"Gesture Recognition by Pattern Matching using Sensor Fusion on an Internet of Things device"

Gios, Sébastien

ABSTRACT

Internet of Things devices are small computers with embedded hardware and minimal software. They are increasingly present in everyday life (home automation, household) and can be equipped with many sensors to collect data or have a small motor to perform a physical movement. But as they are present in the human environment, interaction with them should be simple and reliable. Our goal is to recognize a human gesture, using only an IoT device attached to a wrist. No cloud or other dependencies because of the many disadvantages. This gesture can then be used to control another device. The IoT we will use is a GRiSP, it is a prototype board for IoT that supports Erlang and Elixir on bare hardware. It has sockets for attaching up to 5 PMOD sensors and actuators. PMOD devices include navigation, barometer, humidity, range finding, motor control, etc. The approach is to program an IoT capable of learning and recognizing gestures. A gesture is a movement of a human arm in the environment. We will combine data from sensors passed through a filter. Then recognize the different gestures. It is the fusion of 3 sensors from the PMOD-NAV: an accelerometer, a gyroscope, and a magnetometer. Sent to a Kalman Filter, it estimates the state of a system from measurements. But it is robust against incomplete data and noise. Return an acceleration in real-time, and sent it to the machine learning algorithm to recognize the gesture. A gesture is recognized with a classifier by pattern matching. Each gesture is a different category. There is an initial list of learned gestures, and more can be ...

CITE THIS VERSION

UCLouvain

epl

École polytechnique de Louvain

Gesture Recognition by Pattern Matching using Sensor Fusion on an Internet of Things device

Author: **Sébastien Gios**
Supervisor: **Peter Van Roy**
Readers: **Peer Stritzinger, Ramin Sadre**
Academic year 2022–2023
Master [120] in Computer Science

Abstract

Internet of Things devices are small computers with embedded hardware and minimal software. They are increasingly present in everyday life (home automation, household) and can be equipped with many sensors to collect data or have a small motor to perform a physical movement. But as they are present in the human environment, interaction with them should be simple and reliable. Our goal is to recognize a human gesture, using only an IoT device attached to a wrist. No cloud or other dependencies because of the many disadvantages. This gesture can then be used to control another device.

The IoT we will use is a GRiSP, it is a prototype board for IoT that supports Erlang and Elixir on bare hardware. It has sockets for attaching up to 5 PMOD sensors and actuators. PMOD devices include navigation, barometer, humidity, range finding, motor control, etc.

The approach is to program an IoT capable of learning and recognizing gestures. A gesture is a movement of a human arm in the environment. We will combine data from sensors passed through a filter. Then recognize the different gestures. It is the fusion of 3 sensors from the PMOD-NAV: an accelerometer, a gyroscope, and a magnetometer. Sent to a Kalman Filter, it estimates the state of a system from measurements. But it is robust against incomplete data and noise. Return an acceleration in real-time, and sent it to the machine learning algorithm to recognize the gesture.

A gesture is recognized with a classifier by pattern matching. Each gesture is a different category. There is an initial list of learned gestures, and more can be added to diversify possibilities. Several gestures can be detected in succession, but the user must stop moving between 2 gestures.

Contents

Chapter 1

Introduction

This introductory chapter starts with detail on the goal and approach, followed by an explanation of the term *Internet of Things*. Then a section on a concrete use case for the gesture recognition algorithm, and a section on the contribution developed during this thesis. Finally, there is a roadmap detailing the structure of the other chapters.

1.1 Goal & approach

In this Master's Thesis, we will develop a program to detect a gesture in the environment. This movement will not be detected from an image but from a combination of vectors. Those vectors will come from measurements of multiple sensors that will be combined through a Kalman Filter. We will use an IoT device to use the sensor, a "GRiSP". It's a small prototype board that can have multiple sensors attached to it. In particular, the PMOD-NAV can provide the raw data from an accelerometer, a gyroscope, and a magnetometer. This device is attached to a human wrist, analyzing the movement of the arm. Those measures will be passed to a Kalman Filter to have usable information. The output vectors will give us a movement, and this movement will be passed through a classifier so it will learn the different possible movements and be able to recognize and detect which gesture was done. After getting the first version of a classifier adapted for our problem, we are going to improve it and add new features, until reaching a satisfying result.

We are developing this algorithm because, in the future, we want to control another device or the environment. The idea is to use only gestures to control something else because it is way more simple to develop and use movement than voice or direct input. Voice needs more complex algorithms than movement and is

also restricted by the language barrier. An algorithm that works for the English language will not understand a sentence in French. Direct input can also be complicated, you can be a person with limited mobility, the device needs constant input (for example to follow you), the device can be very high, like a drone or a window, etc...

An important part of edge computing is communication, so using algorithms to make devices communicate with each other is perfectly adapted. We also use a GRiSP that works with native Erlang; a language widely used for its capability in communication even with failure, (with the paradigm "let it fail"). So using Erlang is also well adapted because it is good for communication and robust in case of failure. The Hera framework we will use and explain more in-depth in its section below can for example still work without problem with only one device when ideally used with five. That's why we can use only the GRiSP with the sensor PMOD-NAV and still use most of the features of Hera without issues.

More than using only gestures, we do not want to rely on a side device to have the algorithm working, so everything needs to be done on the GRiSP. We don't want a dependency on the cloud either because of the many disadvantages of using something else even if it's the cloud. For example in the future, if we have our algorithm on a connected watch, it will be able to work without anything else. Because if we rely on something else we would need constant communication between the device that obtains the sensor data and the device that computes the classification. If this communication is not good enough, there will be a loss of data or result quality. And if it depends on the cloud you will need a constant connection to the internet which can cost money, and may not be reliable if a user moves a lot. Because of all these disadvantages, we are pushed to depend on nothing else.

A movement should be simple to make, to be performed by anyone, and to make many of them without the risk of overlapping movement. For example a circle with your arms, or going from left to right. There will be an initial list of movements so the algorithm can train and understand them. If the algorithm is workings well, we will add the possibility to add movement, and again if it's working well add them more easily in real-time just by using the GRiSP, with for example a movement to start "recording" a gesture.

If we obtain a good algorithm that detects close movements, for example, 2 circles with your arms with different radii. The next improvement is to take into account the direction of the gesture. With this improvement, we will be able to

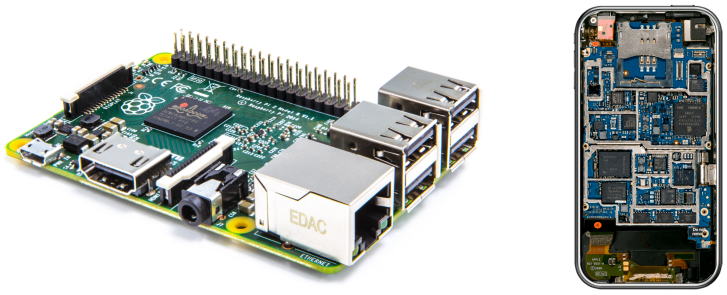
point to another connected device with a gesture and then perform a command with a gesture. After that, if it is working well we would be to add be able to perform more complex gestures. Indeed with the use of the direction, 2 gestures that look the same but aren't done in the same direction are now totally different. So the direction allows a new range of complexity for the movement that can be performed.

1.2 Internet of Things

"*Internet of Things*" or IoT in short, are small devices used for edge computing. They are usually adapted to a particular function but can also be more general to be used as "microcomputers"[1]. In the second category, we have for example the famous Raspberry Pi. They have embedded hardware and minimal software for their purpose, they are usually connected on a network and monitors via a server or a computer. They can receive information or generate their own via captors, compute results, and send it on the network. They are not always inert and can also be equipped with motors that allow them to move the physical world.

They are more and more present around us, and they will continue to grow exponentially in the coming years. We can see them in our houses, in the street, around our wrists, or in our pockets. So as they are in a human environment, they should be reliable, and interaction with them should be as simple as possible.

A complex IoT device we all have is our mobile phones, but during this thesis, a simple IoT with minimal software and hardware, a GRiSP, will be used. It will be talked about in more detail in the background chapter.



(a) Raspberry Pi

(b) Opened Mobile Phone

Figure 1.1: Simple and Complex IoT aren't that much different

1.3 Use case

What we're going to develop will typically be to help people with reduced mobility to interact easily with their environment via simple gestures. A concrete case of what it will be like in the future is:

We have a room that has a very high "IoT connected window", an old woman inside wants to open this window but can't reach it on her own. She has however an IoT device around her wrist with our gesture recognition algorithm. She will point in the direction of the window, and this will be detected as "a gesture pointing to the connected window", so an instruction to listen to what will be the next one is sent to the window. Then the woman performs another gesture that is recognized by the algorithm as "open the window", this new instruction is sent to the window that is listening. The window receives the instruction and remotely opens without having been directly touched.

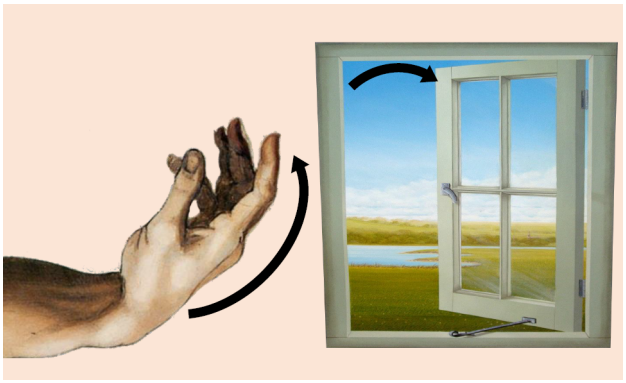


Figure 1.2: A hand performing a gesture to remotely open a window

1.4 Contribution

We have developed a gesture recognition algorithm that classifies a gesture by pattern matching. This algorithm is running on an IoT device, a GRiSP. To perform a gesture, a user must hold the device in his hand or attach it to his wrist, a gesture is thus a movement or a rotation of a human arm. The input of the algorithm is obtained from the data gathered on 3 sensors on a PMOD-NAV attached to the IoT device, an accelerometer, a gyroscope, and a magnetometer. These raw data are fused and passed through the Kalman filter. It reduces the noise and is fault tolerant by giving as output a combination of a prediction and the last raw data. The output of the Kalman filter is a cleaned list of acceleration.

Those accelerations are relative to the position of the 3 axes (X, Y, and Z) of the GRiSP during the calibration. These 3 axes can be precisely viewed as they are written on the PMOD-NAV sensor attached to the GRiSP. If a user performs a gesture, then rotates the GRiSP around an axis and performs the same gesture again, it will be recognized as a different gesture if he didn't do a new calibration.

Any rotation around one or several axes can be recognized as a gesture. It is possible to exceed a rotation of 360° and do multiple loops to create a more complex gesture. For a non-rotating movement, the GRiSP must be moved fast enough to be detected.

The process in an image: (1.3)

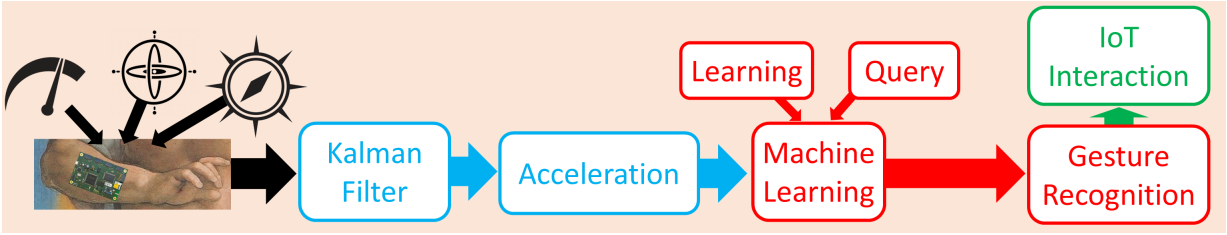


Figure 1.3: From the data collection to the IoT interaction

- The first part in **black** is the 3 physical sensors of the PMOD-NAV, an accelerometer, a gyroscope, and a magnetometer.
- The **blue** part is the Hera sensor fusion created previously by other students. It takes as input the raw data of the sensors and fuse them through a Kalman filter to give a clean acceleration as output.
- The **red** section is our contribution, we have developed a gesture recognition algorithm by pattern matching. The input is the cleaned data that is coming out of the Kalman filter, the machine learning algorithm has learned a list of gestures and new ones can also be learned. When a user will perform a gesture it will be classified according to the learned gestures.
- the **green** part is a future work, where each gesture performed on the GRiSP, will be an instruction sent to another IoT device.

1.5 Roadmap

The second chapter on the **background**, explains the software and hardware already developed by other people used in this thesis. So a talk on Erlang then

details about the GRiSP device, followed by a section on some gesture recognition algorithms talked about in papers. There is also a section on the sensor fusion created in the previous thesis.

The third chapter is on all the **preliminary works** done, with an explanation of the adaptation of the software from the first version of the GRiSP to the new one. Then there is a section on the content of the sensor fusion output, followed by several sections on algorithm attempts before finding the one to keep.

The fourth chapter is on the **main contribution**, so the algorithm created during this thesis, from its first version to the most improved one, with each time explanation of improvements, implementations, and experiments.

There is a last chapter on the **conclusion** with a summary of what was created and the possible future works.

There are also **appendixes**: an explanation of how to use the gesture recognition algorithm, an indication of how to improve the gesture recognition algorithm, and all the source code created or modified from the previous thesis.

Chapter 2

Background

This chapter explains in detail all the software and hardware used for the thesis that was developed by other people. There is a section on the Erlang language, a section on the GRiSP hardware, a section on papers on gesture recognition, and a section on the Hera software.

2.1 Erlang

The Erlang language[2] is used during this thesis because GRiSP runs with this language. This section explains what is this language, its specificity, and its differences from other languages.

Figure 2.1: Erlang Llgo

Erlang[3] is a programming language highly efficient for concurrent programming, thus very well adapted for distributed systems, and fault-tolerant. The concept of "let it crash" works perfectly with Erlang. It was developed in 1986 by Joe Armstrong, Robert Virding, and Mike Williams who worked for Ericsson. They developed it to use it for telecommunications systems because they were systems

that needed high availability and fault tolerance.

It was created because popular languages at that time weren't very well suited for those systems. Ericsson that was using a telecommunication system needed something very good, so they aim to develop their language. Erlang is based on many other languages but the biggest inspiration is from Prolog. The language evolve over the years, and the Erlang/OTP (for Open Telecom Platform) was created around 1990. It is a collection of libraries, frameworks, and tools that extend the basic functionality of Erlang to make it easier for developers to use it. Now Erlang/OTP is still evolving with its recent version 25.3.1 as it is from 1998 a free and open-source software.

Erlang is efficient for concurrent programming thanks to its approaches. It doesn't use thread like many other languages but processes that are very light, so many of them can be created or destroyed quickly. Each process can communicate with the others, by passing messages. One process can send a message to another one, that will listen if they are any incoming messages, it can also do some pattern matching to listen to only certain types of messages. That is why Erlang, is a good choice when building a distributed system.

Because processes are easy to make or destroy, Erlang is really good for fault tolerance. A process can crash or stop without impacting the other one because they are isolated from each other. So they don't share resources, but can still transmit a message to another process if they know its name. They can be monitored and restarted if a failure is detected, creating fault-tolerant systems, with minimal downtime is easily achievable with Erlang.

Erlang has also some big differences with other popular modern languages like C, Java, and Python. All of its variables are immutable. That means once a value has been set to a variable it can not be changed later. In concurrent programming, it is a guarantee that the variable you are using will not be affected by other processes, or changed during the execution.

Erlang also uses pattern matching for its programming. So you can change the behavior of what will happen based on how is a variable. You can also have a function with the same name but not the same arguments, or not the same quantity of them, and they will be 2 different functions.

And speaking of function, Erlang uses functional programming, which is a paradigm where functions are treated like any other type of data and can be easily

used as input or output of another function. That means Erlang doesn't have complex programming structure like for or while loops because it can be done easily with a combination of pattern matching and recursion.

2.2 GRiSP

The GRiSP[4] is, according to its creator: "A combination of embedded hardware and software aiming at being the best prototyping solution for Erlang and Elixir developers"[5].

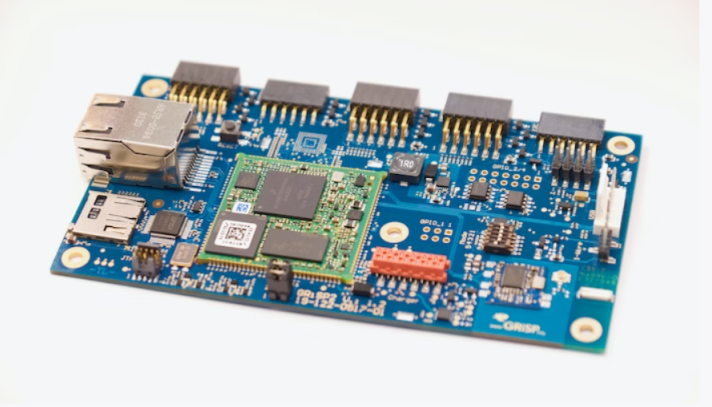


Figure 2.2: Prototype GRiSP2 board

It is an Internet of Things device, a small computer adapted for edge computing. Generally, an IoT will be specialized for a specific function and have minimal components to perform it. The GRiSP is still only a prototype and therefore for the moment has many more elements than we need to use. It supports native Erlang and Elixir, with the processor and the RAM running in Erlang. To avoid having too many components, it is possible to connect more elements thanks to specialized ports. They are called **PMOD**, the available for the moment are:

- PMOD-NAV, is equipped with a 3-axis accelerometer, a gyroscope, and a magnetometer. It has also a barometer, so with all of this it, you can have acceleration rotation, and orientation. It also detects pressure.
- PMOD GYRO is just equipped with a 3-axis gyroscope, but it is enough to get the angular velocity. There are also other smaller features, like interrupt pins and signal filtering.

- PMOD HYGRO is a sensor adapted to track the humidity and the temperature, thus it is equipped with sensors for this 2 information. It has also a heating element to remove condensation accumulated after being in a humid environment.
- PMOD TPH is equipped with 6 external pin headers, so you can test if the signal passes through them.
- PMOD MAXSONAR is equipped with sonar so you can detect the nearest obstacle. It can detect object up to 6m with a precision of around 2.5cm. It has also the possibility to run continuously to track a moving object.
- PMOD MTDS is equipped with a small touch screen that can display a small interface and take input by touching. It is usable thanks to its library that allow you to easily code what you want on it.
- PMOD RF2 allows you to transmit information, through RF communication. It supports some popular network protocols like ZigBee or MiWi.
- PMOD HB5 is equipped with the right circuit to allow you to control a small motor. It was made to work with the Diligent motor, but you can also your motor.
- PMOD TPH2 is like the PMOD TPH, but this time equipped with 12 external pin headers.

More information can be found on the online GRiSP shop[6].

There are 2 versions of the GRiSP, and the sensor fusion used to obtain our sensor data was created on the first version of the GRiSP. But this GRiSP was too slow to try doing some gesture recognition. So a first step explained a section later was to adapt the program to the GRiSP2. This new version improved the software for the new hardware, which allows us to be much more accurate in detecting movements, the list of hardware improvements can be found on the Kickstarter[5], they are :

- Real bare-metal Erlang using the RTEMS RTOS
- Support for Elixir via Nerves and Linux
- More CPU power for better peak performance and enhanced power efficiency
- Better booting capabilities, aiming at ultra-fast boot time
- Ethernet port for more network configuration choices

- Overall improved IO throughput
- More modular design to ease the move from development to production
- Improved tooling
- Complete Erlang project backward compatibility

And about the specification of the hardware[7]:

- CPU
  - NXP iMX6UL, ARM Cortex-A7 @ 696 MHz, 128 KB L2 cache
  - Integrated power management
  - 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
- 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)
  - One Digilent Pmod™ Type 4 interface (UART)

- User Interface
  - Two RGB LEDs
  - 5 DIP switches
  - Reset Key
- Debug & 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 is needed for power, console, and on-board JTAG

The sensor fusion Hera works ideally with 5 GRiSPs. In a room, there is 1 central GRiSP with a PMOD-NAV, and 4 at each corner of the room with PMOD-MAXSONAR, but using all those GRiSP to detect the gesture done on the central one is too much and has many disadvantages, like having dependency and a constant need of communication between the GRiSPs.

So we are only going to use the GRiSP with PMOD-NAV that can give us enough data with all the sensors on it. A PMOD-NAV looks like this: (2.3)

Figure 2.3: PMOD-NAV: 9-axes IMU plus barometer

And it is attached via the 12 pins to one of the GRiSP-adapted ports. As it is still a prototype, it is very sensible, it's not a "*plug and play*" device. It must always be plugged or unplugged when the GRiSP is shut down.

2.3 Gesture recognition

Gesture recognition is a hot topic for many years, we have for example this well-cited Survey[8] about gesture recognition from 2007, where it talk about many of the application that can use gesture recognition. It can range from recognizing sign language to track on objects. But this survey and most of the popular papers on the subject, are based on images to capture the gesture. They are many tools and algorithms to recognize those gestures like the Hidden Markov Model[9], Deep Neural Networks, and even Finite-state Machine. But each time they used images or equivalent input.

There is this paper[10] from 2020 using a sensor from a mobile phone rather than images. Indeed those devices have many sensors, like a gyroscope and an accelerometer, so it is a really good idea to use them. This paper uses a neural network for its algorithm going a bit too far from what we want to do, as we want a simple algorithm that can run easily on an edge computer, not a deep learning algorithm that consumes a lot of resources.

There is this paper[11] on gesture recognition on an Internet of Things device that aims to recognize a gesture on an IoT for human-to-computer interaction. The goal this time is near what we also want to achieve. This algorithm is also based on an adapted version of the Hidden Markov Model for its problem in combination with the D-S evidence theory[12]. This theory makes this algorithm more robust to uncertainty by using a combination of different sources of evidence and their associated belief. But it is based on images, and not on IoT sensors that give vectors.

Another paper[13] do gesture recognition on an IoT device, and they don't use an image as input for their research but they use a radar, detecting the micro movement of someone. It also creates to make human to computer interaction. So you can do small movements with your hand to create a gesture and with the precision of the radar, it will detect the change, creating a gesture recognition without using any images.

This other paper[14] is using an accelerometer as input for its gesture algorithm. But this time the research is not aiming for the interaction but the final goal is to create a password by "drawing" in the air a symbol that will be recognized as a gesture. They don't use the same method that we are planning to do, by having a personalized gesture for each of their participants because they each make their password. This indicates it is possible to develop an algorithm of gesture recognition from data of an accelerometer.

Those were examples of research on the subject, which indicate that there is

an interest in gesture recognition for human-to-computer interaction. With the advance of technology, the way we interact with computerized devices is always evolving rapidly and becoming more important. HCI (for Human to Computer Interaction)[15] aims to increase the user experience, availability, accessibility, and usability. Also, new technology become more complex, but is also available to more people, some of whom have little or no experience of interacting with those devices, that is why the user experience is a key component of the new technology.

We are interacting with digital interfaces more frequently than ever before. It touches many types of people around the world and also many domains, like healthcare, education, entertainment, and business. For healthcare, we have devices that could help people move but are also easy to interact with. For education, we can have tools that help children to learn something. In entertainment, there is the emergence of virtual reality and it's an immersive experience. And in business we intuitive digital interfaces.

So gesture will be the new way to interact with the technology of tomorrow because we can control devices with natural and intuitive movement. It can be the whole body, the hands, or just a finger. It also gives a new alternative to people with reduced mobility or who can't use regular input to interact with a device. And many new applications are still to be found based on the gesture as interaction. Also with the evolution of machine learning algorithms and technology, we have an increasing quality in the input, it can be better images, better sensors, or more efficient algorithms. The last important part is the apparition of IoT where the gesture is perfect to interact with them. They are also really good to obtains sensor information, and as they are more and more popular and more present, gestures will help to interact with them in a natural and "*user-friendly*" way.

A paper studied[16] this theory about the importance of hand gestures for communication. It starts by defining the different types of gestures, based on the fact that knowledge is needed to understand them, or it can also depend on the culture. Gestures are divided into 4 main categories:

- Deictics: these are gestures that refer to a real physical location, you could for example point a direction with your finger.
- Iconic Gestures: these are gestures that explain something, for example, if you need to explain a path to someone lost you could, in addition to explaining, show the path with your hands.
- Metaphoric: This time, gestures that don't have expressed for things that have physical forms. You can express that something is looping by tracing a

circle or raising a finger to show that you just got an idea.

- Beat Gestures: they are small movements we repeat multiple times, it can be when we talk or are enjoying music. It can also represent the loss of patience by beating rapidly a foot, a finger, or a hand on a surface.

They then perform an experiment where they ask people to first classify gestures according to the 4 categories then with another type of classification. They compare the quality of human classification with their algorithm and manage to have a better classification than humans. For certain parts of the experiments, they combine audio with video (where the gesture is played). So combining multiple senses (here a gesture and sounds) to increase the complexity and the possibility for communication. For their test, they used the Kappa[17] metric for reliability, and they made a confusion matrix to sort in percentage the different values for each category because each person doesn't have the same perception of all the gestures. They notice that the sound and gesture are complementary and each contains their complementary information so combining them gives a new and better result. They then talk about their algorithm that tries to classify those gestures only based on textual information. They study the sentences that are linked to a gesture, so obtaining a "gesture sentence" and by using windows of few words and linguistic analysis try to find to right classification. They then use different popular algorithms like an SVM[18] of a Naive Bayes[19] and they look at the most accurate. In this domain of machine learning it is important to try different algorithms before choosing the right one rather than picking the first one. Another interesting point for this thesis is the conclusion and future works. As they notice that gesture and sound are complementary. They talk about combining, for them, textual classifiers and pattern-matching techniques. If we adapt it to our subject. In future works, we could combine 2 algorithms to have probably a better result. Or combine 2 types of input, like a gesture coming from a motion sensor plus a vocal recognition, and have recognition based on that 2 information.

2.4 The sensor fusion Hera

The *Sensor Fusion* also called the "Hera framework", is a software developed by Julien Bastin and Guillaume Neirinckx[20] and improved by Sébastien Kalbush and Vicent Verpoten[21]. This distributed framework is fault-tolerant and works asynchronously from the sensor. Experiments were created to use the sensor of the PMOD-NAV and the PMOD-MAXSONAR, but new measures and protocols could be created for a new sensor. Hera operates by gathering data obtained from the sensor and passing them through a *Kalman Filter*. It's explained in more detail in the next section, but to summarize in a few words, it combines data from sensors

and gives a clean output based on a prediction and the last data obtained on a sensor, it is also robust against noise and failure.

During this thesis, we are going to use the Hera with only data obtained from the PMOD-NAV sensor, because we aren't interested in the movement within the room, just the movement of a person's arm. And because Hera is fault-tolerant we can use most of its features where we would need a PMOD-MAXSONAR, even though it's not in use.

The 3 metrics obtained from the sensor will be passed in the Kalman filter, so data from an accelerometer, a magnetometer, and a gyroscope. From those 3 raw data, a single output will be returned from the filter depending on what we want, it can be an acceleration over an axis or its velocity. And by having a list of those data we can then see how it evolves or change over a certain time. For our gesture recognition algorithm, we will want to obtain an acceleration and a velocity over time.

2.4.1 Kalman filter

A Kalman Filter[22] is an algorithm that uses a series of measurements over time and tries to estimate the state of a system while also reducing the noise.
It was developed by Rudolf Kalman around 1960 for systems where the measurements of data are noisy and incomplete. And it is thus widely used now in many systems that involve sensors of any kind.

The Kalman Filter will try to predict the next state of the system based on the current state and a dynamic model. This is the prediction phase, and it also includes a factor of uncertainty in the prediction. A dynamic model is the expression of the relation between the actual state and the next state, so used to predict the next one from the actual one. It is expressed by a set of equations, with their complexity depending on the usage of the Kalman Filter, the more complexes and adapted for your problem, the more accurate your prediction will be. The dynamic model has a deterministic part, and a non-deterministic one, in its system behavior, it is to "*show*" how the system evolves based on mathematics properties but also takes into account a certain randomness of the system behavior. This is used to take into account the noise from a measurement, this little randomness around the physical property of a measure.

Then the second phase, the update, the filter will combine its prediction with the next measured information to make a new state of the system. This new state

can be calculated, in different ways, for example, compute the difference between the estimation and the measurement. So each time a new measure is obtained it will be passed through the filter, and the filter will continue to update its prediction based on the model and the measurements it receives.

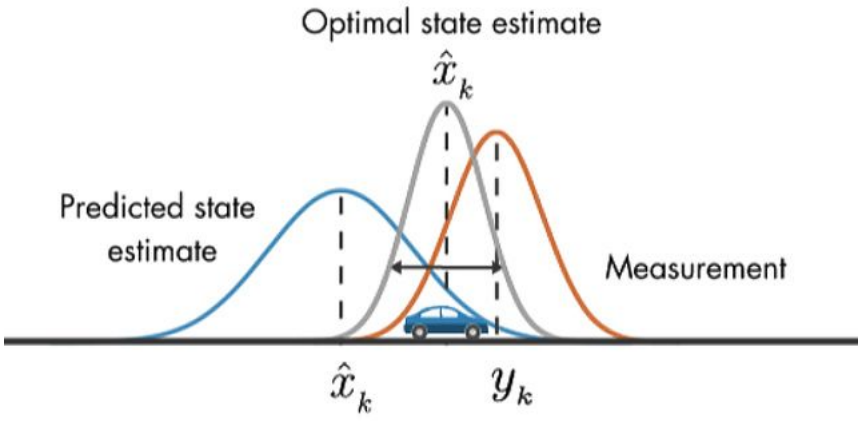


Figure 2.4: Kalman filter illustration[23]

Chapter 3

Preliminary work

In this chapter, there are all the preliminary works done before obtaining the first version of our gesture recognition algorithm. There is an explanation of the adaptation made to use Hera on the new hardware, then details about the output of Hera, and finally an analysis of machine learning algorithms.

3.1 The first GRiSP version

The previous master thesis on Hera[21] was made when the second version of the GRiSP didn't exist and thus was made on the first version. But the performance of collecting new data from all the sensors is too slow to get good gesture recognition. If we run a quick simulation and obtain an output: (3.1)

306	-576460751235	-0.5595133499999999	-1.5211582199999998	-9.159861870000002	0.22845392705311907
318	-576460748267	-1.4822615699999997	-0.09694241999999999	-9.66851037	-0.3254481276829313
319	-576460748062	-1.4553331199999995	0.5870402100000002	-9.64696761	-0.19395938516080855
320	-576460747755	-1.6617845699999998	-0.22201010999999995	-10.2268269	0.07085069559803109
321	-576460747551	-0.7246745099999999	2.20992813	-10.318383630000001	-0.309871064108882
322	-576460747345	0.07899012000000002	2.7909842400000007	-9.060525810000001	-0.09270847192948753
323	-576460747038	2.6802783900000007	2.618642160000001	-8.077338180000002	0.13544969453747122
324	-576460746618	1.8275441400000003	0.9167641200000002	-8.67514977	0.2909148976588662
325	-576460746617	3.412732230000001	0.4589804700000001	-7.084575990000001	-0.192279505755764
326	-576460746336	2.2470295500000006	2.599493040000001	-8.28319122	-0.06567768513922535
327	-576460746116	0.5888354400000002	2.1955662900000004	-10.059870510000001	0.3182511170682274
328	-576460745887	-2.8717695899999995	1.6558004700000004	-11.167527420000003	-0.10507849300299733
329	-576460745647	0.2202148800000006	0.9436925700000003	-9.35554194	-0.28070224898492674
330	-576460745297	1.5450946200000004	2.2775484600000007	-9.585331380000001	-0.03406540906247806
331	-576460745093	1.0968855300000002	1.7078621400000005	-8.277207120000002	0.23471529574464875
332	-576460744880	0.3710142000000001	3.2164537500000008	-9.754083000000001	-0.0387996146585126
333	-576460744581	0.23697036000000007	1.5636453300000002	-9.69663564	0.017399987255704734
334	-576460744373	0.04847121000000015	3.705354720000001	-9.94078692	-0.037425167872567146
335	-576460744080	0.1849086900000004	0.6618414600000002	-7.715300130000001	0.1701162968052086
336	-576460743743	0.9395037000000003	4.165532010000015	-8.60393898	0.1188036167965753
337	-576460743495	0.8694897300000003	4.049440470000001	-10.52782713	-0.13852336479433866
338	-576460743149	1.0567920600000003	5.073319980000002	-11.266863480000001	-0.1760915769435166
339	-576460742942	0.29262249000000007	1.7557349400000004	-9.47941281	-0.14509016610496733
340	-576460742659	0.5236087500000002	1.1573249400000003	-8.867837790000001	0.20722636002573802

Figure 3.1: Example of data collection with the sensor fusion on the GRiSP1

We notice on the second column, that it represents the timestamp of the moment when the data was collected. If we compute the average time between 2 of them, it

is around 300ms, which is way too slow for good accuracy.

Another thing visible in the image is about the first column, it is a list of the indexes. But there isn't always a difference of 1 between two lines. This indicates that in addition to the slow collection of information, much of them is lost before being written in this file. This gives another reason to switch to the newest GRiSP.

The first step to start the gesture detection was to adapt the previous code that was working on the first GRiSP to the GRiSP2. Those changes were mandatory because prototype boards work on precise requirements (precise programs version, compiler version, dependency version, Operating System, etc...)

The first GRiSP, needed:

- Ubuntu 20.04
- Erlang 22.0
- Rebar3 3.13
- Rebar3_hex 6.9.6
- Rebar3_grisp 1.3.0

and also many dependencies explained in the setup of a GRiSP environment, but they were less important as the updated setup for the GRiSP2 works well on the GRiSP1[24]. At the beginning of this thesis, the GRiSP2 was not still yet available, so we did the whole setup and manage to get data (3.1). When the new version was released, we adapted the environment for this new GRiSP.

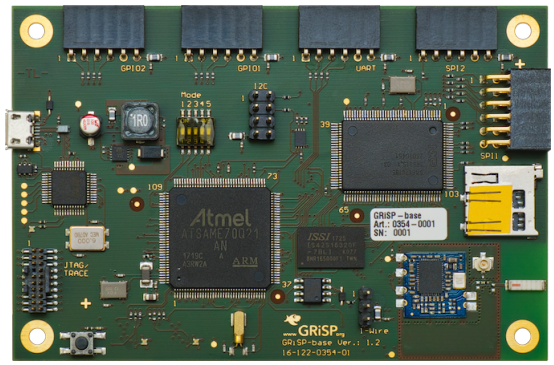


Figure 3.2: First GRiSP prototype board

3.2 Adaptation to the GRiSP 2

The first step was to try to make the GRiSP1 work, and this needed a specific OS and Erlang version because the most recent one aren't compatible anymore.

Most of the versions needed to make the GRiSP1 works aren't the same as the ones required for the GRiSP2, there is no compatibility between the 2 GRiSPs. The new working environment for the GRiSP2 is:

- Ubuntu 20.04
- Erlang 25.2.3.
  At first it was version 22.0, then 23.3.4.11, and now it works with the latest Erlang version available at the moment.
- Rebar3 3.19
- Rebar3_hex 7.0.4
- Rebar3_grisp 2.4.0

There is also, like for the GRiSP1, all the other packages needed for the development environment[24]. In addition to a working environment for the GRiSP2, we needed to adapt some side software, dependencies, and the code of the sensor fusion.

For the code, the biggest part to adapt was the network configuration's file as the files used for the GRiSP1 changed a lot and thus needed to be adapted, and then the change propagate in the code of the sensor fusion. Another important change is that the ports on the 2 GRiSPs aren't the same, so the code was adapted for the new port. To get a CSV of the data collected, we need to run an emulator on a computer connected to the GRiSP, but the emulator was never adapted for GRiSP2 and thus didn't work. So the developers developed a new version that is working on the GRiSP2[25]. Lots of dependencies forced to make the sensor fusion works were outdated for the GRiSP2 and were adapted.

3.3 Study of the output of Hera

When launching the sensor fusion only on the GRiSP, output data are not saved, we must run an emulator on a computer. This is a dependency we will get rid of when using the classifier in real-time. By launching the emulator with the GRiSP2 with a PMOD-NAV, we get an output like this: (3.3)

1	-576460659999	0.31835412	0.13224861000000002	-9.68526585
2	-576460659800	0.31835412	0.13224861000000002	-9.68526585
3	-576460659799	0.31835412	0.13224861000000002	-9.68526585
4	-576460659635	0.31835412	0.13224861000000002	-9.68526585
5	-576460659635	0.28065429000000003	0.08198217000000001	-8.96717385
6	-576460659634	0.28065429000000003	0.08198217000000001	-8.96717385
7	-576460659634	0.28065429000000003	0.08198217000000001	-8.96717385
8	-576460659634	0.28065429000000003	0.08198217000000001	-8.96717385
9	-576460659634	0.28065429000000003	0.08198217000000001	-8.96717385
10	-576460659633	0.28065429000000003	0.08198217000000001	-8.96717385
11	-576460659633	0.28065429000000003	0.08198217000000001	-8.96717385
12	-576460659633	0.28125269999999997	0.09035991000000002	-9.807939900000001
13	-576460659632	0.28125269999999997	0.09035991000000002	-9.807939900000001
14	-576460659632	0.28125269999999997	0.09035991000000002	-9.807939900000001
15	-576460659632	0.28125269999999997	0.09035991000000002	-9.807939900000001
16	-576460659632	0.28125269999999997	0.09035991000000002	-9.807939900000001
17	-576460659631	0.28125269999999997	0.09035991000000002	-9.807939900000001
18	-576460659631	0.28125269999999997	0.09035991000000002	-9.807939900000001
19	-576460659631	0.27826065	0.09514718999999999	-9.534466530000001
20	-576460659631	0.27826065	0.09514718999999999	-9.534466530000001
21	-576460659630	0.27826065	0.09514718999999999	-9.534466530000001
22	-576460659630	0.27826065	0.09514718999999999	-9.534466530000001
23	-576460659630	0.27826065	0.09514718999999999	-9.534466530000001
24	-576460659630	0.27826065	0.09514718999999999	-9.534466530000001
25	-576460659629	0.27826065	0.09514718999999999	-9.534466530000001
26	-576460659629	0.28843362	0.12865815	-9.56438703
27	-576460659629	0.28843362	0.12865815	-9.56438703
28	-576460659629	0.28843362	0.12865815	-9.56438703
29	-576460659628	0.28843362	0.12865815	-9.56438703
30	-576460659628	0.28843362	0.12865815	-9.56438703

Figure 3.3: Nav3 CSV, 30 first lines on the 5 first columns

This is a part of the file when we perform a circle in the air. There are 11 columns. We understand that the first column is the sequence number of the data, and the seconds are a timestamp. It comes from the *Hera* dependency that uses an Erlang timestamp that is calculated in milliseconds. So we have one data each 10 milliseconds, which is a better performance than the GRiSP1 performance but it still can be improved.

The 9 other columns are sensor data obtained during the run-time. If we look at the experiment called *nav3* which is the same as the CSV. We know that 9 other columns are: X-axis acceleration, Y-axis acceleration, Z-axis acceleration, X-axis gyroscope, Y-axis gyroscope, Z-axis gyroscope, X-axis magnetometer, Y-axis magnetometer, and Z-axis magnetometer. There are also multiple other simulations, more than 10 different, each time with different parameters (number of GRiSP, captor used, etc...). There is for example a experiment called *e1* which uses a PMOD-NAV and only tracks the X acceleration and another GRiSP with a PMOD-SENSOR that tracks the range, so just 2 data tracked with their timestamp. The goal of this experiment is just to test forward and backward motion on the GRiSP Another experiment, the *e2*, also uses the 2 types of PMOD, but this time tracks the acceleration on the 3 axes, to track someone in a room.

there is also the experiment named *e13*, which involves tracking the 2 types of PMOD like many other tests, but this time tracked in order, a sequence number, a timestamp, and position on the X-axis, the X-axis velocity, the X-axis acceleration, the Y-axis position, the Y-axis velocity, the Y-axis acceleration. The Z-axis position, the Z-axis velocity, and the Z-axis acceleration.

This is another test than the *nav3* CSV of 11 columns. But this test could give us a really good output for starting our gesture recognition algorithm because starting with using the acceleration or velocity as the input seems a good idea as it can be obtained just with a PMOD-NAV. But there is a problem with the performance, a cause of it is that the experiment is "heavy" as it should involve multiple GRiSP with PMOD-SONAR and 1 PMOD-NAV. Thus it can slow from the start or slowed because we only use a part of the system, so even if it's still working it's slowed down. we will try other experiments and compare their performance.

To obtain data from an experiment the file $sensor\_fusion.erl$ file must be changed to call the desired experiment. After obtaining data from some experiments, we have this table:

Experiment	Average time
e11	14.478 ms
e13	46.838 ms
nav3	9.948 ms

We see that the performances are nearly the same over the different experiments except for *e13*. And as this thesis isn't focused on having better performance, because it is the one of another student, we are going to keep those performances for the moment. We will use data from *nav3* and we are going to make slow movements for the moment and find a way in the algorithm to reduce the impact of having few data per second.

3.4 First search on machine learning algorithm

To develop a classifier, we start with a simple one and then try more complex ones to get specified to this problem. It can be by adapting what we have, or changing the algorithm if finding another more fitting.

After a first search around algorithms for gesture recognition from sensor vectors, several algorithms can be implemented. There is, first, the Decision Tree

Classifier[26] which is a classifier that separates different data with a sequence of conditions and chooses those conditions to try to have as many as possible data classified well at the end. This can a be small starting algorithm because our data are a sequence of positions so if we do the same gesture with the same starting point and the same speed this algorithm could recognize 2 nearly the same gestures done at 2 different times.

Another possibility is to use a Support Vector Machine[18] it's an algorithm that divided your plan or space, by "drawing lines" between points. So we get different clusters and thus points on the same side are grouped. There is for example this paper that uses this kind of classification by cluster, it is about motion recognition for each finger[27].

The last algorithm is the K-means[28] algorithm which also uses a classification by cluster. This one defines several *centroid* and will fit each point to its nearest *centroid*. Then move the *centroid* to the mean position of all its points, and we come back to the first step at the assignment of the point to a *centroid*. This loop is done until a certain number of times or until we don't detect many changes between 2 different loops. There will be one cluster for each *centroid* and its associated point.

We also search for Erlang-specific algorithms, and there's not much public information or algorithm of machine learning in Erlang. There are algorithms in public GitHub projects[29], and a potential library that could be adapted to Erlang (libsvm[30]).

We started by developing a K-means algorithm for 2D points. Then it was adapted to points in a 3D space. The code is the annexes but not in the project GitHub because we've reached a dead end trying to switch from a point cloud to vectors.

3.4.1 Algorithms can't be adapted to our case

What we researched was algorithms using *unsupervised learning*[31], which means that the algorithm learns and trains on unlabeled data. But it is not what we want for our algorithm, because we will feed the algorithm with an initial gesture list that will be labeled, it mean that we will give a name of a gesture and all the data of this gesture. We want to do a *supervised learning*[32], thus previously researched algorithms can't be adapted for what we want to achieve in this thesis. We need to look at *supervised algorithm* or create our algorithm.

3.5 Research on supervised learning algorithm

Many previously talked about algorithms as *unsupervised learning* algorithms[31]. We want to have a *supervised learning*[32], after researchers on algorithms that could be adapted for gesture recognition, we have this list:

- **Neural Network[33]**: It is an algorithm that has multiple layers of states. There is an input layer, some hidden layer, and an output layer. Depending on the state of the input at the start of the algorithm, information will be sent to the 1st hidden layer. Again, depending on what this layer has received, the output to the next layer will differ from input to input. This continues until the last layer, the output layer, is reached. The problem with this algorithm is that it quickly becomes complex and needs heavy computation, and we want a simple algorithm that can run on GRiSP. The layers between input and output are not observable, and we want to see as much as possible what's going on in our gesture recognition algorithm.
- **Hidden Markov Model[9]**: It is a statistical model that describes systems with hidden states generating observable symbols. It utilizes transition and emission probabilities to model the sequential dependencies. There is the same problem with this model, where there is some hidden part.
- **Dynamic Time Wrapping[34]**: It is a technique used to measure the similarity between 2 lists of time sequences that may vary in speed or length. It aligns the sequences by warping the time axis to find the best match, allowing variable alignments between the sequences. This technique can be used if we always keep the same sensor data, so only the acceleration on X for example. It will need an adaptation to take into account more axes and other sensors. There is also still noise even after data has passed through the Kalman filter and this technique isn't made to reduce noise. And the computational complexity of this technique increases exponentially with sequence length, which is not suited for the long data collection on the GRiSP over multiple gestures.

So those algorithms aren't adapted to the goal of this thesis.

Chapter 4

Main Contribution

This chapter explains the development of the gesture recognition algorithm, from its first version to the last one. For each new version, there is an explanation of the improvements, how it was implemented, and experiments.

4.1 Physical Properties of a gesture

The input of a gesture is the clean acceleration values obtained from the Kalman filter. These values are relatives to the rotation of the GRiSP during the calibration. They are not relative to the ground because the magnetometer is also relative to the calibration of the GRiSP(this property is studied in a section below) and thus can't be used to always know a direction regardless of GRiSP's orientation. They are also not relative to the orientation of the GRiSP during the runtime, so any rotation around the Z-axis will not influence gesture recognition. For example, if a user performs a forward-backward gesture, like a "punch". Then turn his body or just his arms 90° around the Z-axis, and do the same gesture, they will be recognized as the same. But if he turns his arms 90° around the X or the Y-axis and now "punches" toward to sky or the ground, it will be recognized as another gesture, because of the rotation.

A reminder of rotation in 3D : (4.1)

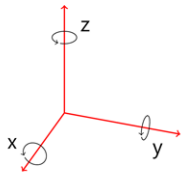


Figure 4.1: Rotation around 3D axis[35]

4.2 Gesture recognition by pattern matching

After seeing that all the previous algorithms aren't working well because they are too focused on each individual point, and reaching a dead end with the previous methods. We've changed the way we identify a gesture and identified it by a "pattern", which means we will not look at precisely each data point, but at the general behavior of each value. So by looking at the acceleration or the velocity of an axis over time, we can understand what is the behavior of the gesture. And by looking at the pattern of a new gesture we could calculate the distance between this gesture and all the one learned by the algorithm and return as output the closest match.

After the first version of the algorithm, we can normalize the gesture in the algorithm, to be used from an absolute direction, for example. We could be to use the magnitude[36], which is a mathematical computation from vector over multiples axis to combine them and find the final vector. So in our case, we could have the position at the start and the end of a gesture or periodical position. It could be used to help with calibration or noise canceling, but can also be used as input for the algorithm and used for gesture recognition.

We decided to do the first version of the pattern matching, by looking just at the acceleration over the 3 axis. We will look at if the data is positive, negative, or near zero. Then looked at each time we have we change in one of those behavior so we have a list of information on how the gesture change over time.

We would have something like that:
{"Gesture 1", "positive acceleration", "No acceleration", "negative acceleration"}
So each value is followed by a different one, as we combine them if they are the same. This will compensate for the bad performance of the sensor fusion and the lack of many data for each second. The threshold to detect a positive or negative value will be above 2, or lower than -2, to take into account the noise.

We did this to overcome performance problems and because we can obtain quick results and see if we will continue in this direction or quickly shift to something else. We are using the acceleration for the moment as it changes are easy to look at even by just looking at the CSV, but the algorithm can be changed or extended to the other metrics.

4.2.1 Implementation

We first make a parser for a CSV file. We will not directly compute the learning and classification in real-time, it will be a step after we have a good algorithm we are satisfied with. We will use the *nav*3 experiment to have as input the acceleration for the 3 axes. Any experiment can be selected by changing some lines in the *sensor_fusion.erl* file. This parser takes a CSV file as input and returns a list of the data in the requested column using the file library[37] from Erlang.

The algorithm has 2 distinct parts, the first one is about learning, and the second step is the classification. It will thus be done in 2 separate files and have a clear distinction between those 2 parts.

The function works like that from CSV with a gesture:
- learn(CSV, "Gesture Name ") => Add the gesture to the list of learned gestures.
- classify(CSV) => Return the closest gesture from the list of learning.

Learning

We will only learn on the X-axis, in the first step. It uses the parser to obtain a list and look at the value and see the behavior of the gesture. Then it computes the pattern to have a list of the change in the value of the acceleration. To keep in memory all the gestures learned even with the GRiSP offline or off, are written in a file. A file called "gesture" where the gestures are saved like this:
{gesture_one, zero, pos, zero}
We first have the name of the gesture and then the list of changes in the acceleration, they are written as 'atom' as it is really easy to use those types of variables in Erlang.

Classification

It imports the learned gesture written in the file to usable variables and lists. So we have a list of gestures that are themselves listed with the name followed by the list flow of the gesture. The "flow" is the change in the acceleration. To compare a new gesture to the list of learned ones, it imports the acceleration from a CSV for the new gesture and like with the learning, turn it into a clean list of the flow of the gesture, so a list of the change in the acceleration. For comparison, it looks at each behavior, one by one, and looks if it's the same between the new gesture and one of the lists of learned ones. It then computes the number of the same behavior on all the comparisons to obtain accuracy.

Then it returns the *name* and the *accuracy* of the gesture with the highest accuracy, so the final output will look like this:

Name : gesture_one, with Acc : 0.77

The equation is:

$$AccI = Same/Comparison$$

$$Closest = MAX([Acc1, Acc2, Acc3, ...])$$

For each learned gesture I, we compute the *Accuracy*, by looking at each value of the list of behavior, each time it is the same between the gesture I and the new one we want to classify, it increases the value $Same$ by 1, $Comparison$ is always increased by 1 for each comparison. When it reaches the end of a list, it divides the value $Same$ by $Comparison$. It is computed for each learned gesture. In the end, it returns the closest learned gesture by returning the highest *Accuracy*.
The computation of accuracy must be improved because it's a very simplistic calculation at the moment.

4.2.2 Improvements

Learning

We extended this algorithm to compute on 3 axes. So in the gesture file, we add between the *name* and *flow list*, the axis. (so a x, y, z) So we will have 3 lists for a gesture in the gesture file. It also makes the file more readable, like this the gesture can be read and understood by a human to understand what is happening and eventually add manually add a gesture for testing the algorithm.

Another improvement is to increase the accuracy and avoid having a new gesture very different from its learned gesture. It is by adding multiple gestures with the same *name* but just a bit different pattern in the file. So each name will be the same, but not the pattern, like this, a new gesture will have more chance to correspond to its learned and be matched with the right gesture. They will not be combined, so just be compared individually by the algorithm like gestures with 2 different names, and the one with the biggest accuracy will be returned.

There is another improvement that must be done because they are still noise in the data even after being passed through the Kalman filter. For example, there change in acceleration if we don't move. This noise changes the flow because it gives a different behavior from the previous one and the next one. To explain with an example:

{no move, zero} What we want when not moving.
But we have this with noise:
{no move, zero, pos, zero, pos, zero, neg, ...} And that is not what we want.

To reduce the noise, we will combine in a group the type of pattern before the step where the algorithm combines them based on the change. Through this, we could reduce noise without losing any information. So take for example 10 data and calculate the most recurrent pattern. So we will create a new list of patterns 10 times smaller with each element being the most present one for 10 elements in the original list.

About the coding, the algorithm calls 3 times the *nav*3 CSV file to request value for the 3 axes. Then before transforming each value to the name of a pattern, it takes the list of values, 10 elements by 10 elements. With those, it calculates the most present pattern and makes a new list with all those most present patterns. And the rest of the algorithm doesn't change much. During the saving to the file, as said before, the name of the axis is added after the name of the gesture. So we pass from a gesture on a single axis, with many noises, to 3 cleans gesture:

From a single noisy gesture:
{no move,zero,pos,zero,pos,zero,neg,pos,zero,pos,zero,pos,zero,pos,zero,pos,
zero,pos,zero,pos,zero,pos,zero,pos,zero,pos,zero,pos,zero,pos,zero,pos,zero,
pos,zero,pos,zero,pos,zero,pos,zero,pos,zero}
To 3 clean gestures:
{no move,x,zero} Value for the x axis
{no move,y,zero} Value for the y axis
{no move,z,neg} Value for the z axis

Classification

There is also a change in the classification because now we need to take into account the accuracy of the 3 axes and calculate the final based on those 3 values. So now rather than compare the new gesture with each gesture one by one, we look at the 3 next gestures (that are 3 axes for a gesture). Then compute the number of

matching patterns on the number of comparisons, then add the accuracy of the 3 axes, and divided them by 3, to have the average.

So *AccI* is now computed like this:

$$AccIX = Same/Comparison$$

$$AccIY = Same/Comparison$$

$$AccIZ = Same/Comparison$$

$$AccI = (AccIX + AccIY + AccIZ)/3$$

4.2.3 Testing

We tested the algorithm with a *round trip half circle*: (4.2)

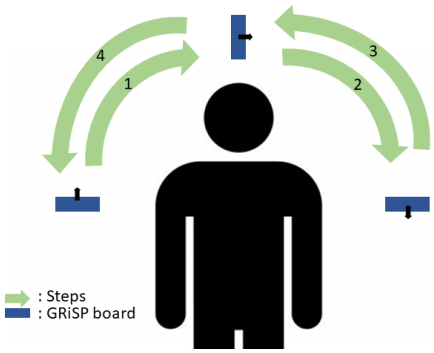


Figure 4.2: A round trip half circle

The result of the learning of this first gesture, in the gesture file:
{halfcircle,x,zero,neg,zero}
{halfcircle,y,zero,pos,zero,pos,zero,pos}
{halfcircle,z,neg,zero,pos,neg,zero,neg,zero,pos}

We also manually write a dummy gesture in the gesture file to see if the algorithm correctly computes and chooses the closest gesture, the dummy gesture is:

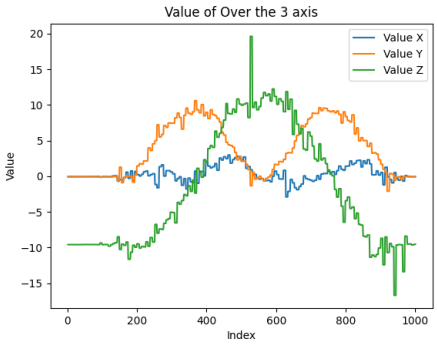
{test,x,xxx}
{test,y,xxx}
{test,z,xxx}

We wrote *xxx* to see the behavior of the algorithm when the gesture is never the same between a new gesture and a learned one.

We perform a second gesture, the same one, we don't use it for the training but try to classify it. We plot the data obtained during the gesture and compute the output: (4.3)

```
11> classify:classify_new_gesture("../measures/hc1.csv").
X : 0.0, Y : 0.0, Z : 0.0
X : 1.0, Y : 1.0, Z : 0.375
Name : halfcircle, with Acc : 0.7916666666666666
```

(a) Output of the classification of the half circle



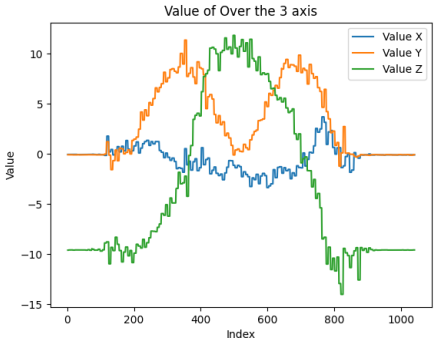
(b) Acceleration values for the 3 axes

Figure 4.3: Output and acceleration of the 2nd gesture

We first see that there isn't any match with the test gesture which is what we expected. And there is a good matching with the first gesture of the half circle, as they got the same pattern 79% of the time. We add this gesture to the learning and do a third gesture: (4.4)

```
3> classify:classify_new_gesture("../measures/hc2.csv").
X : 0.0, Y : 0.0, Z : 0.0
X : 1.0, Y : 1.0, Z : 0.6
X : 1.0, Y : 1.0, Z : 0.8
Name : halfcircle, with Acc : 0.9333333333333332
```

(a) Output of the classification of the half circle on 2 other gestures



(b) Acceleration values for the 3 axes

Figure 4.4: Output and acceleration of the 3rd gesture

We still see 0 matching with the test gesture. But this time it has an even better match than before, this time with the second half circle. So this show that

is it really good to add more sample of the gesture to have the best comparison. We add this gesture to the learned gesture and perform a different movement.

We do a forward and backward movement with the GRiSP, moving 3 times in succession for a gesture. And do 2 gestures, one learned and the second tried to be classified: (4.5, 4.6)

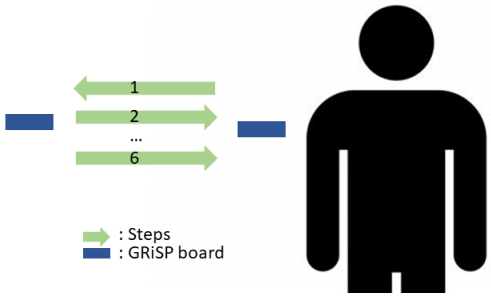
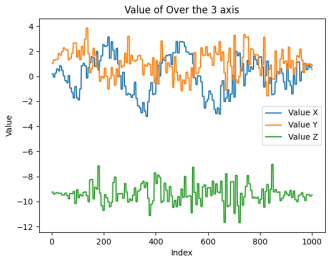


Figure 4.5: 3 times, a forward-backward movement

```
17> classify:classify_new_gesture("../measures/bf1.csv").
N : test, X : 0.0, Y : 0.0, Z : 0.0
N : halfcircle, X : 0.6666666666666666, Y : 0.6666666666666666, Z : 1.0
N : halfcircle, X : 1.0, Y : 1.0, Z : 1.0
N : halfcircle, X : 0.5714285714285714, Y : 0.5714285714285714, Z : 1.0
N : forwardBack, X : 0.6, Y : 0.6, Z : 1.0
Name : halfcircle, with Acc : 1.0
```

(a) Output of the classification of the Forward and backward movement



(b) Acceleration values for the 3 axes

Figure 4.6: Output and acceleration of the for the forward-backward gesture

With this result, we start to notice the limitation of this first version, because the second gesture was matched with the first one. This is caused because we stop the recognition as soon as we reach the end of a list of patterns, and because one of the lists of the *halfcircle* isn't very long, and has the same first pattern, the new gesture is matched to it. For example, if we create a gesture where we don't move, every next gesture will be matched to it because the first element at the start will be the same, and because there will be only 1 pattern in the learned gesture list of the *no move* the recognition will immediately stop and show a 100% accuracy with the no move for any new gesture.

4.2.4 Limitation

With these experiments, we notice a few limitations. The first issue is the algorithm classification limitation of the moment, we compute from patterns difference between 2 lists, but the computation stops when reaching the end of the smallest between the new gesture and the one learned. So this must be changed to be adapted to the size of the list of longest patterns.

Another problem is that the gesture is relative to the GRiSP. Because all the information is given based on the position of the GRiSP, if it is turned around the GRiSP and the same gesture is done as the previous one, you will not have the same movement for the GRiSP. In other words, if we turn the GRiSP 90° on the X-axis, we will have reversed the X and Y-axis, that means if we do a movement previously learned this one will have its X and Y reversed and it will not be recognized as the same one. If we want to recognize a direction, we want to have an "absolute" position. That means if we point in a certain direction, the north for example, even if the GRiSP is turned by 90° around any axis and then we point again to the north with the same movement, it should be recognized as the same movement. For this, we need to normalize the data based on a fixed value, we could use the gyroscope and magnetometer to get this absolute direction.

Another problem will emerge if we start to do an absolute position, little by little there is going to be an accumulation of noise and thus an increasing gap with the initial values. So a protocol to re-calibrate the GRiSP will need to be implemented if used over a long period. For the moment we are only using it for a small period, so it is not a priority.

We also need to have gesture recognition running in real-time. But the learning can be done "offline", so learning the gesture and saving those learned gestures in the file that will be imported on the GRiSP. Then the classification will be done on run-time on the GRiSP.

We also want to add the possibility to learn gestures on run-time. It can be done in different manners. Ask specifically to learn the gesture that will be performed, but it can also be asked after a classification. We could for example ask for any classification or, just if you don't have a good accuracy with all the other gestures, meaning it is probably a new one.

4.3 Improvements

4.3.1 Whole list comparison

We adapted the algorithm to make a better comparison. Before the comparison stopped when reaching the end of the shortest list. Now it will look until the end of both lists for the comparison, where excess elements in the longest list will be counted as different. By performing the last comparison done in the section above, we have this result: (4.7)

```
5> classify:classify_new_gesture("../measures/bf1.csv").
N : test, X : 0.0, Y : 0.0, Z : 0.0
N : halfcircle, X : 0.08695652173913043, Y : 0.08695652173913043, Z : 0.125
N : halfcircle, X : 0.043478260869565216, Y : 0.043478260869565216, Z : 0.1111111111111111
N : halfcircle, X : 0.17391304347826086, Y : 0.17391304347826086, Z : 0.2
N : forwardBack, X : 0.6521739130434783, Y : 0.6521739130434783, Z : 1.0
Name : forwardBack, with Acc : 0.6935817805383023
```

Figure 4.7: Forward-backward movement classification with the new version

We now see that the forward-backward movement is well classified with its learned gesture.

4.3.2 Study on the magnetometer

As said earlier, we want to find a way to have an "absolute" position, like that we would be able to be anywhere and point in the same direction, (for example, pointing to a device) and it would be detected as pointing to the same direction. So we are going to look if the magnetometer gives always the same direction after a rotation or another initialization.

We place the GRiSP on the table facing the north direction, (using a compass to have the direction). Then start recording a gesture, don't touch the GRiSP few second then turn 90° clockwise around the Z-axis, the GRiSP now face the east then wait a few seconds. Repeat this experiment 3 more times to return to facing north: (4.8)

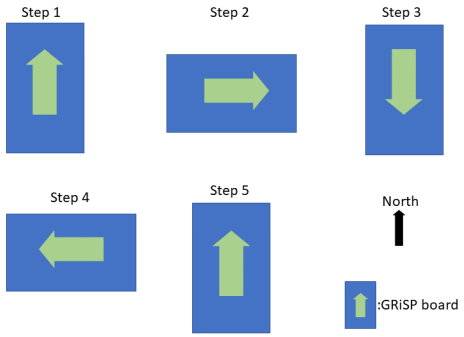
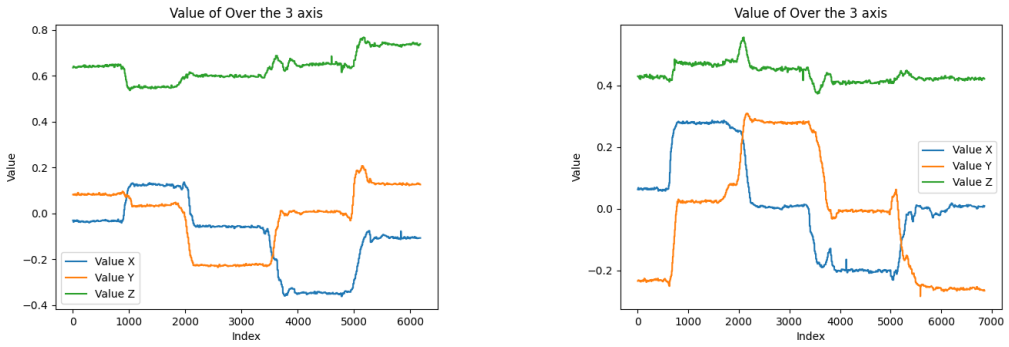


Figure 4.8: First experiment: turning the GRiSP 4 times starting from the north

For the second step, the GRiSP is shut down and restarted facing the south. We will perform the same experiment but turn the GRiSP anti-clockwise, so facing south then east, then north, then west, then south again. We are going to look if we have the same value for each direction for both experiments: (4.9)



(a) Clockwise rotation from north

(b) Anti-clockwise rotation from south

Figure 4.9: Magnetometer values when turning the GRiSP around the Z-axis

For the first experiment, we see these results:

- **North**: X = -0.03, Y = 0.08, Z = 0.63
- **East**: X = 0.12, Y = 0.03, Z = 0.55

- **South**: X = -0.07, Y = -0.22, Z = 0.61
- **West**: X = -0.35, Y = 0.01, Z = 0.65
- **North** again: X = -0.10, Y = 0.12, Z = 0.73

We notice well with the graph and the value that we manage to have the same number when returning to the north, they are not the same because they are very precise, and just a little difference between the 1st north position and 5th one where the GRiSP face again north changed the value. We also notice that the Z value changes a bit, it's because the GRiSP isn't flat when placed on the table, so turning it around change a bit its value, and for this experiment, it is not an important value. So it will not be evaluated during the second experiment.

The value for X and Y in the second experiment are: (Z doesn't interest us)

- **South**: X = 0.06, Y = -0.23
- **East**: X = 0.27, Y = 0.02
- **North**: X = 0.01, Y = 0.27
- **West**: X = -0.20, Y = 0.01
- **South** again: X = 0.01, Y = -0.25

If we look at the corresponding value of X for a cardinal direction for both of the experiments, we see some difference between them, but if we match each value to its closest value in the other experiment it matches only for east and west. This is seen in the graphs above, where the value of X follows the same pattern for both experiments, where the east is the second value and the west is the fourth one, with roughly the same value. The north and south are seen at tested at different moments and because the curve on both graphics are the same they match the opposite value. So the value obtained when facing north in the first experiment is the same as the value obtained when facing south in the second experiment.

About the Y-axis, we see that it is the opposite between the 2 experiments, this means that these values are relative to the direction of the GRiSP during the calibration. So using the value of the magnetometer is impossible to always know which direction we are facing because it depends on the calibration.

4.3.3 Study on the Z-axis

When not moving, the Z-axis is always negative around 9.81, because it's the gravity, the acceleration toward the ground. To see how it evolves depending on the orientation of the GRiSP, we will perform 2 tests. The first one will be rotation around the X-axis of 180°, so at the end, the GRiSP is upside down compared to its starting position. Then returning to the initial position. For the second experiment, we first rotate the GRiSP 90° over the Z-axis, so now the GRiSP is facing a new cardinal direction, then perform the same movement during the first test. Through this, we can see how does evolve the Z-axis over time and see if it is influenced by its orientation. We obtain these results: (4.10, 4.11, 4.12)

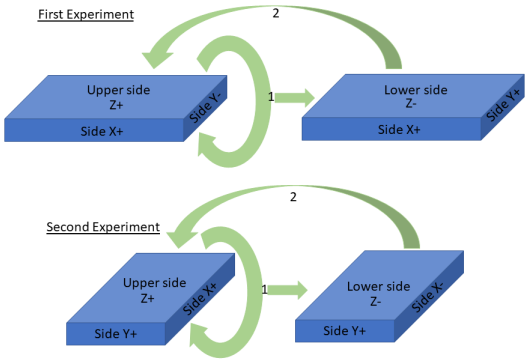
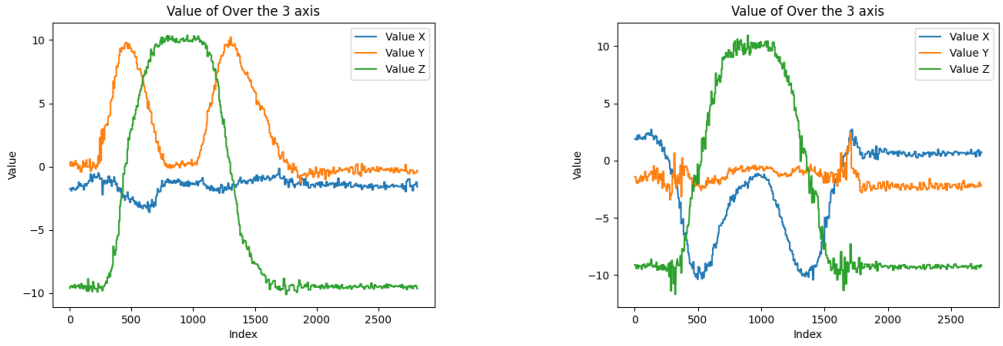


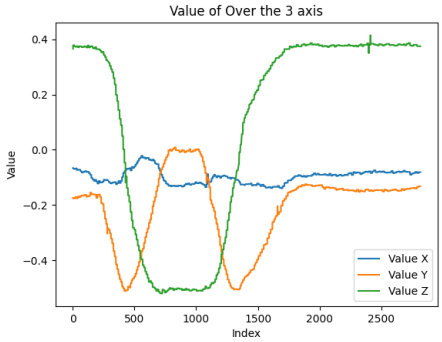
Figure 4.10: 1st experiment: turning around the X-axis; 2nd : turning around the Y-axis



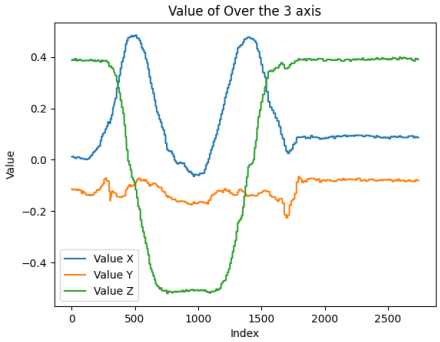
(a) 1st acceleration values

(b) 2nd acceleration values

Figure 4.11: 180° rotation forward-backward, then perform again after turning the GRiSP 90°



(a) 1st magnetometer values



(b) 2nd magnetometer values

Figure 4.12: 180° rotation forward-backward, then perform again after turning the GRiSP 90°

We plotted both the acceleration and magnetometer values to study them. The first thing to notice is that values oscillate between -10 and 10. So this gives the scale to use for the algorithm. Then the green line that represents the Z values is directly linked to the orientation of the GRiSP as this acceleration represents gravity, if the GRiSP's upper side is facing the sky we will have around -9.81 of acceleration. And if the GRiSP is facing to the ground we have an acceleration of 9.81.

Then, when looking at the 2 others axis, if we compare both graphs, we notice that the X and Y value is just exchanged and reversed. So the positive X value in one graph corresponds to the negative Y values. And by looking at the magnetometer value, we notice that they are the same but reversed compared to the accelerometer. They are also on another scale by being between -0.5 and 0.5.

The last thing to notice is the noise created by the movement on an axis that shouldn't change. Because we can't make perfect and stable movements, so even if we don't want to influence the acceleration they are still the noise created because we move. It doesn't exceed 5 but is still bigger than 2 which is the actual threshold to detect a change in acceleration. Thus the threshold should be adapted to fit better real-world experimentation.

After this experiment, we have several choices for using the Z-axis effectively:

- The first choice is to not take into account the Z-axis during the comparison. If we rotate the GRiSP around this axis, it will still be the same movement. That means if we attach the GRiSP to our wrist and we rotate it when we move, this rotation will not be taken into account. This choice can be made

if we want a movement to be recognized as the same even with a slightly different movement.

- An extension to not using the Z-axis during gesture recognition, can be to use it for the separation between 2 gestures in real-time. We could use the Z-axis to indicate the end or start of a movement. For example, turning the GRiSP upside down and then returning to the initial position could indicate the start of a gesture. Because we would have the Z-axis going from -9.81 to 9.81 and this can be detected.
- Another choice is to normalize the Z-axis around 0 and use it like before. We could use the same threshold condition as for X and Y. Or we can adapt them because the Z will oscillate between 0 and 20.
- We can also still use the Z-axis for gesture recognition but with a different threshold than for the X and Y acceleration.
- A last choice is to not change anything.

Change on the Z-axis

We implement an adaptation to add 9.81 to each value of the axis, so when you have your GRiSP just simply placed on a table and face up, the Z acceleration would be around 0. But we notice it didn't change much in gesture recognition. There is just not the same label for the name of the pattern, being around *zero* rather than around *neg*. But no improvement. So we revert the changes.

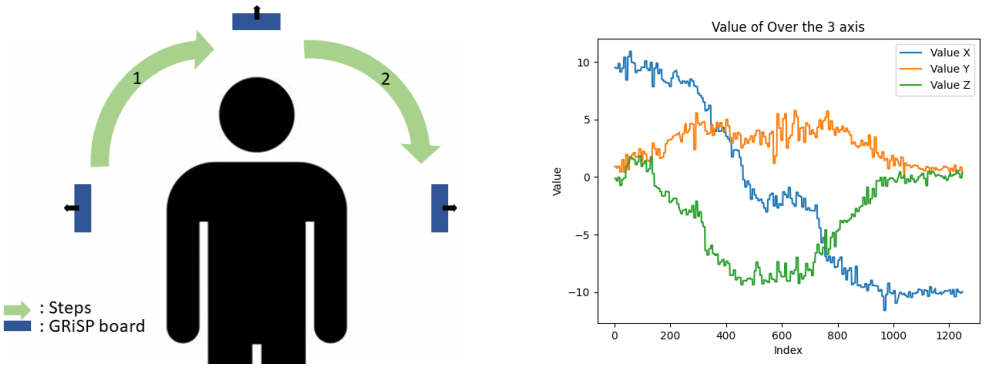
We thought this axis was different from the other 2, but that's not the case. Starting with the GRiSP on a table, if it rotates 90° around the X-axis, for example, depending on the direction of rotation, the acceleration of Y will be + or - 9.81. The axis towards the ground will always be + or - 9.81 because of gravity, which is an acceleration of $9.81 m/s^2$. This acceleration is very fast compared to the speed of the gestures we have performed previously, which is why we thought the Z-axis was different. This indicates that our gesture recognition algorithm which uses acceleration as input is very efficient to detect a rotation in any direction.

We will then keep the Z-axis without any change, but improve the threshold to detect a change in acceleration. Previously if a value was above 2 or under -2, the algorithm detect a change. Now, we have added *highly* positive and negative. As we saw previously that the values can be between -10 and 10, so the new threshold is:

- Highly negative: $<$ -6

- Low negative: >= -6 & < -3
- Around zero: >= -3 & <= 3
- Low positive: > 3 & <= 6
- Highly positive: > 6

We also improve the way the noise is canceled, by now looking at more than 10 values to find to average. To find the right value, we will experiment and make a gesture, a half circle: (4.13)



(a) Gesture of a half circle

(b) Half circle acceleration values

Figure 4.13: Half circle gesture and its acceleration values

For this experiment, the half-circle was made with the arm extended. To have less balance and generate the maximum noise theoretically possible for a movement made by a human. If we keep the noise canceling as it is now we have this learned gesture.

{halfcircle10,x,pp,p,o,n,nn,n,nn,o,nn}
{halfcircle10,y,o,p,o,p,o,p,o,p,o,p,o,p,o,p,o,p,o,p,o}
{halfcircle10,z,o,n,nn,n,nn,n,o}

We notice that they are still too much noise, so we looked at another list size to compute the average. We tried a size of 50 and 100 data:

{halfcircle50,x,pp,p,o,n,nn}
{halfcircle50,y,o,p,o,p,o}
{halfcircle50,z,o,n,nn,n,o}

{halfcircle100,x,pp,o,nn}
{halfcircle100,y,o,p,o,p,o}
{halfcircle100,z,o,nn,n,o}

We now see that using 50 data to compute the average is better, and using 100 is too much as we lose information, we can see that for the X-axis, the isn't any low positive and negative value. But there is still noise for the Y-axis because there was much noise when the acceleration value was near the threshold. If we look at what is done in telecommunication, to avoid interference between 2 close signals, we can implement a "Guard Zone"[38]. It is used in telecommunication to avoid having 2 information on the same frequency or at the same moment by letting a space between 2 reserved slots. We are going to adapt this concept to the algorithm by adding a "Guard zone" between certain changes in the behavior of the acceleration. The new thresholds are now:

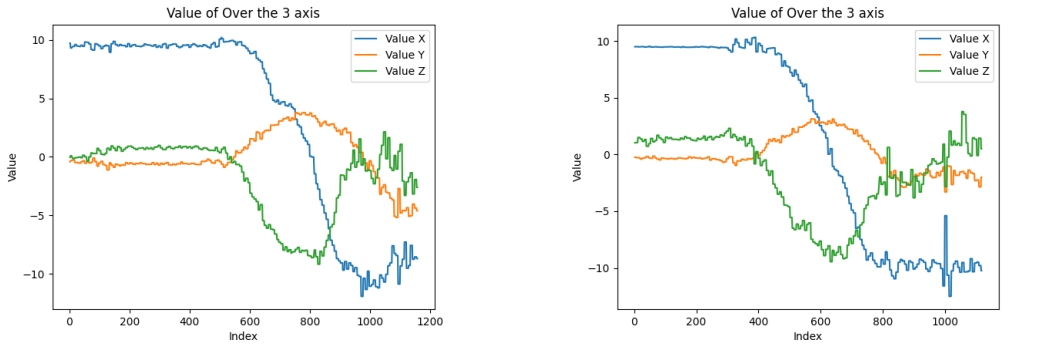
- Highly negative: < -7
- Low negative: >= -7 & < -4
- Around zero: >= -2 & <= 2
- Low positive: > 4 & <= 7
- Highly positive: > 7

So if a value is between -2 and -4, or 2 and 4, this data will be dropped. We tested this new version with a size list of 50 and 75. We obtained those learned gestures:

{hcguard50,x,pp,p,o,n,nn}
{hcguard50,y,o,p,o}
{hcguard50,z,o,n,o,nn,n,o}
{hcguard75,x,pp,p,o,n,nn}
{hcguard75,y,o}
{hcguard75,z,o,n,nn,n,o}

The version with 50 data for the average has still some noise, but with a list of 75, we have what we wanted to obtain for the pattern of this gesture. The next step is now to make other gestures and see if we can match them well.

We will do 2 half-circle with a new calibration to see if everything is working as expected: (4.14)



(a) Acceleration values of the 1st gesture

(b) Acceleration values of the 2st gesture

Figure 4.14: Acceleration values of half circle gesture from right to left

We learned the first gesture and classify the second one: (4.15)

```
X : 0.6, Y : 1.0, Z : 1.0
Name : halfcircle, with Acc : 0.8666666666666667
```

Figure 4.15: Classification of the 2nd half circle

the 2 gestures have been matched together but we see a small difference because the second gesture was done faster and the algorithm didn't detect the light negative value for X, going directly to highly negative.

We will do a new gesture, the GRiSP is turned 360° around the Y-axis, like before we perform 2 gestures, one to learn, the other to classify: (4.16, 4.17)

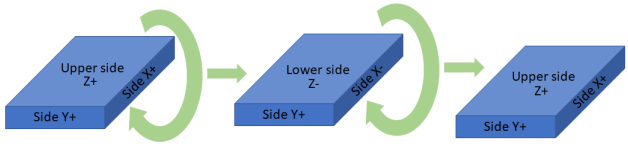
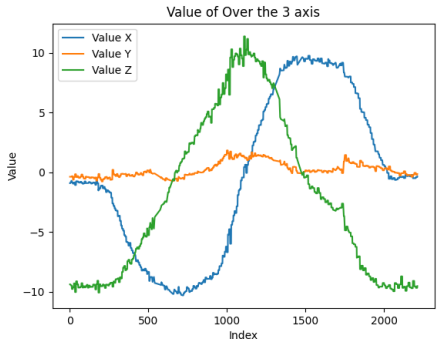
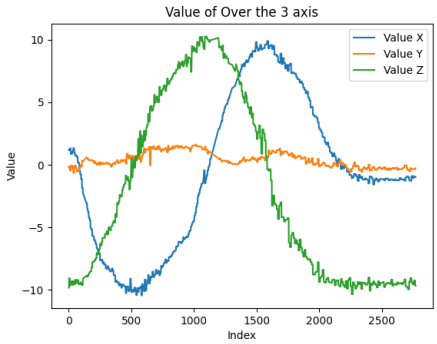


Figure 4.16: 360° rotation around the Y-axis



(a) Acceleration values of the 1st gesture



(b) Acceleration values of the 2st gesture

Figure 4.17: Acceleration values of a 360° rotation over the Y-Axis

We see well that we are turning the GRiSP because the acceleration evolves smoothly, indeed if we turn the GRiSP 90° around an Axis another axis is going to face the ground and have an acceleration of -9.81 because of gravity. If we now perform a classification: (4.18)

```
18> classify:classify_new_gesture("../measures/rot1.csv").
N : halfcircle
X : 0.1111111111111111, Y : 1.0, Z : 0.1111111111111111
N : halfcircle
X : 0.0, Y : 1.0, Z : 0.1111111111111111
N : rotation
X : 1.0, Y : 1.0, Z : 0.5555555555555556
Name : rotation, with Acc : 0.8518518518518517
```

Figure 4.18: Classification over the rotation

The new rotation is matched with the first rotation as expected.

4.4 Real-time classification

We have a correct and satisfying algorithm to classify a gesture, we will now adapt it to do it in real-time on the GRiSP. Until now, when performing a gesture on the GRiSP, it produces a CSV file with the gesture. Then we had to manually launch the classification that read this file and find the best match between all the learned gestures. We will get rid of the last step and compute immediately the classification after the gesture has been performed. They are multiple ways to divide and use the data:

- **Classification over the whole time**: For recognition, it will always use the whole data collected from the start of Hera until it is stopped. And each time enough new data are obtained, it will try to match the best result. So at the start, it will not give a good match because the movement isn't done entirely, but after the gesture has been made, it should give a good match. To stop, it can detect that there is no movement, stopped manually or it can after a certain time.

- **Classification over a period**: The user will define a time window to perform the gesture. So just all the data collected during this time will be used for the gesture. It can be implemented by looking at the timestamp and only keeping the data that are under a certain value.

- **Classification divided by gesture**: The algorithm will divide 2 gestures by performing a specific one, for example by using an axis, if we turn the GRiSP around an axis, it can detect and can mean that we are starting a gesture. Then the user redoes the turn-around movement at the end to indicate the end of the gesture.

- **Classification on matching percentage**: It could detect that a gesture is finished and a new one is starting if it reaches a certain percentage of accuracy to one of the gestures. It will classify each time there is enough new data and compute the best accuracy toward a learned gesture. For example, if we perform a movement and at the end of it, the algorithm detects over 90% of matches for a learned gesture. It will understand that it was a movement and will now start to classify the next one only with data collected from now on.

- **Classification by detecting a stop** Another possibility is to detect when the user stopped moving, meaning a gesture has ended. the algorithm will then classify with collected data up until this time. For the next gesture, the algorithm will not keep the data used for the classification that was just done. It can match a "*no move*" gesture and start collecting for the next from now

on. It can also detect that for the 3 axes, there is no major change, it can be on the value or just the general flow (if it is a positive or negative value). It could also only take into account 1 or 2 axes to detect a stop.

4.4.1 Retrieve data from Hera

Previously, all data were obtained from CSV, but for the real-time classification, we want to get them directly from Hera, the instruction are explained in the previous master thesis[21]. But in a quick summary. It is possible to define what data from the sensor should be collected or take an experiment already implement. Then, the sensor fusion process needs to be started, and Hera will return each time data are collected and passed through its sensor fusion. Data are obtained from the function : hera_data:get(nav3, sensor_fusion@nav_1). And it returns the last data from Hera. An output look like this: (4.19)

```
[{sensor_fusion@nav_1,166,32533,
                      [-0.01436184,-0.008377740000000002,-9.56618226,
                       6.144286187543222e-4,-3.7466401276141977e-4,
                       3.2070425005407873e-4,-0.00487900000000005,
                       0.003051999999999999,0.0144200000000001]}]
```

Figure 4.19: Output of hera_data:get(..)

As it returns only 1 data, it must be called many times and each time added to a list to have the evolution of a movement over time.

4.4.2 Classification over a period

We choose this algorithm because we will rapidly be able to see if everything is working as intended for the classification in real-time. It also removes a factor that could be a source of error. By defining the data collection time before classification, there can be no risk of the algorithm starting classification before having completed the movement. It also solves the question of letting the algorithm run indefinitely or not. By letting it run only over a defined period.

Implementation

For the implementation, a new module was created. It can be started via the *sensor_fusion* module. There is 2 main function. The first one is a simple count-down so the user can be prepared to do the gesture after he entered the instruction in the terminal. The second function collects the data from Hera. It looks in a loop if Hera has new data, and each time there is a new add it to the list of data.

After a defined time by the user, it calls the function already implemented for the classification from CSV. There was a little adaptation to use directly a given list rather than reading a CSV. It is still possible to read and classify a CSV (by adding "_CSV" at the end of the function). The path to get the file with all the learned gestures was also adapted to match the path on the GRiSP board.

Explanations on how to run the gesture recognition classification are written in the **Appendix A**.

The real-time classification is not automatically executed during the start of the sensor fusion because a user might want to do something else. When the real-time classification is launched, there is first a countdown so the user can be ready to perform the gesture. Then the algorithm will look in a loop at the timestamp of the last data given by Hera, if it is not the same timestamp as the last one, it's a new one, and it will be added to the list of data. It keeps all the next arriving data during the defined time. In the end, it performs a classification with all the data collected.

Testing

Using the classification algorithm and the real-time implementation, the output is like this: (4.20)

```
sensor_fusion:realtimeOnce(20).
Start Realtime with 20 seconds for the gesture
Countdown!
5
4
3
2
1
Move!
StartTime : 71197
Done!
Calculating...
NewX : [o,p,pp,p,o,p,pp,p,o]
NewY : [o]
NewZ : [nn,n,o,p,pp,p,o,n,nn]
N : halfcircle
X : 0.1111111111111111, Y : 1.0, Z : 0.1111111111111111
N : halfcircle
X : 0.1111111111111111, Y : 1.0, Z : 0.1111111111111111
N : rotationFront
X : 1.0, Y : 1.0, Z : 1.0
N : rotationBack
X : 0.1111111111111111, Y : 1.0, Z : 0.5555555555555556
Name : rotationFront, with Acc : 1.0
```

Figure 4.20: Output of the execution in real-time

For this experiment, the function was called in the terminal with 20 seconds

as input. It is as said earlier, the time to do the gesture. Then there is a print of the timestamp of the first data collected, it is to see that the simulation is still running. After 20 seconds, the collection of the data is stopped and there is a pattern-matching computation on the GRiSP. When it is finished it shows what was the pattern of this new gesture and the result of the computation. We see at the end, that it matches perfectly the rotation around the Y-axis because the same gesture was performed twice.

4.5 Classification by detecting a stop

To improve the real-time classification, rather than having to launch manually the detection each time, we want to do it only one time when we start and calibrate the GRiSP and it will detect continuously new gestures performed. For that, we need to change the way to detect or separate 2 gestures. We will separate 2 movements by not moving to indicate the end of a gesture. Like this, it will be very clear when we will start a new one. It will also remove the ambiguity of being between 2 gestures and not knowing if it's a new one, or the continuity of what you were doing. It will be important to tune well the parameters of the detection stop. It should not be detected during a movement.

Implementation

There is only one input, which is the number of times to wait before detecting inactivity and ending the gesture to perform the recognition of what was done. The default value is 2 seconds if a user doesn't put any argument in the function's call. The argument of the function is in seconds. When the function is starting, many variables are initialized to keep track of what is happening.

- **TimeOut**: it's the input value, it's the time to stay without moving.
- **Average Size**: A value that calls the global variable in the *learn* module, that indicates the number of data to take to calculate the most present pattern among those data. It is used to reduce the noise.
- **List**: It is a reduced list used to detect if the user stopped to move. There is also the variable **SizeL**, to efficiently count its size rather than using a function from the *lists*[39] module.
- **Gesture List**: The list of all the data obtained from Hera since the last gesture.
- **LastT**: Timestamp of the last data obtained from Hera, it is used to detect if there is new data available.

- **Time Since Move**: It is the timestamp of the last time a movement was detected, it means the last time when at least one of 3 axes changed its value.
- **LastX, LastY, LastZ**: Those variables keep track of the last value of the 3 axis, it is used to detect if the new value is different or not.

The whole function works like a loop.
First, it gets the last data from Hera, if it's not a new value, it just loops until it gets one. When it's a new value, it adds it to the 2 lists of gestures there are in this function, the list with all the data since the last gesture, and a small list that has a maximum size. Its maximum size is the value set to compute the most present pattern in a list of data. It is used to reduce the noise, because if noise is present among this list, it will not be very present and with the majority of good data, the output will give the good most present value in this subset. When the second list reaches its maximal size, it means we can compute the most present value in this list. It is done with a function already created for the learning and classification when only using CSV output. After the computation of the most present pattern in the list, it obtains 3 values, the pattern for the 3 axes, those values are compared to the value registered last time. If they aren't the same, the last value is updated to the new one it just gets, and the variable that keeps track since the last movement is also updated to the timestamp of the last value obtained from Hera. The small list is emptied and the loop restart. If the value is the same since the last time, it is then going to check if it has waited enough time without moving. If it's too soon it will just empty the list and start to loop again. But if it has waited enough time, the function will call the classification module and compute the nearest learned gesture. When it's finished it will show the result in the terminal, and then start again emptying both lists, the one with the data for the gesture, and the small one to detect a change, because they were just used.

Testing

We perform the same test as before but with this new function: (4.21)

```
Stop detected!
NewX : [o,pp,p,o,p,pp,p,o]
NewY : [o]
NewZ : [nn,n,o,p,pp,p,o,n,nn]
N : halfcircle
X : 0.0, Y : 1.0, Z : 0.1111111111111111
N : halfcircle
X : 0.0, Y : 1.0, Z : 0.1111111111111111
N : rotationFront
X : 0.1111111111111111, Y : 1.0, Z : 1.0
N : rotationBack
X : 0.25, Y : 1.0, Z : 0.5555555555555556
Name : rotationFront, with Acc : 0.7037037037037037

Stop detected!
NewX : [o]
NewY : [o]
NewZ : [nn]
N : halfcircle
X : 0.0, Y : 1.0, Z : 0.0
N : halfcircle
X : 0.0, Y : 1.0, Z : 0.0
N : rotationFront
X : 0.1111111111111111, Y : 1.0, Z : 0.1111111111111111
N : rotationBack
X : 0.16666666666666666, Y : 1.0, Z : 0.14285714285714285
Name : rotationBack, with Acc : 0.4365079365079365
```

Figure 4.21: Output of the real-time classification without time limit

The main change from before is now that the program doesn't stop after 1 execution, it will continue indefinitely until the GRiSP is shut down. A rotation gesture was performed, and it was matched with the good learned gesture. We then see that the algorithm didn't stop, and detected the next movement, where the GRiSP wasn't moved for 2 seconds. The classification was computed a new time and matched one of the learned gestures badly as there is no corresponding gesture learned for this second movement.

To avoid having a match with a low accuracy when a gesture shouldn't be recognized, a condition was added. If the accuracy of the new gesture is lower than 50% with all the learned gestures, it will indicate there isn't any match rather than giving a match with a low accuracy: (4.22)

```
N : rot_x_right_360
X : 0.0, Y : 0.23529411764705882, Z : 0.06666666666666667
Too low Accuracy, No gesture recognized
```

Figure 4.22: If the accuracy is too low, no gesture is recognized

Improvement

The algorithm will run indefinitely, so the only way to stop it is by unplugging the GRiSP from its energy source, and this is not *"user-friendly"*. An improvement to the algorithm is to let the user define a period for the algorithm. It will only run on this period and when this period ends, it will compute a last gesture and then stop. It will be possible to let the algorithm run indefinitely if it's what the user wants.

After, the implementation of the improvement, the function *realtime* takes a new argument which is the time limit in seconds, a user can set it to whatever he wants. If it is a negative value, the algorithm will run indefinitely, this can be useful to perform tests on a long period without needing to access a terminal. If you call the function without any argument, the default value will be 60 seconds. Also, there is a last classification with the data collected when the time runs out, before stopping the algorithm. There is nothing new to show for the improvement as it is only some internal change.

4.6 Learning in real-time

The improvement done is to learn a new gesture in real-time. It will be after a classification, a message will appears in the terminal and ask the user if he wants to add this gesture to the learned one. He will also have the possibility to choose the name of this gesture. It will only be implemented for the gesture recognition done 1 time (so *realtime_once*) because, for the endless method, the user should interact as little as possible with a terminal, where the number of interactions is the minimum, it's 1 time, to call the function. A part of the goal for this thesis is to have as little as possible of dependency, so the method that classifies multiples time should not be interrupted by the terminal after each classification. It will continue indefinitely or until the time limit. But because it is also important to be able to learn on real rather than passing through CSV, the *once* method will be adapted. A user already needs to interact with a terminal when it's finished, to start another gesture or stop the sensor fusion, so it doesn't add a new dependency to the terminal.

4.6.1 Implementation

For the implementation, a new function was implemented, it is called at the end of *realtime_once* after the classification. It will ask the user if he wants to learn the performed gesture, if it is *yes*, the user will then be asked to enter the name of this new gesture. If it doesn't answer (by pressing *enter*) or writes *no*, it ends like

before. Some adaptations for the learning were made about reading the input and using the file path of the GRiSP.

4.6.2 Testing

The experiment was done by performing a real-time classification and just leaving the GRiSP on the table without moving. After the classification, the gesture was learned with the new feature. Then another classification was executed without moving again. We don't move because we only want to see that the learning feature is working as intended. The terminal output : (4.23)

```
Do you want to learn this gesture? (y/n/ENTER) : y
y
What is the name of the gesture (use _ for space) : no_move
no_move
ok
```

(a) 1st execution after the classification

```
N : rotationFront
X : 0.1111111111111111, Y : 1.0, Z : 0.1111111111111111
N : rotationBack
X : 0.16666666666666666, Y : 1.0, Z : 0.14285714285714285
N : no_move
X : 1.0, Y : 1.0, Z : 1.0
```

(b) Classification of the 2nd execution

Figure 4.23: The learning in real-time

During the first execution, the user writes **y** to indicate that he wants to add this gesture, then he writes **no_move**. Like this, the gesture was added with the name **no_move** to the list of learned gestures. Then during the classification of the second gesture, we see that the first gesture (that was learned) is compared like all the other learned gestures by the algorithm. It is a perfect match between the 2 gestures.

Chapter 5

Conclusion

This chapter summarizes the contribution of this thesis in the results section and explains future works' ideas to extend the gesture recognition algorithm.

5.1 Results

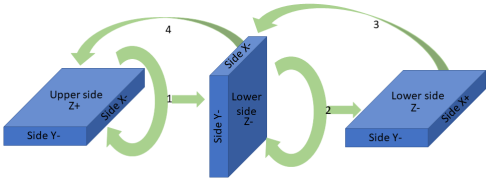
We manage to create an algorithm that recognizes gestures using a sensor fusion. The data obtained from an accelerometer, a gyroscope, and a magnetometer are combined and passed through a Kalman filter to give an output that is robust to noise and failure. Those sensor data are obtained from a PMOD-NAV attached to a GRiSP2 device, this computation is done directly on this Internet of Things device. The GRiSP is attached to a human wrist or held in a hand, a gesture is thus a movement or a rotation of a human arm in the environment. The gesture will be relative to the rotation of the GRiSP during the calibration. To look at the result of the gesture recognition classification, there is 2 way, the first one is to start the sensor fusion and connect a GRiSP to a computer where a GRiSP simulator is running, the data obtained during the run-time will be transferred to the computer in a CSV file. When the gesture is finished and the data collection is stopped, it is possible to compute the classification on the computer by using the CSV as input. But one of the goals of this thesis is to have as little as possible dependency, so another possibility is to do the computation for the classification of the gesture recognition in real-time on the GRiSP. However, it still needs a terminal to start it and look at the result as the is no visual interface on the GRiSP. From a connected terminal to the GRiSP, multiple operations can be performed, it is possible to start a gesture recognition one time, so just performing one movement and see how it will be classified with the learned gesture, it is also possible to learn the gesture that was just performed for the next classifications. A user can also start multiple classifications, performing different gestures in succession and see for each of them

the result of the classification, it can be run endlessly or stopped after a certain time. The learned gesture can be viewed and edited via a file saved on the GRiSP, it is also possible to clear all the gestures on the GRiSP from the terminal.

An experiment was performed with the final version of gesture recognition. 4 gestures were learned with real-time features: A 180° rotation around the Y-axis by going down then returning to the initial position, a 90° rotation around the X by going to the left the returning to the initial position, a 360° rotation around the X-axis by going to the right, and a *no move* gesture by letting the GRiSP on the table. Each gesture was learned 2 times, and after they were all learned, they were performed a third time for the classification: (5.1, 5.2, 5.3, 5.4)

```
Calculating...
NewX : [o]
NewY : [o]
NewZ : [nn]
N : no_move
X : 1.0, Y : 1.0, Z : 1.0
N : rot_Y_down_180
X : 0.1111111111111111, Y : 1.0, Z : 0.1111111111111111
N : rot_Y_down_180
X : 0.125, Y : 1.0, Z : 0.125
N : rot_x_left_90
X : 1.0, Y : 0.2, Z : 0.2
N : rot_x_left_90
X : 1.0, Y : 0.16666666666666666, Z : 0.2
N : rot_x_right_360
X : 1.0, Y : 0.1111111111111111, Z : 0.1111111111111111
N : rot_x_right_360
X : 1.0, Y : 0.1111111111111111, Z : 0.1111111111111111
Name : no_move, with Acc : 1.0
```

Figure 5.1: Classification of *no move*

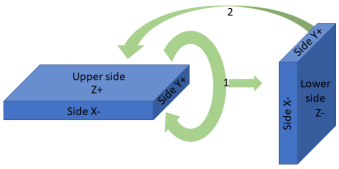


(a) Gesture of *rot Y down 180*

```
Calculating...
NewX : [o,n,nn,n,o,n,nn,n,o]
NewY : [o]
NewZ : [nn,n,o,p,pp,p,o,n,nn]
N : no_move
X : 0.1111111111111111, Y : 1.0, Z : 0.1111111111111111
N : rot_Y_down_180
X : 1.0, Y : 1.0, Z : 1.0
N : rot_Y_down_180
X : 0.3333333333333333, Y : 1.0, Z : 0.1111111111111111
N : rot_x_left_90
X : 0.1111111111111111, Y : 0.2, Z : 0.3333333333333333
N : rot_x_left_90
X : 0.1111111111111111, Y : 0.16666666666666666, Z : 0.3333333333333333
N : rot_x_right_360
X : 0.1111111111111111, Y : 0.1111111111111111, Z : 1.0
N : rot_x_right_360
X : 0.1111111111111111, Y : 0.1111111111111111, Z : 1.0
Name : rot_Y_down_180, with Acc : 1.0
```

(b) Classification of *rot Y down 180*

Figure 5.2

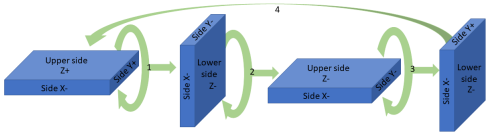


(a) Gesture of *rot X left 90*

```
Calculating...
NewX : [o]
NewY : [o,p,pp,p,o]
NewZ : [nn,o,n,nn]
N : no_move
X : 1.0, Y : 0.2, Z : 0.25
N : rot_Y_down_180
X : 0.1111111111111111, Y : 0.2, Z : 0.1111111111111111
N : rot_Y_down_180
X : 0.125, Y : 0.2, Z : 0.25
N : rot_x_left_90
X : 1.0, Y : 1.0, Z : 0.2
N : rot_x_left_90
X : 1.0, Y : 0.3333333333333333, Z : 0.2
N : rot_x_right_360
X : 1.0, Y : 0.2222222222222222, Z : 0.1111111111111111
N : rot_x_right_360
X : 1.0, Y : 0.2222222222222222, Z : 0.1111111111111111
Name : rot_x_left_90, with Acc : 0.7333333333333334
```

(b) Classification of *rot X left 90*

Figure 5.3



(a) Gesture of *rot X right 360*

```
Calculating...
NewX : [o]
NewY : [o,n,nn,n,o,p,pp,p,o]
NewZ : [nn,n,o,p,pp,p,o,n,nn]
N : no_move
X : 1.0, Y : 0.1111111111111111, Z : 0.1111111111111111
N : rot_Y_down_180
X : 0.1111111111111111, Y : 0.1111111111111111, Z : 1.0
N : rot_Y_down_180
X : 0.125, Y : 0.1111111111111111, Z : 0.1111111111111111
N : rot_x_left_90
X : 1.0, Y : 0.2222222222222222, Z : 0.3333333333333333
N : rot_x_left_90
X : 1.0, Y : 0.1111111111111111, Z : 0.3333333333333333
N : rot_x_right_360
X : 1.0, Y : 1.0, Z : 1.0
N : rot_x_right_360
X : 1.0, Y : 1.0, Z : 1.0
Name : rot_x_right_360, with Acc : 1.0
```

(b) Classification of *rot X right 360*

Figure 5.4

We can see that each gesture was well classified into one of its corresponding gestures learned by the algorithm.

5.2 Future works

5.2.1 Continuous gesture recognition

The actual version of the algorithm that detects several gestures in succession still needs a clear indication to separate 2 gestures. The user needs to stop moving for a certain time, it is 2 seconds by default but another value can be specified. However, in a human environment where the IoT device that runs the gesture recognition is attached to the wrist of a person, it will never stop moving. Thus an important future work is to be able to detect a gesture without the user specifically specifying the beginning or end of it.

There are multiple challenges to overcome, one of the biggest will be to distinguish 2 gestures that have a part of their pattern that is the same and knowing

without ambiguity which gesture was performed. It should also detect when the user is just moving around without performing a precise gesture. There is also the problem that if there is a gesture that has all its pattern that is only a part of another more complex gesture, the algorithm should distinguish without a doubt if just the small gesture was performed or if it was the complex one.

5.2.2 Gesture recognition from the velocity

For the moment, the gesture recognition algorithm is using acceleration over the 3 axes, which means it detects very well when we are turning the GRiSP around but will be less efficient if we don't rotate the GRiSP when doing the gesture. A solution can then be to use velocity rather than acceleration. But for the moment it is not usable with only a GRiSP equipped with a PMOD-NAV. Using the test called *e13* where the velocity is used, also requires other GRiSPs with PMOD-MAXSONAR. It will work without a crash, but the data collected and computation for the velocity and position can not be corrected by PMOD-MAXSONAR, so if we run the test without moving we obtain those values for the velocity on the 3 axes: (5.5)

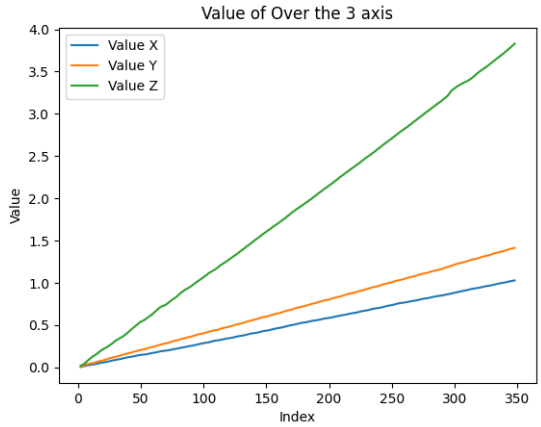


Figure 5.5: Velocity values for the 3 axes when not moving

We see that without the correction from the PMOD-MAXSONAR, the value is increasingly rising when it should be not changing and staying at 0. We can try another test where we will perform a simple gesture. Going up and then down with GRiSP, like this we could see how are evolving the value, as it is maybe just some small error that doesn't interfere much with a real gesture.

Doing this test results in those values: (5.6)

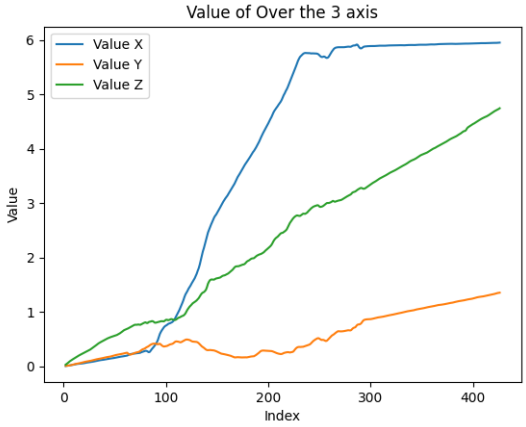


Figure 5.6: Velocity values for the 3 axes when going up and down

With this new test, we see that those weren't small noise values as they heavily impact the value. The moment where we are supposed to see the descent is not visible, theoretically, we should observe a negative velocity. So it is currently impossible to use a GRiSP with only a PMOD-NAV to obtain the velocity from Hera. And this is why it can be a good improvement to do, as a whole new set of gestures will be possible.

5.2.3 Communication between devices

Another improvement that is the next step of this series master thesis, is to add communication between devices. Perform a gesture with a device and it will communicate an instruction to another device. As communication between devices is already possible it should be possible to improve and extend to any other IoT that can receive information or instruction from an external source. The version of the algorithm that is running in real-time endlessly should work well in a real environment where if a user just moves without precision it will just find bad matches to gesture. But if the user, for example, points to an IoT device, to indicate that you want to send an instruction, then he performs a precise gesture and it will be recognized, and the instruction will be sent to the other IoT device.

5.2.4 Adding sound sensor as input

A possible improvement talked about in the background chapter, is to add a new sensor as input. It can be *sound* for example, so combining a gesture and a sound to

diversify possibilities. It is important to be careful not to create a new dependency or add a language barrier. But if it's done right, it'll be a major improvement that could help people who have difficulty moving around but no voice problems.

5.2.5 Merging with the optimized matrix libraries

Another student, Lucas Nélis, also started his thesis on the improved Hera sensor fusion and worked in parallel with this thesis. His goal was to improve the performance of Hera by optimizing the matrix libraries in particular, which have been rewritten and optimized in C. More detailed information can be found on his Github[40]. A future work to do is then to merge this thesis about gesture recognition and the optimization of Lucas. To obtain more data per second from Hera and improve the classification quality.

Chapter 6

Bibliography

[1] Wikipedia contributors. "Microcomputer." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Microcomputer

[2] Erlang language website: https://www.erlang.org/

[3] Wikipedia contributors. "Erlang (Programming Language)." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Erlang_(programming_language)

[4] GRiSP 2: Diving Deeper Into Embedded Systems — GRiSP: https://www.grisp.org/

[5] The GRiSP 2 Kickstarter - Specially designed for Embedded Erlang and Elixir: https://www.kickstarter.com/projects/peerstritzinger/grisp-2

[6] The GRiSP online shop: https://www.grisp.org/shop/

[7] The GRiSP technical specification webpage: https://www.grisp.org/specs/

[8] S. Mitra and T. Acharya, "Gesture Recognition: A Survey," in *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews),* vol. 37, no. 3, pp. 311-324, May 2007, doi: 10.1109/TSMCC.2007.893280.: https://ieeexplore.ieee.org/abstract/document/4154947

[9] Wikipedia contributors. "Hidden Markov Model." Wikipedia, Mar. 2023: https://en.wikipedia.org/wiki/Hidden_Markov_model

[10] Ji-Hae Kim, Gwang-Soo Hong, Byung-Gyu Kim, Debi P. Dogra, *deepGesture: Deep learning-based gesture recognition scheme using motion sensors, Displays,* Volume 55, 2018, Pages 38-45, ISSN 0141-9382: https://www.sciencedirect.com/science/article/abs/pii/S0141938217302032

[11] G. Li, H. Wu, G. Jiang, S. Xu and H. Liu, "Dynamic Gesture Recognition in the Internet of Things," in *IEEE Access,* vol. 7, pp. 23713-23724, 2019, doi: 10.1109/ACCESS.2018.2887223. : https://ieeexplore.ieee.org/abstract/document/8580553

[12] Wikipedia contributors. "Dempster–Shafer Theory." Wikipedia, Mar. 2023: https://en.wikipedia.org/wiki/Dempster%E2%80%93Shafer_theory

[13] R. Min, X. Wang, J. Zou, J. Gao, L. Wang and Z. Cao, "Early Gesture Recognition With Reliable Accuracy Based on High-Resolution IoT Radar Sensors," in *IEEE Internet of Things Journal,* vol. 8, no. 20, pp. 15396-15406, 15 Oct.15, 2021, doi: 10.1109/JIOT.2021.3072169. : https://ieeexplore.ieee.org/abstract/document/9399656

[14] Jiayang Liu, Lin Zhong, Jehan Wickramasuriya, Venu Vasudevan, uWave: Accelerometer-based personalized gesture recognition and its applications, *Pervasive and Mobile Computing,* Volume 5, Issue 6, 2009, Pages 657-675, ISSN 1574-1192 : https://www.sciencedirect.com/science/article/abs/pii/S1574119209000674

[15] Wikipedia contributors. "Human–computer Interaction." Wikipedia, Apr. 2023: https://en.wikipedia.org/wiki/Human%E2%80%93computer_interaction

[16] Eisenstein, J., & Davis, R. (2007). Visual and linguistic information in gesture classification. In *ACM SIGGRAPH 2007 courses* (pp. 15-es). : https://dl.acm.org/doi/abs/10.1145/1281500.1281526

[17] Wikipedia contributors. "Cohen's Kappa." Wikipedia, Dec. 2022: https://en.wikipedia.org/wiki/Cohen%27s_kappa

[18] Wikipedia contributors. "Support Vector Machine." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Support_vector_machine

[19] Wikipedia contributors. "Naive Bayes Classifier." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[20] Neirinckx, G., Bastin, J., & Van Roy, P. (2020). Sensor fusion at the extreme edge of an internet of things network (Doctoral dissertation, Master's thesis. UCLouvain. http://hdl. handle. net/2078.1/thesis: 26491): https://dial.uclouvain.be/downloader/downloader.php?pid=thesis%3A26491&datastream=PDF_01&cover=cover-mem

[21] Kalbusch, S., Verpoten, V., & Van Roy, P. (2021). The Hera framework for fault-tolerant sensor fusion on an Internet of Things network with

application to inertial navigation and tracking (Doctoral dissertation, Master's thesis. UCLouvain. http://hdl. handle. net/2078.1/thesis: 30740): https://www.info.ucl.ac.be/ pvr/Kalbusch_22701600_Verpoten_61101500_2021.pdf

[22] Wikipedia contributors. "Kalman Filter." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Kalman_filter

[23] "Understanding Kalman Filters." MATLAB: https://it.mathworks.com/videos/series/understanding-kalman-filters.html

[24] Grisp. "Linux Development Environment." GitHub: https://github.com/grisp/grisp/wiki/Linux-Development-Environment

[25] "GRiSP." GitHub: https://github.com/grisp

[26] Wikipedia contributors. "Decision Tree." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Decision_tree

[27] Y. Ren and F. Zhang, "Hand Gesture Recognition Based on MEB-SVM," *2009 International Conference on Embedded Software and Systems,* Hangzhou, China, 2009, pp. 344-349, doi: 10.1109/ICESS.2009.21. : https://ieeexplore.ieee.org/abstract/document/5066667

[28] Wikipedia contributors. "K-means Clustering." en.wikipedia.org, May 2023: https://en.wikipedia.org/wiki/K-means_clustering

[29] Dimos Michailidis. "GitHub - Mrdimosthenis/Emel: A Simple and Functional Machine Learning Library for the Erlang Ecosystem." GitHub: https://github.com/mrdimosthenis/emel

[30] Wikipedia contributors. "LIBSVM." Wikipedia, May 2023: https://en.wikipedia.org/wiki/LIBSVM

[31] Wikipedia contributors. "Unsupervised Learning." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Unsupervised_learning

[32] Wikipedia contributors. "Supervised Learning." Wikipedia, Mar. 2023: https://en.wikipedia.org/wiki/Supervised_learning

[33] Wikipedia contributors. "Neural Network." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Neural_network

[34] Wikipedia contributors. "Dynamic Time Warping." Wikipedia, Apr. 2023: https://en.wikipedia.org/wiki/Dynamic_time_warping

[35] Rotating a 3D Model Before Conversion With reaConverter: https://www.reaconverter.com/features/rotate-3d-images.html

[36] Wikipedia contributors. "Magnitude (Mathematics)." Wikipedia, May 2023: https://en.wikipedia.org/wiki/Magnitude_(mathematics)

[37] Erlang – File: https://www.erlang.org/doc/man/file.html

[38] Wikipedia contributors. "Guard Interval." Wikipedia, May 2022: https://en.wikipedia.org/wiki/Guard_interval

[39] Erlang – Lists: https://www.erlang.org/doc/man/lists.html

[40] Lucas Nélis. "GitHub - Lunelis/Sensor_Fusion: Aims to Port the Sensor_Fusion Library to Grisp2 and Improve It's Performances." GitHub: https://github.com/lunelis/sensor_fusion

[41] GRiSP Slack: https://erlanger.slack.com/channels/grisp

[42] Sébastien Kalbush. "Sensor_Fusion/README.md at Master · Sebkm/Sensor_Fusion." GitHub: https://github.com/sebkm/sensor_fusion/blob/master/README.md

[43] Sébastien Gios. "GitHub - Roulalou/Sensor_Fusion: Create a Gesture Recognition Algorithm From the Sensor Fusion and Port It to the GRiSP2." GitHub: https://github.com/Roulalou/sensor_fusion

[44] Grisp. "Connecting Over WiFI and Ethernet." GitHub: https://github.com/grisp/grisp/wiki/Connecting-over-WiFI-and-Ethernet

Appendix A

How to use the gesture recognition

The configuration used during the thesis is:

- Ubuntu 20.04
- Erlang 25.2.3.
- Rebar3 3.19
- Rebar3_hex 7.0.4
- Rebar3_grisp 2.4.0

It could have changed with time, you can look at the requirement in the GRiSP tutorial[24] or ask a question on Slack[41]. Some parts of those steps are explained in more depth in the *readme*[42] of the previous thesis:

- **1st:** you will need a development environment, which can be achieved by following the tutorials on the GRiSP wiki[24]
- **2nd:** get the project from the Github[43].
- **3rd:** adapt the file **wpa_supplicant.conf** under **sensor_fusion/grisp/grisp2/common/deploy/files/** to your internet configuration.
- **4th:** compile the program with: **make deploy-nav_1**, you might need to change **rebar.config** to deploy on your micro-SD location.
- **5th:** plug the micro-SD in your GRiSP equipped with PMOD-NAV and link it to your computer (or you can do it remotely, see here[44])

- **6th:** in a terminal execute this command:
  sudo picocom /dev/ttyUSB1 –baud 115200 –echo to interact with the GRiSP.

- **7th:** when the booting phase is finished, calibrate the sensor and follow the steps that are shown: **sensor_fusion:set_args(nav3).**

- **8th:** when the calibration is finished, start the sensor fusion: **sensor_fusion:launch().**

- **9th:** then start the real-time classification : **sensor_fusion:realtime_once()**.
  To classify 1 gesture or the default 10-second timer for the movement, you can also set the time in seconds between the parenthesis. After the classification, you will be asked if you want to add the gesture you just perform.

- **9th bis:** call **sensor_fusion:realtime()** to classify multiple gesture during the default 60 seconds. Two arguments can be set, the first on the time in seconds to stop and detect the end of a gesture, and the second on the running time of the algorithm in seconds, a negative number means indefinitely.

To clean all the learned gestures from the GRiSP you can execute:
sensor_fusion:clean_gesture().
The learned gestures can also be manually edited in the file **gesture** under **src/**.

Appendix B

How to improve the gesture recognition algorithm

It is also possible to directly improve the gesture recognition algorithm or functions around it. All functions are written in Erlang and located in the same folder, under **src/** in the GitHub repository[43]. Any file can be improved and tested on the GRiSP. Instructions on how to use the GRiSP are written in **Appendix A**, the *readme* of the Github, and in the *readme*[42] of the previous master thesis.
All the functions for the gesture recognition were regrouped under a few files:

- **sensor_fusion.erl**: to, among other things, start Hera and the gesture recognition in real-time or clear the learned gestures.
- **csvparser.erl**: to parse CSV file or matrix (list of lists) and obtained a requested column in a list.
- **learn.erl**: regroup all the functions used for the learning part of the algorithm.
- **classify.erl**: regroup all the functions used for the classification part of the algorithm.
- **realtime.erl**: regroup all the functions used for the real-time part of the algorithm.
- **gesture**: it is the file with the learned gesture, if not empty, the gestures in the file will be already learned when the algorithm will run in real-time on the GRiSP.
- **analyze.py**: is not in **src/** but in the root. It's to plot graphs of the CSV.

Appendix C

Source code

C.1 Sensor_fusion.erl

```
% sensor_fusion.erl
-module(sensor_fusion).

-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]).
-export([realtime/0, realtime/2, realtime_once/0, realtime_once/1,
    clear_gesture/0]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 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
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() ->
    try launch(node_type()) of
        ok ->
            [grisp_led:color(L, green) || L <- [1, 2]],
            ok
    catch
        error:badarg ->
            [grisp_led:color(L, red) || L <- [1, 2]],
            {error, badarg}
    end.


launch_all() ->
    rpc:multicall(?MODULE, launch, []).


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).

realtime() ->
    io:format("Start Realtime with 3 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("sensor_fusion/lib/sensor_fusion-1.0.0/src/gesture",
        ""),
    io:format("Clear gesture~n",[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Callbacks
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
start(_Type, _Args) ->
    {ok, Supervisor} = sensor_fusion_sup:start_link(),
    init_table(),
    case node_type() of
        nav ->
            _ = grisp:add_device(spi2, pmod_nav);
        sonar ->
            _ = grisp:add_device(uart, pmod_maxsonar),
            pmod_maxsonar:set_mode(single);
        _ -> % 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.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Internal functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

node_type() ->
    Host = lists:nthtail(14, atom_to_list(node())),
    IsNav = lists:prefix("nav", Host),
    IsSonar = lists:prefix("sonar", Host),
    if
        IsNav -> nav;
        IsSonar -> sonar;
        true -> undefined
    end.


launch(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),
    R0 = ets:lookup_element(args, {e11, node()}, 2),
    % {ok,_} = hera:start_measure(nav, Cn),
    % {ok,_} = hera:start_measure(mag, Cm),
    {ok,_} = hera:start_measure(nav3, Cn),
```

```
    {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.
```

```
% csvparser.erl
-module(csvparser).

-export([parse_CSV/2, parse/2, print_list/1]).

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

% CSV : "../measures/Anav3_sensor_fusion@nav_1.csv"
% return a list of the column Index
% For nav_3 acceleration are in 3, 4, 5
parse_CSV(CSV, Index) ->
    Lines = parser(CSV),
    CL = clean_lines(Lines, Index),
    CL.
```

```
% Used by realtime.erl
% return a list of the column Index
% For nav_3 acceleration are in 1, 2, 3
parse(List, Index) ->
    CL = clean_lines(List, Index),
    CL.

% return a list of the lines
parser(CSV) ->
    {_, Data} = file:read_file(CSV),
    Rows = string:tokens(binary_to_list(Data), "\n"),
    Records = [string:tokens(Row, ",") || Row <- Rows],
    Records.

% Useful for debugging, print a list
print_list([H|T]) ->
    io:format("H : ~p~n", [H]),
    print_list(T);
print_list([]) ->
    ok.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Internal functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% return a list of the column Index
clean_lines(List, Index) ->
    clean_lines(List, Index, []).
clean_lines(List, Index, AccX) ->
    case List of
        [] -> AccX;
        [H|T] ->
            Just_AccX_H = lists:nth(Index, H),
            New_H = lists:append(AccX, [Just_AccX_H]),
            clean_lines(T, Index, New_H)
    end.
```

C.2 Learn.erl

```
% learn.erl
-module(learn).

-export([to_file/1, learn/2, learn_CSV/2, analyze/1, analyze_CSV/1,
    regroup/1, average/1, av_size/0]).
-import(csvparser, [parse_CSV/2, print_list/1]). % can also use
    csvparser:parse(..) instead of import
-define(AXIS, [x, y, z]).
-define(AV_SIZE, 50). % can be tuned depending on the quality of the
    results

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

% Called by realtime.erl
% add a new gesture to the gesture file
learn(List, Name) ->
    % learn for the 3 axis
    learn_axis(List, Name, lists:nth(1, ?AXIS)), %lists:nth(1, ?AXIS) = x
    learn_axis(List, Name, lists:nth(2, ?AXIS)),
    learn_axis(List, Name, lists:nth(3, ?AXIS)).

% CSV : "../measures/bf1.csv"
% example : learn:learn("../measures/bf1.csv", test).
% add a new gesture to the gesture file
learn_CSV(CSV, Name) ->
    % learn for the 3 axis
    learn_axis_CSV(CSV, Name, lists:nth(1, ?AXIS)),
    learn_axis_CSV(CSV, Name, lists:nth(2, ?AXIS)),
    learn_axis_CSV(CSV, Name, lists:nth(3, ?AXIS)).

% analyze the list of acc and determine the pattern FOR REALTIME
analyze(Vector) ->
    analyze(Vector, []).
analyze(Vector, Pattern) ->
    case Vector of
        [] -> Pattern;
        [H|T] ->
            % Rules of patterns
            % It is arbitrary values
```

```
            if H < -7 ->
                analyze(T, lists:append(Pattern, [nn])); % nn : negative
                    high
            H < -4 ->
                analyze(T, lists:append(Pattern, [n])); % n : negative low
            H < -2 ->
                analyze(T, Pattern); % Guard Zone
            H > 7 ->
                analyze(T, lists:append(Pattern, [pp])); % pp : positive
                    high
            H > 4 ->
                analyze(T, lists:append(Pattern, [p])); % p : positive low
            H > 2 ->
                analyze(T, Pattern); % Guard Zone
            true ->
                analyze(T, lists:append(Pattern, [o])) % o : zero
            end
    end.

% analyze_CSV the list of acc and determine the pattern
analyze_CSV(Vector) ->
    analyze_CSV(Vector, []).
analyze_CSV(Vector, Pattern) ->
    case Vector of
        [] -> Pattern;
        [H|T] ->
            % Rules of patterns
            {Int_H, _} = string:to_integer(H),
            % It is arbitrary values
            if Int_H < -7 ->
                analyze_CSV(T, lists:append(Pattern, [nn])); % nn :
                    negative high
            Int_H < -4 ->
                analyze_CSV(T, lists:append(Pattern, [n])); % n :
                    negative low
            Int_H < -2 ->
                analyze_CSV(T, Pattern); % Guard Zone
            Int_H > 7 ->
                analyze_CSV(T, lists:append(Pattern, [pp])); % pp :
                    positive high
            Int_H > 4 ->
                analyze_CSV(T, lists:append(Pattern, [p])); % p :
                    positive low
```

```
            Int_H > 2 ->
                analyze_CSV(T, Pattern); % Guard Zone
            true ->
                analyze_CSV(T, lists:append(Pattern, [o])) % o : zero
            end
    end.

% regroup the pattern to have the general flow
regroup(Pattern) ->
    regroup(Pattern, []).
regroup(Pattern, Flow) ->
    case Pattern of
        [] -> Flow;
        [H|T] ->
            if T == [] -> % If last element
                regroup(T, lists:append(Flow, [H]));
            true ->
                [HT|_] = T,
                Next = HT, % Head Tail
                if H == Next ->
                    regroup(T, Flow);
                true ->
                    regroup(T, lists:append(Flow, [H]))
                end
            end
    end.

% return a list of average value, each time over AV_SIZE data
average(List) ->
    average(List, ?AV_SIZE, []).
average(List, Size, New_L) ->
    if length(List) =< Size ->
        Av = calculate_av(List),
        lists:append(New_L, [Av]); % Return the list of average
    true ->
        Sub_List = lists:sublist(List, Size),
        Av = calculate_av(Sub_List),
        Next_L = lists:sublist(List, Size + 1, length(List)),
        average(Next_L, Size, lists:append(New_L, [Av]))
    end.

% export the gesture to the file
to_file(Gesture) ->
```

```
    file:write_file("sensor_fusion/lib/sensor_fusion-1.0.0/src/gesture",
        io_lib:fwrite("~p\n", [Gesture]), [append]).

% export AV_SIZE for others modules
av_size() ->
    ?AV_SIZE.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Internal functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% hardcoded for : nn, n, pp, p, zero
calculate_av(List) ->
    calculate_av(List, 0, 0, 0, 0, 0).
calculate_av(List, NegH, NegL, PosH, PosL, Zero) ->
    case List of
        [] ->
            if NegL > NegH, NegL > PosH, NegL > PosL, NegL > Zero ->
                n;
            PosL > NegH, PosL > PosH, PosL > NegL, PosL > Zero ->
                p;
            NegH > NegL, NegH > PosH, NegH > PosL, NegH > Zero ->
                nn;
            PosH > NegH, PosH > NegL, PosH > PosL, PosH > Zero ->
                pp;
            true ->
                o
            end;
        [H|T] ->
            case H of
                nn -> calculate_av(T, NegH + 1, NegL, PosH, PosL, Zero);
                n -> calculate_av(T, NegH, NegL + 1, PosH, PosL, Zero);
                pp -> calculate_av(T, NegH, NegL, PosH + 1, PosL, Zero);
                p -> calculate_av(T, NegH, NegL, PosH, PosL + 1, Zero);
                o -> calculate_av(T, NegH, NegL, PosH, PosL, Zero + 1)
            end
    end.

% Called by learn() for a specific axis
learn_axis(List, Name, Axis) ->
    case Axis of
        x ->
            Index = 1;
```

```
        y ->
            Index = 2;
        z ->
            Index = 3
    end,
    Vector = csvparser:parse(List, Index),
    Pattern = analyze(Vector),
    Clean_Pat = average(Pattern),
    Flow = regroup(Clean_Pat),
    Gesture = lists:append([Name, Axis], Flow),
    to_file(Gesture).

% Called by learn() for a specific axis
learn_axis_CSV(CSV, Name, Axis) ->
    case Axis of
        x ->
            Index = 3;
        y ->
            Index = 4;
        z ->
            Index = 5
    end,
    Vector = parse_CSV(CSV, Index),
    Pattern = analyze_CSV(Vector),
    Clean_Pat = average(Pattern),
    Flow = regroup(Clean_Pat),
    Gesture = lists:append([Name, Axis], Flow),
    to_file_CSV(Gesture).

% export the gesture to the file
to_file_CSV(Gesture) ->
    file:write_file("gesture", io_lib:fwrite("~p\n", [Gesture]),
        [append]).
```

C.3 Classify.erl

```
% classify.erl
-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),
    Clean_PatX = average(PatternX),
    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 >= 0.5 ->
```

```
        io:format("Name : ~p, with Acc : ~p~n", [Name, Accuracy]);
    true ->
        io:format("Too low Accuracy, No gesture recognized~n")
    end.

% CSV : "../measures/hc1.csv"
classify_new_gesture_CSV(CSV) ->
    List_gestures = import_gesture_CSV(),

    VectorX = parse_CSV(CSV, 3), % 3 is the index of the x axis
        acceleration
    PatternX = analyze_CSV(VectorX),
    Clean_PatX = average(PatternX),
    NewX = regroup(Clean_PatX), % New is the general flow of the new
        gesture

    VectorY = parse_CSV(CSV, 4), % 4 is the index of the y axis
        acceleration
    PatternY = analyze_CSV(VectorY),
    Clean_PatY = average(PatternY),
    NewY = regroup(Clean_PatY), % New is the general flow of the new
        gesture

    VectorZ = parse_CSV(CSV, 5), % 5 is the index of the z axis
        acceleration
    PatternZ = analyze_CSV(VectorZ),
    Clean_PatZ = average(PatternZ),
    NewZ = regroup(Clean_PatZ), % New is the general flow of the new
        gesture

    {Name, Accuracy} = compare_gesture(NewX, NewY, NewZ, List_gestures),

    % for the moment a simple print
    io:format("Name : ~p, with Acc : ~p~n", [Name, Accuracy]).

% For execution on the GRiSP board
import_gesture() ->
    {_, Data} =
        file:read_file("sensor_fusion/lib/sensor_fusion-1.0.0/src/gesture"),
    Gestures = string:tokens(binary_to_list(Data), "\n"),

    Cleaned_Gestures = [string:substr(G, 2, length(G)-2) || G <-
        Gestures],
```

```
    List_Gestures = [str_to_atom_list(G) || G <- Cleaned_Gestures],
    List_Gestures.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Internal functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% import the gesture file to a list of gesture
import_gesture_CSV() ->
    {_, Data} = file:read_file("gesture"),
    Gestures = string:tokens(binary_to_list(Data), "\n"),

    Cleaned_Gestures = [string:substr(G, 2, length(G)-2) || G <-
        Gestures],
    List_Gestures = [str_to_atom_list(G) || G <- Cleaned_Gestures],
    List_Gestures.

% String to list of atoms
str_to_atom_list(Str) ->
    [list_to_atom(E) || E <-string:tokens(Str,",")].

% compare the new gesture to the list of gestures
compare_gesture(NewX, NewY, NewZ, List_gestures) ->
    compare_gesture(NewX, NewY, NewZ, List_gestures, none, 0).
compare_gesture(NewX, NewY, NewZ, List_gestures, Name, Accuracy) ->
    case List_gestures of
        [] -> {Name, Accuracy}; % Empty list
        [H|T] ->
            % Take the 3 next list of flow, to have the 3 axis
            [GName|_] = H, % GName = Gesture Name
            TriList = lists:sublist([H|T], 3),
            io:format("N : ~p~n", [GName]),
            New_Accuracy = tri_compare(NewX, NewY, NewZ, TriList),
            Next_G = lists:sublist([H|T], 4, length([H|T])), % remove
                the 3 axis for the gesture compared
            if New_Accuracy > Accuracy ->
                compare_gesture(NewX, NewY, NewZ, Next_G, GName,
                    New_Accuracy);
            true ->
                compare_gesture(NewX, NewY, NewZ, Next_G, Name, Accuracy)
            end
    end.
```

```
% compare the 3 axis
tri_compare(NewX, NewY, NewZ, TriList) ->
    GXT = lists:nth(1, TriList), % Gesture X Axis, with TOO MUCH
        variables
    GX = lists:sublist(GXT, 3, length(GXT)),
    GYT = lists:nth(2, TriList),
    GY = lists:sublist(GYT, 3, length(GYT)),
    GZT = lists:nth(3, TriList),
    GZ = lists:sublist(GZT, 3, length(GZT)),
    Acc_X = direct_compare(NewX, GX),
    Acc_Y = direct_compare(NewY, GY),
    Acc_Z = direct_compare(NewZ, GZ),
    io:format("X : ~p, Y : ~p, Z : ~p~n", [Acc_X, Acc_Y, Acc_Z]),
    Acc = (Acc_X + Acc_Y + Acc_Z) / 3, % Average over the 3 axis
    Acc.


% compare NEW to 1 GESTURE
direct_compare(New, Gesture) ->
    direct_compare(New, Gesture, 0, 0).
direct_compare(New, Gesture, Okay, Comparison) ->
    % For the moment not very opti, we look if a list is empty and then
        we return the accuracy


    if New == [] ->
        Total_Comp = finish_list(Gesture, 0) + Comparison,
        Okay/Total_Comp;
    true ->
        if Gesture == [] ->
            Total_Comp = finish_list(New, 0) + Comparison,
            Okay/Total_Comp;
        true ->
            [NH|NT] = New,
            [GH|GT] = Gesture,
            if NH == GH ->
                direct_compare(NT, GT, Okay+1, Comparison+1);
            true ->
                direct_compare(NT, GT, Okay, Comparison+1)
            end
        end
    end.
```

```
% Count the number of elements in a list
finish_list(List, N) ->
    case List of
        [] -> N;
        [_|T] -> finish_list(T, N+1)
    end.
```

C.4 Realtime.erl

```
% realtime.erl
-module(realtime).

-export([start/3]).

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

start(Type, Maxtime, Period) ->
    [{_, _,StartTime,_}] = hera_data:get(nav3, sensor_fusion@nav_1),
    case Type of
    once ->
        io:format("Countdown!~n"),
        countdown(5),
        io:format("StartTime : ~p~n", [StartTime]),
        collect_data_over_time(StartTime + Maxtime + 5000); % 5000 for
            the countdown
    loop ->
        if Period < 0 ->
            grdos(Maxtime, Period); % Keep Period negative to loop
                indefinitely
        true ->
            grdos(Maxtime, Period + StartTime) %
                gesture_recognition_division_over_stop
        end
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% For the First Method
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

collect_data_over_time(Maxtime) ->
    collect_data_over_time(Maxtime, [], 0).

collect_data_over_time(Maxtime, List, LastT) ->
    [{_, _,Time, Data}] = hera_data:get(nav3, sensor_fusion@nav_1),
        %[{_, _,Time, Data}]

    if Time > Maxtime ->
        io:format("Done!~nCalculating...~n"),
```

```
        classify:classify_new_gesture(List),
        learning(List); % Call the function to ask if the user want to
            learn the gesture
    true ->
        if Time == LastT ->
            collect_data_over_time(Maxtime, List, Time);
        true ->
            NewList = lists:append(List, [Data]),
            collect_data_over_time(Maxtime, NewList, Time)
        end
    end.

countdown(Count) ->
    case Count of
        0 ->
            io:format("Start!~n");
        _ ->
            io:format("~p~n", [Count]),
            timer:sleep(1000),
            countdown(Count-1)
    end.

learning(List) ->
    case io:get_line("Do you want to learn this gesture? (y/n/ENTER) :
        ") of
        "y\n" ->
            Name = io:get_line("What is the name of the gesture (use _
                for space) : "),
            RemSlash = string:strip(Name, right, $\n),
            NameAtom = list_to_atom(RemSlash),
            learn:learn(List, NameAtom);
        "n\n" ->
            io:format("No learn ");
        "\n" ->
            io:format("No learn ");
        _ ->
            io:format("Unknown~n"),
            learning(List)
    end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% For the Second Method
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
grdos(Maxtime, Period) ->
    AS = learn:av_size(),
    grdos(Maxtime, Period, AS, [], 0, [], 0, 0, x, x, x). % x for nothing

% 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
grdos(TO, Period, AS, List, SizeL, GestureList, LastT, TSM, LastX,
    LastY, LastZ) ->
    [{_, _,Time, Data}] = hera_data:get(nav3, sensor_fusion@nav_1),
    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);
    true ->
        if Time == LastT ->
            grdos(TO, Period, AS, List, SizeL, GestureList, Time, TSM,
                LastX, LastY, LastZ); % Skip if no new data
        true ->
            NewGestureList = lists:append(GestureList, [Data]),
            NewList = lists:append(List, [Data]),
            NewSizeL = SizeL + 1,
            if NewSizeL >= AS -> % It mean we can compute the average
                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),
                [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"),
                    classify:classify_new_gesture(GestureList),
                    grdos(TO, Period, AS, [], 0, [], Time, Time,
                        LastX, LastY, LastZ);
                true -> % too soon, still need to wait
                    grdos(TO, Period, AS, [], 0, GestureList, Time,
                        TSM, LastX, LastY, LastZ)
                end;
            true ->
                NewLastX = HX,
                NewLastY = HY,
                NewLastZ = HZ,
                NewTSM = Time,
                grdos(TO, Period, AS, [], 0, NewGestureList, Time,
                    NewTSM, NewLastX, NewLastY, NewLastZ)
            end;
        true ->
            grdos(TO, Period, AS, NewList, NewSizeL, NewGestureList,
                Time, TSM, LastX, LastY, LastZ)
        end
    end
end.
```

C.5 Analyze.py

```
# analyze.py
import pandas as pd
import matplotlib.pyplot as plt

# do math
def do_math():
    T1 = 576460741874
    T2 = 576460720188
    iter = 463

    time = T2 - T1
    av = time / iter
    print(av)

def plot(min, max):
    df = pd.read_csv('measures/forealonce0.csv')
    last_three_columns = df.iloc[:, min:max]
    index = df.iloc[:, 0]

    plt.plot(index, last_three_columns.iloc[:, 0], label='Value X')
    plt.plot(index, last_three_columns.iloc[:, 1], label='Value Y')
    plt.plot(index, last_three_columns.iloc[:, 2], label='Value Z')
    plt.title('Value of Over the 3 axis')
    plt.xlabel('Index')
    plt.ylabel('Value')
    plt.legend()
    plt.show()

# plot(2, 5)
# plot(5, 8)
# plot(8, 11)

do_math()
```

C.6 Gesture

[no_move,x,o]
[no_move,y,o]
[no_move,z,nn]
[rot_Y_down_180,x,o,n,nn,n,o,n,nn,n,o]
[rot_Y_down_180,y,o]
[rot_Y_down_180,z,nn,n,o,p,pp,p,o,n,nn]
[rot_Y_down_180,x,o,n,nn,o,n,nn,n,o]
[rot_Y_down_180,y,o]
[rot_Y_down_180,z,nn,o,p,pp,p,o,n,nn]
[rot_x_left_90,x,o]
[rot_x_left_90,y,o,p,pp,p,o]
[rot_x_left_90,z,nn,n,o,n,nn]
[rot_x_left_90,x,o]
[rot_x_left_90,y,o,p,o,pp,p,o]
[rot_x_left_90,z,nn,n,o,n,nn]
[rot_x_right_360,x,o]
[rot_x_right_360,y,o,n,nn,n,o,p,pp,p,o]
[rot_x_right_360,z,nn,n,o,p,pp,p,o,n,nn]
[rot_x_right_360,x,o]
[rot_x_right_360,y,o,n,nn,n,o,p,pp,p,o]
[rot_x_right_360,z,nn,n,o,p,pp,p,o,n,nn]

C.7 MiniCluster.erl

```
% miniCluster.erl not used in the final version of the algorithm
-module(miniCluster).

% -compile(export_all).
-define(DATASET, [{1,8}, {1,9}, {2,9}, {8,1}, {9,1}, {9,2}, {2,5},
    {6,4}]).
-define(MAX, 10).
-export([km/0, kmean/2]).

km() ->
    kmean(?DATASET, 2).

kmean(Dataset, K) -> % First occurence
    io:format("Dataset : ~p~n", [Dataset]),
    Centroids = random_centroids(Dataset, K),
    io:format("Initial Centroid : ~p~n", [Centroids]),
    kmean_loop(Dataset, Centroids, 0).

kmean_loop(Dataset, Centroids, Occurence) ->
    Clusters = assign_clusters(Dataset, Centroids),
    io:format("Clusters : ~p~n", [Clusters]),
    New_centroids = compute_centroids(Clusters),
    io:format("New Centroid : ~p~n", [New_centroids]),
    Sorted_centroids = lists:sort(Centroids), % Sort the centroids
        because == donn't work if element aren't at the same place
    Sorted_new_centroids = lists:sort(New_centroids),
    if
        Sorted_centroids == Sorted_new_centroids -> Clusters; % If the
            centroids are the same, return the clusters
        Occurence >= ?MAX -> Clusters; % If the number of occurence is
            greater than MAX, return the clusters
        true -> kmean_loop(Dataset, New_centroids, Occurence + 1) %
            Else, repeat the process
    end.

random_centroids(Dataset, K) ->
    random_centroids(Dataset, K, []).

random_centroids(Dataset, K, Centroids_list) ->
    Centroid = {rand:uniform(10), rand:uniform(10)},
    New_centroids_list = [Centroid | Centroids_list],
```

```
    if
        length(New_centroids_list) == K ->
            New_centroids_list;
        true ->
            random_centroids(Dataset, K, New_centroids_list)
    end.

assign_clusters(Dataset, Centroids) ->
    N = length(Centroids),
    Clusters = [ [] || _ <- lists:seq(1,N) ], % N = length(Centroids)
    assign_clusters(Dataset, Centroids, Clusters).

assign_clusters(Dataset, Centroid, Clusters) ->
    case Dataset of
        [] -> Clusters; % Parse through the whole Dataset : return
            clusters
        [H|T] ->
            Closest = closest_centroid(H, Centroid),
            % io:format("Clusters : ~p~n", [Clusters]),
            Actual_point = lists:nth(Closest, Clusters),
            New__actual_point = [H | Actual_point],
            New_clusters = replace_nth_element(Closest,
                New__actual_point, Clusters),
            assign_clusters(T, Centroid, New_clusters)
    end.

closest_centroid(Point, Centroids) -> % Return the index of the closest
    centroid
    closest_centroid(Point, Centroids, 1, 0, 9999). % 9999 is a random
        large number

closest_centroid(_, [], _, ClosestIndex, _) -> % When the cluster list
    is empty
    ClosestIndex;

closest_centroid(Point, [Centroid|Rest], Index, ClosestIndex,
    ClosestDistance) ->
    Distance = euclidean_distance(Point, Centroid),
    if
        Distance < ClosestDistance ->
            closest_centroid(Point, Rest, Index+1, Index, Distance);
        true ->
```

```
            closest_centroid(Point, Rest, Index+1, ClosestIndex,
                ClosestDistance)
    end.

euclidean_distance({X1, Y1}, {X2, Y2}) ->
    math:sqrt(math:pow(X2-X1, 2) + math:pow(Y2-Y1, 2)).

replace_nth_element(N, Value, [_ | T]) when N =:= 1 ->
    [Value | T];
replace_nth_element(N, Value, [H | T]) when N > 1 ->
    [H | replace_nth_element(N-1, Value, T)];
replace_nth_element(_, _, []) ->
    [].

compute_centroids(Clusters) ->
    compute_centroids(Clusters, []).

compute_centroids(Clusters, New_centroids) ->
    case Clusters of
        [] -> New_centroids;
        [H|T] -> % H is a cluster
            New_centroid = compute_centroid(H),
            compute_centroids(T, [New_centroid | New_centroids])
    end.

compute_centroid(Cluster) ->
    X = lists:sum([X || {X, _} <- Cluster]) / length(Cluster),
    Y = lists:sum([Y || {_, Y} <- Cluster]) / length(Cluster),
    {X, Y}.
```

C.8 MiniCluster3D.erl

```
% miniCluster3D.erl not used in the final version of the algorithm

% Small Cluster algorithm based on kmean for 3D point between 0 and 10
-module(miniCluster3D).

% -compile(export_all).
-define(DATASET, [{1,8,0}, {1,9,0}, {2,9,0}, {1,9,3}, {8,1,9}, {9,1,9},
    {9,2,9}, {9,1,7}]).
-define(MAX_OCC, 10).
-define(MAX_VAL, 10).
-define(RANDOM_LARGE_NUMBER, 9999).
-export([km/0, kmean/2]).

km() ->
    kmean(?DATASET, 2).

kmean(Dataset, K) -> % First occurence
    io:format("Dataset : ~p~n", [Dataset]),
    Centroids = random_centroids(Dataset, K),
    io:format("Initial Centroid : ~p~n", [Centroids]),
    kmean_loop(Dataset, Centroids, 0).

kmean_loop(Dataset, Centroids, Occurence) ->
    Clusters = assign_clusters(Dataset, Centroids),
    io:format("Clusters : ~p~n", [Clusters]),
    New_centroids = compute_centroids(Clusters),
    io:format("New Centroid : ~p~n", [New_centroids]),
    Sorted_centroids = lists:sort(Centroids), % Sort the centroids
        because == donn't work if element aren't at the same place
    Sorted_new_centroids = lists:sort(New_centroids),
    if
        Sorted_centroids == Sorted_new_centroids -> Clusters; % If the
            centroids are the same, return the clusters
        Occurence >= ?MAX_OCC -> Clusters; % If the number of occurence
            is greater than MAX, return the clusters
        true -> kmean_loop(Dataset, New_centroids, Occurence + 1) %
            Else, repeat the process
    end.

random_centroids(Dataset, K) ->
    random_centroids(Dataset, K, []).
```

```
random_centroids(Dataset, K, Centroids_list) ->
    Centroid = {rand:uniform(?MAX_VAL), rand:uniform(?MAX_VAL),
       rand:uniform(?MAX_VAL)},
    New_centroids_list = [Centroid | Centroids_list],
    if
        length(New_centroids_list) == K ->
            New_centroids_list;
        true ->
            random_centroids(Dataset, K, New_centroids_list)
    end.

assign_clusters(Dataset, Centroids) ->
    N = length(Centroids),
    Clusters = [ [] || _ <- lists:seq(1,N) ], % N = length(Centroids)
    assign_clusters(Dataset, Centroids, Clusters).

assign_clusters(Dataset, Centroid, Clusters) ->
    case Dataset of
        [] -> Clusters; % Parse through the whole Dataset : return
           clusters
        [H|T] ->
            Closest = closest_centroid(H, Centroid), % Return the index
               of the closest centroid
            Actual_point = lists:nth(Closest, Clusters),
            New__actual_point = [H | Actual_point],
            New_clusters = replace_nth_element(Closest,
               New__actual_point, Clusters),
            assign_clusters(T, Centroid, New_clusters)
    end.

closest_centroid(Point, Centroids) -> % Return the index of the closest
    centroid
    closest_centroid(Point, Centroids, 1, 0, ?RANDOM_LARGE_NUMBER). %
       9999 is a random large number

closest_centroid(_, [], _, ClosestIndex, _) -> % When the cluster list
    is empty
    ClosestIndex;

closest_centroid(Point, [Centroid|Rest], Index, ClosestIndex,
    ClosestDistance) ->
    Distance = euclidean_distance(Point, Centroid),
```

```
    if
        Distance < ClosestDistance ->
            closest_centroid(Point, Rest, Index+1, Index, Distance);
        true ->
            closest_centroid(Point, Rest, Index+1, ClosestIndex,
                ClosestDistance)
    end.

euclidean_distance({X1, Y1, Z1}, {X2, Y2, Z2}) ->
    math:sqrt(math:pow(X2-X1, 2) + math:pow(Y2-Y1, 2) + math:pow(Z2-Z1,
        2)).

replace_nth_element(N, Value, [_ | T]) when N =:= 1 ->
    [Value | T];
replace_nth_element(N, Value, [H | T]) when N > 1 ->
    [H | replace_nth_element(N-1, Value, T)];
replace_nth_element(_, _, []) ->
    [].

compute_centroids(Clusters) ->
    compute_centroids(Clusters, []).

compute_centroids(Clusters, New_centroids) ->
    case Clusters of
        [] -> New_centroids;
        [H|T] -> % H is a cluster
            New_centroid = compute_centroid(H),
            compute_centroids(T, [New_centroid | New_centroids])
    end.

compute_centroid(Cluster) ->
    case Cluster of
        [] -> % If the cluster is empty, return a random point
            {rand:uniform(?MAX_VAL), rand:uniform(?MAX_VAL),
                rand:uniform(?MAX_VAL)};
        [_|_] ->
            X = lists:sum([X || {X, _, _} <- Cluster]) / length(Cluster),
            Y = lists:sum([Y || {_, Y, _} <- Cluster]) / length(Cluster),
            Z = lists:sum([Z || {_, _, Z} <- Cluster]) / length(Cluster),
            {X, Y, Z}
    end.
```

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