problems when trying to perform Pulse-Width-Modulation, as the maximum frequency of a process is thus limited to 1958.86 Hz, which is fairly slow when it comes to controlling motors[2].

  Lastly, the fact that Erlang variables are immutable would require to work with a `gen_server` and big records to store the states, as well as defining many Erlang macros for constant values.

- About PWM: there are no drivers on the GRiSP2 that support it. A driver which allows to drive DC motors using the GPIO pins exists, but it only allows ON/OFF commands without supporting PWM operation[3]. In discussions with Peer Stritzinger, he mentioned the fact that the CPU used inside the GRiSP2 board certainly has PWM hardware and pins, but that a driver should be created to be able to access it, using dedicated NIFs, which could be the subject of a thesis on its own[4].

- There is no GRiSP equivalent to the `attachInterrupt()` function of the Arduino IDE, which is used to update the encoder count reliably. In discussions with Peer Stritzinger, he mentioned that it would not be hard to add this to the GPIO driver, so this should not pose too much of a problem.

It is to be noted that Cédric Ponsard and François Goens, who are also doing their theses this year, are reportedly developing a PID controller in Erlang, which they are adding as an extension to the Hera platform. So this part should not be a problem either.

[1] This has been found by averaging 10 values. The timestamp was printed in microseconds.

[2] https://electronics.stackexchange.com/questions/242293/is-there-an-ideal-pwm-frequency-for-dc-brush-motors

[3] https://erlangforums.com/t/grisp-2-support-pwm-to-dc-motor/1976

[4] Note: to avoid entirely the PWM issue, one could use other motors like stepper motors, which would be well suited for an application that requires precise control.

Porting this application entirely on the GRiSP2 would have the advantage that it could be possible to use the already existing sensor fusion techniques and even extend them by adding other sensors to keep a precise control over the robot at any point in time. This could allow a future application where the robot has to follow a certain path decided by the user's gestures and a Kalman filter could be used to precisely keep track of its position and/or trajectory, for example.

7.1.2 Improve the algorithm used to detect gestures

One way of improving future applications is to rework the gesture recognition algorithm, as it is rather slow and can prove to be imprecise, and it only uses the accelerometer data, which can be noisy. In the future, several ideas could be explored:

1. Clean the accelerometer data by performing Kalman filtering with the accelerometer data and another sensor's data, like the gyroscope, for example. A special model that allows to work with both an angular rate in degrees per second and a measured acceleration would need to be developed.

2. Change the way the accuracy is computed to be able to better deal with human mistakes in the gestures.

3. Change the whole algorithm to not use the raw accelerometer data, but rather to use, for example, the board orientation. This orientation is the output of the Kalman filter implemented in `e11.erl` and that uses the accelerometer and magnetometer of the PNAV to find the DCM, or Direct Cosine Matrix, and finally output a quaternion, which are explained in Appendix C.1.2. One way of changing the existing algorithm to use this measurement would be the following: instead of collecting data and then performing averages and using pattern matching, one could immediately detect the orientation of the detector in real-time and associate a particular orientation with a defined command. This could highly improve the speed at which the system responds to a "gesture", though the algorithm is no longer really performing gesture recognition: as long as a precise orientation is detected the associated command would be sent.

   This idea could be used in association with other techniques to improve the final result, or simply on its own. Depending on how precisely the orientation can be detected, this could open the gate to many opportunities, like turning more or less quickly depending on the orientation of the board or increasing gradually the velocity the more the board is upright, etc.

4. Use another sensor. In fact, a sensor that would be very interesting to explore is the UWB, which stands for Ultra-Wide-Band. This sensor would be able to continuously tell the GRiSP2 board the distance separating it from a second GRiSP2 board, acting as the "reference point", allowing easy 3D gesture recognition. Having the opportunity to detect linear gestures would tremendously improve the algorithm by allowing many new gestures. This Pmod™ could have many uses, for example:

   - a special gesture could be used to signal the end of the algorithm, instead of waiting for three `stopCrate` gestures as is currently the case;

   - using the receiver inside the robot as the "reference point" and the detector as the remote command, a "follower" mode could be added, where the robot follows the detector from a given distance.

5. Without the use of other sensors, more commands could be added whilst preserving the same number of gestures: the idea would be to allow the same gesture to have different meanings depending on the "mode" in which the robot is. To this end, the algorithm performing the

gesture recognition should be able to output different results for the same gesture, given some additional information.

6. Finally, a small change to the algorithm would be to clean the `grdos/12` function to stop performing the data processing twice for each gesture. This will improve very slightly the speed at which the algorithm runs.

7.1.3 Improve Hera

The platform is slowing down the process of fetching the data from the PNAV. If the data could be fetched more quickly, it would mean that an average would take less time to be calculated and thus, the gesture recognition algorithm would be faster. Overall, a faster platform is necessary to start performing the control of objects, etc.

More and more, the limitations of the platform are appearing. Its second version has been around for several years now, and since it was developed on the GRiSP board, a new and upgraded version for the GRiSP2 board is called for. This would open the gate to many more years of working with Hera and the GRiSP2 board, and many new applications.

7.1.4 Improvements to the application itself

Currently, the application relies on a computer to monitor the state of the system, launch the algorithm, etc. To this end, the detector is connected to a USB port of the computer via a micro-USB cable. This supplies the detector, whilst also allowing the computer to access the board's Erlang shell and launch modules and functions from there. In the context of this project, this was acceptable, as this is only a prototype and it allowed to bypass the problems mentioned in Section 6.5.3. However, in order to be more user-friendly, the detector should be powered by batteries and no cabled connection should exist between the board and the computer.

As was already mentioned, one way to launch remotely the `grdos/12` function would be to have an outdated version of the application running on the computer, without the `numerl` NIF. However, there would be no way of knowing the state of the system. But there could be a way around this problem: the Digilent Pmod™ BTN (which stands for "Buttons")[60]. This board, shown in Figure 7.1, provides four buttons and follows the Pmod interface specification type 1 (GPIO), for which a driver exists on the GRiSP2 board. One could use these buttons to, for example, start the gesture recognition algorithm on the board, or stop it. Combined with the LEDs of the GRiSP2 board, this would allow the user to know what the application is currently doing and to keep a full control over the remote controller, without having the need of a cabled connection.

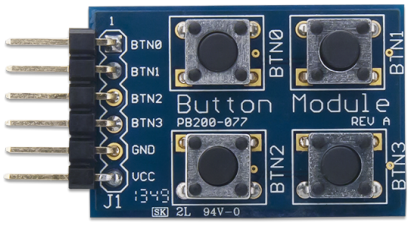


Figure 7.1: Illustration: The Digilent Pmod™ BTN

Lastly, on a side note, it is to be mentioned that the `sendOrder` module can certainly be improved. For such applications, a Finite-State-Machine (FSM) could be very well suited.

7.1.5 Improve the robot

This first prototype, although it works, is far from optimal. Based on the robot's flaws, the following can be improved to get a better robot if someone else wants to reproduce the work done in this thesis:

- Change the motors: the chosen motors were far from being the best suited for this application. They were chosen mainly due to their immediate availability, but much time would have been saved had the motors been better from the start. Stepper motors are an interesting option for this kind of application.
- Use other wheels: once the gestures are more precise, it could become interesting to use omnidirectional wheels. This would greatly increase the mobility of the robot on flat ground and allow diagonal movements, for example.
- Use a different starting point than a pre-existing crate: despite turning out well, the crate as the basic structure was challenging and in the end, the robot could have been much more compact had a dedicated structure been created for the occasion.

Bibliography

[1] IBM. *What is the Internet of Things (IoT)?* https://www.ibm.com/topics/internet-of-things. Accessed on 24-05-2024.

[2] Wikipedia contributors. *Cloud computing - Wikipedia, the free encyclopedia*. https://fr.wikipedia.org/wiki/Cloud_computing. Accessed on 24-05-2024. Apr. 2024.

[3] Robin R. Murphy. "Smart houses and domotics". In: *Science Robotics* 3.24 (2018). DOI: https://doi.org/10.1126/scirobotics.aav6015.

[4] Wikipedia contributors. *Home automation - Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Home_automation. Accessed on 24-05-2024. May 2024.

[5] Cristina Simonet and Alastair J. Noyce. "Domotics, Smart Homes, and Parkinson's Disease". In: *Journal of Parkinson's Disease* 11.s1 (2021), pp. 55–63. DOI: https://doi.org/10.1126/scirobotics.aav6015.

[6] Goldi Remington. *How To Control IoT Devices*. https://robots.net/tech/how-to-control-iot-devices/. Accessed in 24-05-2024. Sept. 2023.

[7] Lucas Nélis. *Low-cost high-speed sensor fusion with GRiSP and Hera*. Master thesis, Ecole polytechnique de Louvain, UCLouvain, 2023.

[8] Sébastien Gios. *Gesture Recognition by Pattern Matching using Sensor Fusion on an Internet of Things device*. Master thesis, Ecole polytechnique de Louvain, UCLouvain, 2023.

[9] Peer Stritzinger. *GRiSP 2*. https://www.kickstarter.com/projects/peerstritzinger/grisp-2. Accessed on 29-05-2024. Feb. 2022.

[10] Raspberry Pi Ltd. *Raspberry Pi Pico and Pico W*. https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html. Accessed on April 2024.

[11] Patrick Fromaget. *What is Raspberry Pi Pico? Everything You Need to Know*. https://raspberrytips.com/what-is-raspberry-pi-pico/. Accessed on April 2024. Feb. 2023.

[12] Digilent. *Pmod™*. https://digilent.com/reference/pmod/start. Accessed in March 2024. 2023.

[13] Digilent. *Digilent Pmod™ Interface Specification™*. https://digilent.com/reference/_media/reference/pmod/pmod-interface-specification-1_3_1.pdf. Accessed in March 2024. Oct. 2020.

[14] Digilent. *Pmod NAV*. https://digilent.com/reference/pmod/pmodnav/start. Accessed in March 2024. Jan. 2017.

[15] Sébastien Kalbusch and Vincent Verpoten. *The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with application to inertial navigation and tracking*. Master thesis, Ecole polytechnique de Louvain, UCLouvain, 2021.

[16] Digilent. *Pmod HB5*. https://digilent.com/reference/pmod/pmodhb5/start. Accessed in March 2024. Oct. 2016.

[17] Wikipedia contributors. *H-bridge - Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/H-bridge. Accessed in May 2024. Aug. 2023.

[18] Ericsson AB. *Erlang main page*. https://www.erlang.org/. Last accessed in May 2024.

[19] Ericsson AB. *What is Erlang*. https://www.erlang.org/faq/introduction.html. Last accessed in May 2024.

[20] Ericsson AB. *Processes*. https://www.erlang.org/doc/system/ref_man_processes.html. Last accessed in May 2024.

[21] Ericsson AB. *Data Types*. https://www.erlang.org/doc/system/data_types#pid. Last accessed in May 2024.

[22] Wikipedia contributors. *Functional programming - Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Functional_programming. Accessed in May 2024. May 2024.

[23] Ericsson AB. *Distributed Erlang*. https://www.erlang.org/doc/system/distributed.html. Last accessed in May 2024.

[24] Ericsson AB. *rpc*. https://www.erlang.org/doc/apps/kernel/rpc.html. Last accessed in May 2024.

[25] Man Lok Fung, Michael Z. Q. Chen, and Yong Hua Chen. "Sensor fusion: A review of methods and applications". In: *2017 29th Chinese Control And Decision Conference (CCDC)* (2017), pp. 3853–3860. DOI: 10.1109/CCDC.2017.7979175.

[26] Wikipedia contributors. *Sensor fusion - Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Sensor_fusion. Accessed in May 2024. Apr. 2024.

[27] Warren Triston D'souza and Kavitha R. "Human Activity Recognition Using Accelerometer and Gyroscope Sensors". In: *International Journal of Engineering and Technology (IJET)* 9.2 (2017). DOI: 10.21817/ijet/2017/v9i2/170902134.

[28] Ahmad Jalal, Majid Ali Khan Quaid, Sheikh Badar ud din Tahir, and Kibum Kim. "A Study of Accelerometer and Gyroscope Measurements in Physical Life-Log Activities Detection Systems". In: *Sensors* 20.22 (2020). ISSN: 1424-8220. DOI: https://doi.org/10.3390/s20226670. URL: https://www.mdpi.com/1424-8220/20/22/6670.

[29] Siregar B, Andayani U, Bahri R P, Seniman, and Fahmi F. "Real-time monitoring system for elderly people in detecting falling movement using accelerometer and gyroscope". In: *Journal of Physics: Conference Series* (2018). DOI: 10.1088/1742-6596/978/1/012110.

[30] Timothy Hirzel. *Pulse-width modulation*. https://docs.arduino.cc/learn/microcontrollers/analog-output/. Accessed in April 2024. Dec. 2022.

[31] Wikipedia contributors. *Pulse-width modulation - Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Pulse-width_modulation. Accessed in May 2024. May 2024.

[32] Shawn Dietrich. *Understanding the Basics of Pulse Width Modulation (PWM)*. https://control.com/technical-articles/understanding-the-basicsof-pulse-width-modulation-pwm/. Accessed in April 2024. Mar. 2022.

[33] Scott Campbell. *Basics of the I2C communication protocol*. https://control.com/technical-articles/understanding-the-basicsof-pulse-width-modulation-pwm/. Feb. 2016.

[34] Wikipedia contributors. *I^2C - Wikipedia, the free encyclopedia*. https://fr.wikipedia.org/wiki/I2C. Last accessed in May 2024. Jan. 2024.

[35] Benoit Raucent. "Specifications and pilot study". In: *LMECA2801 - Machine Design, Ecole polytechnique de Louvain, UCLouvain* (2016).

[36] Wright Mr. *Ultimate Guide to Different Types of Wheels for Robots*. https://www.awerobotics.com/ultimate-guide-to-different-types-of-wheels-for-robots/. Accessed in March 2024. 2024.

[37] Wikipedia contributors. *Hyperstatisme - Wikipedia, the free encyclopedia*. https://fr.wikipedia.org/wiki/Hyperstatisme. Last accessed in May 2024. Apr. 2023.

[38] Nathanaël Jarrassé and Guillaume Morel. "A formal method for avoiding hyperstaticity when connecting an exoskeleton to a human member". In: *2010 IEEE International Conference on Robotics and Automation* (2010), pp. 1188–1195. DOI: https://doi.org/10.1109/ROBOT.2010.5509346.

[39] Renaud Ronsse. "Lecture 2: Mobile Robot Kinematics". In: *LELME2732 — Robot modelling and control, Ecole polytechnique de Louvain, UCLouvain* (2021 - 2022).

[40] Wikipedia contributors. *Differential wheeled robot - Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Differential_wheeled_robot. Accessed in May 2024. May 2024.

[41] *6V, DC-Motor with integrated gearbox, IG220053X00085R Model*. 141209. Shayang Ye Industrial Co., LTD. 2014. URL: https://digilent.com/reference/_media/motor_gearbox/290-008_ig220053x00085r_ds.pdf.

[42] Curio Res. *How to control a DC motor with an encoder*. https://www.youtube.com/watch?v=dTGITLnYAY0&t=185s&ab_channel=CurioRes. Feb. 2021.

[43] Dejan. *How Rotary Encoder Works and How To Use It with Arduino*. https://howtomechatronics.com/tutorials/arduino/rotary-encoder-works-use-arduino/. Accessed in March 2024. July 2016.

[44] Dynapar. *Quadrature Encoder Overview*. https://www.dynapar.com/technology/encoder_basics/quadrature_encoder/. Last accessed in May 2024. Jan. 2013.

[45] *Two Channel Hall Effect Magnetic Encoder*. Shayang Ye Industrial Co., LTD. URL: https://digilent.com/reference/_media/motor_gearbox/magnetic-encoders.pdf.

[46] Dynapar. *Encoder Resolution, Encoder Accuracy and System Repeatability*. https://www.dynapar.com/Knowledge/Encoder_Resolution_Encoder_Accuracy_Repeatability/. Last accessed in May 2024. Mar. 2018.

[47] Surgay C. *A Complete Arduino Rotary Solution*. https://www.instructables.com/A-Complete-Arduino-Rotary-Solution/. Accessed in April 2024. Mar. 2018.

[48] Neil Cameron. *ESP32 Formats and Communication*. Springer Link: Maker Innovations Series. 2023. URL: https://link.springer.com/chapter/10.1007/978-1-4842-9376-8_1.

[49] Xavier Lemil. *What is an Interrupt?* https://www.geeksforgeeks.org/interrupts/. Last accessed in May 2024. Feb. 2024.

[50] PID-Explained. *PID Controller Explained?* https://pidexplained.com/pid-controller-explained/. Last accessed in May 2024. Nov. 2018.

[51] Wikipedia contributors. *Proportional–integral–derivative controller - Wikipedia, the free encyclopedia*. Last accessed in May 2024. Apr. 2024. URL: https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller.

[52] Vance J. Vandoren. *PID: Still the One*. https://www.controleng.com/articles/pid-still-the-one/. Oct. 2003.

[53] Vance J. Vandoren. *PID: Still the One*. https://www.controleng.com/articles/the-three-faces-of-pid/. Mar. 2007.

[54] Fred Herbert. *Callback to the Future*. https://learnyousomeerlang.com/clients-and-servers#callback-to-the-future. Jan. 2013.

[55] Ericsson AB. *gen_server behaviour*. https://www.erlang.org/doc/apps/stdlib/gen_server.html. Last accessed in May 2024.

[56] Ericsson AB. *Records*. https://www.erlang.org/doc/system/ref_man_records.html. Last accessed in May 2024.

[57] Arduino. *loop()*. https://www.arduino.cc/reference/en/language/structure/sketch/loop/. Last accessed in May 2024.

[58] Renaud Ronsse. "Lecture 6: Trajectory planning, revisited". In: *LELME2732 — Robot modelling and control, Ecole polytechnique de Louvain, UCLouvain* (2021 - 2022).

[59] Wikipedia contributors. *Erlang (programming language) - Wikipedia, the free encyclopedia*. https://en.wikipedia.org/wiki/Erlang_(programming_language). Accessed in May 2024. May 2024.

[60] Digilent. *Pmod BTN*. https://digilent.com/reference/pmod/pmodbtn/start. Accessed in May 2024. May 2016.

[61] *ESP32 Series, Datasheet*. Version 4.5. Espressif Systems. 2024. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

[62] Raspberry Pi Ltd. *Welcome to RP2040*. https://www.raspberrypi.com/documentation/microcontrollers/rp2040.html#welcome-to-rp2040. Last accessed in May 2024.

[63] Claudio Urrea and Rayko Agramonte. "Kalman Filter: Historical Overview and Review of Its Use in Robotics 60 Years after Its Creation". In: *Journal of Sensors* 2021 (Sept. 2021).

Publisher: Hindawi. ISSN: 1687-725X. DOI: 10.1155/2021/9674015. URL: https://doi.org/10.1155/2021/9674015.

[64] Greg Welch, Gary Bishop, et al. "An introduction to the Kalman filter". In: (1995).

[65] Ron Goldman. "Understanding quaternions". In: *Graphical models* 73.2 (2011), pp. 21–49.

[66] Wikipedia contributors. *Quaternions - Wikipedia, the free encyclopedia*. https://fr.wikipedia.org/wiki/Quaternion. Last accessed in May 2024. Mar. 2024.

[67] Jay A Farrell. "Computation of the Quaternion from a Rotation Matrix". In: *University of California* 2 (2015).

[68] Ericsson AB. *Erlang main page*. https://www.erlang.org/docs/17/man/ets. Last accessed in May 2024.

[69] Dave Mckay. *How to Add a Directory to Your $PATH in Linux*. https://www.howtogeek.com/658904/how-to-add-a-directory-to-your-path-in-linux/. Accessed in February 2024. Sept. 2023.

[70] Karl Söderby. *Downloading and installing the Arduino IDE 2*. https://docs.arduino.cc/software/ide-v2/tutorials/getting-started/ide-v2-downloading-and-installing/. Accessed in March 2024. Feb. 2024.

[71] Abhishek Gandal. *How to Install 'Python-Serial' package on Linux?* https://www.geeksforgeeks.org/how-to-install-python-serial-package-on-linux. Accessed in March 2024. Nov. 2021.

[72] Marko Aleksic. *How to Resolve the 'Cannot connect to the Docker daemon' Error*. https://phoenixnap.com/kb/cannot-connect-to-the-docker-daemon-error. Accessed in March 2024. Nov. 2023.

Appendix A

Hardware comparison: ESP32 NodeMCU-32S vs Raspberry Pi Pico W

Table A.1 compares some relevant aspects of the ESP32 Node-MCU32S[61] and the Raspberry Pi Pico W[10] in the context of the current application.

Table A.1: Comparative table: ESP32 NodeMCU-32S vs Raspberry Pi Pico W

	ESP32 NodeMCU-32S	Raspberry Pi Pico W
CPU	Xtensa single-/dual-core 32-bit LX6 microprocessor(s). Typ. frequency: 240 MHz	Dual ARM Cortex-M0+ @ 133 MHz
Memory	448 KB ROM, 520 KB SRAM and 16 KB SRAM in RTC	264 kB of SRAM, and 2 MB of on-board flash memory
Pins	34 programmable GPIOs	26 × multi-function GPIO pins
Peripherals	4 × SPI, 2 × I2C, 3 × UART, motor PWM and LED PWM up to 16 channels	2 × SPI, 2 × I2C, 2 × UART, 3 × 12-bit ADC and 16 × controllable PWM channels

As one can see, they have very similar hardware, with more memory and CPU speed for the ESP32, while the Raspberry Pi Pico W has more PWM channels.

Appendix B

Detailed specifications of the GRiSP2 and Raspberry Pi Pico W board

B.1 GRiSP2 board

For a complete description of the GRiSP2 board, please refer to its Kickstarter page[9].

- CPU:
  - NXP iMX6UL, ARM Cortex-A7 @ 696 MHz, 128 KB L2 cache
  - Integrated power management and TRNG, Crypto Engine (AES/TDES/SHA), Secure Boot
- Memory:
  - 128 MB of DDR3 DRAM
- Storage:
  - 4 GB eMMC
  - 4 KBit EEPROM
- Networking:
  - Wi-Fi 802.11b/g/n WLAN
  - 100 Mbit/s Ethernet port with support for IEEE 1588
- External Storage:
  - MicroSD Socket for standard MicroSD cards[1]
- Exposed Input/Output:
  - Dallas 1-Wire via 3-pin connector
  - Digilent Pmod™ compatible I²C interface
  - Two Digilent Pmod™ Type 1 interfaces (GPIO)
  - One Digilent Pmod™ Type 2 interface (SPI)
  - One Digilent Pmod™ Type 2A interface (expanded SPI with interrupts)

[1]Note: this is used to flash the application's code and store data for further analyses. Refer to Appendix E.2 and Section 6.5.4 for more explanations on this.

  - One Digilent Pmod™ Type 4 interface (UART)
- User Interface:
  - Two RGB LEDs
  - 5 DIP switches
  - Reset Key
- Debug and Power Supply:
  - Serial port via Micro USB for console (Erlang shell or RTEMS Console)
  - On-board JTAG debugger via Micro USB
  - JTAG / Trace connector for external debuggers
  - Power supply via Micro USB connector
  - Only one USB cable needed for power, console and on-board JTAG

B.2 Raspberry Pi Pico W

For a complete description of the Raspberry Pi Pico W board, please refer to its documentation page[10]. Raspberry Pi Pico is a low-cost, high-performance microcontroller board with flexible digital interfaces. Key features include:

- RP2040 microcontroller[62] chip designed by Raspberry Pi in the United Kingdom;
- Dual-core ARM Cortex M0+ processor, flexible clock running up to 133 MHz;
- 264 kB of SRAM, and 2 MB of on-board flash memory;
- USB 1.1 with device and host support;
- Low-power sleep and dormant modes;
- Drag-and-drop programming using mass storage over USB;
- 26 × multi-function GPIO pins;
- 2 × SPI, 2 × I2C, 2 × UART, 3 × 12-bit ADC, 16 × controllable PWM channels;
- Accurate clock and timer on-chip;
- Temperature sensor;
- Accelerated floating-point libraries on-chip;
- 8 × Programmable I/O (PIO) state machines for custom peripheral support;

Note that the Raspberry Pi Pico W adds on-board single-band 2.4 GHz wireless interfaces (802.11n) using the Infineon CYW43439 while retaining the Pico form factor. The on-board 2.4 GHz wireless interface has the following features:

- Wireless (802.11n), single-band (2.4 GHz);
- WPA3;
- Soft access point supporting up to four clients;
- Bluetooth 5.2;
  - Support for Bluetooth LE Central and Peripheral roles;

  - Support for Bluetooth Classic;

- The antenna is an onboard antenna licensed from ABRACON (formerly ProAnt). The wireless interface is connected via SPI to the RP2040 microcontroller.

The board's pinout can be found in Figure B.1. To see its design files, etc., please refer to its documentation page[2].

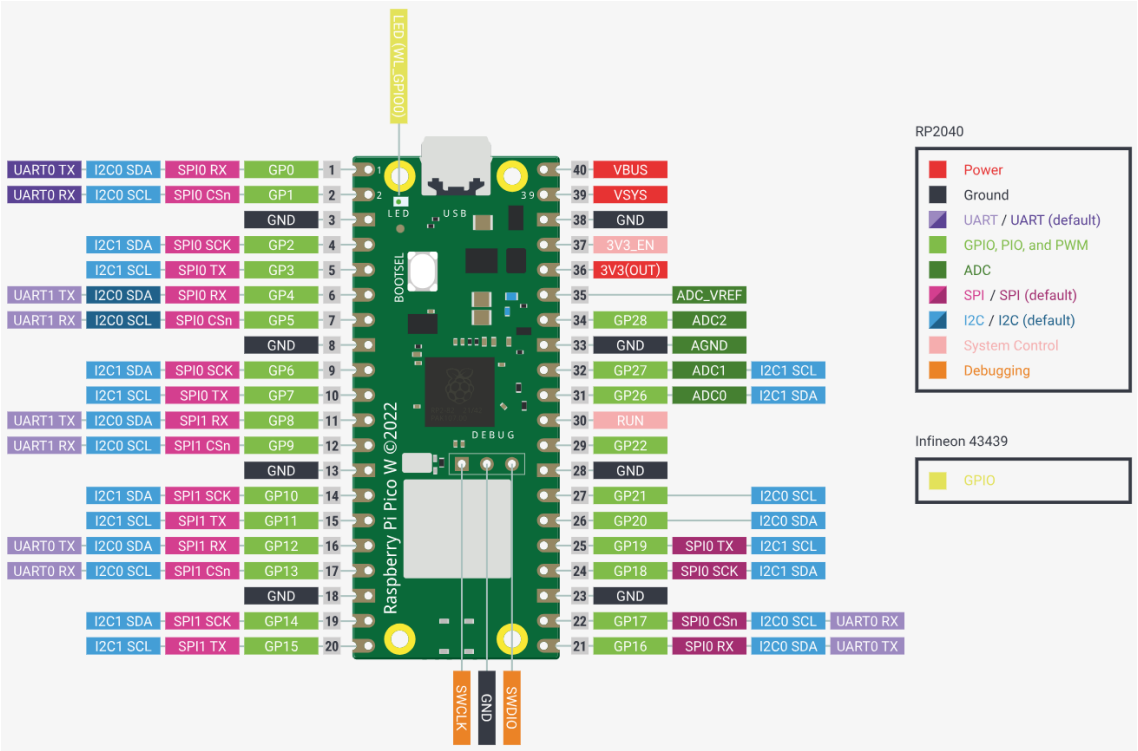


Figure B.1: Pinout of the Raspberry Pi Pico W board

[2] https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html

Appendix C

Additional information and illustrations

C.1 Background complementary informations

C.1.1 Kalman filters

Kalman filters are a powerful mathematical tool which was originally designed by Rudolf E. Kalman in 1960 to estimate non-observable state variables based on observable variables that may have some measurement error[63]. It is a powerful tool for data sensor fusion which deals well with noisy sensor data and supports the estimation of past, present, and even future states[64].

The Discrete Kalman Filter is its simplest form: it gives a recursive solution to the discrete data linear filtering problem[64]. It tries to estimate a normally distributed random state of a discrete-time controlled process that is governed by the linear stochastic difference equation visible in Equation C.1:

$$x_{k+1} = A_k x_k + B u_k + w_k \tag{C.1}$$

with a measurement whose form is shown in Equation C.2:

$$z_k = H_k x_k + v_k \tag{C.2}$$

where w_k and v_k are random variables that represent the process and measurement noise, respectively. It is assumed that they are independent from one another, white, and with normal probability distributions. B is the control-input model and u_k is the control vector, while x_k is the state to be determined and A relates the state at time step k to the state at step $k+1$, in the absence of either a driving function or process noise[64]. It can be seen as the physical model of the system. Finally, H_k relates the state to the measurement and is called the observation model.

The filter works in two steps. It first does a *prediction phase*, then goes through an *update phase*.

The prediction phase starts by estimating the future state of the system using the physical model and previously estimated state, as visible in Equation C.3:

$$\hat{x}_{k+1}^{-} = A_k \hat{x}_k + B u_k \tag{C.3}$$

Then, it computes the next step's *a priori* estimate error covariance, using Equation C.4:

$$P_{k+1}^{-} = A_k P_k A_k^T + Q_k \tag{C.4}$$

where Q_k is the process noise's covariance matrix and P_k is the *a posteriori* estimate error covariance from the previous computation.

Next comes the update phase, which will correct the state estimation. It first computes the Kalman gain K_k, using Equation C.5:

$$K_k = P_k^{-} H_k^T \left(H_k P_k^{-} H_k^T + R_k \right)^{-1} \tag{C.5}$$

This Kalman gain will minimise the *a posteriori* error covariance. Using this gain and by measuring the state of the system, the algorithm will next output the *a posteriori* state estimate, using Equation C.6:

$$\hat{x}_k = \hat{x}_k^{-} + K_k \left(z_k - H_k \hat{x}_k^{-} \right) \tag{C.6}$$

in which $H_k \hat{x}_k^{-}$ is the measurement prediction, compared to the real measurement to give what is called the residual, which reflects the discrepancy between the predicted measurement and the actual one[64].

Finally, it computes the *a posteriori* error covariance estimate, using Equation C.7:

$$P_k = \left(I - K_k H_k \right) P_k^{-} \tag{C.7}$$

and the process is repeated, using the *a posteriori* estimates to project or predict the new *a priori* estimates.

To extend to non-linear systems, the creators of Hera added the Extended-Kalman-Filter, or EKF. It is a Kalman filter that linearises about the current mean and covariance. Now, the process is governed by a non-linear stochastic difference equation, as can be seen in Equation C.8:

$$x_{k+1} = f(x_k, u_k, w_k) \tag{C.8}$$

and with a measurement given by Equation C.9:

$$z_k = h(x_k, v_k) \tag{C.9}$$

where f is a non-linear function that relates the state at time step k to the state at step $k+1$ and includes as parameters any driving function u_k and the zero-mean process noise w_k, while h is a non-linear equation that relates the state x_k to the measurement z_k.

Again, the filter is composed of two parts: the prediction phase and update phase. But here, the equations will change a little. The complete equations of the prediction phase become:

$$\hat{x}_{k+1}^{-} = f(\hat{x}_k, u_k, 0) \tag{C.10}$$
$$P_{k+1}^{-} = A_k P_k A_k^T + W_k Q_k W_k^T \tag{C.11}$$

where W_k is the Jacobian matrix of partial derivatives of f with respect to w, and where A_k is not to be confused with the previously defined A quantity, as it is here the Jacobian matrix of partial derivatives of f with respect to x.

Finally, the complete update phase equations become:

$$K_k = P_k^{-} H_k^T \left(H_k P_k^{-} H_k^T + V_k R_k V_k^T\right)^{-1} \tag{C.12}$$
$$\hat{x}_k = \hat{x}_k^{-} + K_k \left(z_k - h(\hat{x}_k^{-}, 0)\right) \tag{C.13}$$
$$P_k = (I - K_k H_k) P_k^{-} \tag{C.14}$$

where V_k is the Jacobian matrix of partial derivatives of h with respect to v, and where H_k is not to be confused with the previously defined H quantity, as it is here the Jacobian matrix of partial derivatives of h with respect to x.

C.1.2 Quaternions

Quaternions are vectors in four dimensions used to rotate vectors in three dimensions. To rotate a vector in three dimensions, one needs to sandwich it between a unit-quaternion and its conjugate. Note that the quaternion multiplication is associative but not commutative. On the other hand, this mathematical tool offers an unambiguous representation of the orientation of an object in 3D, as opposed to the Euler angles, for example, and in contrast to a rotation matrix which is 3×3, a quaternion contains only four entries, leading to a reduced computational effort. They are composed of one real number and three imaginary numbers and can be modeled as in Equation C.15[65]:

$$q = q_4 + q_1 i + q_2 j + q_3 k \tag{C.15}$$

where i, j and k are linked by the relationship given in Equation C.16[66]:

$$i^2 = j^2 = ijk = -1 \tag{C.16}$$

Note also that there exist many ways of converting a quaternion into a rotation matrix and vice-versa. This will not be detailed in this master thesis, but these methods can be found here[67].

C.2 Visual of the final version of the prototype

Figure C.1: Lambert the robot: outside view

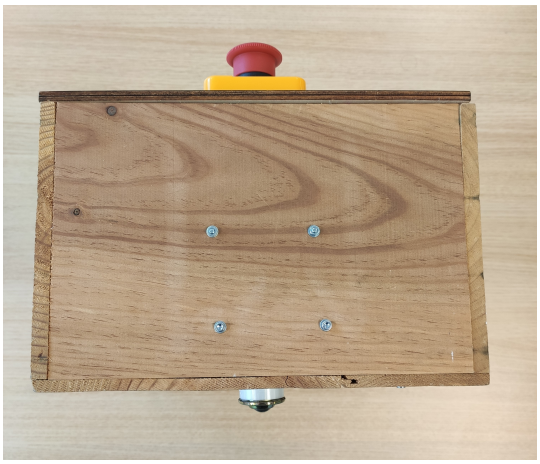


Figure C.2: Front of the robot: the emergency-stop button is placed above the freebear wheel

Figure C.3: Placement of the wheels and underside of the robot

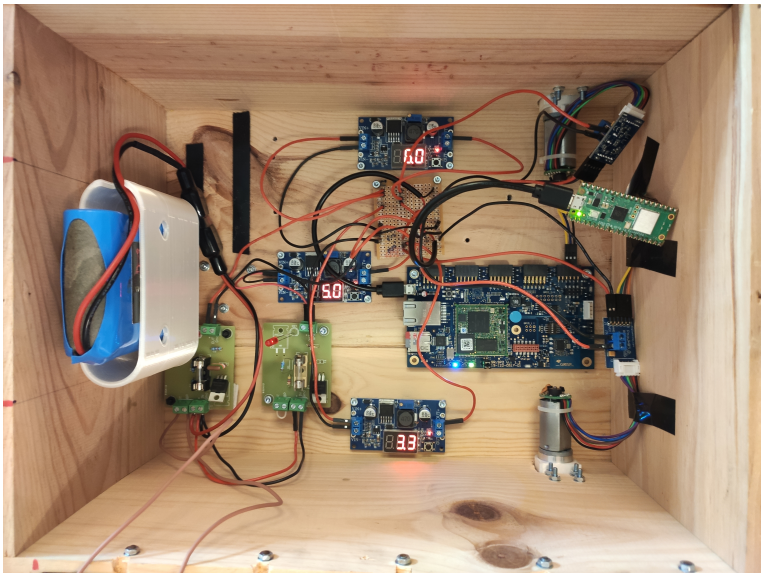


Figure C.4: Final hardware layout in the robot

C.3 The gestures: complete visual description, associated commands and states

This section illustrates the gestures by showing schematics of the gestures and summarises in Table C.1 the gesture names, commands and existence or not of a fast version of the gesture. Reminder: the command format is given by:

`[Velocity1, Direction1, Velocity2, Direction2, command index]`

Table C.1: Summary of the gesture names, the associated commands and if a fast version of the gesture exists

Gesture name	Command	Fast gesture?
`stopCrate`	`[0,1,0,0,2]`	Yes
`forward`	`[currentVelocity,1,currentVelocity,0,0]`	Yes
`backward`	`[currentVelocity,0,currentVelocity,1,0]`	Yes
`forwardTurnRight`	`[110,1,80,0,0]`	Yes
`forwardTurnLeft`	`[80,1,110,0,0]`	Yes
`backwardTurnRight`	`[110,0,80,1,0]`	Yes
`backwardTurnleft`	`[80,0,110,1,0]`	Yes
`turnAround`	`[100,1,100,1,3]`	No
`changeVelocity`	`[0,1,0,0,2]`	No
`accelerate`	`[0,1,0,0,0]`	No
`decelerate`	`[0,1,0,0,0]`	No
`testingVelocity`	`[currentVelocity,1,currentVelocity,0,1]`	No
`exitChangeVelocity`	`[0,1,0,0,2]`	No

Note that when the velocity sent to the Raspberry Pi Pico W is said to have value `currentVelocity`, it means that it sends the velocity which is currently stored in the state, which can be either 100, 110 or 120 RPM.

The following list details the state stored in the `gen_server` following each gesture. Remember that the state, named *movState*, is of the following format:

$$\{currentVelocity, prevName, movName, movMode\}$$

- `stopCrate`: $\{keepVelocity, stopCrate, stopCrate, normal\}$
- `forward`: $\{keepVelocity, forward, forward, normal\}$
- `backward`: $\{keepVelocity, backward, backward, normal\}$
- `forwardTurnRight`: $\{keepVelocity, forward, forwardTurnRight, normal\}$
- `forwardTurnLeft`: $\{keepVelocity, forward, forwardTurnLeft, normal\}$
- `backwardTurnRight`: $\{keepVelocity, backward, backwardTurnRight, normal\}$
- `backwardTurnleft`: $\{keepVelocity, backward, backwardTurnleft, normal\}$
- `turnAround`: $\{keepVelocity, stopCrate, turnAround, normal\}$
- `changeVelocity`: $\{100, changeVelocity, stopCrate, changeVelocity\}$
- `accelerate`: $\{updatedVelocity, changeVelocity, accelerate, changeVelocity\}$
- `decelerate`: $\{updatedVelocity, changeVelocity, decelerate, changeVelocity\}$
- `testingVelocity`: $\{keepVelocity, changeVelocity, testingVelocity, changeVelocity\}$
- `exitChangeVelocity`: $\{keepVelocity, exitChangeVelocity, stopCrate, normal\}$

Note that when the value of the *currentVelocity* state variable is *keepVelocity*, it simply means that it does not change the velocity currently stored in the state, and when the value is *updatedVelocity*, it means that it stores the new value of the velocity. Moreover, special states happen when the guard-rails intervene:

- When the user tries to go from a gesture labelled as *forward* to a gesture labelled as *backward* or vice-versa, the following state is stored:

  $$\{currentVelocity, prefixDetected, stopCrate, normal\}$$

  where *prefixDetected* is the prefix of the detected gesture (either *forward* or *backward*).

- When the user tries to perform a `turnAround` gesture before having stopped the robot, the following state is stored:

  $$\{currentVelocity, stopCrate, stopCrate, normal\}$$

Figure C.5 and Figure C.6 explain how to interpret the following schematics. The black arrow in the schematic indicates the top of the GRiSP2 board. The black dot (corresponding to the yellow dot in Figure C.5, for more visibility) means that the board is seen from the top and the white dot indicates that it is the back of the board[1]. The two axes on each figure represent the cartesian coordinate system at the start position for the given image. It is directly taken as the one printed on the PNAV, shown in Figure 2.3a, plugged on the side of the GRiSP2 board. The *right-hand convention* is used.

[1]Note: due to how the PNAV is plugged in, the black arrow indicates the -z direction.

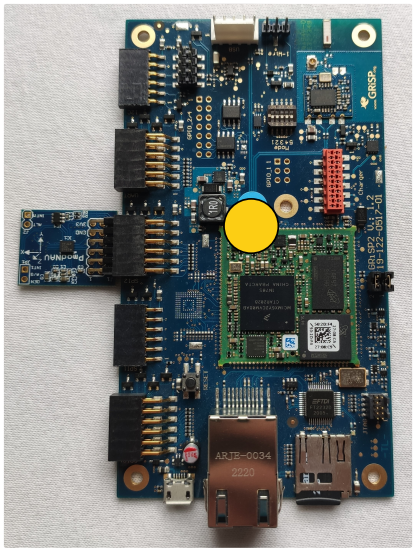


Figure C.5: Illustration: top view of the GRiSP2 board and the corresponding schematic and coordinate system

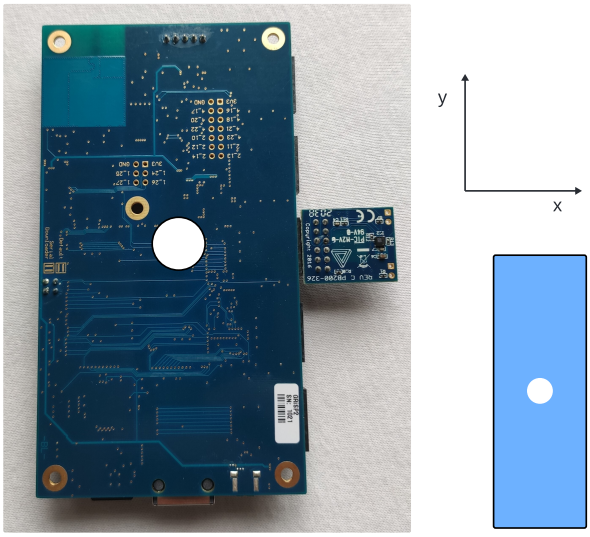


Figure C.6: Illustration: back view of the GRiSP2 board and the corresponding schematic and coordinate system

Hereafter can be found all the schematics illustrating the gestures. Note that for some gestures (`forward`, `backward`, `forwardTurnRight` and `forwardTurnLeft`), there is one gesture which is not shown, which is "staying at the final position of the gesture leading to that name", e.g., for the `forward` gesture, it would mean that keeping the GRiSP2 board upright and not moving is registered as a gesture and leads to the gesture name `forward`. This is illustrated in Figure C.7.

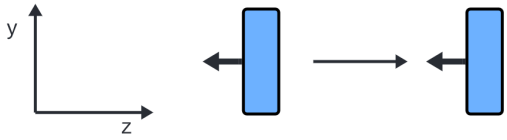


Figure C.7: Keeping the board upright leads to a `forward` gesture

forward

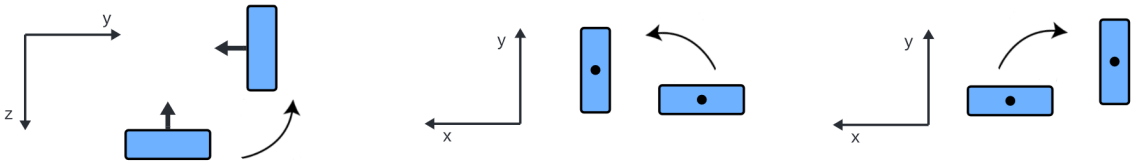


Figure C.8: Gestures associated with the `forward` gesture name

backward

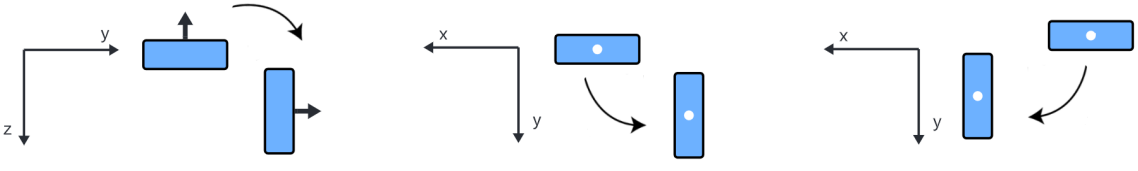


Figure C.9: Gestures associated with the `backward` gesture name

Turning in forward direction

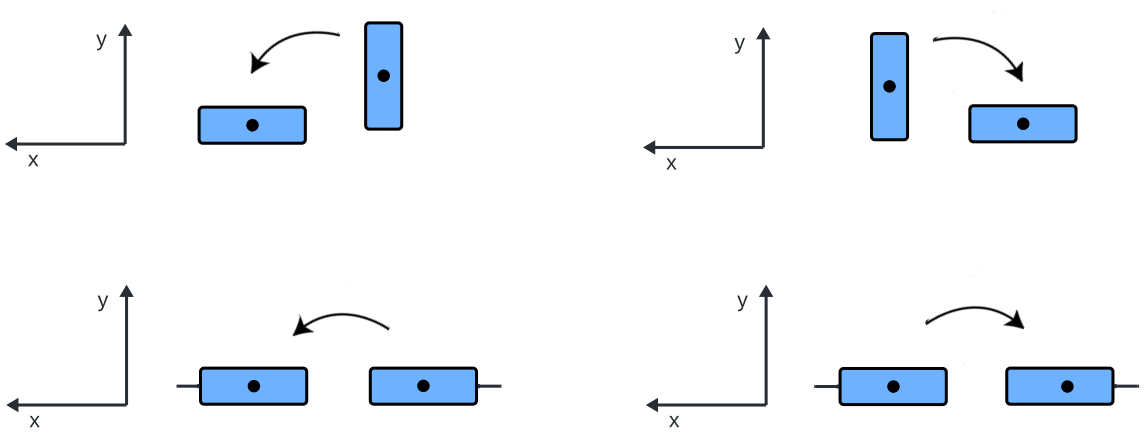


Figure C.10: On the left side, gestures associated with the `forwardTurnLeft` gesture name. On the right side, gestures associated with the `forwardTurnRight` gesture name. The small "needle" in both bottom figures are there to help visualise the movement

Turning in backward direction

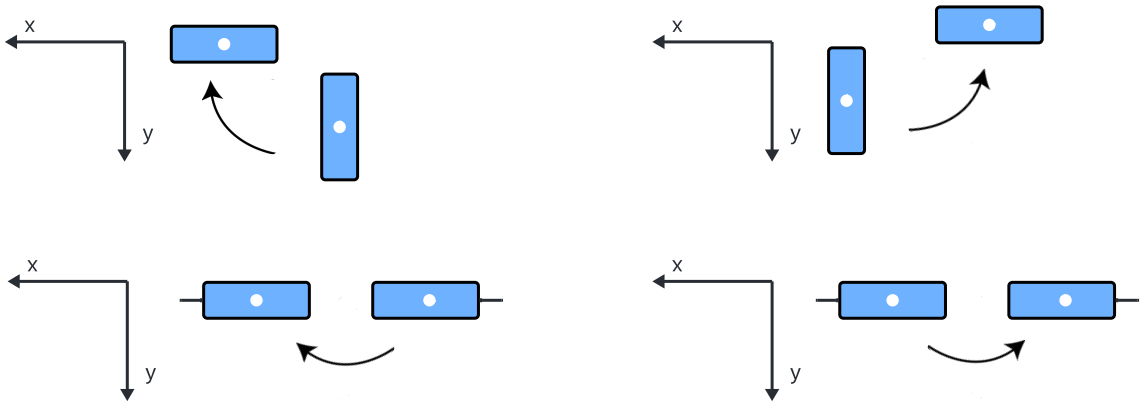


Figure C.11: On the left side, gestures associated with the `backwardTurnLeft` gesture name. On the right side, gestures associated with the `backwardTurnRight` gesture name. The small "needle" in both bottom figures are there to help visualise the movement

turnAround

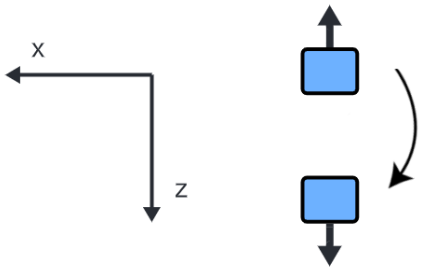


Figure C.12: Gestures associated with the `turnAround` gesture name

stopCrate

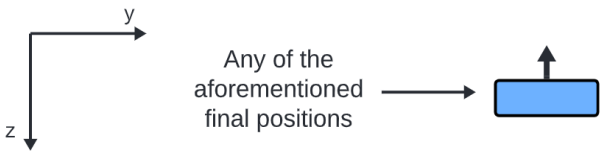


Figure C.13: Gestures associated with the `stopCrate` gesture name. Note: in this gesture, the reference frame is linked to the final position of the board

Movements in changeVelocity mode

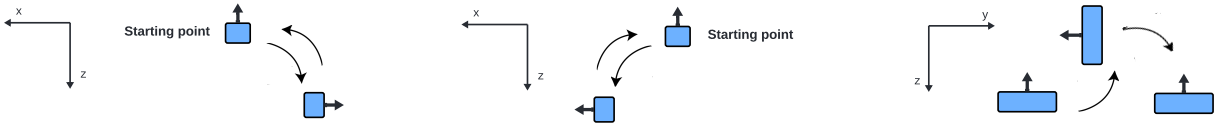


Figure C.14: From left to right, gesture associated with the `accelerate`, `decelerate` and `testingVelocity` gesture name

changeVelocity

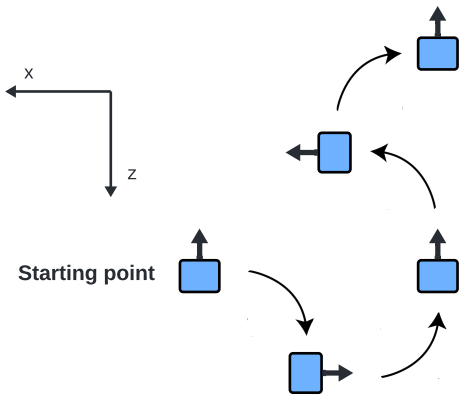


Figure C.15: Gesture associated with the `changeVelocity` gesture name

exitChangeVelocity

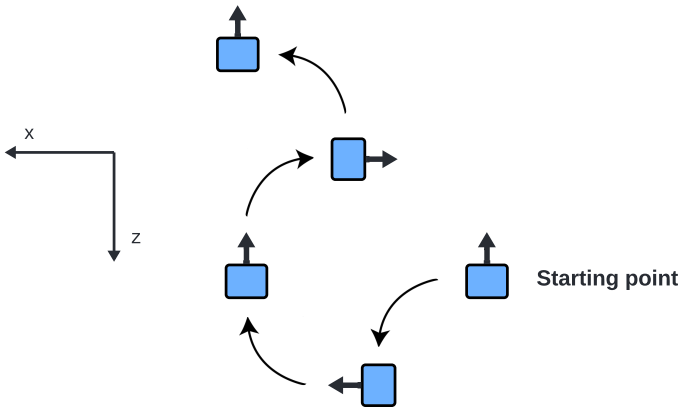


Figure C.16: Gesture associated with the `exitChangeVelocity` gesture name

C.4 More explanation about the logger's working principle

C.4.1 Goal of the new logger

As it was explained in Section 6.5.4, Hera offered the possibility to log the data from the sensors, but this made the whole platform crash. Indeed, it was writing to the SD-card at each new measurement. The accumulation of write requests combined with the processing time for each write to the SD-card seemed to be saturating the GRiSP2 board. The GRiSP2 then gave the impression of freezing during intervals whose duration increased with each new log request, and timeouts occurred.

As this feature of logging the data can prove useful to debug, etc., a new logger was created.

C.4.2 Working principle

Log requests are first stored in RAM, without having the need of SD-card accesses, and then written to the SD-card (flush) at regular intervals. The principle is to reduce the number of SD-card accesses as much as possible. In this case, the write interval is set by a certain amount of data stored in memory being exceeded. This quantity of data stored between two writes to the SD-card must be adjusted according to the hardware and the processes:

- if it is too small, the initial problem will reappear;
- if it is too large, each write will take a certain amount of time and give the impression of jerky execution of commands.

It is also necessary to manage the writing of all the data still in memory in the event of the program being stopped, so that nothing is lost.

C.4.3 Technical realisation

Using the `gen_server` behaviour, the module `buffered_logger` is created. It uses Erlang's ETS module[68], designed for data storage with constant access times. In what follows, a table structure, with each ETS table behaving like an array of hashes `[key|value]`, will be used. As several processes can run in parallel and each can request to log its data (`nav3`, `e11`, etc.), two levels of dictionary will be used:

- The first level will be a table whose key will be a module and whose value will be another ETS table, as illustrated in Figure C.17;

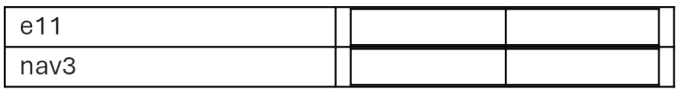


Figure C.17: First level of dictionary

- The second level will be the storage of the data itself, depending on the module, in the sub-tables. Here, for ease of use, the (sub-)key will be an incremental number at the global `buffered_logger` level, according to the order in which the data are stored, as illustrated in Figure C.18.
- When the content of a sub-table reaches the limit set for the number of data that can be stored in said table, e.g. five:
  - the content of this sub-table is saved on the SD-card;
  - the content of this sub-table is emptied from memory;

e11	1	{data}
	3	{data}
	7	{data}
nav3	2	{data}
	4	{data}
	5	{data}
	6	{data}
	8	{data}

Figure C.18: Second level of dictionary

In the example above, this will be the case for `nav3`. This is illustrated in Figure C.19.

e11	1	{data}
	3	{data}
	7	{data}
nav3		

Figure C.19: The data stored in `nav3`'s sub-table are discarded

The data are stored on the SD-card in "append" mode, in the SD-card "measures/" sub-directory, in a file whose name is that of the module (e.g.: `nav3`) followed by ".csv". The format of the saved data follows the format of the previous logger.

C.4.4 How to use the logger

The module's supervisor launches the logger at the start, and the logger will wait for data to arrive. In the launch configuration, choose the number of data to be stored in the ETS tables before writing to the SD-card. Empirically, 200 appeared to be a good number. This is illustrated in Figure C.20.

```
ChildSpecs = [#{id => tested_buffered_logger,
                start => {buffered_logger, start_link, [{200}]},
                restart => transient,
                shutdown => 5000,
                type => worker,
                modules => [buffered_logger]}],
```

Figure C.20: Define the number of data to be stored locally in the ETS tables, in the `buffered_logger_sup` module

Still in the supervisor, one can indicate the time that the `buffered_logger` has to finalise the saving of the data in memory before timing-out, when the measurements are stopped. This is illustrated in Figure C.21.

Here, five seconds are allocated. This may not be much if there is a lot of data to be stored. Much depends on the hardware and the type of data.

The request to save data in the log is made using the `buffered_logger:store_record/2` command. The first argument is the table's name, or the name of the measurement, e.g. `nav3`. The second is the data to be saved. At any point in time, `print_all_contents/0` is used to output the contents of the logger to the console. But there is no explicit request to write to the SD-card: the logger handles writing automatically.

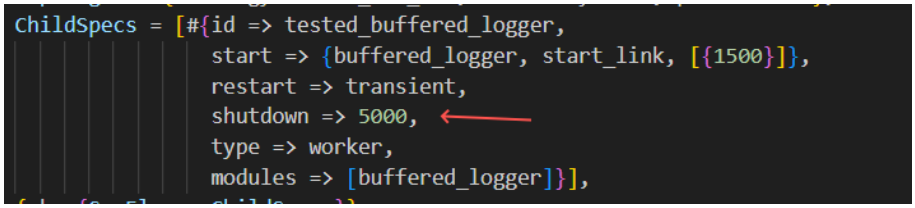


Figure C.21: Define the time that the `buffered_logger` has to store the data on the SD-card when the measurements are stopped

Note: to keep the module execution to a minimum and maintain fluidity, it is possible to:

- Set the limit on the number of data that can be stored in the ETS table to a number greater than the expected amount of data to be collected during the experiment. Thus, all the logs will be stored in dynamic memory, instantly, and then flushed at the end;
- To store everything at the end, put a larger timeout value in the supervisor to allow the logger to have enough time to write everything.

But be careful: to set this number, you need to take into account the RAM available when the program is launched, so as not to saturate the GRiSP2 board's memory.

Appendix D

System performance and limitations: illustration of experiments and complete result tables

General note for this Appendix: except for the velocities and PWM values, computed directly by the micro-controller code, the measured distances and times elapsed were all measured "by hand". It is important to bear in mind that the presented values for distances and times are thus impacted by potential human errors, reaction times, rounding, etc.

D.1 Controller performance

D.1.1 The continuous velocity profile

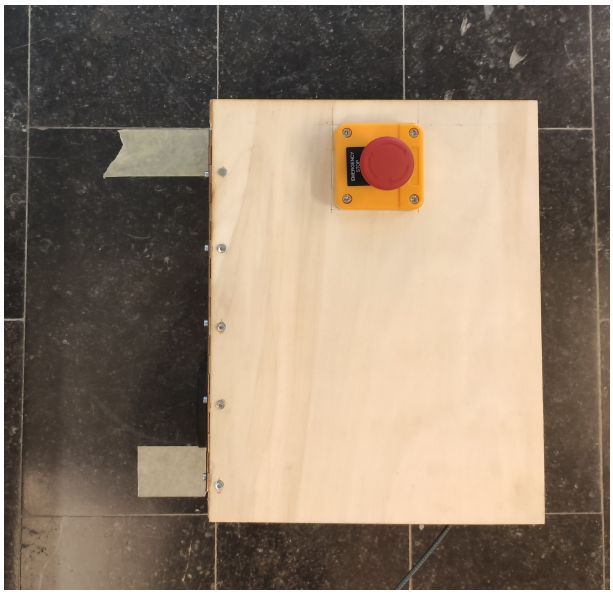


Figure D.1: Beginning of the first experiment: the robot is placed parallel to the two marks on the ground

Figure D.2: End of the first experiment: the distance is measured between the front of the robot and where its front originally was

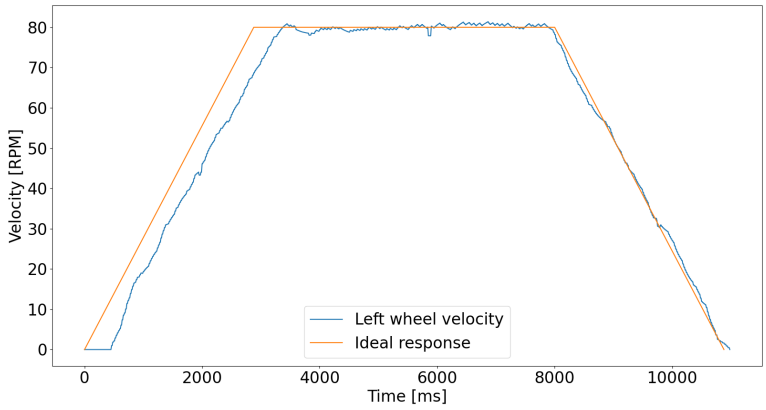


Figure D.3: Enlarged version of the velocity profile at 80 RPM

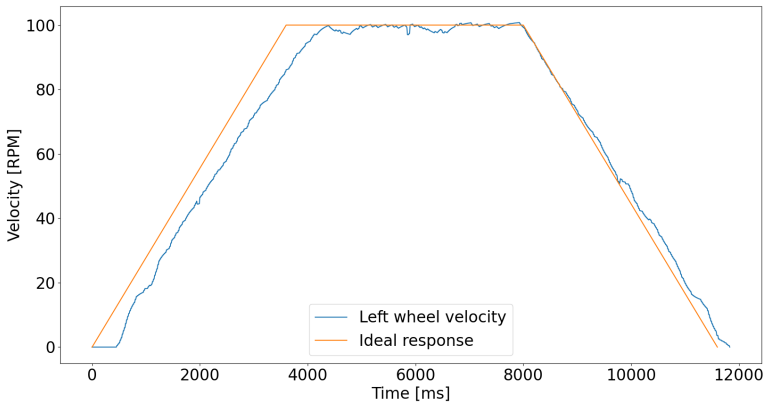


Figure D.4: Enlarged version of the velocity profile at 100 RPM

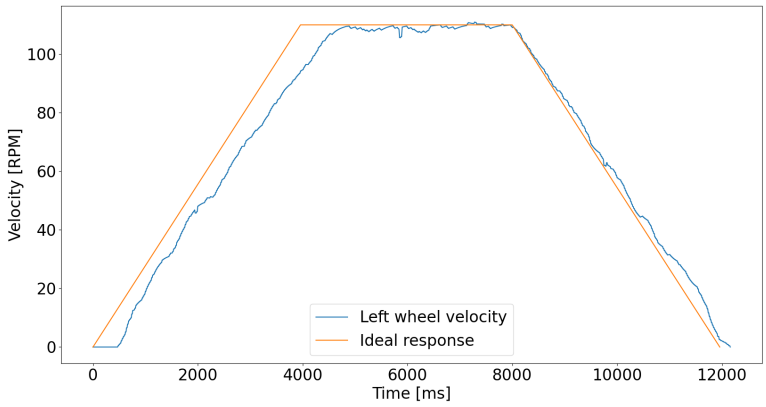


Figure D.5: Enlarged version of the velocity profile at 110 RPM

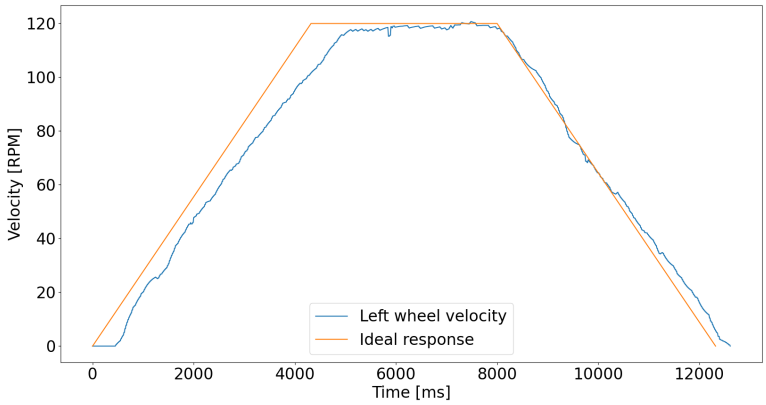


Figure D.6: Enlarged version of the velocity profile at 120 RPM

Figure D.7 illustrates the strange behaviour of the motors observed: they seem to slow down before slowly trying to reach for the desired velocity again. Note: this bug on the motors happened most of the time when the freebear wheel hit a joint, or got stuck in one, but it sometimes happened for no apparent reason.

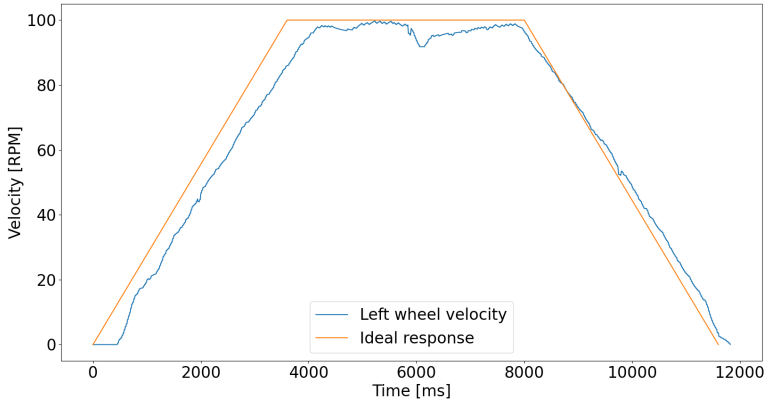


Figure D.7: Enlarged version of the velocity profile at 100 RPM with the bug on the motor, which occurs right after the 6000 ms mark

D.1.2 PWM values at steady-state

Each average shown in Table D.1 and Table D.2 has been computed as the average of three other averages based on ten values, as explained in Section 6.1.2.

Table D.1: PWM value observed on the wheels for a given velocity. Full data

Tested velocity [RPM]	Wheel	Test 1	Test 2	Test 3
100	Left	155.767	154.8	153.83
	Right	151.63	151.367	151.3
110	Left	180.1	180.43	184.167
	Right	174.7	177	179.7
120	Left	215.467	210.067	211.83
	Right	214.63	220.53	212.967

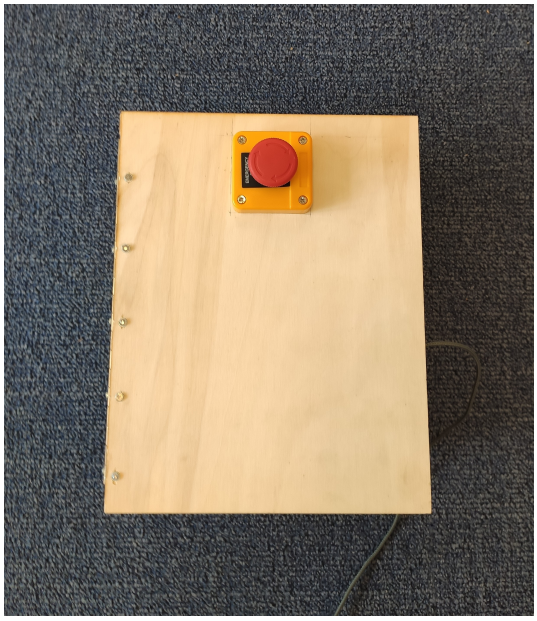


Figure D.8: Illustration: testing the velocity profile on a carpet floor

Table D.2: PWM value observed on the wheels on a carpet floor. Full data

Tested velocity [RPM]	Wheel	Test 1	Test 2	Test 3
100	Left	241.8	240.83	240.867
	Right	243.03	242.4	241.9

D.1.3 Travelled distance during the velocity profile experiment

Table D.3: Measured travelled distance by the robot, in [cm], during the velocity profile experiment. Note the lower value for the last test at 110 and the first two at 120, due to the freebear wheel getting stuck in a joint and the motor behaving strangely. Full data

Tested velocity [RPM]	Test 1	Test 2	Test 3
100	335	334.5	334.5
110	367	368	354
120	395	394	401

D.1.4 Deviation from the straight path

As has been mentioned in Section 6.1.4, the robot would not go in a straight line most of the time. It was found that this deviation was mostly random and due to how the robot would react to the

joints between the tiles, as well as to the small impact of the slight wheel placement difference. The following tables summarise the observed deviations from a straight path. The average deviation angles in Table D.4 are computed by taking the values from Table 6.4 for the distance travelled during the test.

Table D.4: Deviations, in [cm], observed at the end of the velocity profile experiment. A positive value indicates a deviation to the left of the straight path, while a negative value indicates a deviation to the right. Full data

Tested velocity [RPM]	Test 1	Test 2	Test 3	Average deviation angle [°]
100	11.1	4	5.9	1.198
110	3.9	4	1	0.468
120	3.3	-4.2	-1.1	-0.096

Note: the distances used to compute the average deviation angles in Table D.5 are taken as twice the distances shown in Table 6.5.

Table D.5: Deviations, in [cm], observed at the end of the `testingTheNewVelocity()` experiment. A positive value indicates a deviation to the left of the straight path, while a negative value indicates a deviation to the right. Full data

Tested velocity [RPM]	Test 1	Test 2	Test 3	Average deviation angle [°]
100	12	13	2.5	0.954
110	0.8	1.4	4.7	0.225
120	9.2	-0.7	5.4	0.426

D.1.5 Performance of a turning motion

Figure D.9: Turning motion experiment: two boxes were placed, one for each time the robot would be parallel to the tiles. The distance was measured between these two boxes to obtain the diameter of the achieved circle

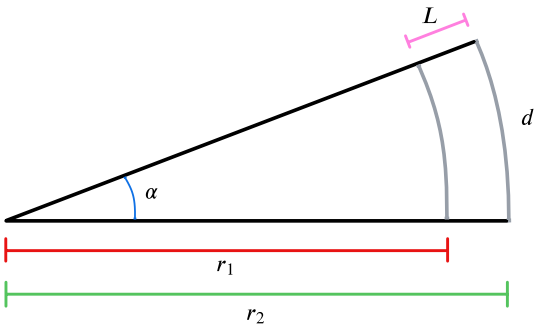


Figure D.10: Mathematical illustration of the wheels' path during a turning motion

Table D.6: Radius, in [cm], of the circle described by the outer wheel. Full data

Test 1	Test 2	Test 3
108	108.25	108.5

D.2 Performance of the control routines

D.2.1 Performance of `testingTheNewVelocity()`

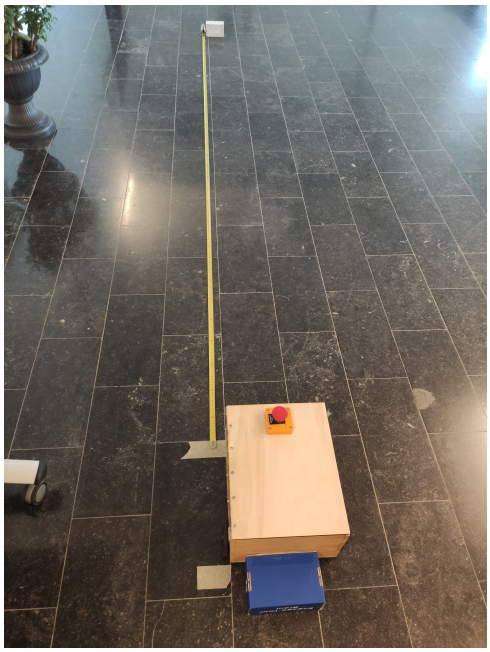


Figure D.11: In this experiment, the robot performs a back and forth motion. Two boxes are placed as follows: the blue one is placed behind the robot when it is at rest, before starting the test, while the white box is placed where the front of the robot arrived at the end of the first phase

Table D.7: Measured travelled distance by the robot, in [cm], during the `testingTheNewVelocity()` experiment. Full data

Tested velocity [RPM]	Test 1	Test 2	Test 3
100	274.5	274.7	276.7
110	292.4	292.8	292.6
120	311.5	311	311.5

D.2.2 Performance of the `turnAround` routine

Figure D.12: In this experiment, the robot performs a 180° turn. This deviation angle from the joint next to which the robot was initially placed is computed using the distance from the front and the back of the robot to that joint

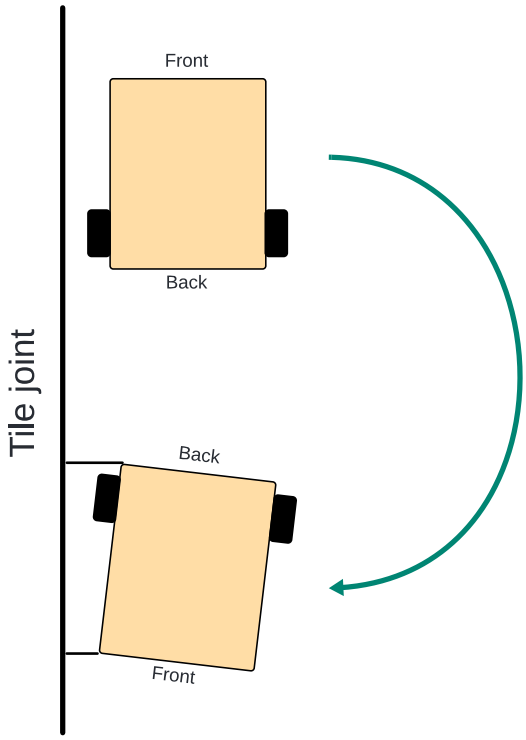


Figure D.13: Illustration: the `turnAround` experiment

Table D.8: Measured distance, in [cm], between the starting line of the experiment and the robot during the `turnAround` experiment. Full data

Test number	Distance back	Distance front
1	2.5	1.5
2	2.7	2.7
3	2.6	2
4	2.5	2.7
5	2.5	4.5

D.3 Guard-rail performance

D.3.1 Guard-rail between the two GRiSP2 boards

Table D.9: Time measured, in [s], for the robot to reach a full stop starting from 100 RPM when the detector is unplugged from its power source. Full data

Test 1	Test 2	Test 3	Test 4	Test 5
3.961	4.618	4.132	4.242	4.93

As can be seen in Table D.9, the time in the second test is slightly longer than the theoretical maximum. This is most certainly due to human error.

D.3.2 Guard-rail between the receiver and Raspberry board

Table D.10: Time measured, in [s], for the robot to reach a full stop starting from 100 RPM when the receiver is unplugged from its power source. Full data

Test 1	Test 2	Test 3	Test 4	Test 5
3.841	4.159	4.4	4.071	4.143

D.4 Gesture recognition timings

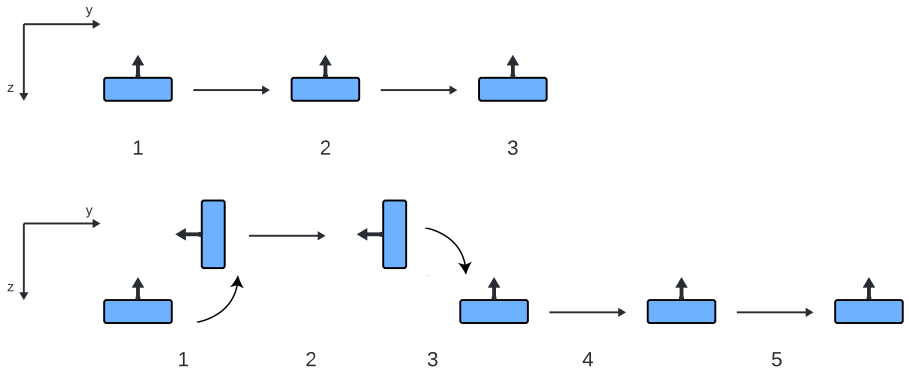


Figure D.14: From top to bottom: illustration of the first and second experiments to evaluate the speed at which the algorithm is detecting gestures

Table D.11: First experiment: time it takes, in [s], for the gesture recognition algorithm to detect each gesture. Full data

Test number	Gesture 1	Gesture 2	Gesture 3
1	2.796	1.921	1.896
2	2.888	1.937	1.84
3	2.946	1.912	1.8
4	2.825	1.88	1.656
5	3.024	1.896	1.656

Table D.12: Second experiment: time it takes, in [s], for the gesture recognition algorithm to detect each gesture. Full data

Test number	Gesture 1	Gesture 2	Gesture 3	Gesture 4	Gesture 5
1	4.899	1.632	3.895	2.017	1.76
2	3.384	1.84	3.575	1.889	1.728
3	3.224	1.911	2.896	1.912	1.776
4	3.337	2.256	3.08	1.84	1.728
5	3.216	2.008	4.136	2.048	1.808

Note: Table D.12 shows interesting behaviours: the time for the first gesture in the first test is bigger than the four other tests: this is because the author started the gesture late whilst he started the other four almost immediately after launching the experiment. For the third gesture, in test 3, one can see a shorter time to detect the gesture: this is because the author anticipated the classification and started performing the next gesture even before the algorithm printed the results in the shell. The longer time recorded for gesture 3, test 5, is due to the same reason as the first gesture of test 1.

Appendix E

Updated user manual

In previous theses, the authors have created and updated a user manual detailing how to use their application, whilst also globally explaining how to start up with the GRiSP board and Erlang. However, it was never updated for the GRiSP2 board.

The author encountered numerous problems, which were solved one by one over the course of several weeks. To avoid future generations of Erlang programmers struggling with the GRiSP environment, the author here proposes a guide on how to install Erlang, Hera, etc., followed by a user manual explaining how the application developed in this thesis can be used.

Both of these are heavily inspired by the previous user manual, which can be found here[7]. This user manual is also a great source of information concerning the older versions of the system.

E.1 How to install Erlang, Hera, ...

Each part starts with the "go-to" procedure to complete the installation. An erratum and summary of problems faced at each step by the author can be found in the *Struggles faced* part of each section.

Disclaimer

The following tutorial was done on a computer running on Linux-Ubuntu 20.04, installed in February 2024. The author cannot guarantee it will work for newer versions (though it should) or for other OS[1], nor does he guarantee that updated versions of the packages required to run Erlang and compile it will work with GRiSP. This tutorial, furthermore, is not as detailed as one would hope it to be: the installation process took place in early February, whilst the writing of this section was done in late May. If any questions emerge, the author will gladly help future users with their struggles. Note: another great source of help is the GRiSP Slack channel[2].

Some parts of this tutorial are directly inspired by the GRiSP wiki. Refer to this website[3].

Required hardware

[1] E.g.: do not even try to use wsl on Windows. There are a lot of dependencies that do not exist in wsl but that are required to make this whole thing work. Linux is the go-to OS for the GRiSP environment.

[2] https://app.slack.com/client/T055DJ9UH/C8DSPE9AP

[3] https://github.com/grisp/grisp/wiki

For this tutorial, one needs:

- A computer running on Linux;
- One Wi-Fi access point: it is recommended to use a smartphone (or computer) as a hotspot, since the Eduroam network does not really like connecting on prototyping boards (for safety reasons);
- One or more GRiSP2 boards (with SD-card);
- Pmod™ sensors;

You can find all the GRiSP related hardware at https://www.grisp.org/shop/.

Required software

To use the system, one needs to have installed on its computer:

- Erlang/OTP 25.0, Erts 13.2;
- rebar3 3.22.1;
- rebar3_hex;
- rebar3_grisp;

The following explains in more detail how to install these.

E.1.1 Setting up a development environment

Installing Erlang

It is recommended to use a version manager to get your **Erlang installation**. *asdf* or *kerl* are fine. In this tutorial, *kerl* will be explained.

The first thing to do is to follow this tutorial[4] to install *kerl*. If you are using a new computer, many packages need to be installed in parallel. A list of such packages can be found here[5]. Note that this is technically for *asdf*, but the same goes for *kerl*. Moreover, the package "*wxWidget*" did not work for the author. Next, install Erlang itself using *kerl*. Then, it is recommended to add the installation's activation to the system's PATH. A tutorial on how to add something to the PATH can be found in this website[69]. Be careful: if you modify directly the *bashrc* file, it uses different commands from those which can be used in a Linux shell.

Struggles faced

Upon trying : "kerl install 25.3 /usr/local/lib/erlang/25.3", an "Access Denied" message appeared. This was due to the fact that the user profile on the computer did not have access to the directories mentioned. For any directory to which access is denied but is required, in the shell, go to said directory and type:

```
sudo chmod 777 directoryName
```

[4] https://github.com/kerl/kerl
[5] https://github.com/asdf-vm/asdf-erlang/pull/10/files

This will allow to read, write and execute on the directory for any user on the computer. If the error message happens on any sub-directory, do the same on it. Beware: it is ugly and can be dangerous. Anyone on your computer can now modify your Erlang installation. But, seriously, who would?

The rest of the installation process should go fine. You now have Erlang installed on your computer.

rebar3

This is required to build Erlang projects. Follow the "getting started" tutorial on this web-page[6].

Struggles faced

At the "*cd rebar3*" followed by "*./bootstrap*" step, the following error message appeared:

```
OTP Application crypto not available. Please fix your Erlang
install to support it and try again.
```

This happened because the author installed ssl (ssh command in earlier steps) *after* installing Erlang. Do not do that. If you did, delete the installation then build and redo with all packages installed. Once it is done, "*./bootstrap*" should work better (green lines with "===>" at the start). You can then proceed with "*./rebar3 local install*".

GRiSP Rebar3 plugin

In the previous user manual, it says to add the plugin configuration for GRiSP in the file "*/.config/rebar3/rebar.config*"... Which did not exist. Create the directory and "*rebar.config*" file and then add the following plugin configuration[7]:

```
{plugins, [
    {rebar3_hex,
        {git, "https://github.com/erlef/rebar3_hex.git",
            {branch, "main"}
        }
    },
    {rebar3_grisp,
        {git, "https://github.com/grisp/rebar3_grisp.git",
            {branch, "master"}
        }
    }
]}.
```

If you get a "no escript" error message, it is because the system detects no active Erlang/OTP installations. Make sure to have an active one in the shell in which you are doing the installation procedures.

[6] https://github.com/erlang/rebar3

[7] Note: this is not the same as in the previous user manual. Indeed, it struggled to load the plugins. This update version details where to find the files, which helps the process.

Some notes at this point:

- To find a command which you already typed earlier rapidly, in the shell, type: "*ctrl+r*" and the beginning of the command;
- Do not forget that in order to run rebar3, etc., you should always have an active and valid Erlang session running. Typically, every time you start a shell, type:

```
. /usr/local/lib/erlang/25.3/activate
```

The version being whatever version you have. You can also add rebar3 to your PATH (recommended, at each shell launch, in order to have the rebar3 command work everywhere), using:

```
export PATH=\$PATH:$\sim$/.cache/rebar3/bin
```

Typically, the author would start each new shell by using "*ctrl+r*" and typing the beginning of one of the commands. Do not hesitate to put them one after the other using "&&" so that everything is done in one go.

Lastly, to verify that rebar3 is correctly installed (and to update the plugins if need be), you can type the following in the shell:

```
rebar3 update && rebar3 plugins list
```

First tests

With that, everything should be good regarding Erlang! You can try the tutorials given by the GRiSP team[8] that flashes the LED. To create a new app, in the rebar3 folder, type:

```
rebar3 new grispapp name=Name dest=/whereisyourSDcard
```

e.g.:

```
rebar3 new grisapp name=Whatever dest=/media/nicolas/GRISP
```

This creates a folder in rebar3. You can take this folder wherever you want, the commands for rebar3 will work if you added rebar3 to your PATH.

E.1.2 Connection over serial

To open the Erlang shell on your computer to directly work on the GRiSP2, use:

```
sudo picocom /dev/ttyUSB1 --baud 115200 --echo
```

in your application's folder. To leave picocom without cancelling the program (which is achieved by "*q().*" in the Erlang shell), use "*ctrl+a*" followed by "*ctrl+q*". Note that this only works for tty, as when you use a remote shell (see later), type first "*q().*" then "*ctrl+c*".

[8] To create: https://github.com/grisp/grisp/wiki/Creating-Your-First-GRiSP-Application and deploy: https://github.com/grisp/grisp/wiki/Deploying-a-GRiSP-Application the app.

E.1.3 The Wi-Fi and connecting remotely

The author recommends that you to follow this tutorial from scratch in order to better understand what is happening. Start from a random project, typically the one which you created to do the LED example. Important note: you only need to have the IP address of the boards if you wish to connect to them remotely. Otherwise, simply doing the "*wpa_supplicant.conf*" part should be enough.

First, follow the GRiSP tutorial[9] up until "Finding out the IP address of the GRiSP board when using DHCP", at which point you could face some problems, as the board maybe never connected to the Wi-Fi beforehand. Jump to the **GRiSP INI** section, follow it, and then follow **Configuring Wi-Fi**. Next, deploy the application on your SD-card as per usual. Put the SD-card in the board and supply it. You should see messages in the shell about *wlan0* mentioning errors (*File exists*). This is not a problem. To find the IP-address of the GRiSP2 board, type:

```
inet:getifaddrs().
```

Go back to the tutorial to complete everything regarding the "*erl_inetrc*" file and add the IP-address of the board to your host file by typing, in a normal shell:

```
sudo vi /etc/hosts
```

For example, a completed "*/etc/hosts*" file would look like:

```
127.0.1.1       hostname_computer
192.168.43.215  board_1
192.168.43.6    board_2
```

and a completed "*erl_inetrc*" would look like:

```
{host, {192,168,43,32}, ["neackow_z4"]}.
{host, {192,168,43,215}, ["nav_1"]}.
{host, {192,168,43,6}, ["orderCrate"]}.
```

You can now connect to the GRiSP2 remotely using:

```
erl -sname my\_remote\_shell -remsh my\_project\@my\_grisp\
_board -setcookie MyCookie
```

and replace everything with the names you have given to your project, etc.

Be careful: connecting remotely to the GRiSP2 board does not connect directly to the board itself: it creates a distributed Erlang node solely for the purpose of doing a remote shell. This means that there is an intermediate node between the computer and the GRiSP2 board.

Also, all this is already done in the applications files[10]. All you have to do is to adapt the files to your computer's name, IP address, etc. Moreover, in the applications directory, one can use the "*makefile*" to connect remotely, using:

```
make deploy-NameOfBoard
```

[9] https://github.com/grisp/grisp/wiki/Connecting-over-WiFI-and-Ethernet
[10] More on how to change the applications file to adapt to your computer in Section E.2.

Be careful: the board's name is **very important** since it dictates the behaviour of the board, as has been explained in previous chapters. Note that this command takes the name of the board as a variable value and attributes it to the "*env.NAME*" variable in the "*grisp.ini.mustache*" file.

Some notes:

- The author tried to be smart and put his computer node twice in the "*erl_inetrc*" file, so he would not have to care about whether he was at home or not. It seems like it does not like having twice the same name, however. Comment the node which you do not need and rebuild. Note: this only applies if you use different Wi-Fi depending on where you are working. Again, it is advised to use your smartphone as a Wi-Fi hotspot, since it will *always* give the boards, computers, etc. the same IP address.
- Obviously, make sure the Wi-Fi network that the GRiSP2 board is trying to reach is not disabled.
- Beware of syntax error in the "*erl_inetrc*" file. Instead of separating the numbers by a "," one could type the IP address with ".", but this does not work.
- If you wish to connect remotely to a node, the author believes that the computer should be on the same Wi-Fi as the boards, as he got a "*** ERROR: Shell process terminated! (^G to start new job) ***" error if not connected on the same network. This only applies when you try to connect remotely.
- Using your smartphone as a hotspot can lead to unexpected behaviours. Often, the GRiSP2 board would not manage to connect to the smartphone's Wi-Fi, despite the computer being able to. Or if it did, it would, out of nowhere, disconnect and never find the Wi-Fi ever again. It was found that this was due to the "Saving energy" mode of the phone, which forbids/periodically interrupts any remote connection from GRiSP2 boards to its hotspot. Upon deactivating it, the boards would find the Wi-Fi (almost) all the time. It was also observed that the "Silent" mode of the phone could disturb the connection.

Struggles faced

Due to the `numerl` NIF, the author never managed to run the Erlang application on his computer. Hereafter are some troubles he faced while trying to do so.

Using "make shell" worked, but the clean start "make local_release && make run_local" did not, with error: "Could not start kernel pid, application_controller, invalid config data: application: grisp; duplicate parameter: devices". In the author's "*home/nicolas/TFE/sensor_fusion/_build/computer/rel/sensor_fusion/releases/1.0.0/*" folder, several files were available. In "*sys.config*", two {*devices*} variables were set. These variables were coming from the "*computer.config.src*" file in the "*config*" folder. The author tried to comment one of the two and rebuild.

It now said that it did not find "*numerl.so*". The error pointed to "*sensor_fusion/src/numerl.erl, line 9*". Upon analysing, it was found out that line 9 calls "*erlang:load_nif*", which takes as first argument the file path to the shareable object/dynamic library, minus the OS-dependent extension (.so in Linux), so here, *numerl*. However, "*numerl.so*" seems to be moved in the trash, as was found out by using the command "*find /home -name numerl.so*". So there was a problem here, as it is necessary that the system accesses this file to correctly compile the whole project. According to this wiki[11], there are some special requirements, such as having a main C file named "*NAMEOFDRIVER_nif.c*" and this file must reside in the application's top level folder, under the following path:

[11] https://github.com/grisp/grisp/wiki/NIF-Support

```
    grisp/$<$platform$>$/$<$version$>$/build/nifs/NAMEOFDRIVER\_nif
   .c
```

In the current case, the author tried to following path:

```
    grisp/grisp2/default/build/nifs/numerl\_nif.c
```

Initially, the file was located under:

```
    grisp/grisp2/common/build/nifs/numerl\_nif.c
```

But this attempt in changing the file from folders returned an even weirder error. Upon rebuilding the project, the following message appeared:

```
    * Patching
        [grisp] 00100-rtems.patch (already applied, skipping)
        [grisp] 00300-drivers-nifs.patch ===> sh(git apply
    --ignore-whitespace 00300-drivers-nifs.patch)
    %% Note: this file can be found in
    %% _grisp/grisp2/otp/<your_version>/build/
    failed with return code 1 and the following output:
    error: patch failed: erts/emulator/Makefile.in:938
    %% At this line, we find:
    %% ASMJIT_PCH_OBJ=$(TTF_DIR)/asmjit/asmjit.hpp.gch
    error: erts/emulator/Makefile.in: patch does not apply
```

where lines starting with "%%" indicate a comment from the author.

In the face of this error message, the author realised that either his method was wrong, or there was something beyond his understanding. So, he tried putting the aforementioned file to where it previously was. Upon trying to build once more, the following message appeared in the shell:

```
    ===> Preparing
    * Patching
        [grisp] 00100-rtems.patch (already applied, skipping)
        [grisp] 00300-drivers-nifs.patch (already applied, skipping
   )
    * Copying files
        [grisp] erts/emulator/sys/unix/erl\_main.c
        [grisp] xcomp/erl-xcomp-arm-rtems5-25.conf
        [grisp] xcomp/erl-xcomp-arm-rtems5.conf
    * Copying drivers
        [grisp] erts/emulator/drivers/unix/grisp\_termios_drv.c
    * Copying nifs
        [grisp] erts/emulator/nifs/common/grisp\_gpio_nif.c
        [grisp] erts/emulator/nifs/common/grisp\_hw\_nif.c
        [grisp] erts/emulator/nifs/common/grisp\_i2c\_nif.c
        [grisp] erts/emulator/nifs/common/grisp\_rtems\_nif.c
        [grisp] erts/emulator/nifs/common/grisp\_spi\_nif.c
        [sensor_fusion] erts/emulator/nifs/common/numerl\_nif.c
        %% SO apparently, it DOES copy it. So the question is:
        %% why is it not available?
```

As the author later realised that he did not need to launch the application on his computer, he gave up on the matter. This problem is thus left with no answer.

Note about "*make local_release && make run_local*": the problem is that "*numerl.so*" is not loading, and apparently, according to Sébastien Kalbusch (one of the two Hera2.0 developers), it is normal and if one wanted to have the possibility to run the code directly on the shell, the NIF should be discarded on the computer. It is to be noted that the only advantage of having the application run on the computer is to have the logs directly appearing on it without having to remove the SD card at each test.

E.1.4 Installing the Arduino IDE

The author would like to point out that this section refers a lot to the ESP32, as it was the first micro-controller that he attempted to use. However, every manipulation done here will be useful for the Raspberry Pi Pico W.

The Arduino IDE has been used to code the C++ code of the micro-controller. Another way of coding in Arduino but in *VSCode* is to use *Platformio*, but the author does not especially recommend it, as it seemed unreliable on his computer. It worked (vaguely), but was not really intuitive nor fast to deploy the code on the board.

First, install the Arduino IDE itself by following the tutorial you can find here[70]. Next, upon following this tutorial, the author faced several difficulties:

- The *PySerial* package was not installed on the computer. Follow this tutorial[71] to install it;
- He had some struggles with the ports not being detected (typically, *dev/ttyUSB0*): following what was said on this page[12], by using the part mentioning the following command:

```
sudo usermod -a -G dialout $USER
```

After restarting the computer, the problem was solved.

- One could face port problems if the appropriate driver (UART-USB CP120X) is not installed. On Ubuntu, the driver should be installed by default. If not, you can follow this tutorial[13];

E.1.5 Installing the application

Go to the git repository (here[14]). Clone the git wherever you want it to be on your computer. *hera* and *herasynchronization* are added using the build files, like "*rebar.config*". Technically, you do not need to install them on your computer in order for the whole application to work.

This version of the application uses `numerl`, a custom NIF for fast matrix operations. To use the `numerl` NIF, you must first compile a custom version of OTP. This can be achieved using a Docker or a toolchain. The previous user manual says to follow this tutorial[15] and install a Docker by adding the following line to the "*grisp/build/toolchain*" section of "*rebar.config*":

```
{grisp, [
    {build, [
            {toolchain, [
```

[12] https://askubuntu.com/questions/133235/how-do-i-allow-non-root-access-to-ttyusb0
[13] https://askubuntu.com/questions/941594/installing-cp210x-driver
[14] https://github.com/Neackow/movement_detection
[15] https://github.com/grisp/grisp/wiki/Building-the-VM-from-source

```
                        {docker, "grisp/grisp2-rtems-toolchain"}
                ]}
        ]}
]}
```

and then run the command:

```
    rebar3 grisp build --docker
```

which should generate a "*_grisp*" folder containing the VM running on the GRiSP.

However...

Struggles faced

The build was not working. It appears that rebar3 does not know what Docker is, despite having installed it. To install Docker, see this link[16]. After basically one hour of scanning the Internet, the author went on the Docker hub website, into the "*grisp/grisp2-rtems-toolchain*" and did the following Docker pull:

```
    sudo docker pull grisp/grisp2-rtems-toolchain
```

The goal was to have the Docker image locally. Doing this without sudo failed ("daemon socket connection error"), but this sadly did not correct the problem.

To get the Docker running, on this page[17], they mention adding the Docker toolchain possibility. Downloading the .zip, in one of the files of the "*src/*" directory, the author saw that the build was trying to ask for "*docker info*", which returned an error, leading to the problem. To avoid this, he followed the advice given on this page[72]. This finally resulted in a functioning Docker. But next, after putting the right Erlang/OTP version in the "*rebar.config*" file, the author reached a new error, saying it did not find a C compiler for the code, despite having "*gcc*" installed on the computer.

Despite best efforts, he could not make the Docker work. Instead:

Installing a toolchain

Go to this page[18] and "*git clone*" the repository, then type, in your shell:

```
    make install
```

within the repository. This will take a while. A very long while[19]. Then, in "*rebar.config*", instead of

```
    {docker, "grisp/grisp2-rtems-toolchain"}
```

put:

```
    {directory, "where_is_your_toolchain_installed_on_your_computer
   "}
```

[16] https://docs.docker.com/engine/install/ubuntu/
[17] https://github.com/grisp/rebar3_grisp/releases
[18] https://github.com/grisp/grisp2-rtems-toolchain
[19] Almost an hour, in this case.

e.g., for the author:

```
{directory, "/home/nicolas/TFE/grisp2-rtems-toolchain/rtems/5"}
```

You should now be able to compile the application. Go to the directory and type "*rebar3 grisp build*". Again, the build will take a while.

Having your own Hera repositories

In order to modify the code of Hera and debug the application, you need to create your own repositories. Here, step by step, is how to do it:

- Go on your GitHub account;
- Create the repositories: the author recommends keeping "*hera*" and "*hera_synchronization*" as default names for these. However, you can call your main repository whatever you want (in this project, it was named "*movement_detection*");
- Locally clone all the newly created repositories, then download the original repositories' content from the author's git (download the .zip and paste everything in the repository, it is easier). Place everything inside the correct folder;
- Then, in the shell, for each local folder:

```
git add *
git commit -m "My amazing first comment"
git push -u origin main
```

- Once the repositories are set up, you can now go and modify "*rebar.config*" to put your names and reference to your repositories in the {*deps*} part, at the beginning of the file. You will have to change things in the "*hera*" repository and in the main repository. There is nothing to change in "*hera_synchronization*";
- Then:
  - Push every changes;
  - Delete "*_grisp*" and "*_build*" in the main folder, if they appear;
  - Do "*rebar3 grisp build*" in the shell within the application's main folder;

  And you should now have an application fully built with your repositories.

E.1.6 Other notes

- If your SD-card goes into "read only mode", it is because the little lid on the side of the micro-SD ⟶ SD-card converter is put on "locked". Simply unlock it as shown here[20].
- The author reminds the reader that if he faces any problems, the GRiSP team is very reactive on the GRiSP Slack.

E.2 User Manual: the *movement_detection* application

E.2.1 Adapting the configuration files to your system

There are three files in the "*/config*" folder:

[20] https://askubuntu.com/questions/213889/microsd-card-is-set-to-read-only-state-how-can-i-write-data-on-it

computer.config.src

The only thing you need to change is within the {*sync_node_optional*} part, where you need to put your boards' name and your computer's name. Note: your computer's name is here required only if you are intent on launching the application on it, and use the computer as a node of the network. E.g.:

```
{sync_nodes_optional, [
    movement_detection@neackow_z4,
    movement_detection@nav_1,
    movement_detection@orderCrate
]},
```

Be careful: if you have "-" in your computer's name, the deployment of the application will not work. To change your computer's name on Linux:

- On the top left corner, go to "*Activities*";
- Search for "*About*";
- The first thing will be the computer name: you can change it to whatever, without a "-" or arithmetic sign.

Note: if you add any "*hera*" environment variables and that you want them to affect your computer, add them in the {*hera*} part of the file. E.g.:

```
{hera, [
    {log_data, false},
    {show_log, false},
    {show_log_spec,false},
    {log_BL,false}
]},
```

sys.config

Here, again, change the {*sync_node_optional*} part. Nothing more is required.

vm.args

This sets the arguments of the Virtual Machine. You can set a new Cookie name (be careful to then change it every where) in the "`-setcookie`" line, or the name of the deployed application in the "`-sname`" line.

Next, if you followed everything that was mentioned previously, there is only one thing left to do in the "*rebar.config*" file, which contains all the information to build and deploy the system: tell the system the path to the SD-card on which you wish to deploy the application. Typically, for the author:

```
{deploy , [
    {pre_script, "rm -rf /media/nicolas/GRISP/*"},
    {destination, "/media/nicolas/GRISP"},
    {post_script, "umount /media/nicolas/GRISP"}
]}
```

Within the */src* folder

The only file that requires your attention is "*movement_detection.app.src*". If you added any modules, it is required to add them to the {*modules*} part.

Files for networking

These files can be found in the "*grisp/grisp2/common/deploy/files/*" folder. If you followed the previous steps, "*erl_inetrc*" and "*wpa_supplicant.conf*" should already be adapted to your system and Wi-Fi networks. Note: in "*grisp.ini.mustache*", at the end of the second line, you can change the name of the Erlang cookie if you desire.

For the Hera application

If you did not created your own Hera repository, this step is not useful. If you did, here are some notes about it:

- In "*ebin/hera.app*", you need to specify the {`env`} variables and every new module that you add to Hera. The same goes for "*src/hera.app.src*".
- In "*rebar.config*", do not forget to change the git reference to "*hera_synchronization*".

E.2.2 How to properly build and deploy the application

Start by removing any "*rebar.lock*" files, either by renaming them or completely removing them. Check that absolutely **no** "*rebar.lock*" are left, either in your main application folder or in Hera folders. The first user manual mentions that it is only needed to deal with that in the main folder, but it is not the case: if you modify anything in Hera, in order to have the new version in the GRiSP2 board, you need to remove that file everywhere. Note: "*rebar.lock*" is where the version of dependencies are locked. This way our builds are reproducible, meaning if no code has been changed in a repository, the build on the repository should always have the same result[21]. If you forget to delete this and that you brought changes to the files, you will waste time building the application, then testing it on the board only to realised it is still the old version.

After any modifications to Hera, push every change to your git repositories. This has been automated using a bash script named "*push_hera.sh*". This pushes the "*hera*" and "*hera_synchronization*" folders, using your GitHub credentials. Modify it to put yours. Beware: put the bash script in the "*.gitignore*" if you directly put your GitHub password there. However, there is a cleaner way of doing it: instead of your password, use something like: "$ENVVAR" and set the environment variable in your session using:

```
export ENVVAR=yourpassword
```

This only works *in your current shell session*, just like the rebar3 variable or the activation of the Erlang installation. **Remember to always do the three commands upon starting a shell.**

To use the script, in the main folder, type in the shell:

```
make COMMIT="Your amazing commit message" push_hera
```

If you want to create your own bash, feel free. But do not forget to type, in the shell:

```
chmod +x bashFileName.sh
```

[21] https://github.com/erlang/rebar3/issues/2604

to add execution rights.

Next, make sure to delete both the "*_build*" and "*_grisp*" folders everywhere you find one. Otherwise, it could have the same effect as the "*rebar.lock*" file. This can be done using a new command of "*makefile*":

```
make clear
```

You can finally build and deploy the application. To build, type:

```
rebar3 grisp build
```

Next, to deploy, the first thing to do is to format each SD-card as *fat32*. It is also suggested to name it *GRISP*. The easiest way to achieve that is to use a partitioning tool like *KDE Partition Manager* or similar. You only need to do this once. Generally, the SD-card of your board should already be formatted. Beware to adapt the *vm.args* file to the name you give here.

Now, plug the SD-card in your computer and use the *makefile* again to deploy the software on each SD-card with the command:

```
make deploy-hostname
```

where *hostname* is the desired name of the GRiSP2 board.

E.2.3 How to use the application

Now that the SD-card is ready, you can plug it into the GRiSP2 board. If your board's name is starting by *nav*, make sure to also plug a PNAV in the SPI2 port of the GRiSP2. If the board's name starts by *order*, then no sensors are expected by the application. Also, make sure to have your Wi-Fi hotspot up and ready.

The detector

Let us suppose that the board's name is "*nav_1*"[22]. After booting, the two LEDs should turn red: this is because no calibration information was found in the system. Indeed, certain sensors require a calibration in order to be used. Within the network, in case of restart, as long as there is at least one node staying alive, the information will remain available.

The gesture recognition algorithm uses the measurements from the `nav3.erl` module, which requires calibration to correct the gyroscope and magnetometer defaults. However, since for now the algorithm only uses the accelerometer data, no real calibration is required. Though, the system requires calibration data. The procedure to calibrate a "*nav*" board is the following:

- Connect serially to the Erlang shell using picocom;
- In the shell, type:

```
movement_detection:set_args(nav3).
```

If you wish to use another sensor or another measurement method than *nav3*, refer to the previous user manual[23] for more detail.

[22] This was chosen for historical reasons, as the previous users of Hera would have many nodes with a PNAV plugged-in and number them to distinguish the nodes. Here, as there is only one board with a PNAV, it is not really useful.

[23] https://github.com/lunelis/sensor_fusion

- Text will appear in the shell. As it is not necessary to calibrate for this application, you can press three times "Enter" to finish the calibration step.

The LEDs are still showing red. They will only become green, indicating that there is calibration information in the system, when the measurements are launched. To launch measurements, type next:

```
movement_detection:launch().
```

Note: if your system comprises many nodes, you can launch every node at once using:

```
movement_detection:launch_all().
```

Next, all one has to do is launch the gesture recognition algorithm. To do so, type:

```
movement_detection:realtime(1.5,-1).
```

Using these arguments, the `grdos/12` function will launch and keep looping forever due to the negative period. It will wait a minimum of 1.5 s between gestures to classify them, as explained earlier. Now, you can move the board around to perform the predefined gestures.

If you have trouble with getting the function started, check your Wi-Fi connection: the gesture recognition algorithm does not work without a stable Wi-Fi connection.

Note: if you want to test the effect of specific gestures without using the gesture recognition algorithm, you can type the following two lines in the detector's shell:

```
net_adm:ping(movement_detection@orderCrate).
rpc:call(movement_detection@orderCrate,sendOrder,
set_state_crate,[Name])
```

where `Name` is the movement whose effect you want to test. The first command will attempt to connect the two boards. If it outputs `pang`, it means there is a network issue. If it outputs `pong`, then the two boards managed to connect to each other. The second command then calls the function `set_state_crate/1` from the `sendOrder` module on the receiver, with the `Name` as argument.

The receiver

Let us suppose the board's name is "*orderCrate*". There is nothing special to do here, beyond booting the board. If it was named correctly, the two LEDs should light up, respectively in a light blue and yellow colour, from left to right. Beware that it does not mean that the board connected to the Wi-Fi. This is where having a hotspot is useful, because the user can see how many devices are connected to it. In this application, two devices should be connected for everything to work: the detector and the receiver.

What gestures can be performed in a successive manner?

Every movement combinations are possible, however, some of them were deemed "unsafe" in the present application and thus lead to a stop of the robot:

- If one movement labelled as *forward* is followed by one movement labelled as *backward* or vice-versa, the robot is stopped.

- If the user wants to turn the robot around but did not stop prior to this, the robot is stopped.

Extensions to the platform and further development

If you wish to add new dynamic measurements, other sensors or new sensor fusion models, the author invites you to refer to the previous user manual[24], as nothing has changed since.

[24] https://github.com/lunelis/sensor_fusion

Appendix F

Source code

This application is based on the already existing *sensor_fusion* application. Some of the code implemented for this thesis has been simply added to pre-existing files. In these cases, only the modified portion of the file will be shown. However, if minor changes have been made at many places in the code, the whole code is shown. For each file, the code creator(s) will be credited. Also, for the sake of completeness, some files have been renamed to fit the new application's name. If it is the case, it will be mentioned. Finally, "[...]" means that some code is not shown but present.

Note: the initial code for Hera and the main application's code came with barely any comments. Many comments were added whilst the author was attempting to debug the logger problem. They were left there for future users to more easily understand the code. Also, two functions (`output_log/2` and `output_log_spec/2`) that can be activated by setting a Hera environment variable to *true* are present in most Hera files but also in the main application's files. They can be used for debugging purposes and are visible at the start of the files in which they have been added. They will be shown only once, in the first code.

In some codes, only comments were added. This is the case with `csvparser.erl`, `learn.erl`, `e11.erl`, `nav3.erl` and all the Hera source code files. They will not be shown here but can be found in the application's GitHub[1].

[1] *hera*: https://github.com/Neackow/hera/tree/main/src, *movement_detection*: https://github.com/Neackow/movement_detection/tree/main/src

F.1 hera_data.erl

Credits and comments

This code was created by Sébastien Kalbusch and Vincent Verpoten. Only the beginning is shown to illustrate the `output_log/2` and `output_log_spec/2` functions.

Code

```
    [...]

    % Decide whether or not to print the comments. Remember to
   change it in your environment.
    output_log(Message, Args) ->
        ShowLogs = application:get_env(hera, show_log, false),
        if
            ShowLogs ->
                io:format(Message,Args);
            true ->
                ok
        end.

    % Decide whether or not to print the comments. Remember to
   change it in your environment.
    output_log_spec(Message, Args) ->
        {{Year, Month, Day}, {Hour, Min, Sec}} = calendar:
   now_to_datetime(erlang:timestamp()),
        DisplayedTime = list_to_binary(io_lib:format("~.2.0w:~.2.0w
   :~.2.0w", [Hour, Min, Sec])),

        ShowLogs = application:get_env(hera, show_log_spec, false),
        if
            ShowLogs ->
                if Args == [] ->
                    io:format("[~p]: ~p.~n",[DisplayedTime, Message
   ]);
                   true ->
                    FullMessage = "[~p]: " ++ Message,
                    io:format(FullMessage, [DisplayedTime|Args])
                end;
            true ->
                ok
        end.

    [...]
```

F.2 classify.erl

Credits and comments

This code was created by Sébastien Gios. It was adapted to send the appropriate command using `rpc:call/4`. Only that part is shown.

Code

```
-module(classify).

-import(csvparser, [parse_CSV/2, parse/2, print_list/1]).
-import(learn, [analyze/1, analyze_CSV/1, regroup/1, average/1]
).
-export([import_gesture/0, classify_new_gesture/1,
classify_new_gesture_CSV/1]).

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

% Used by realtime.erl
classify_new_gesture(List) ->
    List_gestures = import_gesture(),

    VectorX = parse(List, 1), % 1 is the index of the x axis
acceleration
    PatternX = analyze(VectorX),    % This will generate the
initial pattern, made of nn, n, o, p and pps.
    Clean_PatX = average(PatternX), % This then cleans the
pattern by making averages, in order to reduce noise.
    NewX = regroup(Clean_PatX), % New is the general flow of
the new gesture
    io:format("NewX : ~p~n", [NewX]),

    VectorY = parse(List, 2), % 2 is the index of the y axis
acceleration
    PatternY = analyze(VectorY),
    Clean_PatY = average(PatternY),
    NewY = regroup(Clean_PatY), % New is the general flow of
the new gesture
    io:format("NewY : ~p~n", [NewY]),

    VectorZ = parse(List, 3), % 3 is the index of the z axis
acceleration
    PatternZ = analyze(VectorZ),
    Clean_PatZ = average(PatternZ),
    NewZ = regroup(Clean_PatZ), % New is the general flow of
the new gesture
    io:format("NewZ : ~p~n", [NewZ]),

    {Name, Accuracy} = compare_gesture(NewX, NewY, NewZ,
List_gestures),
```

```
      % print the result of the classification
      if Accuracy >= 0.7 ->
          % Here: call the function from the other GRiSP to send
the movement.
          % Only done when the accuracy is good enough (security
measure). This will help a bit with forbidding some movements.
          net_adm:ping(movement_detection@orderCrate), % To
connect the node. I could do it only once, but since the same
code is used on both GRiSP,
                                                    % the risk
would be that GRiSP2 tries to connect... to GRiSP2. So, redo it
here, it's no big deal.
          rpc:call(movement_detection@orderCrate, sendOrder,
set_state_crate, [Name]),
          io:format("Name : ~p, with Acc : ~p~n", [Name, Accuracy
]);
      true ->
          net_adm:ping(movement_detection@orderCrate),
          rpc:call(movement_detection@orderCrate, sendOrder,
set_state_crate, [stopCrate]), % Stop the robot in case of wrong
 gesture.
          io:format("Too low Accuracy, No gesture recognized~n")
      end,
      {Name, Accuracy}.

  [...]
```

F.3 realtime.erl

Credits and comments

This code was created by Sébastien Gios. The function `grdos/11` was adapted to give the `grdos/12` function. Only that part is shown.

Code

```
    [...]

    % ====== <EXPLANATIONS FOR THE GRDOS FUNCTION> ======
    % TO(TimeOut) : time to stay without moving
    % AS(Average Size) : Size of the List where we will do the
   average (it is too reduce the noise)
    % List : List used to detect the stop
    % SizeL : Size of the List
    % GestureList: List of collected data since last gesture
    % LastT : Time of the last data, used to detectif Hera produce
   a new data
    % TSM(Time Since Move) : Last time a movement was detected, if
   greater than TO, then we have a stop
    % LastX, LastY, LastZ : Last gesture for an axis
    % grdos => gesture_recognition_division_over_stop
    % ====== </EXPLANATIONS FOR THE GRDOS FUNCTION> ======

    grdos(TO, Period, AS, List, SizeL, GestureList, LastT, TSM,
   LastX, LastY, LastZ, Counter) ->
        [{_, _,Time, Data}] = hera_data:get(nav3,
   movement_detection@nav_1),
% hera_data:get answers back with the structure {Node, Seq,
   Timestamp, Data}
        if Time > Period andalso Period > 0 ->
            io:format("~n~n~n"), % Just to make it more readable
            io:format("End of Timer!~nCalculating...~n"),
%classify:classify_new_gesture(GestureList); % Initially: classify
   the gesture with whatever data is present. Bad for me.
% New version: at the end of the timer, automatically stop the
   crate.
            net_adm:ping(movement_detection@orderCrate),
            rpc:call(movement_detection@orderCrate, sendOrder,
   set_state_crate, [stopCrate]);
        true ->
            if Time == LastT ->
                grdos(TO, Period, AS, List, SizeL, GestureList,
   Time, TSM, LastX, LastY, LastZ, Counter); % Skip if no new data
            true ->
                NewGestureList = lists:append(GestureList, [Data]),
                NewList = lists:append(List, [Data]),
                NewSizeL = SizeL + 1,
                if NewSizeL >= AS ->  % It means we can compute the
    average. AS = Average Size.
```

```
                    ListX = csvparser:parse(NewList, 1),
                    ListY = csvparser:parse(NewList, 2),
                    ListZ = csvparser:parse(NewList, 3),
                    PatternX = learn:analyze(ListX),
                    PatternY = learn:analyze(ListY),
                    PatternZ = learn:analyze(ListZ),
                    AvgX = learn:average(PatternX),
                    AvgY = learn:average(PatternY),
                    AvgZ = learn:average(PatternZ),

% This is used to extract the atom from the list.
                    FirstElement = lists:nth(1,AvgZ),

% Immediately detects a stop and send it. Do not increment the
   counter, as it would interfere with the end of algorithm routine
   .
                    if FirstElement == nn ->
                        net_adm:ping(movement_detection@orderCrate)
   ,
                        rpc:call(movement_detection@orderCrate,
   sendOrder, set_state_crate, [stopCrate]);
                    true ->
                        ok
                    end,

                    [HX|_] = AvgX,
                    [HY|_] = AvgY,
                    [HZ|_] = AvgZ,
                    if LastX == HX andalso LastY == HY andalso
   LastZ == HZ -> % If the last gesture is the same as the new one
                        if Time >= TSM + TO ->
                            io:format("~n~n~n~n"), % Just to make
   it more readable
                            io:format("Stop detected!~n"),
                            {Name, Accuracy} = classify:
   classify_new_gesture(GestureList),
                            if Accuracy >= 0.7 ->
                                if Name == stopCrate ->
                                    NewCounter = Counter#counter{
   value = Counter#counter.value + 1};
                                true ->
                                    NewCounter = Counter#counter{
   value = 0}
                                end;
                            true ->
                                NewCounter = Counter
                            end,
                            if NewCounter#counter.value == 3 ->
% We need to do it on NewCounter, otherwise, it would take one
   useless step more to stop.
% We put the period at 1, so that it is immediately over at the
   next call. Reset the counter for good measure.
```

```
                                grdos(TO, 1, AS, [], 0, [], Time,
Time, LastX, LastY, LastZ, CounterReset = #counter{value = 0});
                            true ->
                                grdos(TO, Period, AS, [], 0, [],
Time, Time, LastX, LastY, LastZ, NewCounter)
                            end;
                        true -> % too soon, still need to wait
                            grdos(TO, Period, AS, [], 0,
GestureList, Time, TSM, LastX, LastY, LastZ, Counter)
                        end;
                    true ->
                        NewLastX = HX,
                        NewLastY = HY,
                        NewLastZ = HZ,
                        NewTSM = Time,
                        grdos(TO, Period, AS, [], 0, NewGestureList
, Time, NewTSM, NewLastX, NewLastY, NewLastZ, Counter)
                    end;
                true ->
                    grdos(TO, Period, AS, NewList, NewSizeL,
NewGestureList, Time, TSM, LastX, LastY, LastZ, Counter)
                end
            end
        end.
```

F.4 sendOrder_sup.erl

Credits and comments

Code created by the author. It implements the supervisor of the `gen_server` which converts received gestures into the appropriate commands and the `gen_server` for the I2C communication.

Code

```
-module(sendOrder_sup).
-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).


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

init(_Args) ->
    SupFlags = #{
      strategy => one_for_one,
      intensity => 1,
      period => 5
    },
    SendOrder = #{
      id => sendOrder,
      start => {sendOrder, start_link, []},
      restart => transient,
      shutdown => 5000,
      type => worker,
      modules => [sendOrder]
    },
    I2C = #{
        id => i2c_communication,
        start => {i2c_communication, start_link, []}
    },

    ChildSpecs = [SendOrder,I2C],
    {ok, {SupFlags, ChildSpecs}}.
```

F.5 sendOrder.erl

Credits and comments

Code created by the author. It implements the `gen_server` which converts received gestures into the appropriate commands.

Code

```
 % Module to convert gestures to orders on an object.
 % This V1.0 offers the ability to control a wine crate on
wheels, with gestures associated to defined orders.
 % This is called by the set_state_crate(MovementDetected),
which calls the 'ctrlCrate' version of the handle_call function.
 % Extensions of this code are straightforward: either do your
own functions and redirect the orders to it, or extend the
handle_call procedures
 % for more functionalities. E.g.: if the PMOD Ultrawideband
comes out, a follower function can be added.
 % This could be called via gen_server:call(?MODULE, {follower,
WhateverArgument}) and redirect to an appropriate handle_call
procedure.

 -module(sendOrder).

 -behaviour(gen_server).


 -export([start_link/0, set_state_crate/1, checkingConnection/1]
).
 -export([init/1, handle_call/3, handle_cast/2, handle_info/2,
terminate/2]).

 % At initialisation (see init/1 function): currentVelocity =
100 RPM (in forward direction).
 % prevName: used to forbid unauthorised gesture succession. Set
 at stopCrate, by default.
 % movName: store the movement name as detected by the GRiSP2
board.
 % movMode: this is to deal with submodes. E.g.: if I want to be
 in 'changeVelocity' mode, then it stores it in this variable.
 % By default: stopCrate and normal, normal being the 'by
default' mode.
 -record(movState, {currentVelocity, prevName, movName, movMode}
).

 % At initialisation, this counter is set to 0 and the function
verifying that the other node is connected is launched.
 -record(counter, {value = 0}).


 % ===============================================
 % ============= <public functions> =============
```

```
 % ================================================

 start_link() ->
     % Initiates the gen_server. Calls init/1.
     gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

 %% Synchronous call: sets the new state according to the
detected movement. This function is called by rpc:call on the
other GRiSP2.
 set_state_crate(MovementDetected) ->
     if(MovementDetected == testRPC) -> % To test the good
connection between the two GRiSPs. Useless otherwise. Manual
command.
         grisp_led:color(1,white),
         grisp_led:color(2,black);
         true ->
             ok
     end,
     gen_server:call(?MODULE, {ctrlCrate, MovementDetected}).

 % Checking if the other board is still answering and available
to send orders. This is a safety measure, a guard-rail.
 % Has to be public in order to be called by apply_after.
 % Also, in parallel, "ping" the Raspberry board to tell it the
receiver is still alive.
 checkingConnection(Counter) ->
     Connected = net_adm:ping(movement_detection@nav_1),
     if Connected == pang ->
         NewCounter = Counter#counter{value = Counter#counter.
value + 1};
     true ->
         NewCounter = Counter#counter{value = 0}
     end,

     if NewCounter#counter.value == 1 ->
         i2c_communication:send_data([0,1,0,0,2]), % Stop the
crate.
         FinalCounter = Counter#counter{value = 0};
     true ->
         FinalCounter = NewCounter
     end,
     i2c_communication:send_data([1]), % We send a 1, which will
 be added to a counter in the Rasp.
     timer:apply_after(1000, sendOrder, checkingConnection, [
FinalCounter]).

 % ================================================
 % ============= </public functions> =============
 % ================================================


 % ================================================
```

```
 % ============= <private functions> =============
 % ================================================

 % Structure of Order: [V1, DIR1, V2, DIR2, command_index].
 % The command indicator allows the controller to know if it's
turning, simply moving forward, etc.
 % This could have been simply implemented within the controller
 by comparing velocities, etc. but since it was needed in
changeVelocity mode to try it out,
 % it may aswell be reused for simplicity.
 % command_index = 0 -> continuous mode, smooth position profile
 ; 1 -> test velocity in changeVelocity mode ; 2 -> crate-on-
wheels stopping ; 3 -> turning around.
 order_crate(State) ->
     Order = case State#movState.movMode of
         changeVelocity ->
             case State#movState.movName of
                 stopCrate ->   % Stop the crate from whatever
it was doing. Reset values to baseline.
                     NewState = State#movState{currentVelocity =
 100},
                     [0,1,0,0,2];
                 accelerate ->   % When we are changing the
velocity, do not move the crate, by default. Just change the
state.
                     if State#movState.currentVelocity == 100 ->
                         NewState = State#movState{
currentVelocity = 110};
                     State#movState.currentVelocity == 110 ->
                         NewState = State#movState{
currentVelocity = 120};
                     State#movState.currentVelocity >= 120 ->
                         io:format("Cannot accelerate further!~n
"),
                         NewState = State#movState{
currentVelocity = 120};
                     true ->
                         NewState = State#movState{
currentVelocity = State#movState.currentVelocity}
                     end,
                     [0,1,0,0,0];
                 decelerate ->
                     if State#movState.currentVelocity == 120 ->
                         NewState = State#movState{
currentVelocity = 110};
                     State#movState.currentVelocity == 110 ->
                         NewState = State#movState{
currentVelocity = 100};
                     State#movState.currentVelocity =< 100 -> %
Just in case the sun wants to play with me.
                         io:format("Cannot decelerate further!~n
"),
```

```
                            NewState = State#movState{
    currentVelocity = 100};
                        true ->
                            NewState = State#movState{
    currentVelocity = State#movState.currentVelocity}
                        end,
                        [0,1,0,0,0];
                    testingVelocity ->
                        NewState = State,
                        [State#movState.currentVelocity,1,State#
    movState.currentVelocity,0,1];
                    _ ->
                        NewState = State,
                        io:format("Invalid command when in
    changeVelocity mode.~n"),
                        [0,1,0,0,2]
% When order is invalid, automatically set to 0. Send to "slow down
     to 0", we stop the crate.
                end;
            normal ->
                case State#movState.movName of
                    stopCrate ->   % Default command, crate does
    not move.
                        NewState = State#movState{prevName =
    stopCrate},
                        [0,1,0,0,2];
                    forward ->
                        NewState = State#movState{prevName =
    forward},
                        [State#movState.currentVelocity,1,State#
    movState.currentVelocity,0,0];
                    backward ->
                        NewState = State#movState{prevName =
    backward},
                        [State#movState.currentVelocity,0,State#
    movState.currentVelocity,1,0];
                    forwardTurnLeft ->
                        NewState = State#movState{prevName =
    forward},
                        [80,1,110,0,0];
% When turning, we stay in "continuous" mode, an a dedicated
    function will adapt the velocities.
% The velocites are fixed. This is a design choice, to have a slow
    turn, to keep as much control on the crate as possible.
                    forwardTurnRight ->
                        NewState = State#movState{prevName =
    forward},
                        [110,1,80,0,0];
                    backwardTurnleft ->
                        NewState = State#movState{prevName =
    backward},
                        [80,0,110,1,0];
```

```
                    backwardTurnRight ->
                        NewState = State#movState{prevName =
    backward},
                        [110,0,80,1,0];
                    turnAround ->
                        NewState = State#movState{prevName =
    stopCrate},
% prevName = stopCrate is necessary to be able to call turnAround
    multiple times in a row, due to the condition on the gesture.
                        io:format("*briiight eyes* EVERY NOW AND
    THEN I FALL APART!~n"),
                        [100,1,100,1,3]; % Turn on itself, towards
    the right. Fixed at 100 RPM, could be less.
                    _ ->
                        NewState = State,
                        io:format("Unknown movement name while in
    movMode normal.~n"),
                        [0,1,0,0,2]
                end;
            _ ->
                NewState = State,
                io:format("Bad movement mode, crate will not move
    as a security measure."),
                [0,1,0,0,2]
        end,

        % Send command to the micro-controller.
        i2c_communication:send_data(Order),
        io:format("Order is ~p~n", [Order]),
        io:format("Current state is ~p~n", [NewState]),
        NewState.

    % Reads the first 'Nbr' of letters from the Movement variable.
    movementComparison(Movement,Nbr) ->
        NewList = atom_to_list(Movement),
        lists:sublist(NewList,1,Nbr).


    % ================================================
    % ============= </private functions> =============
    % ================================================


    % ================================================
    % ============ <gen_server functions> ============
    % ================================================

    init([]) ->
% Not really useful if I don't need to deal with something when the
    gen_server goes down.
        process_flag(trap_exit, true),
```

```
% Display message to the console: allows to see if the function is
   correctly being setup from the shell.
        io:format("Object controller is being setup!~n"),

% Change LED colors: allow to visually tell if the process has been
    launched, or not.
        grisp_led:color(1,aqua),
        grisp_led:color(2,yellow),

% Initialise the counter and launch the function immediately.
        checkingConnection(Counter = #counter{}),

% Set default state and return {ok, state}. State is the internal
   state of the gen_server.
        {ok, #movState{currentVelocity = 100, prevName = stopCrate,
    movName = stopCrate, movMode = normal}}.

    handle_call({ctrlCrate, MovementDetected}, From, State = #
   movState{currentVelocity = CurrentVelocity, movName = MovName,
   movMode = MovMode}) ->
        Available = i2c_communication:read_data(),
        SuffixMovement = movementComparison(MovementDetected,7), %
   This can't be used as a guard. So, define variable outside.
        PreviousSuffix = movementComparison(State#movState.prevName
   ,7), % Only on 7 letters, so as to not double everything.
        if Available == 1 ->
            if MovementDetected == changeVelocity ->
                NewState = State#movState{prevName = changeVelocity
   , movName = stopCrate, movMode = changeVelocity};
% When entering changeVelocity mode, stop the crate. It allows easy
    reset of the changeVelocity mode, in case the user is lost.
% We set here that the previous move was changeVelocity, only to do
    it once at entry and same for exitChangeVelocity.
            MovementDetected == exitChangeVelocity ->
                NewState = State#movState{prevName =
   exitChangeVelocity, movName = stopCrate, movMode = normal};
% When exiting changeVelocity mode, stop the crate, once again as
   security measure.
            SuffixMovement == "forward" ->  % If the current move
   says "forward"...
                if PreviousSuffix == "backwar" -> % And that now,
   we would like to go "backward"...
                    NewState = State#movState{prevName = forward,
   movName = stopCrate}; % Stop the crate.
                true ->
                    NewState = State#movState{movName =
   MovementDetected}
                end;
            SuffixMovement == "backwar" -> % Same as priori
   condition, but the other way around.
                if PreviousSuffix == "forward" ->
                    NewState = State#movState{prevName = backward,
```

```
movName = stopCrate};
                true ->
                    NewState = State#movState{movName =
MovementDetected}
                end;
            SuffixMovement == "turnAro" -> % For the turnAround
rail-guard.
                if PreviousSuffix == "stopCra" ->
                    NewState = State#movState{prevName = turnAround
, movName = turnAround}; % If previous gesture was stopCrate, it
's ok.
                true ->
                    NewState = State#movState{prevName = stopCrate,
 movName = stopCrate}    % If it was anything else, problem. We
stop the robot.
                end;
            true ->
                NewState = State#movState{movName =
MovementDetected}
            end,
            FinalNewState = order_crate(NewState);
        true ->
            io:format("The controller is currently unavailable.
Please wait.~n"),
            FinalNewState = State
        end,
        {reply, ok, FinalNewState};

 handle_call(stop, From, State = #movState{}) ->
        io:format("Handling stop.~n", []),
        {stop, normal, ok, State}.


 handle_cast(Request, State) ->
        {reply, ok, State}.

 handle_info(Msg, State) ->
        io:format("Unexpected message: ~p~n",[Msg]),
        {noreply, State}.


 terminate(normal, State) ->
        io:format("sendOrder: normal termination~n",[]);

 terminate(shutdown, State = #movState{currentVelocity=
CurrentVelocity,movName=MovName,movMode=MovMode}) ->
        % terminating
        io:format("sendOrder: managing shutdown~n",[]);

 terminate(Reason, State) ->
        io:format("sendOrder: other termination with reason: ~p~n",
 [Reason]).
```

```
% ====================================================
% ============= </gen_server functions> =============
% ====================================================
```

F.6 i2c_communication.erl

Credits and comments

Code created by the author. It implements the `gen_server` for I2C communication.

Code

```
% Module to safely send messages to any slave from a GRiSP.
% This piece of code has been implemented in order to avoid any
bus conflict, since multiple functions could be wanting to use
the bus at the same
% time to do different tasks.

-module(i2c_communication).

-behaviour(gen_server).


-export([start_link/0, send_data/1, read_data/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
terminate/2]).

% Store the bus name in the state, to open it once at
initialisation and continuously be able to call it.
-record(busI2C, {busName}).

% ============================================
% ============ <public functions> ============
% ============================================

start_link() ->
    % Initiate an I2C communication.
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

send_data(Command) ->
    gen_server:cast(?MODULE, {write, Command}).

read_data() ->
    gen_server:call(?MODULE, {read}).

% ============================================
% ============ </public functions> ============
% ============================================


% ============================================
% ============ <private functions> ============
% ============================================

% Send the command using the I2C ports of the GRiSP. Thank you
Cedric Ponsard for this piece of code!
```

```
 % For more details on this port: https://digilent.com/blog/new-
i2c-standard-for-pmods/

 send_i2c(Command, Bus) ->
     List_data = lists:flatten(lists:map(fun(X) -> X end, lists:
map(fun(X) -> lists:sublist(binary_to_list(term_to_binary(X))
,3,8) end, Command))),
     grisp_i2c:transfer(Bus, [{write, 16#40, 1, List_data}]), %
16#40 is the fixed address of the Raspberry Pi Pico W, in
hexadecimal format (0x40).
     io:format("Command sent to the micro-controller!~n").

 % Read from the controller if we are avaiable or not.
 read_i2c(Bus) ->
     Message = grisp_i2c:transfer(Bus, [{read, 16#40, 1, 1}]), %
 Reads 1 byte from slave at address 0x40, in register 1.
     io:format("Message received is ~p~n", [Message]),
     Available = lists:nth(1, binary_to_list(lists:nth(1,
Message))). % Convert the binary coded on 8 bits and received as
 [<<val>>] to an integer.


 % ================================================
 % ============ </private functions> ============
 % ================================================


 % ================================================
 % =========== <gen_server functions> ===========
 % ================================================

 init([]) ->
     process_flag(trap_exit, true), % Not really useful if I don
't need to deal with something when the gen_server goes down.
     % Display message to the console: allows to see if the
function is correctly being setup from the shell.
     io:format("An I2C communication is being setup!~n"),
     I2CBus = grisp_i2c:open(i2c1), % Open the bus.
     % Set default state and return {ok, state}. State is the
internal state of the gen_server.
     {ok, #busI2C{busName = I2CBus}}.


 % Deal with reading from Raspberry.
 handle_call({read}, From, State = #busI2C{busName = BusName})
->
     FinalState = State,
     Available = read_i2c(BusName),
     io:format("The bus is ~p.~n", [BusName]),
     {reply, Available, FinalState};

 handle_call(stop, From, State = #busI2C{}) ->
```

```
        io:format("Stopping I2C communication.~n", []),
        {stop, normal, ok, State}.


    % Deal with writing a command to the Raspberry.
    handle_cast({write, Command}, State = #busI2C{busName = BusName
}) ->
        FinalState = State,
        send_i2c(Command, BusName),
        {noreply, FinalState};

    % If anything else.
    handle_cast(_Request, State = #busI2C{}) ->
        {noreply, State}.


    handle_info(Msg, State) ->
        io:format("Unexpected message: ~p~n",[Msg]),
        {noreply, State}.


    terminate(normal, State) ->
        io:format("i2c_communication: normal termination~n",[]);

    terminate(shutdown, State = #busI2C{busName=BusName}) ->
        % terminating
        io:format("i2c_communication: managing shutdown~n",[]);

    terminate(Reason, State) ->
        io:format("i2c_communication: other termination with reason
: ~p~n", [Reason]).

    % ================================================
    % ============ </gen_server functions> ============
    % ================================================
```

F.7 hera_sup.erl

Credits and comments

This code was created by Sébastien Kalbusch and Vincent Verpoten. It implements the supervisor of the *hera* application and has been adapted to launch the `gen_server` which is used to store the data at the start.

Code

```
-module(hera_sup).

-behaviour(supervisor).

-export([start_link/0]).
-export([init/1]).

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

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

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

init([]) ->
    SupFlags = #{
        strategy => one_for_one,
        intensity => 6,
        period => 3600,
        shutdown => 2000 % Used to give the time to
buffered_logger to log the data.
    },
    HeraData = #{
        id => hera_data,
        start => {hera_data, start_link, []}
    },
    HeraCom = #{
        id => hera_com,
        start => {hera_com, start_link, []}
    },
    HeraMeasureSup = #{
        id => hera_measure_sup,
        start => {hera_measure_sup, start_link, []},
        type => supervisor
    },
    HeraBufferedLogger = #{
        id => buffered_logger,
        start => {buffered_logger, start_link, [{200}]} % 200 =
 number of data to be saved in one go. Each 200 data, save them.
```

```
        },
        ChildSpecs = [HeraData, HeraCom, HeraMeasureSup,
    HeraBufferedLogger],
        {ok, {SupFlags, ChildSpecs}}.
```

F.8 buffered_logger.erl

Credits and comments

Code created by the author, with great help of his father. It implements the `gen_server` which is used to store the data.

Code

```
    -module(buffered_logger).

    -behaviour(gen_server).

    -export([start_link/1, store_record/2, print_all_contents/0]).
    -export([init/1, handle_call/3, handle_cast/2, terminate/2]).

    % Values by default for the state of the buffered_logger.
    -record(stateBL, {internal_counter=0, max_buffer_size = 200,
   root_table}).


    start_link(Arguments) ->
        % Arguments must have the form {BufferSize = 10}
        gen_server:start_link({local, ?MODULE}, ?MODULE, Arguments,
    []).

    % ================================================
    % ============= <public functions> =============
    % ================================================

    print_all_contents() ->
        gen_server:call(?MODULE, {print_all_content}).

    store_record(TableName, Record) ->
        gen_server:call(?MODULE, {store, TableName, Record}).

    % ================================================
    % ============= </public functions> =============
    % ================================================


    % ================================================
    % ============= <private functions> =============
    % ================================================

    create_subtable(SubTableName) ->
        ets:new(SubTableName, [ordered_set]).

    insert_data(Table, Key, Record) ->
% It is always {Key, Record}: Key is incremental, Record is the
   data to be stored. For nav3, Record={Seq, T, Ms}.
        ets:insert(Table, {Key, Record}).
```

```
    % Print the content of one table. Debugging function.
    print_table_records([]) ->
        ok;
    print_table_records([H|T]) ->
        {Key, Record} = H,
        io:format("    [~p]: ~p ~n",[Key, Record]),
        print_table_records(T).

    % Print table list. Debugging function.
    print_table_list([]) ->
        ok;
    print_table_list([H|T]) ->
        {Key, Table} = H,
        io:format("Table [~p]: ~n",[Key]),
        TableRecords = ets:tab2list(Table),
        print_table_records(TableRecords),
        print_table_list(T).

% Store table to file. Used when the buffer overflows, or when at
   the end we flush the rest of the data.
    file_name(Name) ->
        lists:append(
            ["measures/", atom_to_list(Name), ".csv"]).

% Adapt this function to be compatible with the measure you want to
    store.
% Acc = accumulator. SubTable is the sub-directory containing the
   log, to be added to the file. TableName = name given to SubTable
   , used for filename.
    save_data_to_file(TableName, SubTable) ->

        Content = ets:foldl(fun({Key, Values}, Acc) -> % Here:
   format the data as before. Values is of the format {Seq, T, Ms}
   -> use pattern matching.
            {Seq, T, Ms} = Values,
            Vals = lists:map(fun(V) -> lists:flatten(io_lib:format(
   "~p", [V])) end, Ms),
            S = string:join(Vals, ","),
            Bytes = io_lib:format("~p,~p,~s~n", [Seq, T, S]),
            Acc ++ Bytes end, [], SubTable),
        ok = filelib:ensure_dir("measures/"),
        FileName = file_name(TableName),
        ok = file:write_file(FileName, Content, [append]),
        ok.

    % ================================================
    % ============= </private functions> =============
    % ================================================


    % ================================================
```

```
    % ============= <gen_server functions> =============
    % ================================================

    init(Arguments) ->
        % Create a 'root' table that will store sub-tables
        % Initialize the State with default values

        {MaxBufferSize} = Arguments,
        process_flag(trap_exit, true), % Needed for the flushing
   when crash down of the app.
        NewRootTable = ets:new(buffered_logger_root,[set, private])
   ,
        % Display success message
        io:format("Root table: ~p; max buffer size: ~p.~n",[
   NewRootTable, MaxBufferSize]),
        % Return initial state
        {ok, #stateBL{internal_counter = 0, max_buffer_size =
   MaxBufferSize, root_table = NewRootTable}}.

    % When storing the current content in a file.
    storeAndEmptyDictionary(RootTable, SubTable, TableName) ->
        io:format("Storing dictionary on SD-cards: ~p~n",[TableName
   ]),
        save_data_to_file(TableName, SubTable),
        ets:delete(RootTable, TableName).

    handle_call({store, TableName, Record}, From, State = #stateBL{
   internal_counter=InternalCounter, max_buffer_size=MaxBufferSize,
    root_table=RootTable}) ->
        % Search for TableName in stored tables
        Match = ets:lookup(RootTable, TableName),
        SubTable =
            if Match =:= [] ->
                % The TableName was not created
                LocalSubTable = create_subtable(TableName),
                insert_data(RootTable, TableName, LocalSubTable),
                LocalSubTable;
            true ->
                % The TableName already exists
                {_, ExistingSubTable} = lists:nth(1, Match),
                ExistingSubTable
        end,

        % The counter is shared between subtables to keep evolution
        NewCounter = InternalCounter + 1,
        insert_data(SubTable, NewCounter, Record),

% If the sub-table size is > max_buffer_size, store the content on
   SD-card and delete from memory

        SubTableSize = ets:info(SubTable, size),
        %io:format("~p size: ~p.~n",[TableName, SubTableSize]),
```

```
      StoreResult =
          if SubTableSize >= MaxBufferSize ->
              storeAndEmptyDictionary(RootTable, SubTable,
TableName);
          true ->
              ok
      end,

      % Return the updated state

      {reply, ok, #stateBL{internal_counter = NewCounter,
max_buffer_size = MaxBufferSize, root_table = RootTable}};

 handle_call({print_all_content}, From, State = #stateBL{
internal_counter=InternalCounter, max_buffer_size =
MaxBufferSize, root_table=RootTable}) ->
      AllTableList = ets:tab2list(RootTable),
      print_table_list(AllTableList),
      {reply, ok, State};

 handle_call(stop, From, State=#stateBL{}) ->
      io:format("Handling stop ~n", []),
      {stop, normal, ok, State}.

 handle_cast(Request, State) ->
      {reply, ok, State}.

 handle_info(Msg, State) ->
      io:format("Unexpected message: ~p~n",[Msg]),
      {noreply, State}.

 terminate(normal, S) ->
      io:format("Buffered_logger: normal termination~n",[]);

 terminate(shutdown, State = #stateBL{internal_counter=
InternalCounter, max_buffer_size = MaxBufferSize, root_table=
RootTable}) ->
      % Terminating
      % Store each remaining dictionary on SD-card
      io:format("Buffered_logger: managing shutdown~n",[]),
      TableList = ets:tab2list(RootTable),
      lists:foreach(fun(LocDictionaryEntry) ->
                  {TableName, SubDictionary} = LocDictionaryEntry
,
                  storeAndEmptyDictionary(RootTable,
SubDictionary, TableName)
              end, TableList);

 terminate(Reason, S) ->
      io:format("Buffered_logger: other termination: ~p.~n", [
Reason]).
```

```
% ====================================================
% ============= </gen_server functions> ==============
% ====================================================
```

F.9 movement_detection.erl

Credits and comments

Initially named `sensor_fusion.erl`, this code was created by Sébastien Kalbusch and Vincent Verpoten and later reworked by Sébastien Gios and Lucas Nélis. The versions of the two last aforementioned authors were merged and the final application was built upon the resulting code.

Code

```
-module(movement_detection).

-behavior(application).

-export([set_args/1, set_args/4]).
-export([launch/0, launch_all/0, stop_all/0]).
-export([update_code/2, update_code/3]).
-export([start/2, stop/1]).
% Integrate the movement detection.
-export([realtime/0, realtime/2, realtime_once/0, realtime_once
/1, clear_gesture/0]).

[...]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%      MODULE API      %%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% set the args for the nav and mag modules
set_args(nav) ->
    Cn = nav:calibrate(),
    Cm = mag:calibrate(),
    update_table({{nav, node()}, Cn}),
    update_table({{mag, node()}, Cm});

%% set the args for the nav3 and e11 modules
%% Calibration function for the 3D orientation test.
set_args(nav3) ->
    Cn = nav3:calibrate(),
    R0 = e11:calibrate(element(3, Cn)),
    update_table({{nav3, node()}, Cn}),
    update_table({{e11, node()}, R0}).


%% set the args for the sonar module.
% Any measure above RangeMax [m] will be ignored.
% For e5 to e9, X and Y are the coordinate of the sonar in [m].
% For >= e10, X and Y are the Offset in [m] and Direction (1 or
-1).
set_args(sonar, RangeMax, X, Y) ->
    update_table({{sonar, node()}, {RangeMax,X,Y}}).
```

```
    launch() ->
% Calling this function will lead to the launch of the correct
   function: either a nav or a sonar one. ADDED 2024: or an order
   board.
        case node_type() of
            order -> % If we have an "order" board, then do not try
    to launch the measurement.
                ok;
            _ ->
                try launch(node_type()) of
                    ok -> % This will be sent if the calibration
   data is present. Otherwise, red leds.
                        [grisp_led:color(L, green) || L <- [1, 2]],
                        ok
                catch
                    error:badarg ->
                        [grisp_led:color(L, red) || L <- [1, 2]],
                        {error, badarg}
                end
        end.


    launch_all() ->
        rpc:multicall(?MODULE, launch, []).
% rpc:multicall/3 is equivalent to multicall([node()|nodes()],
   Module, Function, Args, infinity).
% This will start the function 'launch' of the module
   movement_detection in every node connected to the network of
   boards.


    stop_all() ->
        _ = rpc:multicall(application, stop, [hera]),
        _ = rpc:multicall(application, start, [hera]),
        ok.


    %% to be called on the source node
    update_code(Application, Module) ->
        {ok,_} = c:c(Module),
        {_,Binary,_} = code:get_object_code(Module),
        rpc:multicall(nodes(), ?MODULE, update_code,
            [Application, Module, Binary]).


    %% to be called on the destination node
    update_code(Application, Module, Binary) ->
        AppFile = atom_to_list(Application) ++ ".app",
        FullPath = code:where_is_file(AppFile),
        PathLen = length(FullPath) - length(AppFile),
        {Path,_} = lists:split(PathLen, FullPath),
        File = Path ++ atom_to_list(Module) ++ ".beam",
```

```
    ok = file:write_file(File, Binary),
    c:l(Module).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%      ADDING MOVEMENT DETECTION      %%%%%%%%%%%%
%%%%%%%%%%%%         IN NUMERL VERSION           %%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

realtime() ->
    io:format("Start Realtime with 2 seconds between gesture,
during 60 seconds~n",[]),
    realtime:start(loop, 2000, 60000).

realtime(Time, Period) ->
    if Period >= 0 ->
        io:format("Start Realtime with ~p seconds between
gesture, during ~p seconds~n",[Time, Period]);
    true ->
        io:format("Start Realtime with ~p seconds for the
gesture, indefinitely~n",[Time]) % set Period to a negative
number to loop indefinitely
    end,
    realtime:start(loop, Time * 1000, Period * 1000).

realtime_once() ->
    io:format("Start Realtime with 10 seconds for the gesture~n
",[]),
    realtime:start(once, 10000, 0). % Last argument not used

realtime_once(Time) ->
    io:format("Start Realtime with ~p seconds for the gesture~n
",[Time]),
    realtime:start(once, Time * 1000, 0). % Last argument not
used

clear_gesture() ->
    file:write_file("movement_detection/lib/movement_detection
-1.0.0/src/gesture", ""),
    io:format("Clear gesture~n",[]).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%      MODULE CALLBACKS      %%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

start(_Type, _Args) ->
    {ok, Supervisor} = movement_detection_sup:start_link(),
    init_table(),
    case node_type() of
        nav ->              % This will be detected upon
```

```
    calling node_type().
                _ = grisp:add_device(spi2, pmod_nav); % Adds the
    device.
            sonar ->
                _ = grisp:add_device(uart, pmod_maxsonar),
                pmod_maxsonar:set_mode(single);
            order ->
% At start, launch the supervisor. It is independent of hera.
    Starts sendOrder and sets up the I2C communication.
                _ = sendOrder_sup:start_link();
% This supervisor is not supervised, thus, if the node goes down,
    it goes down.

            _ -> % needed when we use make shell
                _ = net_kernel:set_net_ticktime(8),
                lists:foreach(fun net_kernel:connect_node/1,
                    application:get_env(kernel, sync_nodes_optional
    , []))
        end,
        _ = launch(),
        {ok, Supervisor}.


    stop(_State) -> ok.


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%     MODULE INTERNAL FUNCTIONS     %%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    node_type() ->
        Host = lists:nthtail(14, atom_to_list(node())),
% Returns the actual node, so here, movement_detection@nav_1. It
    returns AFTER the 14th element, meaning the @.

        % Detects the nav in the name.
        IsNav = lists:prefix("nav", Host),
        IsSonar = lists:prefix("sonar", Host),
        IsOrder = lists:prefix("orderCrate", Host),
        if
            IsNav -> nav;
            IsSonar -> sonar;
            IsOrder -> order;   % Need to detect if the board is an
     order one.
            true -> undefined
        end.


    launch(nav) ->
% This is the function called by movement_detection:launch() when
    we are working with a nav.
```

```
        % Cn = ets:lookup_element(args, {nav, node()}, 2),
        % Cm = ets:lookup_element(args, {mag, node()}, 2),
        Cn = ets:lookup_element(args, {nav3, node()}, 2),    % Takes
 the calibration data and puts it in a variable.
        %R0 = ets:lookup_element(args, {e11, node()}, 2),
        % {ok,_} = hera:start_measure(nav, Cn),
        % {ok,_} = hera:start_measure(mag, Cm),

        % For debugging purposes.
        %output_log("I'm calling hera:start_measure(nav3, Cn)~n",[]
),

        % Starts a measure process from Hera.
        {ok,_} = hera:start_measure(nav3, Cn),

        % For debugging purposes.
        %output_log("I'm calling hera:start_measure(e11, R0)~n",[])
,

        %{ok,_} = hera:start_measure(e11, R0),
        ok;

 launch(sonar) ->
        Cs = ets:lookup_element(args, {sonar, node()}, 2),
        {ok,_} = hera:start_measure(sonar, Cs),
        %{ok,_} = hera:start_measure(bilateration, undefined),
        %{ok,_} = hera:start_measure(e10, undefined),
        ok;

 launch(_) ->
        ok.


 init_table() ->
        args = ets:new(args, [public, named_table]),
        {ResL,_} = rpc:multicall(nodes(), ets, tab2list, [args]),
        L = lists:filter(fun(Res) ->
            case Res of {badrpc,_} -> false; _ -> true end end,
ResL),
        lists:foreach(fun(Object) -> ets:insert(args, Object) end,
L).


 update_table(Object) ->
        _ = rpc:multicall(ets, insert, [args, Object]),
        ok.
```

F.10 MotorControl.ino

Credits and comments

Code created by the author. It is the code of the low-level controller, in C++.

Code

```
// Final version of velocity controller.
// Inspired from: https://github.com/curiores/ArduinoTutorials

#include <Wire.h>

// ********** <Pin definition> **********
const int dir[] = {9,18};  // Direction pins
const int en[]  = {8,19};  // Enable pins: sets PWM
const int sa[]  = {7,20};  // Encoder pins SA: interrupt required
const int sb[]  = {6,21};  // Encoder pins SB: interrupt required
// ********** </Pin definition> **********

// ********** <Motor values> **********
#define GB_RATIO  53  // Gearbox ratio of the motor. In my case - 53:1.
#define PPR       6   // Pulse Per Rotation for the encoder. In my case, 3x2
↪  since considering RISING & FALLING.
#define NMOTORS   2   // Number of motors to control.

#define TICKSPERTURN  318     // Number of encoder ticks per wheel turn. 6 x
↪  53 (6 ticks per shaft rotation, with the gearbox added to the mix).
#define DISTPERTURN   0.2513  // Distance per wheel turn in [m]: 2xpix0.04
↪  (wheel diameter: 8cm).
#define NUM_TURN_TEST 7.9586  // To accomplish 2 [m], we need this number of
↪  turns due to the wheel diameter.
#define TURN_CRATE    1.40    // Number of wheel turns required to turn the
↪  crate by 90 degree, in either direction, or on itself. Tuned to limit
↪  overshooting (not perfectly 1/4 a circle: 1.78125).
// ********** </Motor values> **********

// ********** <Controller variables> **********
int   pos[NMOTORS]          = {0,0};  // To safely get the value of
↪  posi[NMOTORS] within noInterrupt().
volatile int posi[NMOTORS]  = {0,0};  // Update the encoder count.
int   posPrev[NMOTORS]      = {0,0};  // Previous value of the encoder
↪  count.

long  prevT_V[NMOTORS]    = {0,0};      // Previous time at which a velocity
↪  computation was made.
long  prevT[NMOTORS]      = {0,0};      // Previous time at which a
↪  computation was made.
float eprev[NMOTORS]      = {0.0,0.0};  // Previous error in the controller.
float eintegral[NMOTORS]  = {0.0,0.0};  // Store the evolution of the
↪  integral error.
```

```
float vFilt[NMOTORS]  = {0.0,0.0};  // Filtered velocity.
float vPrev[NMOTORS]  = {0.0,0.0};  // Previous velocity.
float v[NMOTORS]      = {0.0,0.0};  // Computed velocity (live).
float e[NMOTORS]      = {0.0,0.0};  // Current error.
int diff[NMOTORS]     = {0,0};      // Used to store the counter difference.
const long interval   = 18000;      // Used to filter in the controller.

// Control command
float output[NMOTORS] = {0.0,0.0};    // Output of the controller
float target[5]       = {0,0,0,0,0};  // Will be used to get desired motor
↪  behaviour. Intermediary variable to have a smooth profile.
int order[5]          = {0,0,0,0,0};  // For manual testing.

unsigned long previousMillis[NMOTORS] = {0,0}; // Store previous time.
int coeffMotor[NMOTORS]   = {1,-1};            // To deal with motor
↪  direction differences.
unsigned long tempsActuel = 0;                 // Used in manual command.
int previousDir[NMOTORS]  = {0,0};             // Used for hard reset of the
↪  integral term.
// ********** </Controller variables> **********

// ********** <Variables for the control modes> **********
int initialValueTS            = 0; // This is to store the initial counter
↪  value for this test.
int startingTestingVelocity   = 0; // Not the cleanest way: define an int
↪  which allows to know if we just started the mode, since all the rest is
↪  updated each loop.

int initialValue      = 0;
int startingTurning   = 0;

int startingSlowingDown         = 0;
int initialValueOutput[2]       = {0,0};
long lastTimeSlowingDown        = 0;
long lastTimeEvolveVelocity[2]  = {0,0};
int storeTarget[5]              = {0,0,0,0,0}; // Used to store the values
↪  received by I2C. Contains desired final motor behaviour.

int testNewVelocityPhase = 0;   // Allows to deal with slowingDown in the
↪  testingTheNewVelocity protocol.
int available            = 1;   // By default, the controller says that it is
↪  available to receive more commands.
// ********** </Variables for the control modes> **********

// ********** <PID constants> **********
float kp = 8.0;
float ki = 8.0;
// ********** </PID constants> **********

// ********** <Guard-rail GRiSP2 -> Raspberry> **********
int previousCounterValue  = 0;
```

```
int currentCounterValue   = 1;    // Initialise at one, to avoid any problems
↪  at the first loop.
const long oneSecond      = 1000; // Will be compared to millis().
long lastGuardRailTime    = 0;    // Used to test out the condition.
// ********** </Guard-rail GRiSP2 -> Raspberry> **********


void setup() {
  Serial.begin(9600);

  for(int k = 0; k < NMOTORS; k++){
    pinMode(dir[k],OUTPUT);
    pinMode(en[k],OUTPUT);
    pinMode(sa[k],INPUT);
    pinMode(sb[k],INPUT);
    analogWrite(en[k], 0);
    digitalWrite(dir[k], 0);
  }

  attachInterrupt(digitalPinToInterrupt(sa[0]),changeEncoder<0>,CHANGE);  //
  ↪  It does not work to define 2 attachInterrupt function (RISING and
  ↪  FALLING), only the last is executed. Thus: CHANGE.
  attachInterrupt(digitalPinToInterrupt(sa[1]),changeEncoder<1>,CHANGE);

  // Allows me to have the time to correctly setup the rest of the robot.
  delay(1000);

  for(int k = 0; k < NMOTORS; k++){
    prevT[k] = micros();
    prevT_V[k] = micros();
    previousMillis[k] = micros();
    lastTimeEvolveVelocity[k] = micros();
  }
  lastGuardRailTime = millis();

  Wire.begin(0x40);                // Defines the board's slave address.
  Wire.onReceive(receiveMessage); // To receive the command.
  Wire.onRequest(sendMessage);    // To answer when the master wants to
  ↪  read.

  // Has the board started correctly?
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH);
}


void loop(){

  // Is the GRiSP2 board still up and running?
  check_GRiSP_is_connected();

  noInterrupts();
```

```
pos[0] = posi[0];
pos[1] = posi[1];
interrupts();

// ********** <Manual control of the command: debugging tool> **********

/*int velocityReq = 100;
if(tempsActuel < 7000){
  storeTarget[0] = 0;
  target[1] = 1;
  storeTarget[2] = 0;
  target[3] = 0;
  target[4] = 0;
}
else if(tempsActuel >= 7000 && tempsActuel < 15000){
  storeTarget[0] = velocityReq;
  target[1] = 1;
  storeTarget[2] = velocityReq;
  target[3] = 0;
  target[4] = 0;
}
else{
  storeTarget[0] = 0;
  target[1] = 1;
  storeTarget[2] = 0;
  target[3] = 0;
  target[4] = 2;
}
tempsActuel = millis();*/
// ********** </Manual control of the command: debugging tool> **********

// The target is acquired via I2C. See function receiveEvent() and in
↪  setup().

// Compute velocity & control the motors. Depending on the fifth argument
↪  of target, the desired behaviour varies.
if(target[4] == 0){ // Comparing a float with integer is internally taken
↪  to be comparison between two floats, so it's OK.
  // In the normal behaviour, simply apply the commands. Use evolveVelocity
  ↪  for smooth profile.
  evolveVelocity<0>();
  evolveVelocity<1>();
} else if (target[4] == 1){
  // Test the newly applied velocity.
  testingTheNewVelocity();
} else if(target[4] == 2){
  // This is called whenever we want to stop.
  slowingDownToZero();
} else if(target[4] == 3){
  // Control mode which will make the crate spin on itself.
  turning();
} else {
```

```
      Serial.println("Invalid order for the crate-on-wheels.");
      for(int k = 0; k < NMOTORS; k++){
        setMotor(0, 1-k, dir[k], en[k]);
      }
    }

    // ********** <Printing block: debugging tool> **********
    /*Serial.print(tempsActuel);
    Serial.print(" ");
    Serial.print(output[0]);
    Serial.print(" ");
    Serial.print(output[1]);

    Serial.print(uFilt[0]);
    Serial.print(",");

    Serial.print(storeTarget[1]);
    Serial.print(" ");
    Serial.print(storeTarget[2]);
    Serial.print(" ");
    Serial.print(storeTarget[3]);
    Serial.print(" ");
    Serial.print(storeTarget[4]);
    Serial.println();*/
    // ********** </Printing block: debugging tool> **********
    //delay(10); // To have data only every 10 ms. Avoids overloading the
    ↪  textfiles.
}


// ********************************************* //
// ************ CHECK GRISP IS ALIVE ************ //
// ********************************************* //

void check_GRiSP_is_connected(){
  long currentTime = millis();
  if(currentTime - lastGuardRailTime > oneSecond){   // Check every two
  ↪  seconds that the GRiSP is pinging correctly. One ping should arrive
  ↪  every second.
    if(currentCounterValue == previousCounterValue){  // The counter did not
    ↪  evolve. There is a problem. Send the target to 0 and call the
    ↪  slowingDown function.
      for(int k = 0; k < 5; k++){
        storeTarget[k] = 0.0; // We set everything to 0. The crate will not
        ↪  start again after the full stop.
      }
      slowingDownToZero();
    }
    lastGuardRailTime = currentTime;
    previousCounterValue = currentCounterValue;
  }
}
```

```
// ********************************************** //
// ************* I2C COMMUNICATION ************** //
// ********************************************** //

void receiveMessage(int howMany)
{
  if(howMany == 1){
    currentCounterValue += Wire.read(); // Increment the counter by the value
    ↪  sent by the GRiSP2, which is 1.
  } else {
    for(int i=0; i < howMany; i++){
      storeTarget[i] = Wire.read();
      //Serial.println(storeTarget[i]);
    }
    //Serial.println();
    target[1] = storeTarget[1];  // Direction 1
    target[3] = storeTarget[3];  // Direction 2
    target[4] = storeTarget[4];  // command_index. storeTarget is only used
    ↪  for storage of velocity, do not modify its value.
  }
}

void sendMessage()
{
  byte message = byte(available);
  Wire.write(message);
}


// ********************************************** //
// ************** CONTROL MODES ***************** //
// ********************************************** //

// This function is used to have a smooth velocity profile when used
↪  continuously. To slow down to 0, refer preferentially to the appropriate
↪  function.
template <int j>
void evolveVelocity(){
  int index = 0;
  if(j == 1){
    index = 2;
  }
  long currentTime = micros();
  if(currentTime - lastTimeEvolveVelocity[j] >= interval){ // Re-use the
  ↪  interval of 18 ms. Totally arbitrary, could be more, but less would
  ↪  lead to a rapid evolution that may be hard to follow.
    lastTimeEvolveVelocity[j] = currentTime;
    if(target[index] > storeTarget[index]){ // Comparison between a float and
    ↪  an int: it will work.
      target[index] = target[index] - 0.5;
```

```
    } else if (target[index] < storeTarget[index]){
      target[index] = target[index] + 0.5;
    } else {
      target[index] = target[index];
    }
  }
  computeVelocityAndController<j>();
}

// To test the new velocity : go forward for 2 [m], then backward over the
↪  same distance. Then, stop. We will go further than 2 [m] since we slow
↪  down nicely.
// This is quite an ugly implementation. Some kind of FSM would have been
↪  better suited for this task. But it works.
void testingTheNewVelocity(){
  if(!startingTestingVelocity){
    available = 0; // We say that we are no longer available.
    startingTestingVelocity = 1;
    initialValueTS = fabs(pos[0]);
  } else if(startingTestingVelocity){
    // NUM_TURN_TEST [turns] comes from the fact that we want 2 [m] and that
    ↪  1 turn = 0.2513 [m].
    // While not 2 [m] done and going forward, keep going.
    if(fabs(initialValueTS - fabs(pos[0])) < NUM_TURN_TEST * TICKSPERTURN &&
    ↪  target[1] == 1){ // Full of fabs() to ensure a count going up, no
    ↪  matter the case.
      evolveVelocity<0>();
      evolveVelocity<1>();
    } else if(fabs(initialValueTS - fabs(pos[0])) >= NUM_TURN_TEST *
    ↪  TICKSPERTURN && target[1] == 1){ // We have gone over 2 [m]. Now,
    ↪  come back.
      testNewVelocityPhase = 1; // First step done, now slow-down smoothly.
      slowingDownToZero();
    } else if(fabs(initialValueTS - fabs(pos[0])) < NUM_TURN_TEST *
    ↪  TICKSPERTURN && target[1] == 0){  // Back for 2 [m]
      evolveVelocity<0>();
      evolveVelocity<1>();
    } else {
      Serial.println("Testing the new velocity: test done!");
      if(vFilt[0] != 0.0){
        testNewVelocityPhase = 2; // Second step done, now slow-down
        ↪  smoothly.
        slowingDownToZero();
      } else {  // When we return in this loop after slowing down, we enter
      ↪  this and finish the protocol.
        startingTestingVelocity = 0;
        storeTarget[0] = 0; // Required to avoid going backward when the
        ↪  protocol has finished.
        storeTarget[2] = 0;
        target[4] = 0;      // Back to 'normal' mode.
        available = 1;
      }
```

```
    }
  } else {
    Serial.println("Invalid value for startingTestingVelocity.");
  }
}

void slowingDownToZero(){
  if(!startingSlowingDown){
    available = 0; // We say that we are no longer available.
    startingSlowingDown = 1;
    if(target[0] > target[2]){
      target[0] = target[2];  // In case we were turning, we now say that the
      ↪  target is the same velocity for both wheels, but the lowest one, in
      ↪  order to already slow down.
    } else {                  // At most, step of 30 RPM in the target, will
    ↪  be barely noticeable by the human eye.
      target[2] = target[0];  // This will have no impact if the targets were
      ↪  the same.
    }
    if(vFilt[0] < 0){ // Correct the order sent by stopCrate (by default,
    ↪  forward directions) if we were going backward.
      target[1] = 0;
      target[3] = 1;
    }
    target[4] = 2; // This is used when calling this function from another
    ↪  protocol.
    lastTimeSlowingDown = micros();
    // Get the controller started on the new command.
    computeVelocityAndController<0>();
    computeVelocityAndController<1>();
  } else if(startingSlowingDown){
    if(fabs(vFilt[0]) > 5.0){ // Arbitrary boundary on an arbitrary motor,
    ↪  the goal just being to have a safe-band around 0 to be sure to stop
    ↪  fully.
      long currentTime = micros();
      if(currentTime - lastTimeSlowingDown >= interval){ // Re-use the
      ↪  interval of 18 ms. Arbitrary.
        lastTimeSlowingDown = currentTime;
        for(int k = 0; k < 2; k++){
          target[2*k] = target[2*k] - 0.5;  // Slowly decrease the desired
          ↪  velocity. 0.5 has been found empirically, can be tuned to
          ↪  obtained desired results.
        }
      }
      computeVelocityAndController<0>();
      computeVelocityAndController<1>();
    } else {
      Serial.println("Slowing down: done!");
      // Force 0 output and reset vFilt. We are at full stop.
      for(int k = 0; k < 2; k++){
        target[2*k]   = 0;
        vFilt[k]      = 0.0;  // Say we are at 0 velocity.
```

```
        output[k]     = 0.0;  // Force a 0 output.
        eintegral[k]  = 0.0;
      }
      if(testNewVelocityPhase == 1){
        target[1] = 0;  // Change direction.
        target[3] = 1;  // Change direction.
        target[4] = 1;  // If we used "slowingDown" in the
        ↪  testingTheNewVelocity, we get ready to go back to that mode.
        initialValueTS = fabs(pos[0]); // To exit the other conditions, reset
        ↪  the value here.
      } else if (testNewVelocityPhase == 2) { // Don't change the targets.
      ↪  The protocol testingTheNewVelocity is done.
        target[4] = 1;
        testNewVelocityPhase = 0; // Reset this.
      } else {
        target[4] = 0;
        available = 1;  // We are available again for new commands. We put it
        ↪  here so that the testingTheNewVelocity protocol can keep going.
      }
      startingSlowingDown = 0;
    }
  } else {
    Serial.println("Invalid value for startingSlowingDown.");
  }
}

// To turn, whether it be by 90° or to spin on itself.
void turning(){
  if(!startingTurning){
    available = 0;
    startingTurning = 1;
    initialValue = fabs(pos[0]); // This checks which motor we need to take
    ↪  into account. When spinning on itself, this will lead to motor 1.
    // It doesn't really matter which motor we use to check. Once one of the
    ↪  wheel did its job, assume the crate did a full rotation.
  } else if(startingTurning){
    if(fabs(initialValue - fabs(pos[0])) < TURN_CRATE * TICKSPERTURN){
      evolveVelocity<0>();
      evolveVelocity<1>();
    } else {
      Serial.println("Turning: done!");
      for(int k = 0; k < 2; k++){
        storeTarget[2*k] = 0;
        target[2*k] = 0;
        vFilt[k]    = 0.0;  // Say we are at 0 velocity.
        output[k]   = 0.0;  // Force a 0 output.
      }
      target[4] = 0;
      startingTurning = 0;
      available = 1;
    }
  } else {
```

```
    Serial.println("Invalid value for startingTurning.");
  }
}


// ********************************************** //
// ************* VELOCITY & CONTROL ************** //
// ********************************************** //

template <int k> // Can use prevT[k], posPrev[k], target[k], pos[k]. No need
↪  of an index, thanks to template.
void computeVelocityAndController(){
  unsigned long currentMillis = micros();
  float deltaT  = ((float) (currentMillis - prevT[k]))/1.0e6;
  prevT[k]      = currentMillis;

  if (currentMillis - previousMillis[k] >= interval){
    previousMillis[k] = currentMillis;
    long currT_V      = micros();
    float deltaT_V    = ((float) (currT_V-prevT_V[k]))/1.0e6;

    diff[k]           = coeffMotor[k]*(posPrev[k] - pos[k]); // The
    ↪  coefficient is there to ensure that both motors have positive
    ↪  velocity when rotating in different directions.
    v[k]              = (diff[k]*60.0)/(PPR*GB_RATIO*deltaT_V); // The *60.0
    ↪  is here to get the velocity in RPM and not in RPS.

    posPrev[k]  = pos[k];
    prevT_V[k]  = currT_V;

    vFilt[k]    = 0.894*vFilt[k] + 0.053*v[k] + 0.053*vPrev[k]; // Low-pass
    ↪  filter at 1 Hz, sampling freq of 55.56 Hz -> 56 Hz.
    vPrev[k]    = v[k];
  }


  // ********************************************** //
  // To deal with the way the controller is being told the velocity it needs
  ↪  to reach. Instead of sending negative values, deal automatically with
  ↪  it here.
  float coeffTargetVel = 1.0;
  if(k == 0){
    if(target[2*k+1] == 0){ // If MOTOR1 : POSITIVE VELOCITY when DIR1 and
    ↪  NEGATIVE VELOCITY when DIR0.
      coeffTargetVel = -1.0;
    }
  } else {
    if(target[2*k+1] == 1){ // If MOTOR2 : POSITIVE VELOCITY when DIR0 and
    ↪  NEGATIVE VELOCITY when DIR1.
      coeffTargetVel = -1.0;
    }
  }
```

```
      // ****************************************** //

      float targetVel = coeffTargetVel*target[2*k];
      e[k] = targetVel - vFilt[k];

      // Main control loop.
      if(fabs(targetVel) > 10.0){   // This condition, be it kind off useless,
      ↪  helps the controller when the target is 0, so keep it.

        // ****************************************** //
        // Hard reset of the integral term when switching direction.
        if(previousDir[k] != target[2*k+1]){
          eintegral[k] = 0.0;
        }
        // ****************************************** //

        eintegral[k] += e[k]*deltaT;

        float limit = 70.0;
        float integralTerm = ki*eintegral[k];

        // Limiter: limit the integral term.
        if(fabs(integralTerm) > ki*limit){ // So that it automatically adapts
        ↪  itself when need to change boundary (Ki value, typically).
          if(integralTerm < 0){
            integralTerm = -ki*limit;
          }
          else {
            integralTerm = ki*limit;
          }
        }

        // Compute the necessary command to reach the desired velocity;
        float u = kp*e[k] + integralTerm;

        output[k] = fabs(u);
        if(output[k] > 255){
          output[k] = 255;
        }
        int direction = target[2*k+1]; // target is already in int.

// In theory, this block is actually useless. However, since it helped me with
↪  several situations without impacting the others, keep it. But when rolling,
↪  this is useless.
// It truly is for debugging purposes, when the crate-on-wheels is not on the
↪  floor and that we have to deal with overshoots. Doesn't work with overshoots
↪  in negative direction, however.

        if(u < 0 && targetVel > 0){
          direction = 1 - direction;
```

```
// Change direction if need be, without affecting the initial command. This is
↪  necessary to deal with potential overshoots.
// Added a second condition because when we want to go backward, it is NORMAL
↪  that the error can be negative. But now, risk: overshoot in the negative,
↪  doesn't catch back.
        }

        // Set command
        output[k] = (int) output[k]; // Casting a float to an int results in a
        ↪  rounding to the lower unit: e.g.: 200.9 -> 200.

        // Format: setMotor(direction, targetVelocity, directionPin, enablePin);
        setMotor(direction, output[k], dir[k], en[k]);
      }
      else {
        setMotor(target[2*k+1], 0, dir[k], en[k]);
      }

      // End the loop, update the error.
      previousDir[k] = target[2*k+1];
      eprev[k] = e[k];
    }


    // ********************************************** //
    // ************** ENCODERS READING *************** //
    // ********************************************** //

    template <int i>
    void changeEncoder(){
      int a = digitalRead(sa[i]);
      if(a == HIGH){
        encoder_R(i);
      }
      else encoder_F(i);
    }

    // On rising edge.
    void encoder_R(int index){
      int b = digitalRead(sb[index]);
      if(b > 0){
        posi[index]--;
      }
      else{
        posi[index]++;
      }
    }

    // On falling edge.
    void encoder_F(int index){
      int b = digitalRead(sb[index]);
      if(b > 0){
```

```
    posi[index]++;
  }
  else{
    posi[index]--;
  }
}


// ********************************************** //
// ************* SET MOTOR FUNCTION ************* //
// ********************************************** //

void setMotor(int dir, int pwm_value, int PIN_DIR, int PIN_EN){
  analogWrite(PIN_EN, 0);
  digitalWrite(PIN_DIR, dir);
  analogWrite(PIN_EN, pwm_value);
}
```

F.11 gesture

Credits and comments

This file was created by Sébastien Gios, but completely reworked by the author. It gives the list of known gestures that will be used for pattern matching.

Code

```
[stopCrate,x,o]
[stopCrate,y,o]
[stopCrate,z,nn]
[stopCrate,x,o]
[stopCrate,y,pp,o]
[stopCrate,z,o,nn]
[stopCrate,x,o]
[stopCrate,y,pp,p,o]
[stopCrate,z,o,n,nn]
[stopCrate,x,o]
[stopCrate,y,nn,o]
[stopCrate,z,o,nn]
[stopCrate,x,o]
[stopCrate,y,nn,n,o]
[stopCrate,z,o,n,nn]
[stopCrate,x,pp,o]
[stopCrate,y,o]
[stopCrate,z,o,nn]
[stopCrate,x,pp,p,o]
[stopCrate,y,o]
[stopCrate,z,o,n,nn]
[stopCrate,x,nn,o]
[stopCrate,y,o]
[stopCrate,z,o,nn]
[stopCrate,x,nn,n,o]
[stopCrate,y,o]
[stopCrate,z,o,n,nn]
[forward,x,o]
[forward,y,o,pp]
[forward,z,nn,o]
[forward,x,o]
[forward,y,o,p,pp]
[forward,z,nn,n,o]
[forward,x,pp,o]
[forward,y,o,pp]
[forward,z,o]
[forward,x,pp,p,o]
[forward,y,o,p,pp]
[forward,z,o]
[forward,x,nn,o]
[forward,y,o,pp]
[forward,z,o]
[forward,x,nn,n,o]
```

```
[forward,y,o,p,pp]
[forward,z,o]
[forward,x,o]
[forward,y,pp]
[forward,z,o]
[backward,x,o]
[backward,y,o,nn]
[backward,z,nn,o]
[backward,x,o]
[backward,y,o,n,nn]
[backward,z,nn,n,o]
[backward,x,nn,o]
[backward,y,o,nn]
[backward,z,o]
[backward,x,nn,n,o]
[backward,y,o,n,nn]
[backward,z,o]
[backward,x,pp,o]
[backward,y,o,nn]
[backward,z,o]
[backward,x,pp,p,o]
[backward,y,o,n,nn]
[backward,z,o]
[backward,x,o]
[backward,y,nn]
[backward,z,o]
[forwardTurnRight,x,pp]
[forwardTurnRight,y,o]
[forwardTurnRight,z,o]
[forwardTurnRight,x,o,pp]
[forwardTurnRight,y,pp,o]
[forwardTurnRight,z,o]
[forwardTurnRight,x,o,p,pp]
[forwardTurnRight,y,pp,p,o]
[forwardTurnRight,z,o]
[forwardTurnLeft,x,nn]
[forwardTurnLeft,y,o]
[forwardTurnLeft,z,o]
[forwardTurnLeft,x,o,nn]
[forwardTurnLeft,y,pp,o]
[forwardTurnLeft,z,o]
[forwardTurnLeft,x,o,n,nn]
[forwardTurnLeft,y,pp,p,o]
[forwardTurnLeft,z,o]
[backwardTurnRight,x,o,nn]
[backwardTurnRight,y,nn,o]
[backwardTurnRight,z,o]
[backwardTurnRight,x,o,n,nn]
[backwardTurnRight,y,nn,n,o]
[backwardTurnRight,z,o]
[backwardTurnleft,x,o,pp]
[backwardTurnleft,y,nn,o]
```

```
[backwardTurnleft,z,o]
[backwardTurnleft,x,o,p,pp]
[backwardTurnleft,y,nn,n,o]
[backwardTurnleft,z,o]
[forwardTurnLeft,x,pp,nn]
[forwardTurnLeft,y,o]
[forwardTurnLeft,z,o]
[forwardTurnLeft,x,pp,p,o,n,nn]
[forwardTurnLeft,y,o,p,pp,p,o]
[forwardTurnLeft,z,o]
[forwardTurnRight,x,nn,pp]
[forwardTurnRight,y,o]
[forwardTurnRight,z,o]
[forwardTurnRight,x,nn,n,o,p,pp]
[forwardTurnRight,y,o,p,pp,p,o]
[forwardTurnRight,z,o]
[backwardTurnleft,x,nn,pp]
[backwardTurnleft,y,o]
[backwardTurnleft,z,o]
[backwardTurnleft,x,nn,n,o,p,pp]
[backwardTurnleft,y,o,n,nn,n,o]
[backwardTurnleft,z,o]
[backwardTurnRight,x,pp,nn]
[backwardTurnRight,y,o]
[backwardTurnRight,z,o]
[backwardTurnRight,x,pp,p,o,n,nn]
[backwardTurnRight,y,o,n,nn,n,o]
[backwardTurnRight,z,o]
[turnAround,x,o,p,pp,p,o]
[turnAround,y,o]
[turnAround,z,nn,n,o,p,pp]
[changeVelocity,x,o,p,pp,p,o,n,nn,n,o]
[changeVelocity,y,o]
[changeVelocity,z,nn,n,o,n,nn,n,o,n,nn]
[accelerate,x,o,p,pp,p,o]
[accelerate,y,o]
[accelerate,z,nn,n,o,n,nn]
[decelerate,x,o,n,nn,n,o]
[decelerate,y,o]
[decelerate,z,nn,n,o,n,nn]
[exitChangeVelocity,x,o,n,nn,n,o,p,pp,p,o]
[exitChangeVelocity,y,o]
[exitChangeVelocity,z,nn,n,o,n,nn,n,o,n,nn]
[testingVelocity,x,o]
[testingVelocity,y,o,p,pp,p,o]
[testingVelocity,z,nn,n,o,n,nn]
[templateMove,x,o,n,nn,n,o,n,nn,n,o]
[templateMove,y,o]
[templateMove,z,nn,n,o,p,pp,p,o,n,nn]
```

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