

Improving Human-Robot Interaction In Wearable Robotics Through Comprehensive User Interface Design

BY

GREGORY SACCO

B.S., Politecnico di Torino, Torino, Italy, 2021

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2023

Chicago, Illinois

Defence Committee:

Myunghye Kim, Chair and Advisor

Milos Zefran

Diego Regruto Tomalino, Politecnico di Torino

Stefano Quer, Politecnico di Torino

ACKNOWLEDGMENTS

Acknowledgments Acknowledgments Acknowledgments Acknowledgments Acknowl-
edgments Acknowledgments Acknowledgments Acknowledgments Acknowledgments Ac-
knowledgments Acknowledgments Acknowledgments Acknowledgments Acknowledg-
ments Acknowledgments Acknowledgments Acknowledgments Acknowledgments Ac-
knowledgments Acknowledgments Acknowledgments Acknowledgments Acknowledg-
ments Acknowledgments Acknowledgments Acknowledgments Acknowledgments Ac-
knowledgments Acknowledgments Acknowledgments Acknowledgments Acknowledg-
ments Acknowledgments Acknowledgments Acknowledgments Acknowledgments Ac-
knowledgments Acknowledgments Acknowledgments Acknowledgments Acknowledg-
ments Acknowledgments Acknowledgments Acknowledgments Acknowledgments Ac-
knowledgments Acknowledgments Acknowledgments

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	Motivations	2
1.2	Thesis Structure	3
2	RELATED WORK	4
2.1	Human In the Loop toolkit	4
2.2	Ankle Foot Exoskeleton	5
2.3	ECG biopatch sensor	10
3	IMPLEMENTATION - Wii Balance Board	15
3.1	Wii Balance Board - hardware	17
3.2	WiiBalance Library - software	19
3.3	Calibration functions	24
3.3.1	NewtonTransform	25
3.3.2	CalibrationWdrift	28
3.3.3	CalibrationP	31
3.4	Two Wii Balance Boards integration	35
4	IMPLEMENTATION - Control System Interface	39
5	IMPLEMENTATION - Optimization Interface	40
5.1	Software structure	41
5.1.1	REST API	43
5.1.2	ZMQ	45
5.1.3	Server Architecture	48
5.2	GUI design	52
5.2.1	Initialization tab	53
5.2.2	Optimization tab	54
5.2.3	Signals tab	56
5.2.4	Hyperparameters tab	57
6	IMPLEMENTATION - System Integration	60
7	EXPERIMENTS	61

8 CONCLUSION**62**

LIST OF FIGURES

1	AFO's modular components. Screenshot from [1]	5
2	Subject wearing the AFO on the right leg. Figure from [1]	6
3	Portable actuator system in the off-board emulator setting with the AFO worn by a subject. Figure from [1]	6
4	Torque-angle characteristic	9
5	ECG signal description. Credits https://litfl.com/t-wave-ecg-library/	11
6	Soft flexible bioelectronic system (SFB) integrated with an ankle-foot-orthosis (AFO) exoskeleton. Figure from [5]	12
7	Wii Balance Board from Nintendo. Credits: https://it.wikipedia.org/wiki/File:Wii_Balance_Board_transparent.png	17
8	Extracted pressure sensor. Credits: https://electronics.stackexchange.com/questions/83861/connect-wii-balanceboard-pressure-sensor-to-an-arduino	17
9	Disassembled pressure sensor. Credits: https://www.xsimulator.net/community/threads/diy-load-cell-brake-pedal-short-tuto.6042/page-2	18
10	Strain gauge used in WBB's sensors. Credits [Video on http://www.rehabtools.org/wii-balance-board.html]	19
11	Extracted pressure sensor	20
12	Disassembled pressure sensor	20
13	Weight application device and weights	21
14	Load application method	21
15	Linearity check on WBB's sensors	22
16	Sensors' labels	24
17	Setup for NewtonTransform calibration	26
18	NewtonTransform result	27
19	Raw data and filtered data from Sensor 1	29
20	Second calibration result	30
21	CalibrationP setup	31
22	Map of points selected for calibration	32
23	Measures of CoPx	33
24	Measures of CoPy	33
25	Measures from sensor 1	34
26	Measures from sensor 2	34

27	Measures from sensor 3	34
28	Measures from sensor 4	34
29	Reference system transformation	35
30	Software main architecture	41
31	Initialization tab	53
32	Optimization tab	55
33	Signals tab	57
34	Hyperparameters tab	58

LIST OF CODES

1	WiiLab function for computing CoP	23
2	1-D digital filter	29
3	Portion of WBB.m	36
4	Function for sharing data with UI	44
5	Downloading data from Server	44
6	Setting up the ZMQ communication	46
7	ZMQ sending functions	47
8	ZMQ receiving functions	47
9	Server's subprocess for receiving ZMQ messages	49
10	Methods used by Server code	50
11	Callbacks for sharing data with UI code	50
12	All callbacks used by Server code	51

LIST OF ABBREVIATIONS

WBB Wii Balance Board

HIL Human In the Loop

AFO Ankle Foot Orthosis

HRV Hear Rate Variability

RMSSD Root Mean Square of successive Differences between normal heartbeats

SFB Soft Flexible Biopatch

CoP Center of Pressure

SUMMARY

Summary Summary Summary Summary Summary Summary Summary Summary
Summary Summary Summary Summary Summary Summary Summary Summary Sum-
mary Summary Summary Summary Summary Summary Summary Summary Sum-
mary Summary Summary Summary Summary Summary Summary Summary Sum-
mary Summary Summary Summary Summary Summary Summary Summary Sum-
mary Summary Summary Summary Summary Summary Summary Summary

Chapter 1

INTRODUCTION

asdsda

1.1 Motivations

asdasda

1.2 Thesis Structure

asdasda

Chapter 2

RELATED WORK

In the experiments of this study were used devices and softwares already implemented by students from the Rehabilitation Robotics laboratory of University of Illinois at Chicago¹ and Georgia Institute of Technology². This section will explain in details the design and the main functioning of these products. The section is divided in Human In the Loop (HIL) toolkit, the Ankle Foot Orthosis (AFO) and a biopatch electronic sensor for measuring high-quality ECG.

2.1 Human In the Loop toolkit

¹<https://rehab-robotics.lab.uic.edu>

²<https://www.gatech.edu>

2.2 Ankle Foot Exoskeleton

A fundamental device for the experiments of this thesis is the AFO. This is the mechatronic device that will give assistance to the patient in doing the squatting activity. The device is thoroughly described in the respective paper called *Personalized Wearable Ankle Robot Using Modular Additive Manufacturing Design* [1] and it is a valuable solution to improve research in wearable assistive robotics whose goal is to improve the gait of individuals with reduced mobility. The paper cited above stresses the importance of developing a device that can also mechanically adapt to the variations in body measurements and walking mechanics, while taking into account the operational aspects of the device determined by the user's physical state and objectives in gait training.

The mechanical design is made of two main parts: the Exo-foot and Exo-Tibia. Both

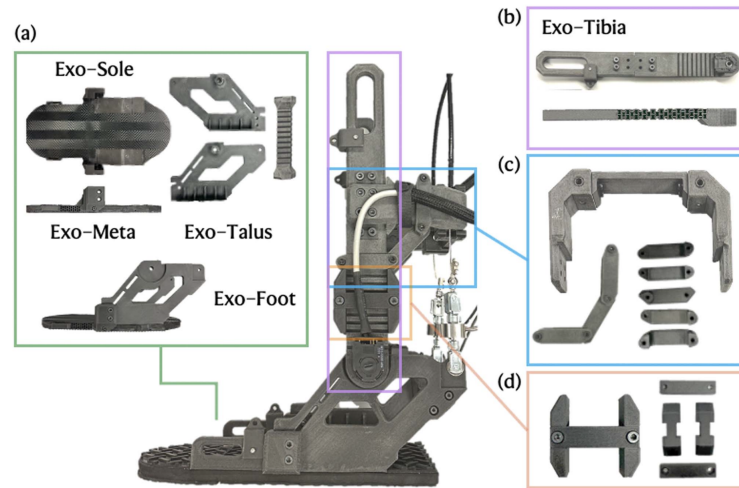


Figure 1: AFO's modular components. Screenshot from [1]

of these components can easily be replaced with other similar parts whose dimensions,

however, meet the characteristics of the patient. Exo-Foot is the bottom part composed by the sole and additional components in order to connect the Exo-Foot with the Exo-Tibia. The latter is important to attach the exo-skeleton to the patient's tibia and make sure that they are securely fixed. On the Exo-Tibia there is connected an additional mechanical supporting piece in order to hold and route the cables coming directly from the motors. The additive manufacturing technique was used to develop the parts mentioned above. This allowed them to make lighter and more anthropometrically efficient components. The material used is nylon-carbon fiber N12 Carbon Fiber and the AFO is around $0.9kg$ in weight.

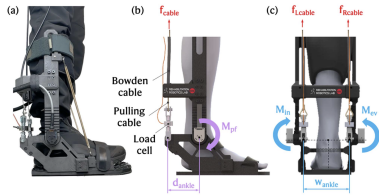


Figure 2: Subject wearing the AFO on the right leg. Figure from [1]

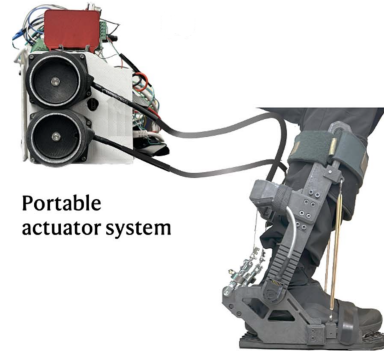


Figure 3: Portable actuator system in the off-board emulator setting with the AFO worn by a subject. Figure from [1]

The portable actuator system as a whole also has two optical encoders (HEDS-5500-A06, Broadcom Inc., CA, USA). They are located in the joints between Exo-Foot and Exo-Tibia. One in the medial side and the other in the lateral side. They are able to measure the angles of the joints in the sagittal plane. Torque feedback is given instead by two DYMH-103 tensile load cells. The torque is given by two motors (EC i-

52s Maxon Group, Switzerland). Two EPOS4 50/15 EtherCAT motor drivers (Maxon Group, Switzerland), on the other hand, take care of controlling the motors and reading the feedback signals coming from the encoders and load cells. The output torque of the AFO is controlled through a Simulink real-time controller. The motors are engaged in supporting plantar flexion. While for dorsiflexion, the device uses two elastic bands connected between the Exo-Tibia and Exo-Foot. Two sub miniaturized, compact, quick-response limit switches are placed on the medial and lateral sides of the AFO in order to do not exceed the maximum expected plantarflexion angle for security purposes. As shown in the Figure 3, the motors can transmit torque to the exoskeleton through two 2 m long Bowden cables. The diameter of the cables is $5mm$. The weight of the whole system shown in the Figure 3 is about $6.5kg$. A portable dual cable-driven system was designed in order to offer assistance with plantarflexion and in/eversion. With the following equations we can define the torques applied for plantarflexion and in/eversion assistance:

$$M_{pf} = M_L + M_R \quad (2.1)$$

$$M_{in-ev} = (M_L - M_R) \cdot \frac{w_{ankle}}{2d_{ankle}} \quad (2.2)$$

where M_L and M_R stands for torque caused by the motor connected to the left and right side of the exoskeleton. D_{ankle} and W_{ankle} are dimensions taken from the geometry of the Exo. It is possible to check these two features in Figure 2.

The controller is run in Simulink (Mathworks, MA, USA) at the frequency of $1KHz$. The controller is divided into mid level control and low level control. The mid level part describes the angle/torque characteristic. So it is responsible for defining the plantar and in/eversion torque needed to assist the human subject. The low-level controller, on the other hand, is responsible for translating the desired torques into velocity commands to be sent to the motors. The latter controller is of the proportional type and its output, that is the command in velocity, is defined by the Equation 2.3.

$$\dot{\theta}_d = k_{gain}(M_d - M_m) \quad (2.3)$$

where M_d and M_m are desired torque and torque measured by the load cells. The Mid level controller can work in different modes: *No control* to completely disable the motors and controller, *Position control* to enable the controller and set the desired torques to zero. This makes it possible to cancel the loosening of the transmission cables. Finally, we have the *Squat control* mode, which instead is used for squatting activity and thus to define the torque needed to support the patient during the various phases of the squats. In Figure 4 it is possible to observe an approximate schematic of the characteristic that represents the torque/angle characteristic used by the controller in this mode.

The characteristic is described by 6 states but here for simplicity we define three main steps: *Descending*, *Bot2Asc* and *Ascending*. For *Descending* and *Ascending* we have a

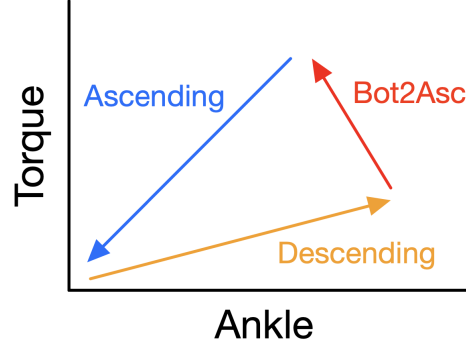


Figure 4: Torque-angle characteristic

function that determines the desired torque based on the angle. The difference between these two steps is that they can have different stiffness, and therefore, different resulting slopes in the characteristic. The Red arrow instead represents the *Bot2Asc* state that stands for “Bottom To Ascend” state. This state is critical to transition as smoothly as possible from descending to ascending. An immediate transition from descending to ascending, assuming that $K_{ascending}$ is greater than $K_{descending}$, could cause exaggerated resistance at the conclusion of the descending movement. In the article, the authors also tested the whole system and obtained satisfactory results. The tests in a laboratory setup indicated that the device could generate $40Nm$ of torque for plantarflexion and $16Nm$ of torque for inversion/eversion movements, achieving rise times of $70ms$ for plantarflexion, $77ms$ for eversion, and $84ms$ for inversion. The control bandwidth for torque exceeded $13Hz$ for both plantarflexion and inversion/eversion. During a squat assistance task involving a human participant, the device was able to closely reach the desired torque pattern, displaying a maximum average root mean square (RMS) error of $2.9 \pm 1.2Nm$ for plantarflexion assistance and $0.7 \pm 0.5Nm$ for inversion/eversion

assistance.

2.3 ECG biopatch sensor

As observed in the preceding chapters, within this field, the utilization of a HIL framework is of paramount importance in order to ensure the personalizing process of a wearable robot capable of assisting the subject. A fundamental constituent of the HIL framework is the Biofeedback signal, which furnishes us with insights into the vital functions of the human subject. Specifically, biofeedback aids in comprehending the level of physical effort experienced by the patient during a given activity. One of the most commonly employed methodologies for this purpose entails estimating the metabolic cost via indirect calorimetry and a respiratory mask, which measures the inhaled oxygen and exhaled carbon dioxide of the subject. Nonetheless, despite its reliability, the aforementioned system is also notably inconvenient, bulky, and impractical. Indeed, devices of this nature necessitate a rigid mask that fits snugly onto the face without gaps and are equipped with a small backpack (worn by the subject) housing the battery system and antenna for data transmission. These components can affect the subject's mobility. These cumbersome systems directly contradict the goal of developing wearable robots aimed at assisting individuals in performing repetitive motions. These systems entail a protracted setup, are confined to laboratory settings, and offer discomfort due to their substantial dimensions and weight. Notably, this system mandates a minimum of 3 minutes to estimate the required physiological sig-

nal. An alternative solution, on the other hand, involves employing sensors capable of interpreting the Electrocardiogram (ECG). The ECG signal is not perfectly periodic [2]; it varies according to the effort experienced by the patient [3], [4]. Greater effort corresponds to a more periodic heart rate. The variable that describes this characteristic is known as Heart Rate Variability (HRV). To compute and quantify this variable, Root Mean Square of successive Differences between normal heartbeats (RMSSD) is used. RMSSD involves taking the square root of the sum of squared intervals between adjacent normal heartbeats. The time intervals are measured between adjacent peaks of the ECG and are referred to as RR intervals, as depicted in Figure 5.

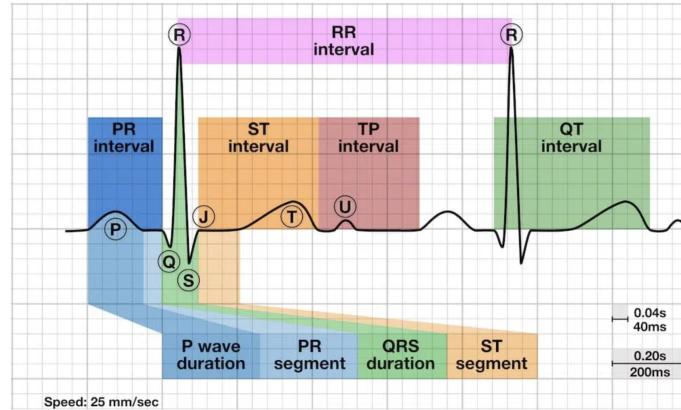


Figure 5: ECG signal description. Credits <https://litfl.com/t-wave-ecg-library/>

Numerous commercial devices are available in the market capable of measuring HRV; however, these systems often entail discomfort during wear, thereby imposing limitations on mobility. The scientific paper titled *Soft wearable flexible bioelectronics integrated with an ankle-foot exoskeleton for estimation of metabolic costs and physical effort* delineates a study centered on the development of a soft, flexible biopatch profi-

cient in collecting ECG data and transmitting them via Bluetooth. The core objective of this article resides in disseminating the design of the Soft Flexible Biopatch (SFB) and elucidating the outcomes of several experiments that underscore the robust negative correlation between normalized metabolic cost and normalized HRV-RMSSD.

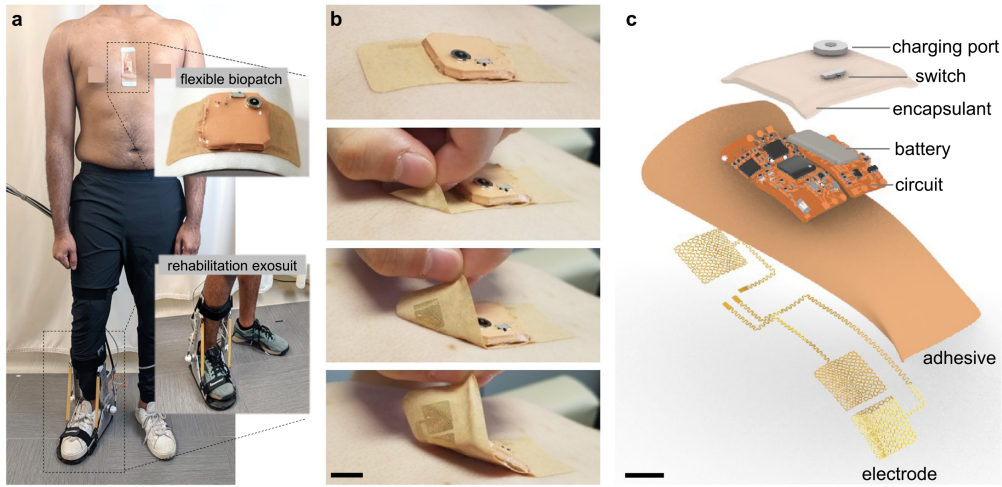


Figure 6: Soft flexible bioelectronic system (SFB) integrated with an ankle-foot-orthosis (AFO) exoskeleton. Figure from [5]

The SFB consists of multiple layers. The initial layer, which makes contact with the skin, comprises electrodes. Subsequently, an adhesive base is present and on top of that there is a flexible Printed Circuit Board (fPCB) and the power supply. The fPCB incorporates all the necessary electronics for data processing. Specifically, within its structure, components such as the ECG analog-to-digital converter (ADS1292, Texas Instruments), Microprocessor (NRF 52832, Nordic), and a high-frequency low-power Bluetooth antenna are encompassed. As depicted in Figure 6, the fPCB is encapsulated by a soft elastomer (EcoflexTM30, Smooth-On), enabling enhanced mechanical and

electrical resistance. Atop of this encapsulation, the charging port and a switch are positioned. Notably, the device is equipped with a $3.7V$, $40mAh$ battery that ensures 9 hours of uninterrupted operations. A complete recharge is accomplished within 30 minutes.

Due to skin irregularities, the device must exhibit flexibility of more than 15° to maintain consistent contact with the sternum. This property has been validated through comprehensive mechanical testing, yielding no damage with regards to mechanics, electronics, or overall performances.

The assessment of SFB quality hinges upon obtaining a reliable ECG signal. To this end, the sensor records raw data which are subsequently filtered using a first-order Butterworth band-pass filter. Cut-off frequencies are $0.5Hz$ and $60Hz$. The Signal-to-Noise Ratio (SNR) is measured at 25 dB, facilitating clear distinction of ECG signal characteristics, in particular the R peaks, across all usage scenarios: squatting, walking, and running. Subsequent to filtering, the signal undergoes convolution through a moving average process, followed by application of an RMS envelope to eliminate noisier periods. They developed an HR detection algorithm that compares the values of the peaks with the threshold shown in Equation 2.4.

$$Threshold = NoiseLevel + 0.25(SignalLevel - NoiseLevel) \quad (2.4)$$

This threshold depends on two parameters: NoiseLevel and SignalLevel. If the value is higher than the threshold, the peak is identified as a valid R peak. The parameters

mentioned before are dynamically updated after every classification with Equation 2.5 and Equation 2.6. If the valid R peak is higher than the result of Equation 2.5, *SignalLevel* will be updated with Equation 2.5. If not, *NoiseLevel* will be updated with Equation 2.6.

$$\textit{SignalLevel} = 0.125\textit{Peak} + 0.875\textit{SignalLevel} \quad (2.5)$$

$$\textit{NoiseLevel} = 0.125\textit{Peak} + 0.875\textit{SignalLevel} \quad (2.6)$$

After this post processing of the ECG data, a python library called Neurokit2 [6] has been used in order to calculate the HRV-RMSSD values.

In summary, this study presents a compact wearable bioelectronic system integrating a flexible biopatch and an ankle-foot exoskeleton. The compact biopatch replaces bulky tools, offering accurate metabolic rate measurement. The conformal device ensures close skin contact for precise ECG and HRV-RMSSD recording. Unlike traditional mask-based calorimetry, this wearable setup, comprising SFB and AFO, exhibits strong performance in various activities with a high signal-to-noise ratio (>25 dB) and robust Pearson R correlation (-0.758 , p-value: $1.2e - 7$) with steady-state metabolic cost.

Chapter 3

IMPLEMENTATION - Wii Balance Board

The Wii Balance Board (WBB) is a peripheral device developed by Nintendo for the Wii console. It was released in 2007 as part of the Wii Fit game package. The Balance Board is a rectangular platform equipped with multiple sensors that can detect shifts in weight and balance. The primary purpose of the Wii Balance Board is to enhance game play and promote physical activity. It allows players to engage in various interactive exercises and games that focus on fitness, balance, and coordination. The board can measure a player's center of gravity and movements in real-time, providing feedback and tracking progress. As already discussed in chapter 2, WBBs are a very cheap alternative for Biomedical applications. In terms of data precision, these systems might not attain the level of accuracy demonstrated by force plates, yet they frequently achieve satisfactory results. Our goal was to be able to detect the pattern drawn by the Center of Pressure (CoP) of each foot of the subject during squatting activity. This information can be used in two possible ways. The first is to use the CoP information in real time as a BioFeedBack signal. This real time data could be the input for an

optimization algorithm like the Bayesian Optimization. In particular, the idea is to compute a cost function using the CoP data and by minimizing the cost function, we can optimize the selection of the control parameters for every subject. The second is to simply compare the CoP data coming from different conditions of squatting during the experiment. For example we can compare the CoP data when the subject is squatting in a *no device* condition and the CoP data when the subject is in a *optimal* condition. For *optimal* condition we mean that the Human subject is squatting wearing the ankle exoskeleton with personalized assistance. By comparing, we can see if the assistance given by the exoskeleton can positively affect the kinematics of the squatting movement. Throughout this section, we will go over the implementation of the software able to connect, calibrate and retrieve the data coming from the WBBs.

3.1 Wii Balance Board - hardware



Figure 7: Wii Balance Board from Nintendo. Credits: https://it.wikipedia.org/wiki/File:Wii_Balance_Board_transparent.png

The working principle is very simple. The device is a platform of dimension $23 \times 43 \text{ cm}$ and it has 4 load cells in the 4 corners. The sensors are cantilevered metal bars with a strain gauge that converts the applied force into a voltage.

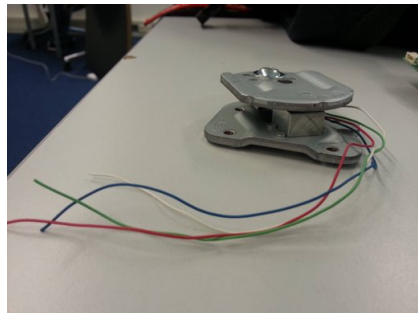


Figure 8: Extracted pressure sensor. Credits: <https://electronics.stackexchange.com/questions/83861/connect-wii-balanceboard-pressure-sensor-to-an-arduino>

Each bar has two strain gauges as represented in Figure 10. When a force is applied to this system, the bar bends and we get a deformation of the two strain gauges. One will tend to become narrower and the other will become wider. These deformations are

translated into a voltage signal for the electronic unit of the WBB. The cantilevered bars are made of duralumin, an alloy with high strength-to-weight ratio. So the straining on the metal is very slight. For example, if a weight of $100kg$ is applied, the bar would only bend by $0.1mm$. However, it is precise enough that it can accurately detect weight differences of 500 grams.

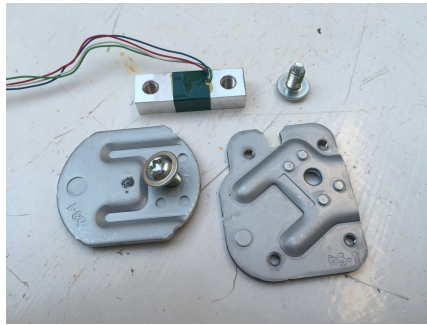


Figure 9: Disassembled pressure sensor. Credits: <https://www.xsimulator.net/community/threads/diy-load-cell-brake-pedal-short-tuto.6042/page-2>

As per the available literature, the Wireless Balance Board is reported to sample each force channel at a notably high frequency of around $100Hz$. This sampling rate exceeds the recommended minimum of $50Hz$ for accurately capturing the Center of Pressure (COP) during postural sway assessments [7]. The literature also suggests that the force sensors utilized in the WBB are designed to be linear and demonstrate COP noise levels of approximately $0.5mm$ [8]. It is also important to notice that because of how this balance is designed, It is possible to detect only vertical reaction forces [9]. This stands as one of the numerous reasons why force plates command a higher cost compared to Wii balance boards.

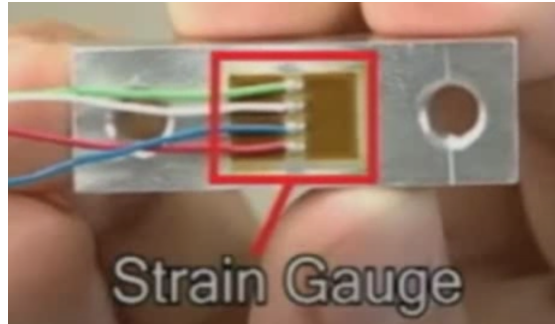


Figure 10: Strain gauge used in WBB’s sensors. Credits [Video on <http://www.rehabtools.org/wii-balance-board.html>]

3.2 WiiBalance Library - software

Over the past few years, Nintendo devices have proven to be very inexpensive and fairly accurate solutions to be implemented in various research applications such as biomedical or virtual reality. For this very reason, some research laboratories have been working on making these devices easier to use. In particular, the University of Notre Dame explained in the paper *WiiLab: Bringing Together the Nintendo Wiimote and Matlab* [10] how they developed low-level programming to facilitate the connection between the Wiimote and Matlab. WiiLab, as described in the paper, is a combined collection of C# and Matlab libraries for Windows that produce an intuitive API that greatly abstracts the difficulty of using the Wiimote. The Wii Remote, commonly referred to as the Wiimote, is the gaming controller for Nintendo’s Wii system. In order to connect to the WBB instead, Pete R Jones, from University of London, developed another Matlab library called WiiBalance¹. This library essentially incorporates additional functionalities to the WiiLab library, enabling it to establish connections not

¹<https://github.com/petejonze/wiibalance>

only with Wiimote devices but also with the WBB. For using these two libraries, it is important to use Matlab 2012 onward and 32 bit Windows as operating systems. On our computer Windows 10 (64 bit), we used Matlab 2015 (32 bit) in order to run the library. Once the installation was completed, we performed a test using the library. This was the outcome of the example given by the WiiLab tool (Figure 12).

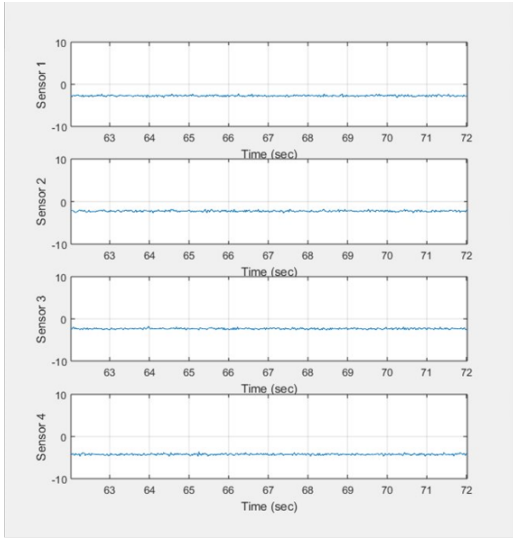


Figure 11: Extracted pressure sensor

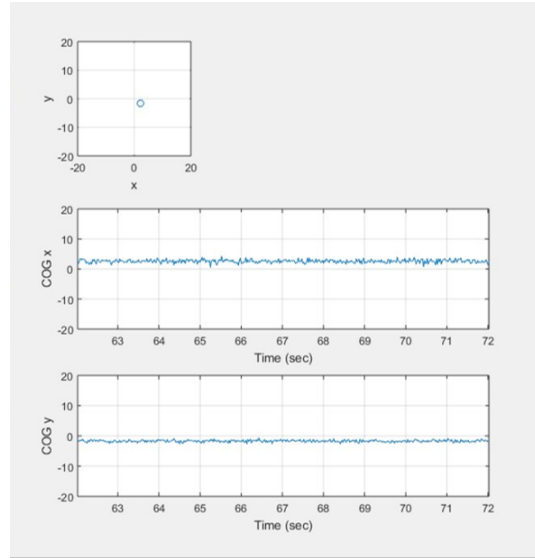


Figure 12: Disassembled pressure sensor

In Figure 12 there are 3 plots. The two on the bottom represents the coordinates (x and y) of the Center of Gravity (CoG) with respect to time. The vertical axes have centimeters as units of measurement. The first plot at the top instead represents the position of the CoG, indeed the x coordinate is on the horizontal axis (cm) and y coordinate is on the vertical (cm). Figure 11 shows 4 plots that we easily implemented in order to display the values read by each sensor in time. Because of the uncalibrated device and little knowledge of how the data is retrieved from the hardware, we couldn't

understand the measurement of the sensors' output. However, the first important thing was to check the linearity of measurements from each sensor. Without linear sensors it would be very hard to compute the CoP. In order to check that, we flipped the WBB with the sensors facing up and we applied several weights to each sensor and saved the values given by the sensors. The Matlab script was developed to connect to the balance board and store single measurements from the sensors. The list of the weights applied was: 4.75, 9.5, 14.12 and 18.77 kg . We applied the force using the system shown in Figure 11. The method for applying the weights is shown in Figure 14. In particular we used a rod with a ring in the middle. The ring is for blocking the weight disks that will be loaded on the rod. On one end of the rod there is a circular pointing tip made of steel whose diameter is 7 mm .



Figure 13: Weight application device and weights Figure 14: Load application method

The procedure was the following one. One disk at the time was applied using the custom pointing device shown in Figure 13. Once the disk was stable on the sensor we

took the measurement. We did this on each sensor by measuring all the weights listed before. When the procedure was done, the scripts showed the plots with the given outputs. The result plots were all very similar like in Figure 15. On the horizontal axis we have the values of the weights applied to the sensors in kilograms, on the y axis instead, there are the respective values measured by the balance board. The red asterisks are the samples, while the blue line is a function interpolating them. The outcome is deemed satisfactory due to the fact that, despite the ambiguity of units of measurement, the sensors exhibit a consistent linearity.

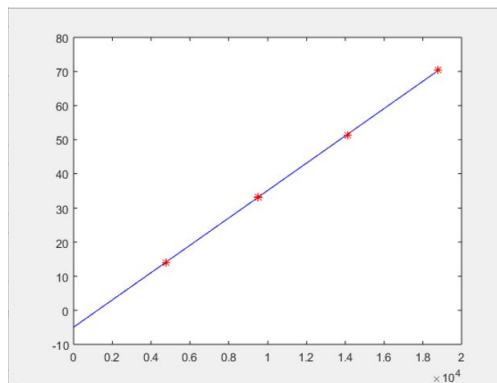


Figure 15: Linearity check on WBB's sensors

In light of the current circumstances, it has become increasingly important to recognize the significance of developing scripts specifically designed to calibrate the WBB. By doing so, we can effectively address and mitigate the challenges presented by this situation. Before working on the calibration functions we found out a problem in the computation of the CoP. The CoP, called CoG in the library WiiBalance, was retrieved by a function that we couldn't open to get more information about:

```
1 CoG = obj.bb.wm.GetBalanceBoardCoGState();
```

Code 1: WiiLab function for computing CoP

However, we wanted to test it in order to understand its behavior. So with our custom application device we applied a force in a specific point on the balance. As expected, we got a coordinate measurement that was within the ranges related to the size of the Balance Board. However, by keeping the same position and sufficiently increasing the pressure, we were able to move the point measured even outside the ranges. This is not tolerable for our application. We are interested in detecting the position where most of the force of the foot is applied. It is intuitively that this CoP point must be located inside the area where the foot is positioned. In the example shown before instead, by changing the force applied to the same point, we were getting different coordinates. The original goal of the WBB from Nintendo was to detect the balance, not the CoP. In order to change this we deleted that function. We used two formulas defined on *Accuracy of force and center of pressure measures of the Wii Balance Board* [8], [9]:

$$COP_x = \frac{L}{2} \frac{(TR + BR) - (TL + BL)}{TR + BL + TL + BL} \quad (3.1)$$

$$COP_y = \frac{W}{2} \frac{(TL + TR) - (BL + BR)}{TR + BL + TL + BL} \quad (3.2)$$

For these equations we considered the center of the balance as the origin and we used the dimensions of the balance for L and W. In particular L is 43,3 cm and W is 22,8 cm.

TL, TR, BL and BR refer to the sensors. They stand for top-right, top-left, button-left and button-right. Here is an explanatory figure (Figure 10):

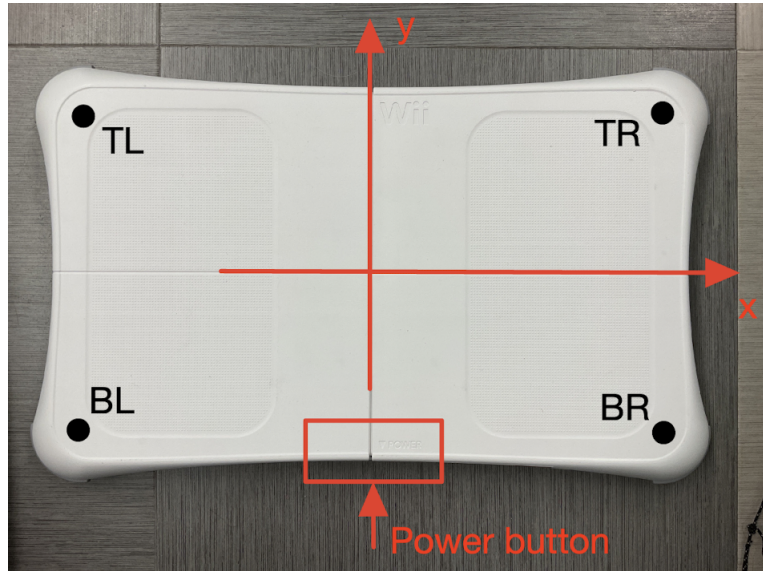


Figure 16: Sensors' labels

Following the aforementioned change, the experiment was conducted again, similar to the previous test. Notably, even by increasing the applied weights, the center of pressure (CoP) remained unchanged if the point of force application remained constant.

3.3 Calibration functions

From the experiment explained above it is clear that all the sensors were all approximately linear, however, the characteristics computed for each sensor were quite different in terms of slope and offset. To contribute to the most precise measurements possible, it is crucial that every sensor has exactly the same characteristic. In order to

calibrate the balance board we designed three calibration functions. *WiiBalance* is a class that calls in its initialization function the *WiiLab* class. In order to add all the calibration functions a higher class called *WiiBwCalibration* has been created. This last class calls the *WiiBalance* library in its initialization function and it contains the three calibration functions.

The first function *NewtonTransform* has the goal to change the characteristic of each sensor to an ideal one. The ideal equation that a sensor needs to have is $y = 9.81 * x$. This equation represents the formula to compute the gravitational force given a mass. The sensor characteristic inputs will be in kilograms and the outputs will be in Newtons. For example, by applying one kilogram on the sensor, we should read $9.81N$. The second calibration function is called *CalibrationWdrift*. As we said before, in the first calibration the WBB is flipped with the sensor facing up to make it easier to place the weights on the sensors. After this calibration, the sensors read zero when no force is applied. However, when we flip again the balance (WBB feet facing down), the sensors will perceive the weight of the balance itself. *CalibrationWdrift* is designed in order to measure the weight of the balance and subtract it from the next measurements. The third calibration is important for correcting the CoP equation.

3.3.1 NewtonTransform

The most complicated part for this calibration is the set up. It is important to maintain the weight application device still on the sensors and to make sure the order of the weights applied is respected. For this purpose, we designed a set up in

order to respect these requirements. As shown in the Figure 17, we used a metallic parallelepiped shaped mechanical structure (available in the lab) and another metallic component. By combining these two mechanical supports we created a structure able to make the custom weight application device standing and still. Because of the height of this structure, we used a rigid black box on which the WBB is then placed along with a scale. In particular we placed the scale on the box and then we arranged the flipped WBB (sensors facing up) on the scale.



Figure 17: Setup for NewtonTransform calibration

The reason for using the scale is quite simple. It is crucial for this calibration function to know exactly the value of the weight we are applying, the absolute value. In fact, even if we know exactly the mass applied to the sensor, this value depends also on how the load is placed on the device. If it is not perfectly vertical, this value could be different. In this case, the weight force will break down into two components, vertical and horizontal components. The WBB, because of the way it is designed, cannot sense horizontal forces. This means we lose information. To resolve this problem, we decided to place a scale under the WBB. So every time we have to start the calibration on a

sensor, we place the WBB on the scale, we make sure everything is stable, then push the Tare button on the scale. After this process we can start the calibration. The Matlab function asks the user to apply the first weight on the sensor, insert the value of the weight read by the scale in the command window and then press Enter to measure.

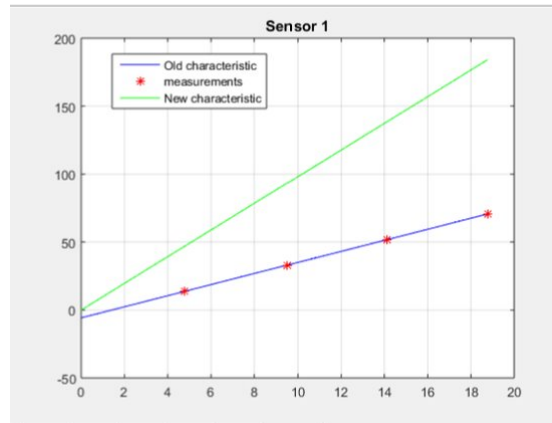


Figure 18: NewtonTransform result

Next step is adding the second weight and measuring again. Continue to the last weight that is 18.77 kg. After the last measure we get the plot showing the result (Figure 18). The red asterisks are the values measured and the blue line is the characteristic of the sensor before the calibration.

From the blue characteristic we can understand slope and offset. These two features will be used in order to get the green characteristic that shows the relationship between weights applied and values measured in Newtons after the calibration. The formula

used to calibrate the sensor is:

$$Y_{new} = (Y_{old} - offset) \cdot \frac{9.81}{slope_{Y_{old}}} \quad (3.3)$$

Where Y_{old} is the blue line and Y_{new} is the green one. This means that after this calibration is performed, every single sensor measurement is subtracted from the offset value of its initial characteristic, and then the result is multiplied by a constant given by the gravitational acceleration over the slope of the characteristic prior to calibration. This step ensures that each new sensor measurement gives in output a value in newtons that is perfectly consistent with the weight applied to the measuring device.

3.3.2 CalibrationWdrift

This calibration is performed immediately after the NewtonTransform function to delete the offset given by the weight of the balance. The user flips and places the WBB on the ground. It is important to mention that this calibration needs to be performed in the place where we want to conduct the experiment and use the calibrated WBB. After this calibration, the balance board shouldn't be moved. This is because this calibration may be different based on the surface where we place the balance. When the device is in the desired position we can press Enter. The Matlab script will start recording the data coming from all the sensors. When little weight is applied to the balance we get very noisy signals from the sensors. In order to get a more accurate estimate of the signals, the script records the signals for ten seconds. Once the data is

recorded, it convolutes the signals with a one dimension digital filter. In particular we take the data as a one dimension vector and then we perform a convolution between this vector and the filter, that is a vector with one hundred components and every element is equal to $1/100$. The design of the filter can be represented by the following code:

```
8 windowSize = 100;
9 b = (1/windowSize)*ones (1, windowSize);
```

Code 2: 1-D digital filter

The design consisted in tuning the value for *windowSize*. This value determines the size of the filter and the magnitude of each element. Every component of this 1D digital filter is equal. Once the convolution is done the Matlab script prints the result in Figure 13.

The example in Figure 13 is for sensor 1. The sensor was reading a mean value of $16N$

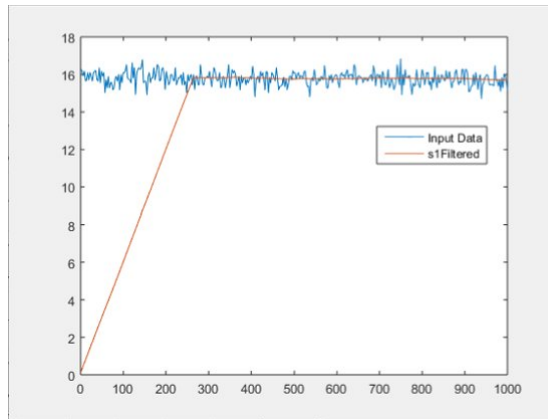


Figure 19: Raw data and filtered data from Sensor 1

because of the weight of the WBB (blue signal). For a larger value of *windowSize* we get a smoother signal in output but the filtered signal will take more time to converge to the real signal. So the design of this filter consisted in trying different values for

windowSize in order to obtain a trade-off between smoothing of the signal and fast convergence of the filtered signal.

After several attempts, we decided the parameter to be equal to 100. We wanted to get a smooth signal, but a fast convergence of the filtered signal was more important. If the delay for convergence is too long, it means that more samples will be neglected and that we have to collect more data, so the recording time should be longer. We wanted to keep the recording time to less than 10 seconds in order to make the calibration procedure faster.

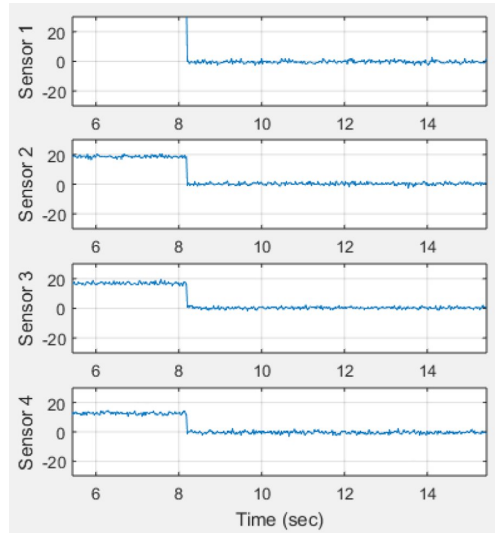


Figure 20: Second calibration result

The *FilterData* function, that is the function that performs this filtering, neglects the first two thirds of the filtered data and then takes the mean of the other samples. This value is computed for each sensor. In Figure 20, we can see the effect of this calibration on the measures of the sensors. Because of the pressure given by the mass of the balance, all the sensors were giving in output values greater than zero. We filter

these noisy signals, take the mean and then we subtract these correction values to the next measurements. As figured in the plots, all the sensors experience a force of $0N$ after the *calibrationWdrift* execution.

3.3.3 CalibrationP

For this calibration the position of the device remains as set in the second calibration. To use the weight application device we again use the mechanical structure mentioned in the first calibration. This time we can not use the box, otherwise we would change the position of the balance. We used the mechanical structure as it is shown in the Figure 15.



Figure 21: CalibrationP setup

The weight is always the same ($18.77kg$) and by moving the metallic support we were able to apply steadily the load in all the points without moving the balance. As can be seen in Figure 21, we covered the balance with transparent tape then we drew a grid on the tape with unit measurements in centimeters. With this grid we were able to mark the points where we wanted to measure. CalibrationP is for the formula that computes

the position of CoP. The idea is to place our custom weight application device with test load in three specific points of the WBB. In each point we keep the weight application system stable and still and in the meanwhile the balance records the data. Once the data was recorded, the matlab function filters the signals like it has been done in the previous calibration function (same filter too) and it computes the errors between actual position and measured position for both x and y coordinates. The matlab script stores these two errors for all the positions and then computes the average error in x and y coordinates.

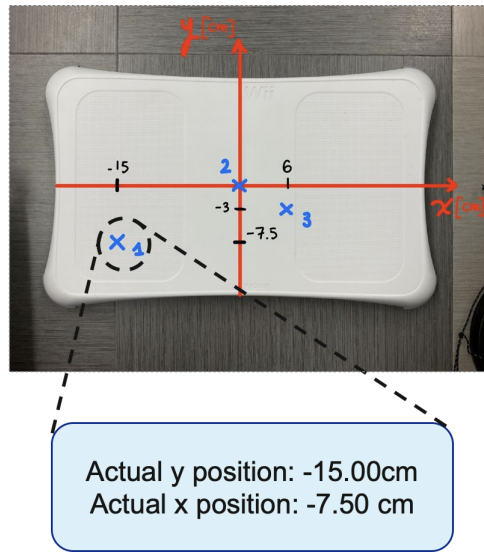


Figure 22: Map of points selected for calibration

These two averages will be the last correcting factors and they will be subtracted to the future calculations of the CoP. In Figure 22 it is possible to notice the positions we decided to explore for this calibration. With respect to the reference system, the positions $[x,y]$ are: $[-15, -7.5]$, $[0,0]$ and $[-3,6]$ where the unit of measure is centimeter and

these coordinates have been chosen randomly. The points selected are represented by the blue crosses drawn on the balance. For all the positions we applied the weight and we collected the data. In order to evaluate the measurements we created an histogram that plots the frequency of each value measured. Here is an example for position 1:

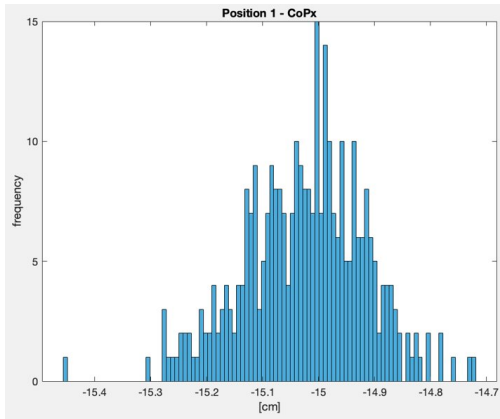


Figure 23: Measures of CoPx

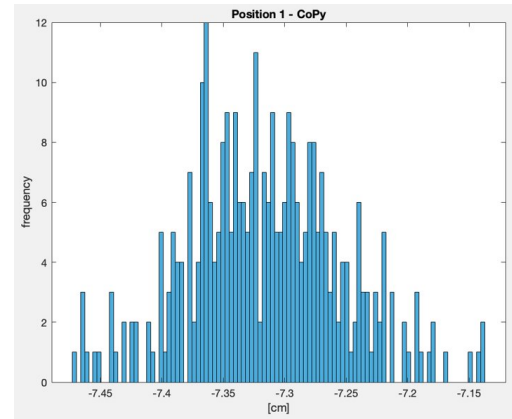


Figure 24: Measures of CoPy

In the Figure 23, we have the x coordinate measured by the balance. With mean equal to -15.0253cm and standard deviation of 0.109cm . For the y coordinate instead we got a mean value of -7.3155cm and standard deviation of 0.062cm . We can therefore easily deduce errors of -0.1845cm for the vertical (y) coordinate and 0.0253cm for the horizontal (x) coordinate. For this example we can also show the signal analysis of the data coming from the sensors. Here we have the frequency of the Newtons measured by the transducers.

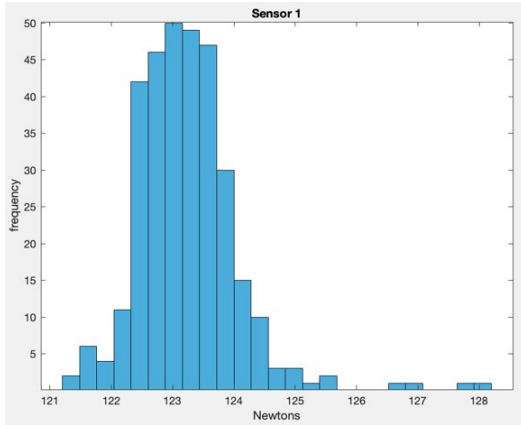


Figure 25: Measures from sensor 1

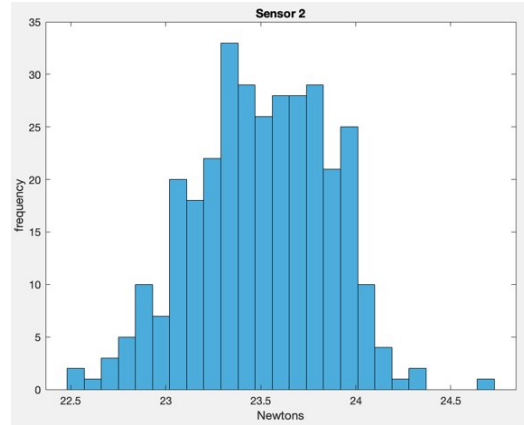


Figure 26: Measures from sensor 2

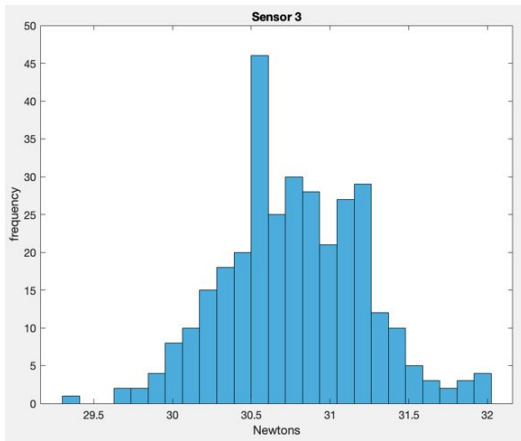


Figure 27: Measures from sensor 3

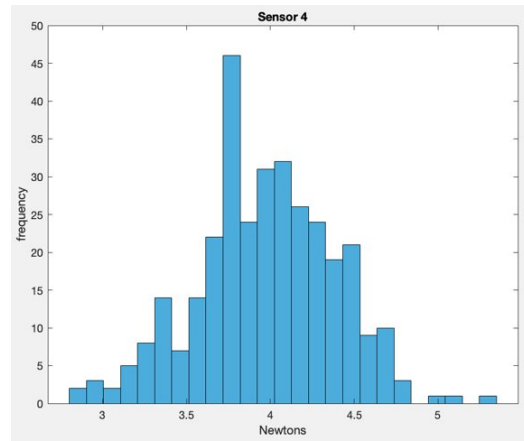


Figure 28: Measures from sensor 4

Sensor one is the closest to the point of application of the weight and it has the highest mean ($123.2447N$). Sensor four is the farthest and it has the lowest mean ($3.9712N$).

3.4 Two Wii Balance Boards integration

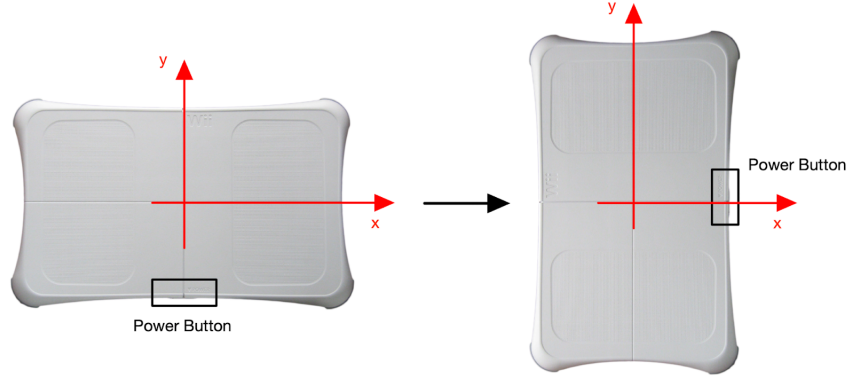


Figure 29: Reference system transformation

Our objective is to analyze the pattern of the Center of Pressure (CoP) signal during the squat activity. When employing a single balance board for both feet, it becomes unfeasible to effectively detect the CoP. As per its definition, the center of pressure is inherently associated with a single foot only. Consequently, utilizing a single WBB and maintaining a state of perfect balance with both feet on it would result in the CoP position being registered somewhere between the two feet. Such a scenario aligns with the concept of balance rather than that of the Center of Pressure. Because of this, once we were able to connect and calibrate one balance, we worked on the code in order to make it easier to connect and save the data coming from two WBBs simultaneously. Basically we rotate the balance anticlockwise and we had to change the reference system accordingly with the convention mentioned in [7]. We changed the points coordinates of the calibration function called `calibrationP`. In fact, we maintained the same points but because of the new reference system, we changed their coordinates in $[7, -15]$, $[0,0]$

and [3, 6]. In general the main matlab script has been changed in order to connect and retrieve the data from two balance boards simultaneously. In the following lines we can explain in detail the main Matlab code.

```

1  plot = true;
2  wiiR = WiiBwCalibration('RIGHT', plot, 70, 0, path);
3  CalibrationModule(wiiR,0,0,0,'RIGHT', path)
4
5  wiiL = WiiBwCalibration('LEFT', plot, 70, 0, path);
6  CalibrationModule(wiiL,0,0,0,'LEFT', path)
7
8  Measure = 1;  % 1 if you want to measure, 0 if not
9  i = 1;
10 %% MEASURING
11 while Measure == 1
12     input('\\n\\nPress Enter to start measuring\\n');
13     tic()
14     t=toc();
15     fprintf('Press A on 1st balance to stop measuring..    ');
16     m = 0;
17
18     starting_time = datestr(now, 'HH:MM:SS:FFF');
19     while ~isButtonPressedP(wiiR.bc.bb,'A')
20         cycleStart_sec = toc();
21
22         % query the wii for data
23         wiiR.update(plot);
24         wiiL.update(plot);
25
26         % when the Plot is off, this will help the user in order to see
27         % if the recording is going on
28         if m == 75
29             fprintf(' Recording...\\n')
30             m = 0;
31         else
32             m = m + 1;
33         end
34         % pause before continuing cycle (subtracting the time
35         % taken to process this loop itself
36         cycleDur_sec = toc()-cycleStart_sec;
37         pause(1/wiiR.Fs - cycleDur_sec);
38
39     end
40     fprintf(' Measuring %d done\\n\\n', i)
41     toc()
42

```

```

43      % save data
44      wiiR.bc.saveAndClearAllData( 'RIGHT', false, path, starting_time);
45      wiiL.bc.saveAndClearAllData( 'LEFT', false, path, starting_time);
46
47      Measure = input('Enter 1 to start measuring again or 0 to stop: ');
48      i = i+1;
49  end
50  % clear memory
51  wiiL.bc.delete();
52  wiiR.bc.delete();

```

Code 3: Portion of WBB.m

This Matlab file called *WBB.m* represented in Code 3 is the main file. Most of the commands are repeated because they are for the two balance boards. On lines 2 and 5 there is the command *WiiBwCalibration* that is the initialization function for the class mentioned few section above. This class takes the properties of the library *WiiBalance* plus the methods defined for calibrating the device. The first input of this function is the label that can be *'RIGHT'* or *'LEFT'*, then there is *plot* (true or false) for showing the plots during the collection of the data. It was important to implement this option because enabling the plotting when more than one WBB is connected can really affect the sampling frequency of the software. The first input number represents the sampling frequency. The second number instead is the number of samples we neglect every second only for updating the plots. This option can be a trade-off in case it is important to show the plots but on the other hand it is necessary to reach a sufficient high sampling frequency. The last entry of this function is the path in the computer for accessing files with old calibration parameters or for saving performed measurements. In lines 3 and 6 there is the *CalibrationModule* function. This function has been defined for starting the calibrations. The first entry is the WBB of interest. The following three

inputs can be only "0" or "1" where "1" is for running the calibration. The first "0" corresponds to the *NewtonTransform* calibration then we have *CalibrationWdrift* and *CalibrationP*. The user, therefore, can decide which calibration to perform. It is not necessary to perform all the calibrations every time.

Another important detail is in lines 23 and 24. The method *update* is the actual function that sample the data from the balance boards. Lines 44 and 45 are responsible for saving the collected data and lines 51 and 52 are important for clearing the memory and disconnecting the devices.

Chapter 4

IMPLEMENTATION - Control System Interface

Chapter 5

IMPLEMENTATION - Optimization Interface

The main goal is to personalize the assistance of an ankle exoskeleton through incorporating user ECG feedback. The optimization algorithm implemented in Python plays a crucial role in this scope, enabling the discovery of optimal solutions to assist the subject wearing the exoskeleton. However, the initial version of the optimization algorithm lacked a comprehensible graphical interface, hindering its accessibility for non-experts in the domain [11]. In this project, we address this limitation by developing a user-friendly application to enhance the configurability and real-time monitoring of the optimization process. The previous implementation required users to manually access the correct directory, access a YAML file and then modify it to change the initial settings, which proved cumbersome for non-technical individuals. Additionally, the absence of real-time progress tracking, pause functionality, and other features further restricted the algorithm's utility.

While the missing functionalities could have been incorporated by adding a few lines of Python code, we opted to create a dedicated application. This approach was

chosen to ensure that users, including those unfamiliar with programming languages, could intuitively interact with the optimization algorithm without requiring extensive technical knowledge.

The newly designed application introduces an intuitive graphical interface, simplifying the process of modifying initial configurations. Furthermore, it allows users to monitor the optimization process in real-time, offering pause and resume capabilities, and providing access to view other features of the optimization process.

In this chapter we will describe the design. In *Software structure* we will show all the Python scripts that we had to implement in order to make the structure reliable and efficient. It will also focus on the design of the *Server.py* file. In *GUI Design* we will talk more in detail about all the features of the web application.

5.1 Software structure

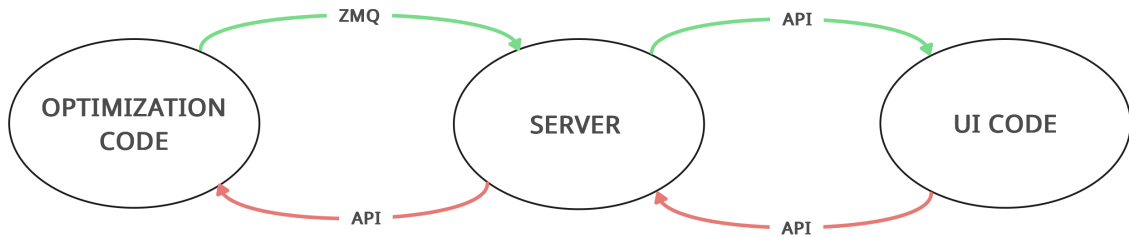


Figure 30: Software main architecture

In this part of the explanation we will often mention an *Optimization Code*, by this we mean the set of Python scripts that had been developed in order to run the Bayesian optimization algorithm. In other words, it is the HIL toolkit described in

[11]. This code contains also the scripts needed to detect and retrieve the data from the ECG sensor. This optimization algorithm is responsible for creating all the outputs that we want to show in the UI. For *UI code* instead, we refer to all the Python scripts that are necessary to build the web application and to make working all the callbacks that it has within. The goal was to design something that was independent from the optimization process. The rationale behind this decision is straightforward: it is possible to encounter errors during the execution of the Bayesian optimization, which may result in a terminal failure. In such a scenario, it is essential to prevent the application from failing altogether. Instead, we aim to enable the user to continue viewing the outputs and, if necessary, receive appropriate notifications regarding the possible errors the optimization process is undergoing. Moreover, it is imperative to ensure that any errors within the UI component do not in any way compromise the optimization process. It is important to bear in mind that the optimization procedure can span anywhere between 15 to 20 minutes approximately. Throughout this duration, the subject is required to perform squats, a physically demanding activity. Consequently, it is absolutely important that a simple UI error does not affect the optimization process, leading to the need for a complete restart of the entire experiment. Ensuring the separation of UI and optimization is essential in this context. The clear distinction between these components will promote a more reliable and resilient system, guaranteeing that both the user interface and the optimization algorithm can proceed harmoniously and without any undue interference.

For this reason, we decided to develop a third Python script called *Server.py*. This file is essential for the separation of the processes mentioned before. Its goal is to constantly listen to the optimization code in order to get new data, store it and share it whenever the UI calls for downloading the information. In Figure 30 we have green and red arrows. Green arrows represent all the data created by the optimization process and shared with the UI. The red arrows instead represent the command that the UI is able to send to the optimization code. These commands are “0” or “1”, they stand for “PAUSE” and “RESUME”. Moreover, on each arrow there is a label. That label says the protocol or mean through which the data is shared. We used the protocol ZMQ for sharing in an asynchronous manner the data of the optimization with the Server class. For the remaining connections we used a REST API.

5.1.1 REST API

REST API stands for Representational State Transfer Application Programming Interface and it is a set of rules on how to connect and communicate between client and server. In particular a REST API is an API that respects a specific architectural style. This style uses HTTP requests to access and use data. The methods that can be used by client/server are GET, PUT, POST and DELETE which refers to reading, updating, creating and deleting data on a specific endpoint of the server. The working principle is that clients can make a HTTP request described by one of the methods listed before. Then the server will respond with a status code. For example “200” means “Ok” and “404” means “Not found”. In order to exploit this communication method

we imported the python library called *Requests*¹. The Python library *Requests* is a widely used and essential tool for handling HTTP-based interactions within Python applications. It provides developers with a simple powerful interface to send various types of HTTP requests and effectively communicate with APIs. By abstracting away the underlying complexities of network communication, it significantly reduces the development time and effort required to implement HTTP functionalities within Python applications. Code 4 represents a portion of *Server.py*. This is an example on how to share the data when the client makes a request.

```

151 @app.get('/OptimizationData')
152 def list_OptimizationData():
153     return saved.share_data()

```

Code 4: Function for sharing data with UI

Whenever the client asks for a GET method on the endpoint `‘/OptimizationData’` the Server returns to the client the outputs of the function *share_data()*. This function simply gives all the data related to the optimization stored by the server in format json. JSON (JavaScript Object Notation) is a text-based data format. It is a set of key-value pairings, where the value can be a number, string, object, array, or boolean and the key must be of the string type.

Here instead we can see the Python command for the client in order to download the data from the server.

```

18 def download_data(obj, config):

```

¹<https://pypi.org/project/requests/>

```

19     n_parm = config[ 'Optimization' ][ 'n_parms' ]
20     server_flag = requests.get( f'http://{obj.serverIP}:{obj.serverPort}/
    OptimizationData' )
21     if server_flag.status_code == 200:
22         obj.flags[ 'server' ] = 'ON'
23     else:
24         obj.flags[ 'server' ] = 'OFF'
25     data = server_flag.json()

```

Code 5: Downloading data from Server

In Code 4, the output of the command on line 20 is the status code of the response by the server. If the response corresponds to 'ok' then we can save all the data in line 25 using the method `.json()` of the library *Requests*.

5.1.2 ZMQ

Optimization code and Server use the ZMQ library in order to exchange the data. ZMQ² is a high-performance, open source asynchronous messaging library. It is built in C++ but there are several libraries translating this set of commands for other programming languages. ZMQ allows you to send messages (binary data, simple strings, objects) over the network through various methods like TCP (*Transmission Control Protocol*). The messages are exchanged through sockets. The sockets differ from type of message and properties. Types of socket are for example REQ/REP and PUB/SUB. We decided to use the PUB/SUB type because, unlike REQ/REP, in this case the publisher pushes the messages out and all the associated subscribers receive the messages without sending responses or acknowledgments. After some tests, we understood it was the best method for exchanging our data. In order to use this library we

²<https://pyzmq.readthedocs.io/en/latest/>

created a class called *ServerCommunication*. In its initialization function there are all the steps to create the socket. In input we have to specify the host IP, the port and the communication type (if it is a socket for sending or receiving). These three inputs are defined in a YAML file called *ECG_configs.yml* that is situated in the folder "configs" of the HIL toolkit. In Code 6 there are two examples of inputs needed for creating a socket. This is a portion of code of *ECG_configs* related to the communication settings.

```

22 Communication:
23     Sending: true
24     RECEIVING: true
25     IP: tcp://192.168.1.49
26     HIL:
27         Cost samples:
28             NAME: Cost_samples
29             TYPE: send
30             PORT: 4501
31         FLAGS:
32             NAME: FLAGS
33             TYPE: send
34             PORT: 4507

```

Code 6: Setting up the ZMQ communication

As clear from Code 6, on line 23 and 24 we enable the communication for sending and receiving data. On the 25th row we set the IP of the device where we run the optimization. The IP is the address by which the device is recognized in the local network. Consequently, for each variable we define name, type and port. Every channel of communication has a different port. For example *cost_samples* is the vector containing the sample of the cost function. We defined this variable as type “sending” and all the messages containing this variable will be exchanged on port 4501. In the class mentioned before, *ServerCommunication* we have listed also the methods that we

can use for all the sockets created.

```

22 def send (self , message: str) -> None:
23     self._socket.send_string(u=message)
24
25 def send_obj(self , obj: object) -> None:
26     self._socket.send_pyobj(obj)
27
28 def send_json (self , json: dict) -> None:
29     self._socket.send_json(json)

```

Code 7: ZMQ sending functions

Code 7 shows the three types of sending functions. Command on line 22 is for sending strings. Command on line 27 is for sending python objects like lists or dictionaries and command on line 31 is for sending messages in json format. We can show the same methods for receiving.

```

22 def receive (self -> str None:
23     try:
24         obj = self._socket.recv_string () #flags=zmq.NOBLOCK
25     except zmq.error.Again:
26         obj = None
27     return obj
28
29 def receive_obj(self) -> object:
30     try:
31         obj = self._socket.recv_pyobj ()
32     except zmq.error.Again:
33         obj = None
34     return obj
35
36 def receive_json(self) -> Any:
37     try:
38         obj = self._socket.recv_json ()
39     except zmq.error.Again:
40         obj = None
41     return obj
42
43 def recv_bytes(self) -> bytes None:
44     try:
45         obj = self._socket.recv ()

```

```

46     except zmq.error.Again:
47         obj = None
48     return obj

```

Code 8: ZMQ receiving functions

We have similar commands for receiving ZMQ messages in the Server code.

5.1.3 Server Architecture

The Server file is divided in two sides: ZMQ and REST API side. The ZMQ side is the one responsible for receiving the data coming from the optimization code and the REST API side is connected to the UI code. For the first side, it is fundamental to be able to run many subprocesses in parallel. We have numerous variables, leading to a substantial number of channels for exchanging messages between Bayesian algorithm and UI. The conventional approach of using a single loop to sequentially receive messages from all the channels is proven to be inefficient. A potential issue arises when the sender transmits a message through a channel that is not actively being monitored at that moment due to our focus on another channel, resulting in information loss. To address this challenge, we propose employing Python's *Asynclibrary*³. This powerful tool enables the execution of multiple subprocesses asynchronously and in parallel. By adopting this approach, we can concurrently run numerous subprocesses, with each subprocess dedicated to listen to a specific channel. Consequently, we can create subprocesses equal to the number of sockets or channels utilized for exchanging messages coming from the optimization code. Such an arrangement ensures that each subpro-

³<https://docs.python.org/3/library/asyncio.html>

cess remains continuously attentive to its designated channel, effectively eliminating the risk of missing any message. In the following figure we can see an example of an asynchronous process for the acquisition of ECG data.

```

87 async def data_ecg():
88     sock = ctx.socket (zmq.SUB)
89     sock.subscribe("")
90     sock.connect(f"{saved.IP}:{saved.port}09")
91     while True:
92         try:
93             msg = await sock.recv_json(flags=zmq.NOBLOCK)
94         except zmq.ZMQError:
95             msg = None
96         if msg is not None:
97             saved.ecg = msg
98             await asyncio.sleep(0.1)

```

Code 9: Server's subprocess for receiving ZMQ messages

We have seven subprocesses. Each of them looks like the one in Code 9 and it has the three commands that are shown in lines 88, 89, 90 for creating the socket and binding it with the sender at the specific address defined by IP and port. Then we have a While loop. In this loop we constantly try to receive a message, if we don't get anything the variable `msg` will be set to `None`. If we receive a message, instead, this will be saved in a local variable called *saved.ecg*. Because of this design with seven subprocesses, every time the system encounters an await function, like on line 93, it means that the system doesn't need to wait for that function to finish but it can move on with the other functions or subprocess. The other side of *Server.py* is related to the Flask app, the REST API that provides the data to the UI. We mentioned the local variable *saved.ecg*. *saved* is an object of the class *Store* that is defined in the beginning of *Server.py*. This

class is important for initializing, storing and sharing all the variables with the web application. Code 10 shows two methods defined in the class *Store*.

```

16 def reset_data(self):
17     self.plot = {'x': [], 'y': []}
18     self.gp = {'mean': [], 'x': [], 'y': []}
19     self.ecg = []
20     self.acq = None
21     self.hyp = {'likelihood.noise_covar.raw_noise': [],
22                'mean_module.raw_constant': [],
23                'covar_module.raw_outputscale': [],
24                'lengthscale_parm1': [],
25                'lengthscale_parm2': [],
26                }
27     self.state = "OFF" #state of the optimization process
28     self.hrv = None
29     self.opt_comand = "1" #command from UI for optimization
30
31 def share_data self:
32     in_memory_datastore = {
33         "data_plot": self.plot,
34         "data_gp": self.gp,
35         "data_acq": self.aca,
36         "data_hyp": self.hyp,
37         "data_ecg": self.ecg,
38         "data_hrv": self.hrv,
39         "state": self.state}
40     return in_memory_datastore

```

Code 10: Methods used by Server code

The function *share_data* on line 31 is called by the client. Indeed the client makes a request with method GET on the endpoint *'/OptimizationData'* and the server will respond by giving the output of the function *share_data*. This connection between client requests and *share_data* is clear from the callback shown in Code 11.

```

151 @app.get('/OptimizationData')
152 def list_OptimizationData():
153     return saved.share_data()

```

Code 11: Callbacks for sharing data with UI code

As explained before, UI code can make different kinds of client requests. Here is Code 12 showing other callbacks in response to client's requests.

```

155 @app.get('/OptCommand')
156 def sendOptcommand():
157     return saved.opt_comand
158
159 @app.post('/OptimizationData')
160 def reset_OptimizationData():
161     saved.reset_data()
162     return 'data reset'
163
164 @app.post('/OptResume')
165 def resume_opt():
166     saved.opt_comand = "1"
167     return 'comand received'
168
169 @app.post('/OptPause')
170 def pause_opt():
171     saved.opt_comand = "0"
172     return 'comand received'

```

Code 12: All callbacks used by Server code

Briefly the first callback is for the Optimization code. The HIL code can download the command "RESUME" or "PAUSE" set by the UI. The second callback is for clearing all the stored variables in the Server. The last two callbacks, `resume_opt` and `pause_opt`, are used by the UI to send the command of "RESUME" or "PAUSE" to the optimization process. Indeed with these two callbacks we basically change the local variable called `saved.opt_comand` and with the callback on line 155, the optimization code retrieves the value of `saved.opt_comand` from the Server.

5.2 GUI design

The design of this application is based on Dash library ⁴. Dash by Plotly is a dynamic web app framework that allows developers to create interactive web applications using Python. It uses Plotly’s visualization features to transform data into user-friendly charts, and graphs. Dash’s straightforward syntax integrates data manipulation, visualization, and user interaction, making it a versatile tool for creating intuitive web interfaces.

The Web application is constituted of two layers. The first layer is fixed and it contains the title of the application, two badges that give some information to the user and then we have a row that is a list of tabs. These tabs are Initialization, Optimization, Signals and Hyperparameters tab. This first layer occupies the top row of the screen and it is fixed, it’s always present. The most important features of this layer are the badges. These badges are indicators of the status of Server code and Optimization code. Based on which color they get, they represent a different status. Table 1 describes the status for each color of the badges. For Server there are only

	RED	GREEN	ORANGE	BLUE
Server	Not working	Working	-	-
Optimization	Not started	Finished	Exploration	Optimization

Table 1: Legend for badges’ colors

two colors because only two status are possible. The user has to know that if the Server badge is red that means that some error may occurred on the Server code or

⁴<https://dash.plotly.com/minimal-app>

the it hasn't been started yet. In this situation it is still possible to run the Bayesian algorithm but it won't be possible to see the outputs on the UI. For the optimization badge instead there are 4 states. Red when the optimization hasn't been started yet, orange when the optimization started and it is in the Exploration phase. Finally blue when the Bayesian algorithm is in the Optimization phase.

The second layer of this UI describes the remaining part of the screen and its configuration depends on which tab has been selected by the user. When UI code is executed and we open the application for the first time, the initial tab selected is the Initialization tab.

5.2.1 Initialization tab

The screenshot shows the 'RRL: Human in the Loop optimization' application interface. At the top, there are two tabs: 'Server' (green) and 'Optimization' (blue). Below these are four main sections: 'Initialization' (selected), 'Optimization', 'Signals', and 'Hyperparameters'. The 'Initialization' section contains the following controls:

- Number of parameters:** A horizontal slider with markers from 1 to 6. The slider is currently set to 2.
- Insert number of steps for optimization:** A text input field labeled 'n_steps' with a value of 2.
- Insert time for each step:** A text input field labeled 'seconds' with a value of 1.
- Parameter 1:** A range selector with 'min' and 'max' values, both set to 0.
- Parameter 2:** A range selector with 'min' and 'max' values, both set to 0.
- Select type of Gaussian process:** A dropdown menu with 'Regular' selected.
- Select type of Acquisition function:** A dropdown menu with 'ei' selected.
- SUBMIT** button: A large grey button at the bottom right.

Figure 31: Initialization tab

In Figure 31 it is shown the appearance of the first tab called *Initialization*. This tab is the interface through which the user is able to change the settings before starting the optimization algorithm. There are many features that can be set for the HIL toolkit, however only the most important settings have been displayed for the UI. The user can select the number of parameters to optimize with the slider in the top left part of the app. By changing the marker's position on the slider, we change the number of boxes in the central column of the screen. If the slider is set on 5 parameters for example, the central column will display 5 couples of boxes, one couple for each parameter. In this central column, the boxes on the left are for setting the minimum value the parameter can be and the right boxes are for setting the maximum. By taking a look at the bottom of the slider it is possible to notice two other boxes. The upper box is useful for determining how many steps and in particular samples, the user wants to analyze for the optimization. The second box determines how many seconds each step should last. Finally, the right column has two dropdown components. By clicking on the first dropdown we can select one of the available Gaussian process models. By clicking on the second it is possible to select the type of Acquisition function.

5.2.2 Optimization tab

Figure 32 illustrates the second tab titled *Optimization*. The primary purpose of this window is to present the evolution of the two main functions utilized in the Bayesian optimization algorithm: the Gaussian process and the Acquisition function. Furthermore, within this tab, users have the capability to manage the optimization

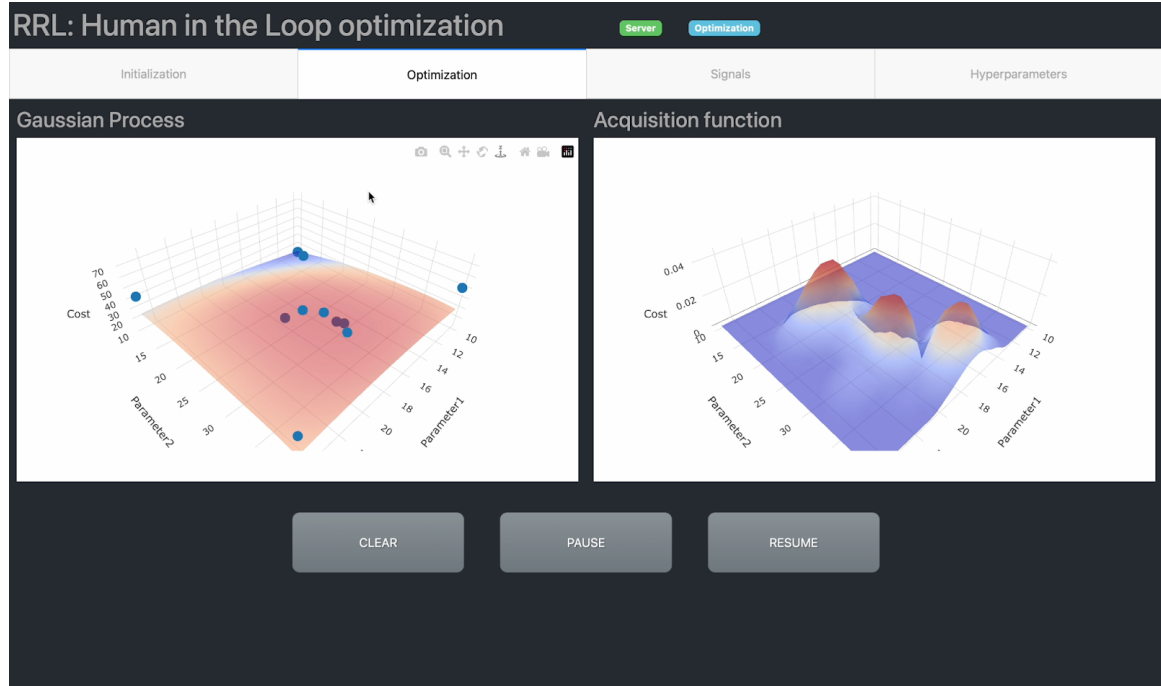


Figure 32: Optimization tab

process through the utilization of three buttons located at the bottom of the window. On the left-hand side of the window, a graph is provided to plot the cost samples represented by blue points, which are utilized in the computation of the Gaussian process. Additionally, the graph displays the Gaussian process itself in the form of a colored surface. A useful feature of this graph is that by positioning the cursor on any of the blue points, a small dropdown is triggered, displaying the corresponding coordinates' values for that particular point. On the right-side of the window instead, it is possible to follow the evolution steps of the Acquisition function. Both the Gaussian process and the Acquisition function are interactive, as users can manipulate the orientation of the surface by dragging it. This enables users to observe the surfaces from various perspectives. However, it is worth noting that the application periodically updates

the surfaces at fixed intervals of every three seconds, restoring their orientation to the original position during each update. Three seconds should ensure that users have enough time to explore different perspectives before the update takes place. The three buttons available in this tab are CLEAR, PAUSE, and RESUME. When users activate the CLEAR button, the application initiates a POST request, prompting the client to request a reset on the server. Upon receiving this request, the server proceeds to delete all variables that were saved up to that point. Consequently, clicking on this button results in clearing all plots within the application, rendering them empty. As for the PAUSE and RESUME buttons, their functionality is explained in detail in the "server architecture" section. These buttons allow users to halt and subsequently resume the optimization process as needed.

5.2.3 Signals tab

This section talks about the third tab labeled *Signals*. Here, we explore other essential signals related to the process. In the top left corner, we find the Biofeedback signal, which, for this project, is represented by the ECG signal used to compute the cost. The bottom left plot displays the trend of the RMSSD value, representing the cost computed using the ECG data. A slight difference exists between the RMSSD plot and the one in the bottom right of the screen. The plot on the bottom left illustrates the cost instant by instant, showing its variations over time. The optimization process samples this function to identify the optimal combination of parameters. All the sampled values from the cost function are shown in the bottom right plot. Lastly, the top-right

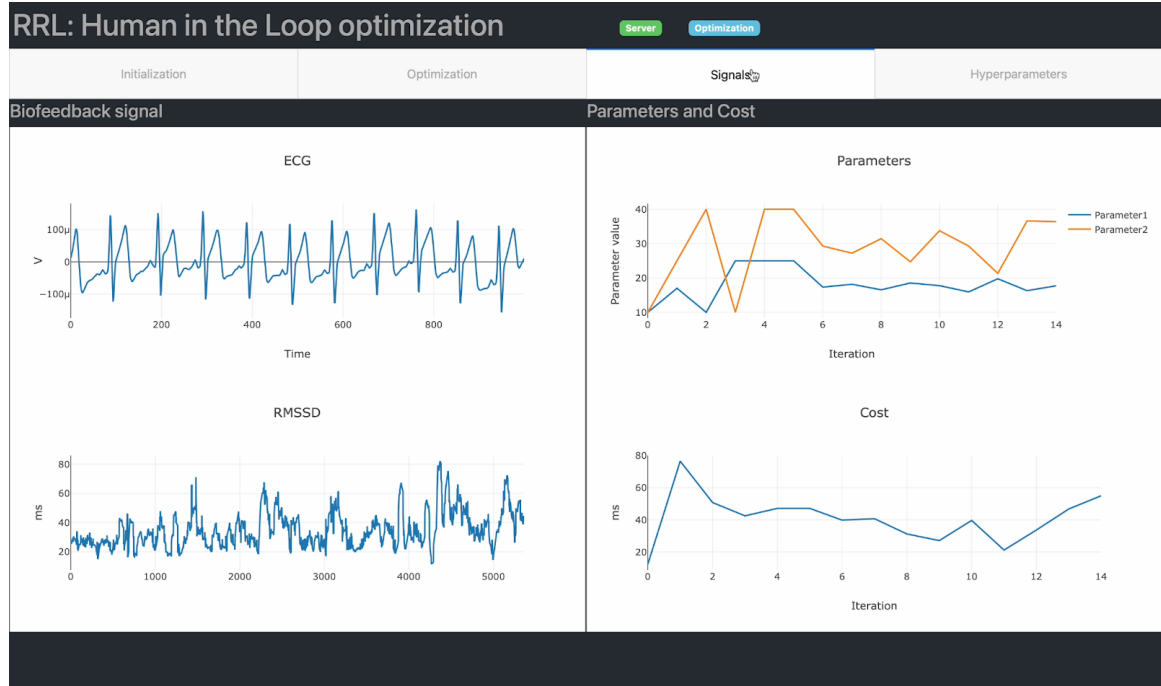


Figure 33: Signals tab

graph demonstrates all the parameter values investigated during the exploration and optimization phase. This graph allows the user to understand how distributed the exploration phase is or whether instead the optimization process found the optimal point more quickly.

5.2.4 Hyperparameters tab

The fourth tab is useful for evaluating the trend of the hyperparameters. Hyperparameters are some parameters describing the Kernel used for calculating the GP at each iteration. As already discussed in chapter 2.1 and in [11], these hyperparameters are very hard to set before starting the optimization. For this reason, a parallel machine-learning algorithm has been employed in the toolkit only for tuning the best

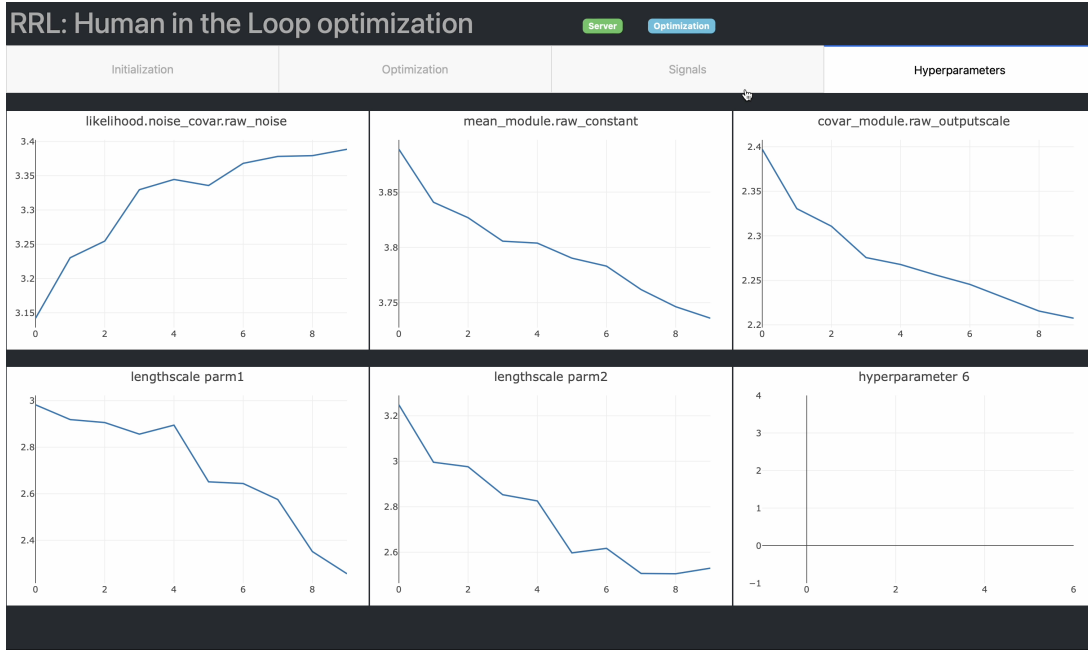


Figure 34: Hyperparameters tab

value for the hyperparameters. Thanks to this tab, the users can evaluate the quality of the tuning of the hyperparameters. In fact, by looking at the plots showing the values over time of these hyperparameters, the user can check the convergence. If a hyperparameter converged to a certain value, there is a better chance of having achieved a good tuning of that hyperparameter. The number of hyperparameters depends on the number of parameters that we want to optimize for the wearable robot of interest. In particular the hyperparameters are as many as the optimized parameters plus three. In the example shown in Figure 34, we were testing optimizing two parameters that result in five hyperparameters to show. Indeed five graphs are filled out with the relative data and the sixth plot on the bottom right is empty. The idea behind this tab is to create as many graphs as number of hyperparameters. If there are more than

six graphs, the user can scroll down this window and check all of them.

Chapter 6

IMPLEMENTATION - System Integration

Chapter 7

EXPERIMENTS

Chapter 8

CONCLUSION

Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion
Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Con-
clusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclu-
sion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion
Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Con-
clusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion Conclu-
sion Conclusion Conclusion Conclusion Conclusion Conclusion Conclusion

APPENDICES

CITED LITERATURE

- [1] Inigo Sanz-Pena, Hyeongkeun Jeong, and Myunghee Kim. “Personalized Wearable Ankle Robot Using Modular Additive Manufacturing Design”. In: *IEEE Robotics and Automation Letters* (2023).
- [2] Fred Shaffer, Rollin McCraty, and Christopher Zerr L. “A healthy heart is not a metronome: An integrative review of the heart’s anatomy and heart rate variability”. In: *Front. Psychol.* (2014).
- [3] Michael Scott, Graham Kenneth S., and Davis Glen M. “Cardiac autonomic responses during exercise and post-exercise recovery using heart rate variability and systolic time intervals—A review”. In: *Front. Physiol.* (2017).
- [4] Laurent Mourot et al. “Quantitative Poincaré plot analysis of heart rate variability: effect of endurance training”. In: *Eur. J. Appl. Physiol.* (2004).
- [5] Jihoon Kim et al. “Soft wearable flexible bioelectronics integrated with an ankle-foot exoskeleton for estimation of metabolic costs and physical effort”. In: *Nature Portfolio Journal* (2023).
- [6] Dominique Makowski et al. “NeuroKit2: A Python toolbox for neurophysiological signal processing”. In: *Behavior Research Methods* (2021).
- [7] Fabio Scoppa et al. “Clinical stabilometry standardization: Basic definitions – Acquisition interval – Sampling frequency”. In: *Gait & Posture* (2012).
- [8] Lena H. Ting, Harrison L. Bartlett, and Jeffrey T. Bingham. “Accuracy of force and center of pressure measures of the Wii Balance Board”. In: *National Institute of Health* (2014).
- [9] Julia M. Leach et al. “Validating and Calibrating the Nintendo Wii Balance Board to Derive Reliable Center of Pressure Measures”. In: *Sensors* (2014).

- [10] Jordan Brindza et al. “WiiLab: Bringing Together the Nintendo Wiimote and MATLAB”. In: *IEEE* (2009).
- [11] Prakyath kantharaju et al. “Framework for Personalizing Wearable Devices Using Real-Time Physiological Measures”. In: *IEEE* (2023).

VITA