CHIC PROGRAM
EPFL

ROOTS
Technical Report

Arthur Bricq
Leonardo Mussa

10/01/2021

Contents

1. Introduction to Roots

During the last semester dedicated to CHIC, our team **Roots** worked on the development of a new sensing technology for detecting interactions between users and plants (*or even, as we will see during this report, with any conductive object*) with much more precision than what we could achieve last semester. The work this semester involved:

— The miniaturization and upgrades of our previous PCB
— The development of a new sensing circuit which allows to detect not only where there is an interaction but also what kind of interaction happened
— The software work to collect data, save it, and process it for machine learning classification
— The creation of a user-friendly interface to present to users a live visualization of the sensor's measurement.

This report will present all those steps starting with a brief reminder of what was done last semester.

1.1 Conclusions of last semester

Last semester, we designed and assembled a PCB which was a **touch sensor**, equipped with a battery and a charging circuit, that could communicate through a **wifi module** to a **host UDP server** held on a **raspberry pi machine**. The sensor was suited to be inserted into its **3D printed box**, itself planted inside a **plant pot**. When users touched the plant, the sensor would detect the touch events (*in a binary way*) and would send them to the UDP server. The server was implemented in Java and could communicate with **Processing**, an open source software which is the reference when it comes to visual design and electronics art. The Processing code generated a live stream of **interactive image**, specified by the graphic designer of our team.

(a) Picture of the set-up (b) The sensor and the central box

Figure 1.1 – An image and a rendering of our product.

Despite the successful achievement of all the points mentioned above, our work last semester had several drawbacks which can be summarized in the following list.

1. The built-in **touch-sensor circuit we used was not satisfactory**. First, it worked in a binary way: the output of the circuit was merely "**did someone touch / or not the object**". Second, even the binary detection wasn't achieving great results: depending on where one user would touch the plant, it would be too sensitive, or not enough. Tuning of the circuit was also impossible.

2. The **setup** to install / turn on the device also was not suited for an efficient deployment of the device. For the **software setup**, specifying the IP address of the UDP host server was required. Since the IP address of a computer often changes, it is not a satisfactory solution. Also, the **electronic setup** wasn't finished. For instance, the touch sensor wouldn't calibrate only when the sensor was turned on, meaning the device had to be hard reset each time before use. Finally, the **programming setup** was not trivial: uploading code required to (i) open the sensor box, (ii) connecting 4 cables to the PCB board, (iii) pressing synchronously 2 buttons, (iv) uploading the code. Since our product is targeting the **marker community**, all those steps are a barrier to an **easy set-up** of our device and its associated technology.
3. There was some minor mistakes in our PCB that needed to be corrected.

1.2 Orientation for the new semester

When we met again at the beginning of September, we had to make strategical decisions with respect to the continuation of the project. Here below are the decisions that were made.

— Focus the work on the development of a **better sensing technology**, which involve at the same time *(i)* the **design of a new PCB** to detect **touch events** on a object (most likely a plant, but as we will see not necessarily), and *(ii)* the **development of the software** required to perform a **classification** on the **type of touched event** that was measured by the sensor.

— Make our project **more accessible**, especially for the maker / open-source community, with **easy procedures** to have the working setup ready... and (well) working.

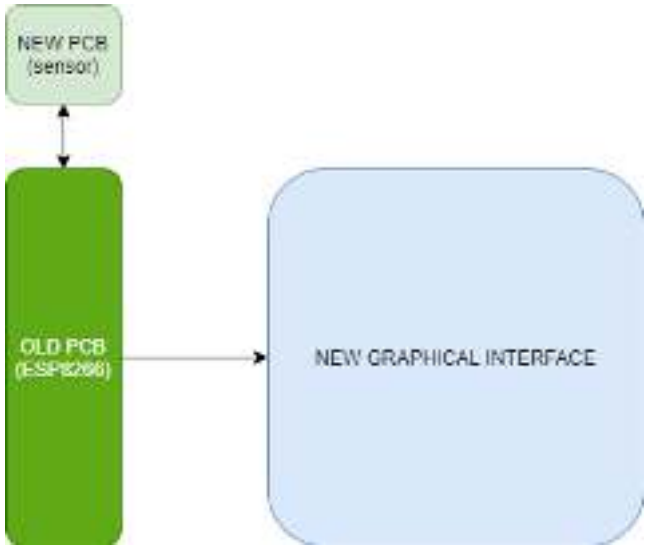


Figure 1.2 – The goals for this semester: design a new sensing PCB, upgrade the old one, and develop a graphical interface. Note that the new PCB is an independent module that can be connected to the old one via a Molex connector.

2. Upgrading of our previous PCB

One part of the work of this semester involved working on the **main PCB** of our project [1]. This section presents the work done on our main PCB board, which the continuation of the PCB that designed last semester.

2.1 A timeline of our PCB boards

The following image illustrates the 4 versions of PCB which were ordered this semester. The leftmost PCB is a corrected version of the first PCB which was ordered last semester. The rightmost PCB is our last stable version, which doesn't contain an embedded touch sensor (as does V1, V2 and V3) but instead a connector to connect to the sensing module.

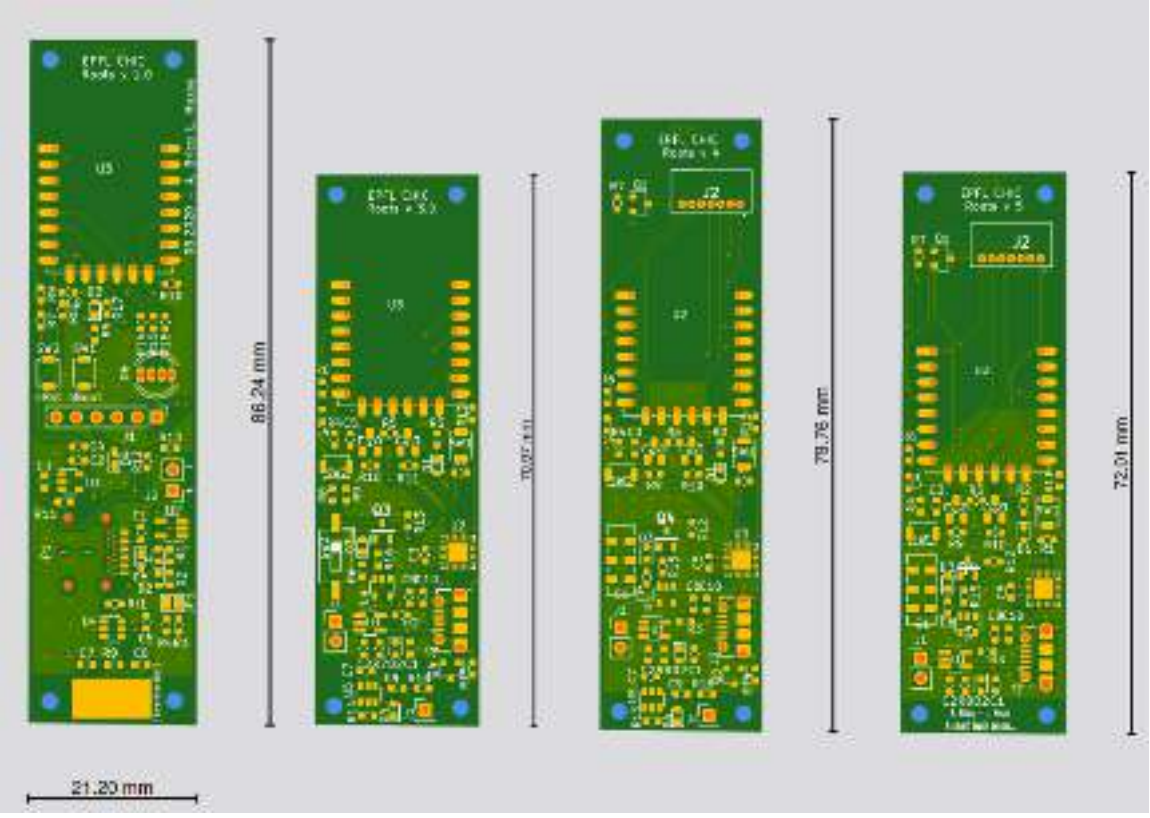


Figure 2.1 – 4 versions of the PCB that were ordered this semester. All boards have the same width.

1. We call this serie of PCB the main PCBs since we have designed another PCB called itself the sensing PCB

2.2 New functionalities

Over the different versions, several functionalities of the PCB changed as other remained rather similar. Let's go through the different changes and explore the functionalities of the last stable version V5 of our main PCB.

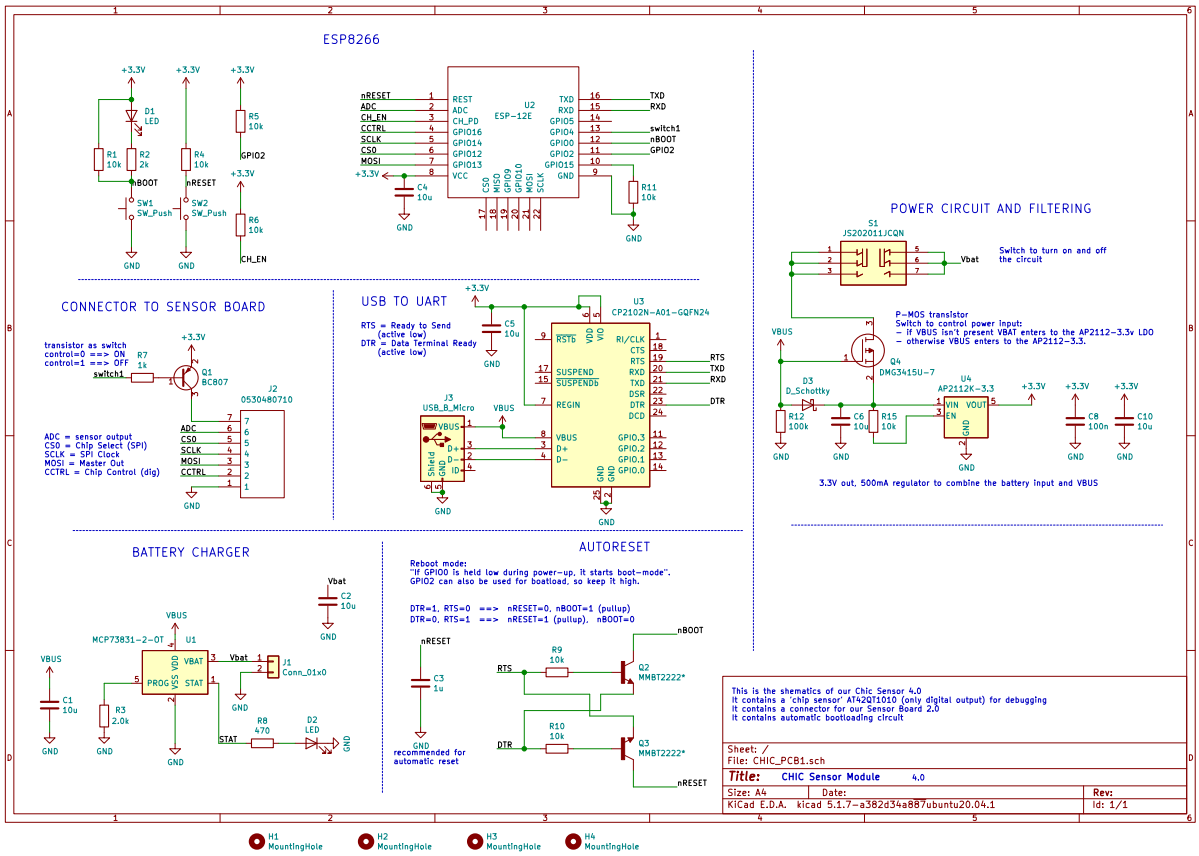


Figure 2.2 – The detailed schematics, last version of the main PCB.

2.2.1 What remained unchanged

Some functionalities of the board that were tested last semester and that proved to be working didn't change.

— The **charging circuit** to charge our battery did not change.

— The **microcontroller board** which is an ESP8266 proved itself very efficient and we did not want to modify it.

2.2.2 Better power circuit

Although the **voltage regulator** used to merge safely the different potential sources of voltage chip did not change (**AP2112K-3.3**), the input circuit did change for a safer charging of the battery.

First noticeable change (especially noticeable for the user-experience) is the add of a switch, allowing the user to turn OFF the board without unplugging the battery.

The second change is the use of a transistor to make a voltage switch between two potential voltage source: either the battery voltage (if high enough) or the USB-voltage (denoted as VBUS in the drawing). If the USB is not plugged, then Vbat will be carried to the input of the charging module. Else, it's going to be the USB voltage which will be used to power circuit, while also charging the battery.

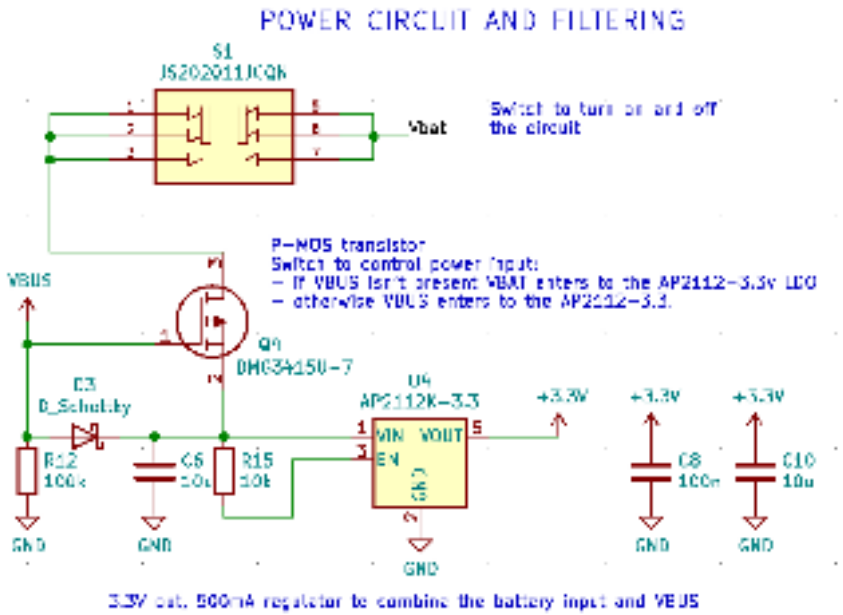


Figure 2.3 – Schematics of the powering circuit

2.2.3 Removal of built-in sensing circuit and connection of new sensing circuit

Since the focus of our work was to design a new sensor, it was logical to remove the last sensor that we used. This modification happened at our very last version (*indeed, it didn't make much sense to remove until the other sensor was finished, mainly for debugging.*), as one can see on the Fig. 2.1.

Also, the two last versions (V4 and V5) contain at the top of the board a connector with 7 pins. This enables to connect by the mean of a cable to link our main PCB to our second PCB: **the sensing circuit**, which will be introduced later. Here is the schematics of this connector, presented here to understand how the sensor communicates with the microcontroller.

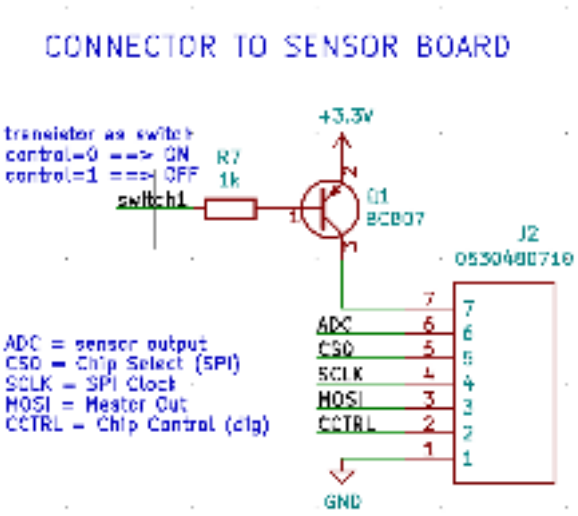


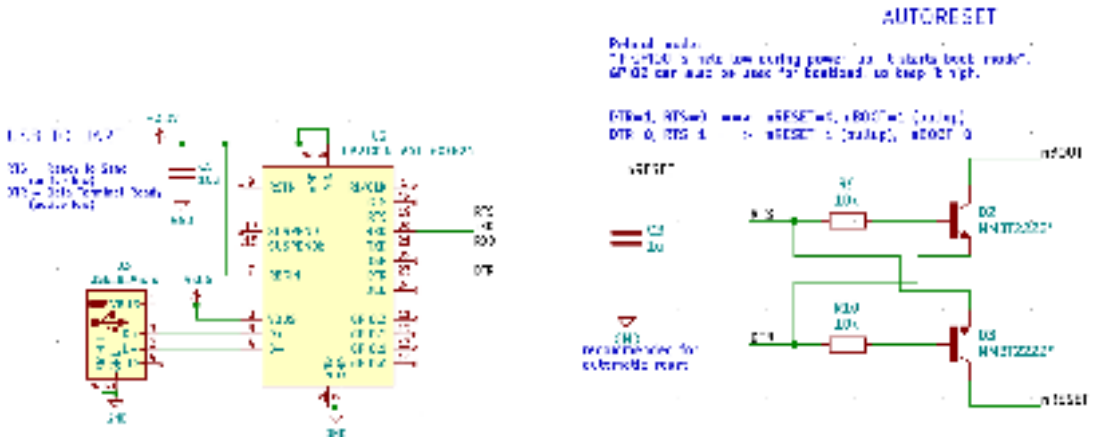
Figure 2.4 – Schematics for the connector

An SPI communication was implemented to **set the registers of the wave generator**, as the output read

from the sensor is the ADC pin (this pin needs to go to the ADC pin of the microcontroller). There is also a **PNP general purpose transistor** Q1 present, to control with the *switch1* pin of the microcontroller when to turn OFF the sensing circuit. When *switch1* is pulled up, the sensing circuit will not be powered anymore. This is really useful since it allows to **switch off** the sensing circuit when it is not used to **save power**, or to reset the circuit without rebooting the entire systems for instance to perform **calibration**.

2.2.4 Uploading code automatically via USB

As mentioned previously, uploading code in our previous versions of the PCB was a rather long process. In order to facilitate the **market entry** by allowing developers and makers to quickly and easily change the code, we added a Single-Chip USB-to-UART bridge for data transfer.



(a) USB to UART bridge

(b) Auto reset circuit

Figure 2.5 – Schematics for uploading / debugging code directly with USB port

This chip comes with an auto-reset circuit, that will automatically reset the microcontroller while pulling down the **GPIO0** pin, which tells the microcontroller to boot in flash MODE (hence, the MC is waiting for new code). This feature is enabled by 2 pins of the USB to UART bridge: **RTS** and **DTR**, working in pair with the **Espressif's Tool** for uploading code, accessible at this link and named 'esptool.py' which specifies how to properly connect those pins.

In a nutshell, **DTR** means Data Terminal Ready and indicates that the connected device is ready to receive data. **RTS** means Request To Send and indicates to the connected device that it wants to send data. It can be seen in source code of esptool.py that before sending the code, the *software* it is going to make sure that two signals are successively sent.

1. (DTR=1, RTS=0): it turns off the microcontroller
2. (DTR=0, RTS=1): microcontroller turns on while pin 0 is down: activates the flash mode

Once this is done, it will transmit the code for the boot-loader to flash the new version.

3. Our new sensing circuit

The first prototype of Roots was equipped with a simple ***charge transfer* capacitive sensor**, the AT42QT1012 chip. We chose this sensor because it is simple to use and, due to its popularity, there is a large amount of documentation available online. Moreover, it was possible to purchase *ready-to-use* development boards which turned out to be very useful during the fast-prototyping and early-integration stage of our project. Nevertheless, the simplicity of the AT42QT1012 comes at a price: the sensor can only detect if an object is touched or not, and it is **unable to detect more complex interactions**. To overcome this limitation we chose to design our own sensing circuit, which uses **electrical resonance** to recognize a wider range of gestures, while keeping the interface with the object simple: only **one cable** is needed to connect the device to the object. Our sensor was inspired by the work of Ivan Poupyrev and others, which propose a similar solution in their paper *"Touché: Enhancing Touch Interaction on Humans, Screens, Liquids, and Everyday Objects"* (accessible here). In our work, we were able to successfully replicate their results while opting for a more **modular design**, since our goal was to create a PCB which could be easily reused in other projects. In fact, our sensor is compatible with any micro-controller equipped with an ADC converter (with at least a range within 0-1 VOlts) and an SPI interface, which makes it accessible to Arduino and RaspBerry users.

3.1 Presenting the technology

The sensing circuit is based on a technique called **Swept Frequency Capacitive Sensing**, which makes use of an **LC resonant circuit** to **measure the capacitance of a given object**. In layman terms, the sensor generates a sinusoidal sweep over a predefined range of frequencies and it analyzes the response of the LC circuit to which the object is attached via an electrode.

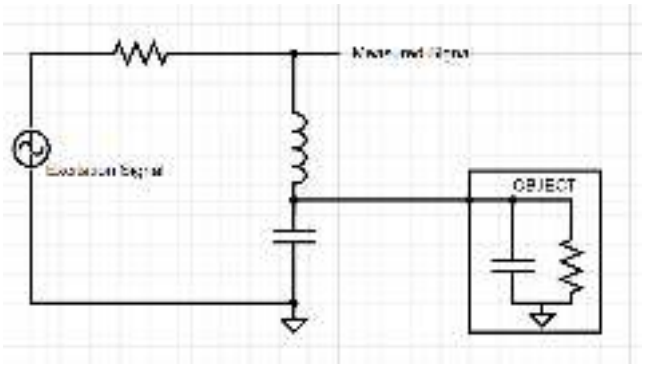


Figure 3.1 – The working principle of the circuit: an AC sweep is used to excite the LC circuit and the voltage after the resistor is measured. At resonant frequency the impedance of the LC series circuit will drop and consequently the measured voltage will decrease in amplitude. Connecting the object to the LC circuit as shown will impact the resonant frequency of the LC circuit and thus the response of the system.

Touching the object will change its electrical characteristics (predominantly its capacitance) and thus affect the response of the LC filter, the most common effect being a **shift** in the resonant peak or a change of the damping factor. The typical response of each object-interaction pair can be analyzed and classified with the help of simple **machine learning algorithms**, in this case we were able to use k-Nearest Neighbours (KNN) classification with success.

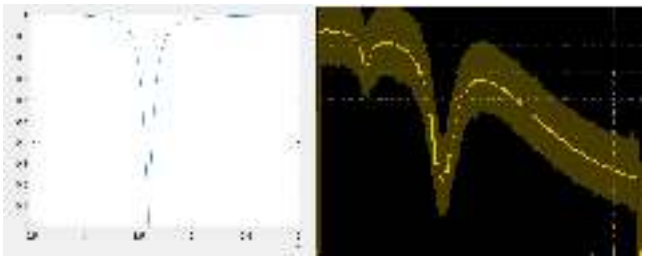


Figure 3.2 – **Left:** the ideal response of an LC filter, **Right:** the response of our physical implementation: the horizontal axis corresponds to the excitation frequency, while the vertical axis corresponds to the amplitude of the measured signal. Notice that the real circuit shows a shallower peak at the resonant frequency and a small secondary peak. Both these effects can be explained by the presence of parasitic elements that were ignored in the Matlab simulation on the left. Notice also that the signal on the right is discontinuous because it is sampled and that the decrease in amplitude at higher frequencies is due to the bandwidth limitation of our circuit (more on that later).

3.1.1 Block Diagram

Below is shown the block diagram of the sensing circuit: a **signal generator** is used to generate a **sweep**. Then the signal is **filtered** to attenuate 50 Hertz noise and to mitigate the disturbances introduced by the digital section of the signal generator. Note that a simple **amplifier** is used to increase the strength of the signal and to improve its SNR (Signal to Noise Ratio). The amplified signal is then fed to the resonant network which was introduced above. Its output is copied with a **buffer** because a direct connection could reduce the sensitivity of the resonant LC circuit. This is because a direct connection would have a low impedance and add parasitic capacitance. Lastly, a very simple **envelope detector**, made with passive components, is used to convert the amplitude of the AC signal into a DC value, which can be measured by the ADC of a micro-controller.

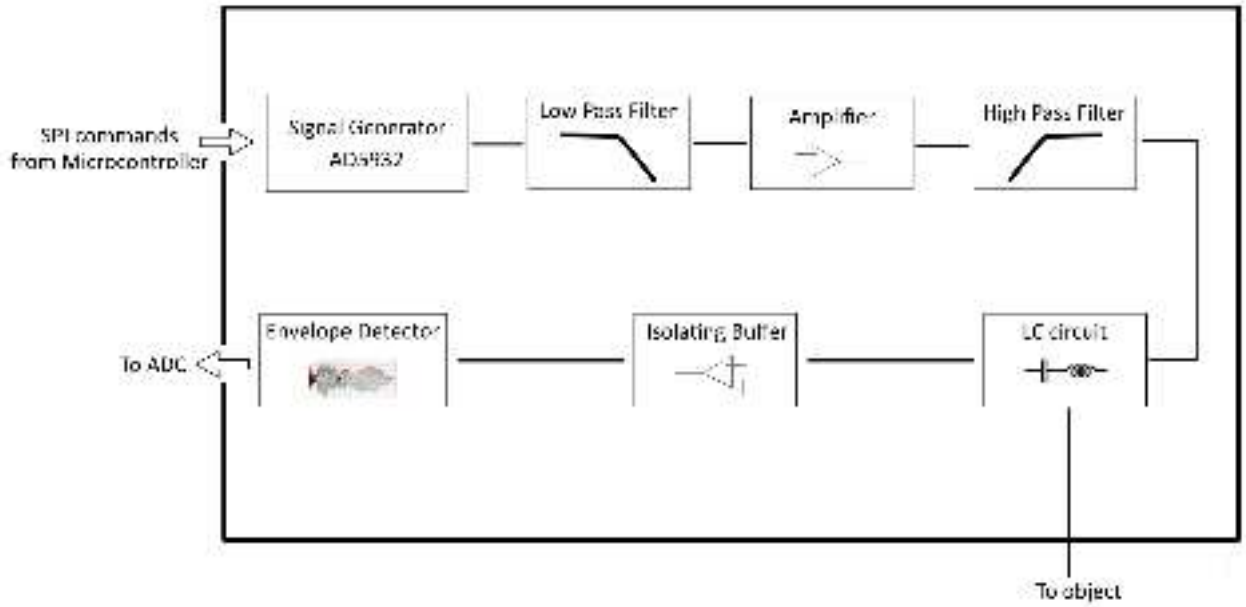


Figure 3.3 – The block diagram of the sensing circuit.

3.1.2 Schematics

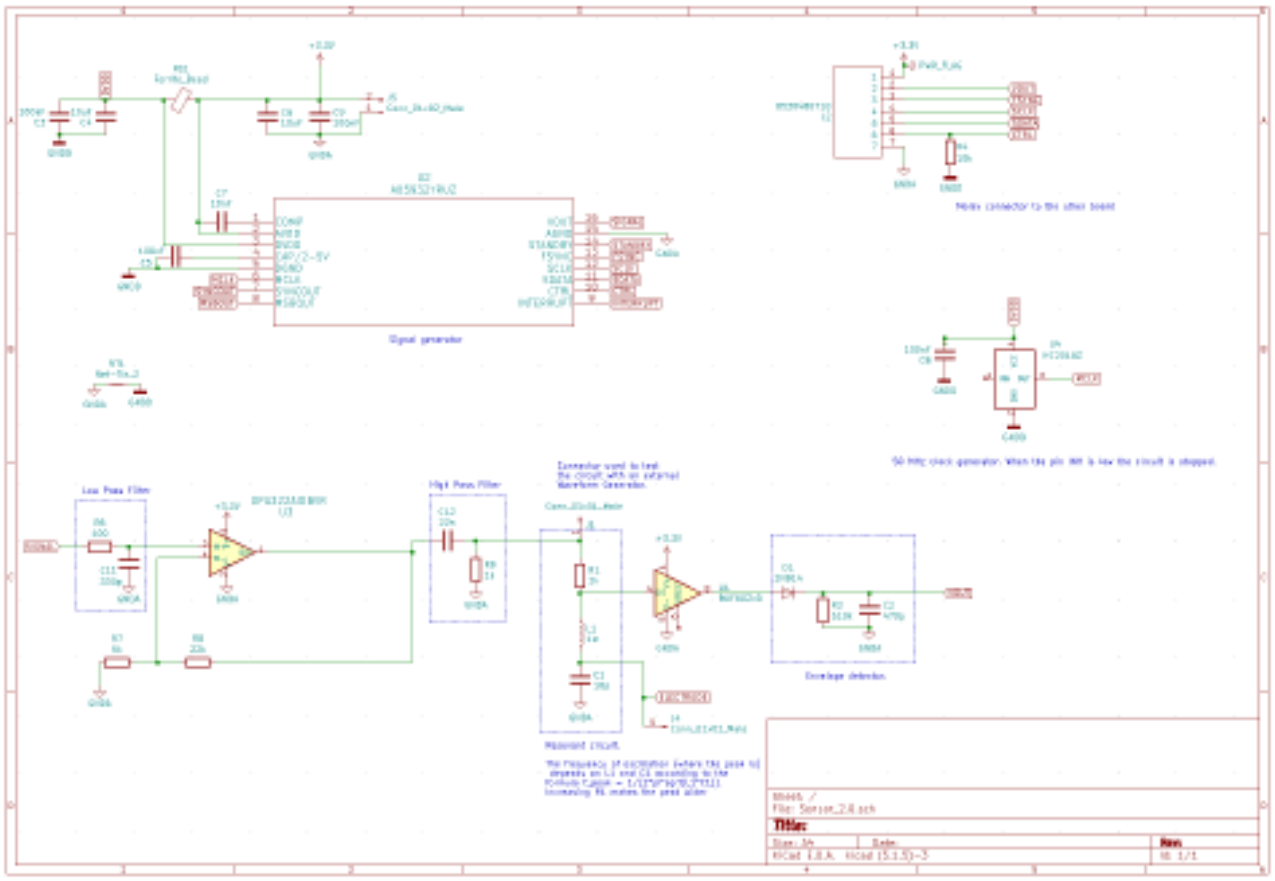


Figure 3.4 – The detailed schematics of the sensor.

3.2 Design Choices

In this section we will briefly explain the reasons behind the choice of each component and the calculations that were done to choose their value. Let's start by saying that we chose to operate within a frequency range **of 500kHz to 3Mhz**. This is because higher frequencies require expensive, high precision components and lower ones need large inductors which are not practical to assemble on a PCB.

3.2.1 Filters

In the first version of our PCB the signal generated by the Signal Generator was fed **directly** to the resonant circuit without any filtering. Even though the noise levels weren't so high to impair the functionality of our circuit, we chose to add the filters and the amplifier stage to **improve the reliability** and the quality of the signal. Since the noise was relatively low we chose to use simple 1st order RC filters, which provide enough attenuation and keep costs and the number of components down.

3.2.1.1 Low-Pass Filter

The value of the components was calculated knowing the frequency range of the sweep, which goes from $f_{min} = 500kHz$ to $f_{max} = 3MHz$. For the low-pass filter we chose a **cutoff frequency** of approximately 5MHz, which would theoretically give an attenuation of -23 Db at 50 MHz (which is the AD5932 clock frequency,

the main source of high frequency noise) and virtually none at f_{max}. The chosen values were $R = 100Ohms$ and $C = 330p$, which correspond to a cut-off frequency of 4.8 MHz.

3.2.1.2 High-Pass Filter

The purpose of the high-pass filter is to **reduce low frequency noise** and to **remove the DC component** from the signal. Note that after filtering the signal becomes (ideally) a sinusoid centered around zero and can thus reach negative values. At first, this may seem counter-intuitive since the power supply of the circuit ranges from 0 to 3.3 Volts, however the presence of reactive components (capacitors) and the fact that we are operating with AC voltages can cause the signals to swing outside of the supply range. The main source of low frequency noise is the **50 Hz electrical grid**, so for the high-pass filter we chose a cutoff frequency of approximately 5kHz, which would theoretically give an attenuation of -43 db at 50 Hz. The chosen values were $R = 1kOhms$ and $C = 33nF$, which correspond to a cut-off frequency of 4.8 kHz.

3.2.2 LC resonant circuit

The LC circuit is a **critical part** of the circuit and its tuning has to be done accurately in order to obtain the maximum sensitivity and operating range from the sensor. Generally speaking, a **smaller** tuning capacitance (C1) corresponds to an **increased sensitivity** and a better capability to detect variations in the object capacitance. However, choosing a very small value could cause the sensor to saturate, because the parasitic capacitance introduced by the cable or by the object itself could become too large relative to the tuning capacitance and thus bring the resonant frequency **outside of the range** covered by the sweep.

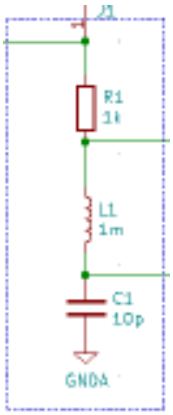


Figure 3.5

The touch of a finger can add about 1-2 pF to the capacitance of an object, so a good range for the tuning capacitance is **within the range 5-20 pF**. Note that the resonant frequency f_{res} is given by the following expression $f_{res} = \frac{1}{2\pi\sqrt{LC}}$, and a **large inductor** is required to compensate for the fact that the capacitance should be small to obtain a high sensitivity. In our case we chose the target resonant frequency to be slightly above the middle frequency of the sweep ($f_{target} = 1.9MHz$) which corresponds to the following values: $L = 1mH, C = 7pF$. An additional factor to consider is the value of the series resistance R1: larger values will cause a greater voltage drop and "dampen" the resonant circuit, which will present a **larger and shallower** resonant peak. Note that for the purpose of classification both very shallow and very sharp peak would be counterproductive, because in the first case the information content of the signal would be condensed in a narrow frequency range and could be lost if the micro-controller sampled the signal with a large interval (for example if it sampled the response of the sensor with steps of 100 kHz). In the second case the resonance could be too damped and the information content would get "smoothened out". For this reasons the resistance value should be chosen carefully to meet these **contrasting requirements**. It should also be noted that a large resistance will **increase the input impedance** of the resonant stage, thus **reducing power consumption**

and distortion of the signal coming from the amplifier stage. In our particular case the size of R1 was chosen to be 1kOhm, which yields satisfying results in both simulations and experimental measurements.

3.2.3 Amplifier

The amplification stage was realized with an Operational Amplifier in **non-inverting configuration**. The value of the gain was chosen taking into account the typical amplitude of the signal generated by the AD5932 signal generator ($V_{AD5932,peak-peak} = 0.58V, V_{AD5932,offset} = 56mV$), the input range of the ADC of the ESP8266 micro-controller ($ADC_{range} = 0 - 1V$) and the voltage losses caused by the rectifying diode in the envelope detector.

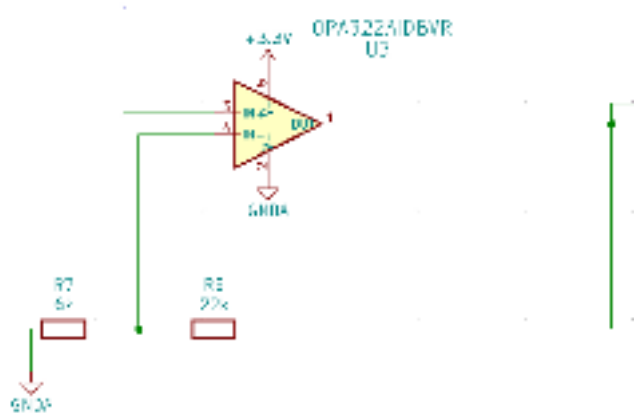


Figure 3.6

In practice we computed the **maximum DC voltage** found at the output of the envelope detector ($V_{out,max}$) and we chose the value of the gain to make it match the maximum value allowed by the ADC. in symbols:

$$V_{out,max} = gain_{max} \cdot (V_{AD5932,offset} + 0.5 \cdot V_{AD5932,peak-peak}) - V_{diode,min} = V_{ADC,max} \tag{3.1}$$

$$V_{out,max} = gain_{max} \cdot (0.056 + 0.29) - 0.6 = 1V \tag{3.2}$$

$$gain_{max} = 4.6 \tag{3.3}$$

Since the circuit contains also 2 integrated circuits (operational amplifier, buffer) which were hard to model by hand, we also **simulated** the circuit with the SPICE models provided by the component producers to check our prediction. The results were **consistent** with the predictions and only a small amount of fine-tuning was performed after experimentation with the real PCB. In the end, the chosen gain was 4.66, which corresponds to $R7 = 6k$ and $R8 = 22k$.

Given the **bandwidth requirements** of the sensor, the operational Amplifier had to be chosen carefully to ensure a high gain even at high frequency. This was a particularly difficult choice since high frequency OpAmps can become very **expensive** and for this reason we had to find the right balance of performance vs price. The chosen OpAmp is the **OPA322**, with stable unity-gain and 20MHz of gain-bandwidth product, which is not very far from the maximum frequency of the circuit, $f_{max} = 3MHz$. This means that the *open-loop* gain of the OpAmp will be far from the ideal value of infinity and the *closed-loop* transfer function of the amplifier (OpAmp plus resistor feedback network) should be analyzed more carefully to obtain accurate results.

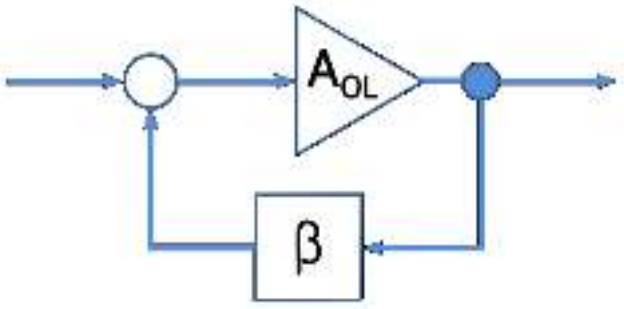


Figure 3.7 – A block representation of the amplifier feedback loop.

The closed loop gain (A_{CL}) of the amplifier is given by the following expression, where A_{OL} is the *open-loop* gain of the OpAmp and β is the gain of the resistor network ($\beta = R7/(R7 + R8)$) :

$$A_{CL} = \frac{A_{OL}}{1 + A_{OL}\beta} \tag{3.4}$$

A_{OL} is linked to the *gain-bandwidth* product by the following expression:

$$A_{OL}(f) = \frac{20MHz}{f} \tag{3.5}$$

Now it is possible to find the *open-loop* gain of the amplifier at $f_{max} = 3MHz$:

$$A_{OL}(3MHz) = 6.6 \tag{3.6}$$

And knowing A_{OL} we can find the *closed-loop* gain of the amplifier:

$$A_{CL}(3MHz) = 2.74 \tag{3.7}$$

Note how the **frequency limitation** of the OpAmp has reduced the gain of the amplifier to 2.74 instead of the desired 4.6 that we computed before. If we repeat the same calculation for $f_{min} = 500kHz$ we obtain a *closed-loop* gain $A_{CL} = 4.18$ which is instead very close to the predicted value. This **degradation** of the gain of the amplifier with the increase of the frequency was observed in both practice and simulation (see section 3.6) and could have been solved by choosing an OpAmp with **larger** *gain-bandwidth* product. However, we opted for this OpAmp because it is relatively **cheap** and this side effect **does not impair the functionality** of the circuit. This is true because the only purpose of amplification is to improve the Signal to Noise Ratio (SNR) of the signal and a slight reduction in gain does not have drastic effects. Furthermore, the gain degradation is a **predictable** source of error (the signal is always modified in the same way), and for this reason the machine-learning classifier retains its ability to classify the various interactions.

3.2.4 Isolating Buffer

The purpose of the isolating buffer is to **decouple** the LC resonant circuit form the envelope detector and the ADC. The buffer is necessary because a direct connection would affect the functionality of the resonant circuit: parasitic capacitance would be introduced and **considerable amounts of current** would flow into the envelope detector and later into the ADC. For example, the 1N914 diode (D1) would add 4pF parallel capacitance to the tuning capacitor (C1) and the current flowing into capacitor C2 could cause a voltage drop across R1.

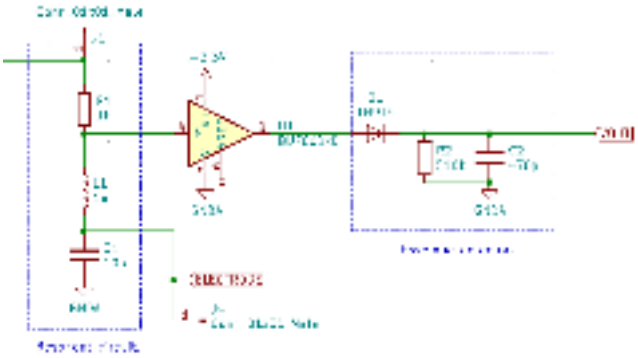


Figure 3.8

Decoupling the two stages with the BUF602 buffer helps **reduce the impact** on the performance of the resonant circuit, in fact the BUF602 shows a **1 MOhm input impedance** (typical value with 3.3V supply) and adds only **2pF of parasitic capacitance**. It should be noted that the buffer was chosen to respect the bandwidth requirements of the circuit (which operates in the 0.5 - 3MHz range), a requirement that is largely met since the BUF602 has a **bandwidth of 600MHz** (with 3.3V supply). Note also that the buffer is unable to follow the input voltage when it gets too close to the power rails voltage (1 V from the power rail) which means that the output voltage **can't drop below 1V** in our case. This doesn't constitute a problem because we are only interested in the **peak voltage** and the distortion introduced by the buffer doesn't have any impact on the functionality of the circuit. The only detrimental effect happens when the circuit is at its **resonant frequency**, or in other words where the **output frequency drops due to the resonant peak**. Since the buffer output is limited to 1V, the output voltage of the sensor can't drop below 0.4 Volts (recall that the voltage drop across D1 is about 0.6 V). A possible way to overcome this problem would be to choose a buffer with a **rail-to-rail output**, but this turned out to be an **expensive** choice (due to the bandwidth requirements) and the increased cost out-weighted the benefits.

3.2.5 Envelope Detector

The envelope detector is used to convert the amplitude of the AC signal coming from the resonant circuit into a **DC voltage** which can be sampled by the ADC of a micro-controller. There are many different ways to realize an envelope detector, which make use of both active and passive components. During the design phase of our circuit we considered a topology based on a *precision-half-wave-rectifier* which would eliminate the voltage drop across the diode D1.

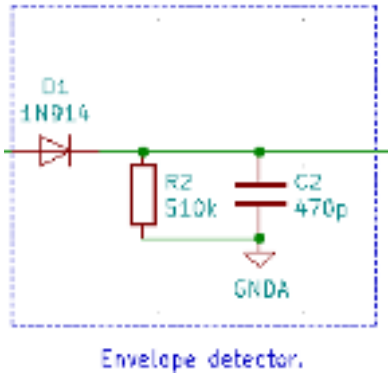


Figure 3.9

However we concluded that the advantages were not worth the added **complexity** and, most importantly, the diode voltage drop was **actually beneficial** due to the limitations of the buffer-follower explained above.

The chosen circuit was then a simple **passive RC envelope detector**, which meets perfectly the requirements of our sensor. The components were chosen taking into account the **frequency requirements** of the circuit, which in the case of the diode corresponds to its **switching speed**, which for the chosen diode (1N914) is 4ns (250MHz). The resistor and the capacitor values were chosen to yield a **maximum ripple of about 2%** and calculated according to the following formula:

$$ripple_{\%} = \frac{1}{fRC} \tag{3.8}$$

$$0.02 = \frac{1}{250kHz \cdot RC} \tag{3.9}$$

The chosen values were: $R2 = 510k$ and $C2 = 470p$ which correspond to a ripple of 1.6%. Note that we chose to leave some margin because the input resistance of the ADC is unknown (we weren't able to find any documentation about the ADC of the ESP8266) and it will most likely increase the ripple seen at the output of the peak detector.

3.2.6 Signal Generator

The chosen signal generator is the **AD5932 chip** by Analog Devices. We chose to use this component because it was already used in the literature and because it **fits perfectly the requirements** of our circuit. We preferred it to other function generators such as the AD9833 or the MAX038DS because they offer more functionalities than needed and add unnecessary complexity. The AD5932 is able to generate sinusoidal signals of $V_{peak-peak} = 0.58V$ with a frequency up to 25 MHz (when clocked at 50 MHz).The Typical power consumption is 6.2 mA (digital plus analog section) while operating and 20 μA in power saving mode. The AD5932 can be programmed via an **SPI interface** and it can be set to generate a sweep **automatically** or to increase the frequency every time a signal is **toggled by the micro-controller** (more details on the programming of the AD5932 can be found in the section about the software). In our sensor we control the AD5932 through the pins ***FSYNC, SDATA, SCLCK, CTRL*** and we ignore the other control pins, which don't provide useful functionalities to us. The first three correspond to the **SPI interface**, while the last is the control pin used to increase the frequency.

3.3 Power Consumption

Let's start by analyzing the power consumption of each component:

DEVICE	MAX CURRENT
Signal Generator (AD5932)	6.7 mA
Amplifier OPA322	0.35 mA
Buffer (BUF602)	5 mA
Crystal Oscillator (KC2016Z)	6 mA

We should then take into account the **losses** due to the various **paths to ground** present in the circuit. To do this we estimate the **average voltage** across $R7 + R8$ and $R1 + R9$, which correspond to the voltage at the output of the amplifier and at the input of the buffer respectively. Finally, we should also consider the current flowing through $R2$. Note that we estimate the average voltages using the **experimental results** that are shown in chapter 3.2. For the voltage across $R2$, an accurate estimate should take into account the shape of the output signal, that can be seen in figure **??**. For the sake of simplicity, we will assume an average voltage of 0.6 V, however it is **largely irrelevant** because $R2$ is very large.

RESISTANCE TO GROUND	AVERAGE VOLTAGE	AVERAGE CURRENT
R7+R8 = 28 kOhm	1.5 V	0.05 mA
R1+R9 = 2 kOhm	0.7 V	0.35 mA
R2 = 510 kOhm	0.6 V	negligible

Adding up all the currents yields the following result:

$$I_{tot,max} = 18.45mA \tag{3.10}$$

If the sensor was powered by a battery of 400 mAh it would last about **20 hours**, note however that this takes into account only the sensing circuit and not the transmitting part, which was characterized during last semester.

3.4 Simulation Results

During the design phase we run a **large number of simulations** with a software called **LTSpice**, which uses on Spice models to simulate analog circuits. This was **extremely helpful** as it allowed us to **spot mistakes** early on and to get a better understanding of the circuit before we actually built it. Another big advantage is that it is possible to **simulate integrated circuits** such as OpAmps or semiconductor components like diodes with the models provided by the constructor, this was very helpful when we had to pick the right component for the circuit. When the correct Spice model of the component is availble, **the accuracy of the predictions is remarkable**, as we noticed when we compared the real measures with the simulations. Unfortunately, the number of models already included in LTSpice is quite **limited**, so we spent a lot of time searching for the models online and creating the corresponding components in the software, a procedure which was quite tedious and that we won't explain here for brevity. Furthermore, we were unable to find the model of some components, like the buffer BUF602 or some models of OpAmps that **we couldn't simulate** and thus decided not to take the risk to use. In the following picture is shown a typical output of the simulations:

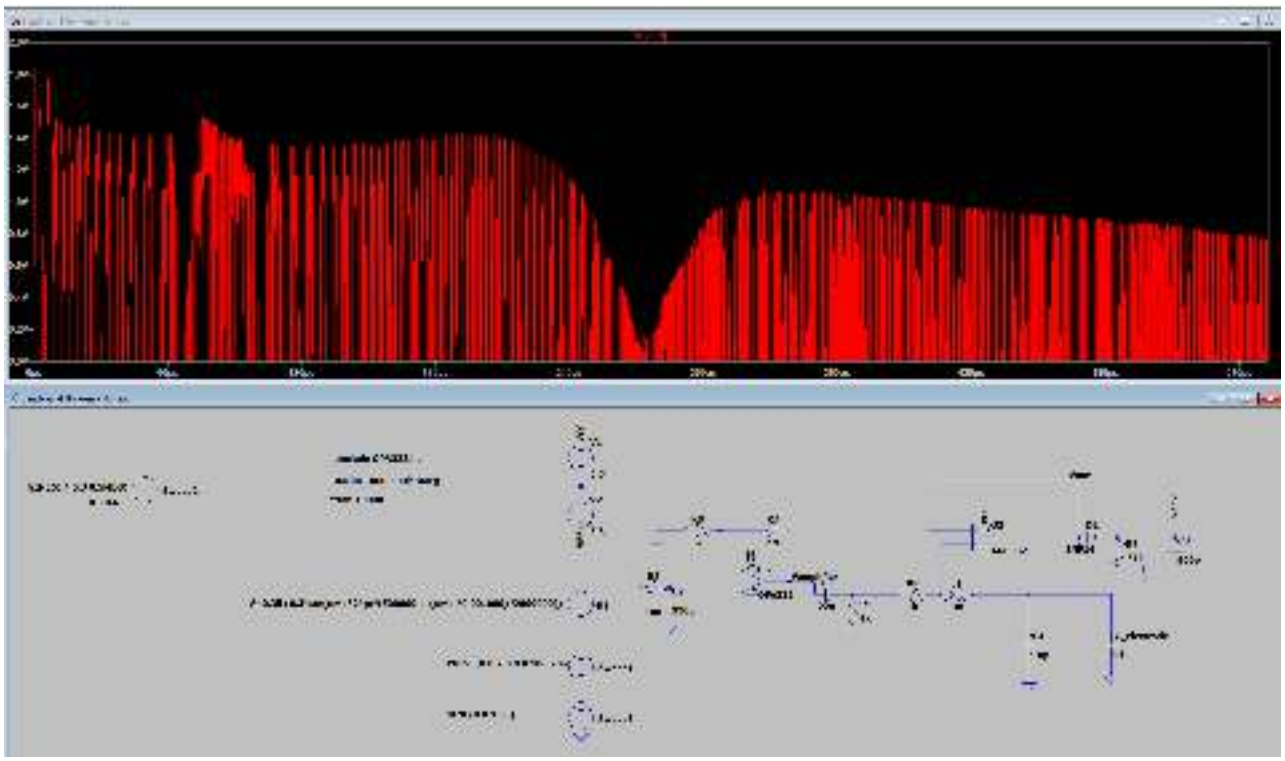


Figure 3.10 – The output of a simulation and the circuit schematics.

In the **bottom** it is possible to see the **schematics** of the circuit under simulation, while on the **top** is shown the **voltage** at the **output of the buffer**, before the peak detection. Note that the **frequency increases from left to right** and ranges from 500 kHz to 2.5 MHz. We didn't show the output of the peak detector because the simulation runs **very slowly** and it was very unpractical to simulate a running time of more than 1 ms. We overcame this problem "compressing" the frequency sweep in about 500 μs, while in reality it happens in about 100ms. For this reason the peak detector response was too slow in the simulation because it was designed to work on another time scale. Note however that it is possible to obtain the envelope of the signal by mentally subtracting 0.6 volts (diode voltage drop) from the red signal. However, the simulation results **resemble closely** what we observed in the real circuit except from a **few details**. For example, since we couldn't simulate precisely the buffer BUF602, the peak is **deeper** than what we observed in reality, for the reasons explained in section 3.2.4. The simulation allowed us to **simulate the effects of noise**, to do this we added **two voltage sources** in series with the source simulating the AD5932 output. One of them generates a 50 Hz sinusoidal noise, while the other generated a 50MHz square wave to emulate digital noise.

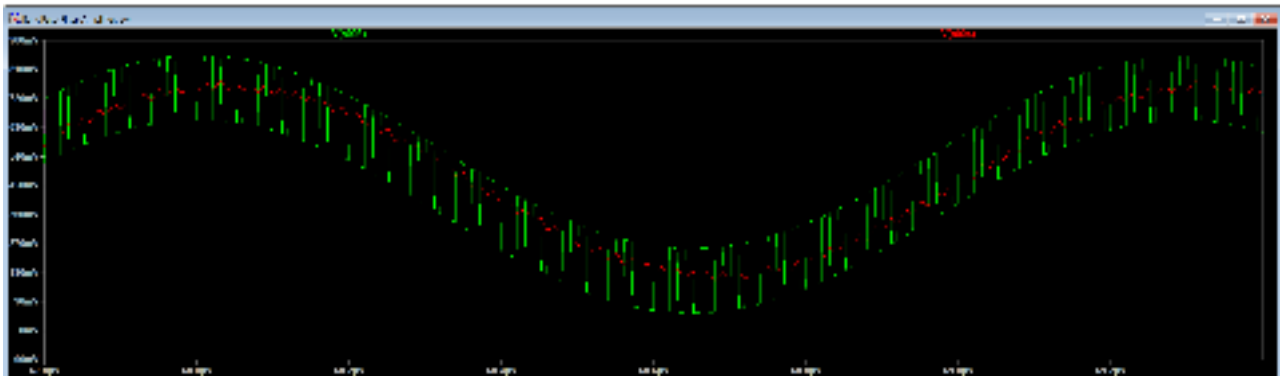


Figure 3.11 – The simulated result of filtering on digital noise. **Green**: noisy signal from AD5932, **Green**: filtered signal

3.5 Design of a new PCB

Designing the PCB was **quite challenging** because it was the first time we had to design a **mixed-signal PCB**, with an analog and digital section. The presence of digital circuits is due to the AD5932 (Waveform Generator), which has to be programmed digitally through an **SPI interface** and is clocked by a **50MHz clock**. The problem with mixed signals PCBs is that **fast-edged signals** on the digital side generate **high frequency noise** that may propagate to the analog section if the right precautions are not taken. The designer should be careful and try to **minimize the coupling** between the analog and digital section, by physically **separating the components** into two distinct areas of the PCB. The most important source of noise to consider are **ground currents**: in fact, if the current return path of a digital component (let's say a clock oscillator) runs near an analog component it may generate large disturbances. This is particularly evident if analog and digital circuitry share the **same ground plane**: the return currents could result in large amount of noise. A possible approach would be to physically isolate the analog ground from the digital ground and to join them at a single point. However, this solution may cause **more harm than good**, because the return currents will be forced to pass through the same **choke-point** and the problems of coupling wouldn't be solved. A much better approach is to keep a single ground plane and to **position the components such that the return paths of the current don't cross**. For example, in our PCB, all digital signals are connected to the lower part of the ground plane and the current coming from the +3.3V terminal can flow back to the GNDA terminal **without mixing** with the currents coming from the analog section on the top (see figure below). Notice that the ground plane is **split on the right** to force the analog currents to flow on the top and the digital currents to flow on the bottom. Another source of coupling could be the fact that both sections share the same connection to the positive supply. This was solved by splitting the power supply in two (analog and digital) and inserting a **ferrite bead** between the two. It should be noted that taking these precautions has **significantly reduced the amount of digital noise** present in the analog section, compared to what was observed with the first PCB.

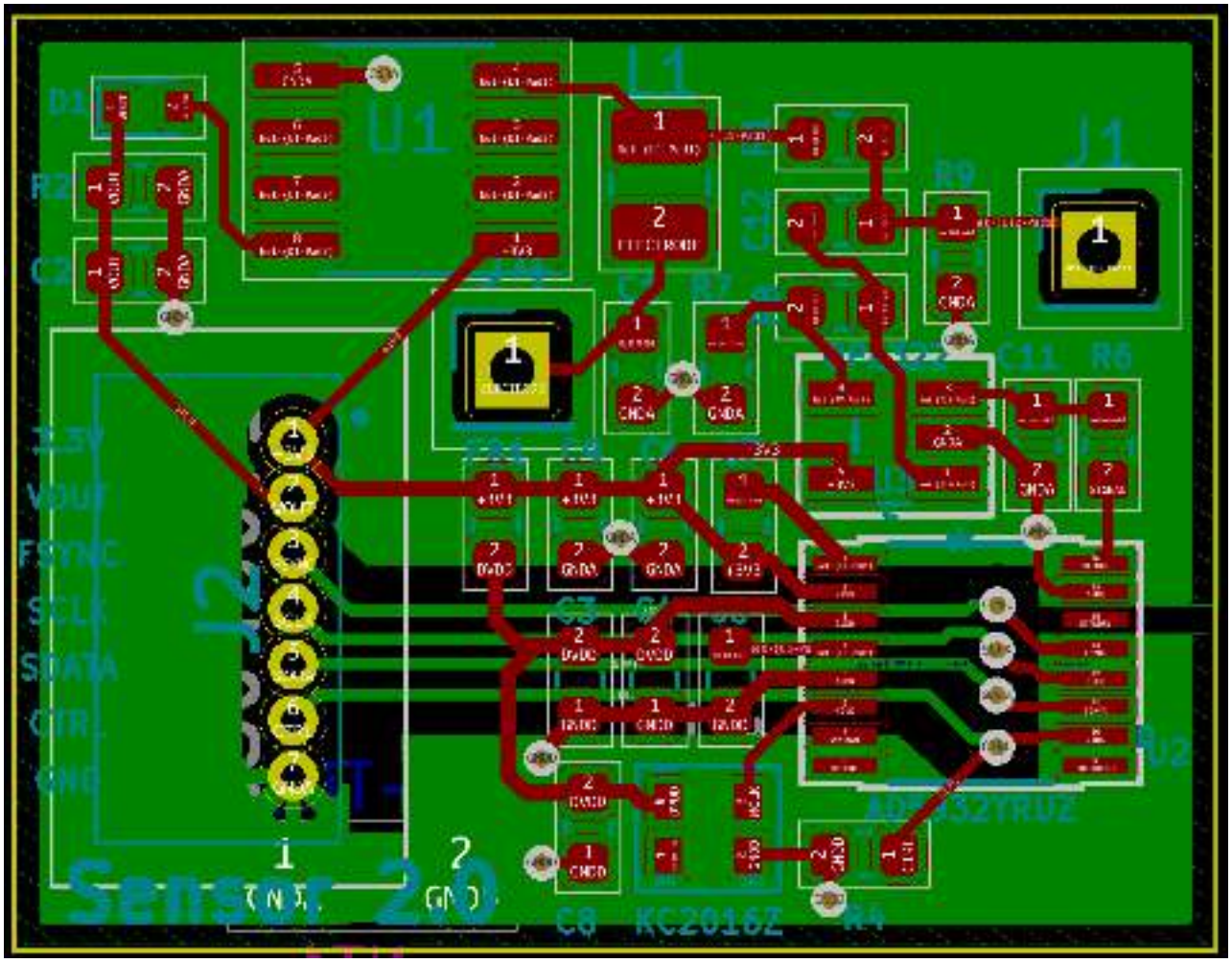


Figure 3.12 – The final design of the sensor PCB. Most passive components belong to the 0603 series, which allowed a good miniaturization. Note that we had to draw by hand the footprint of the Clock Oscillator (KC2016Z) because it wasn't available online.

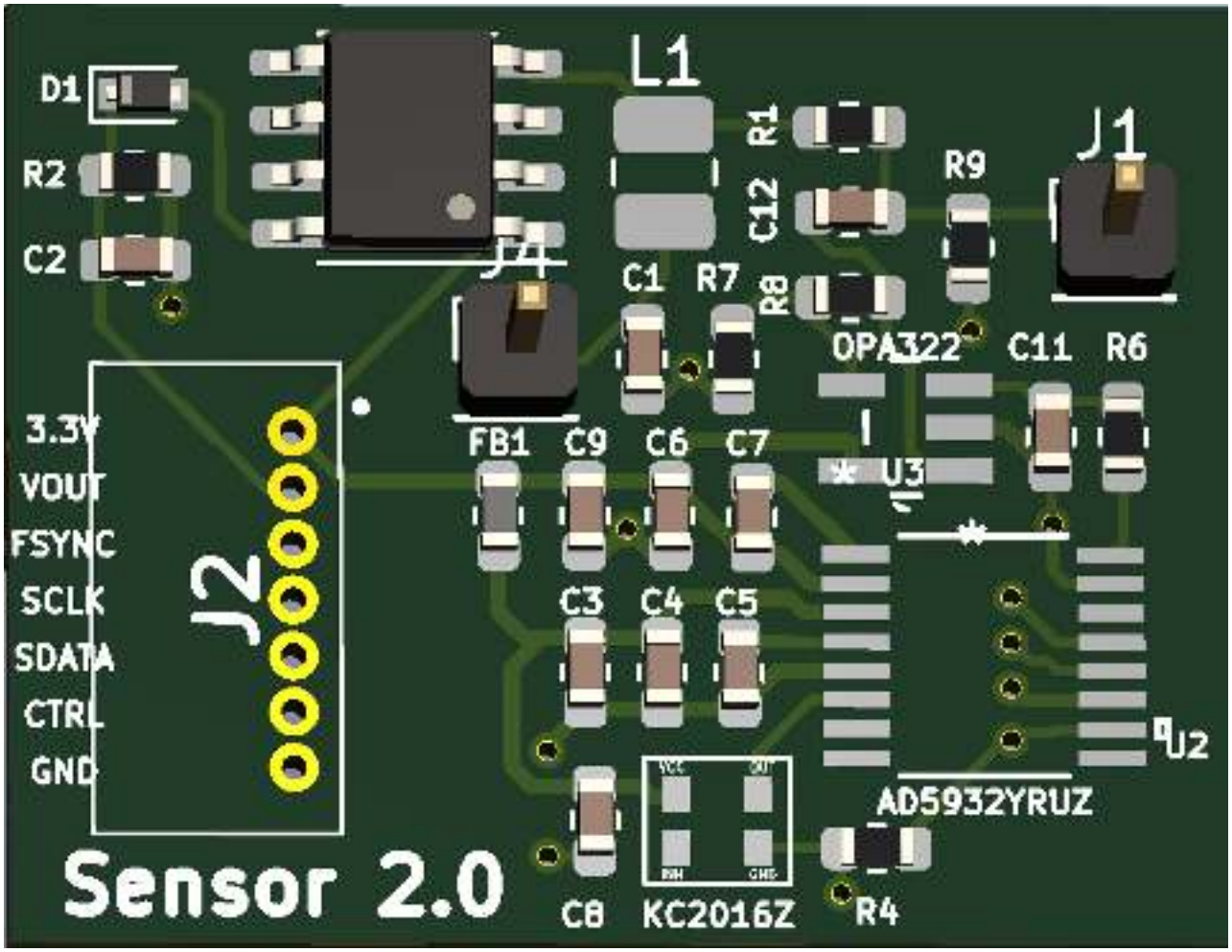


Figure 3.13 – A 3D view of the PCB.

When we designed the last version of our PCBs we considered the option of merging everything into a **single board**, in other words to put the ESP8266 together with the sensor circuit to avoid having to deal with multiple boards. In the end, we chose to keep the **two modules** separate because we liked the idea of a **modular product** which could be reused in different projects. For this reason we added to both PCBs a Molex connector that allows easy connections of the two boards with each other or with an external micro-controller.

3.6 Experimental Results

During the development of our project we performed **a lot of tests** and measures to make sure our predictions were correct and to check if the circuit was working as expected. For most measures we used the **portable oscilloscope** Analog Discovery 2 produced by Digilent, which is a quite incredible tool without which our work would have been much harder.

In the following sections we will show the measures that were taken ad different stages of the circuit, namely:
— At the **output** of the **signal generator** (V_{AD5932}, point A)
— At the **output** of the **amplifier** (V_{amp}, point B)
— At the **output** of the **filter** (V_{filter}, point C)
— At the **input** of the **buffer** ($V_{buff,in}$, point D)
— At the **output** of the **buffer** ($V_{buff,out}$, point E)

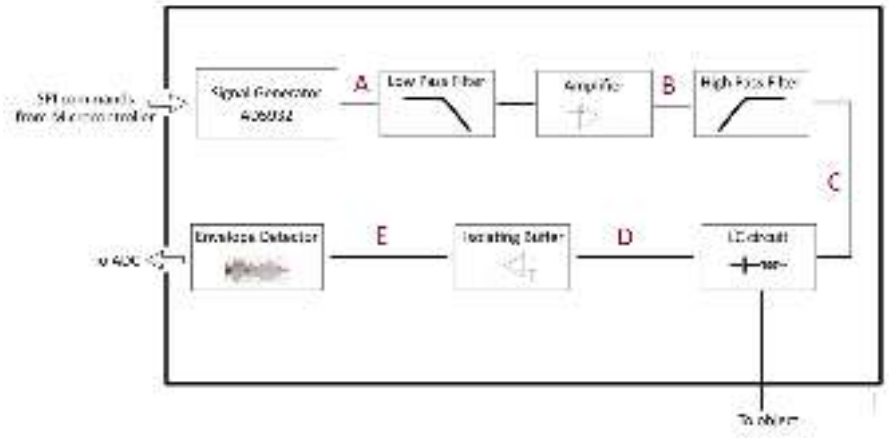


Figure 3.14

3.6.1 Voltage at the output of the signal generator: V_{AD5932}

The first step was to observe the waveform generated by the AD5932 to check whether it **corresponds to the data-sheet specifications** and to see if we were able to program the AD5932 correctly. This step was very important because it **oriented all of our decision** concerning the successive filter and amplifier stages.

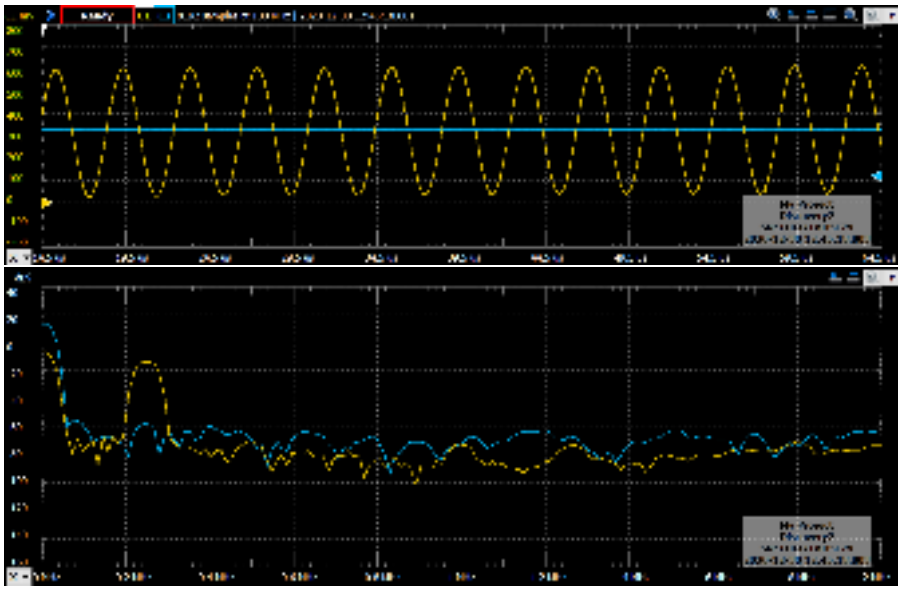


Figure 3.15 – **Top**: the waveform generated at the beginning of the sweep. **Bottom**: the FFT of the signal. The frequency is 250kHz as expected and the sinusoid is clean and free from distortion. This can also be seen in the FFT which shows a single peak at 250kHz and at 0 Hz because of the offset. The amplitude is about 580 mV as indicated in the datasheet, while the offset is about 25 mV (measured between the minimum and 0) which is less than expected.

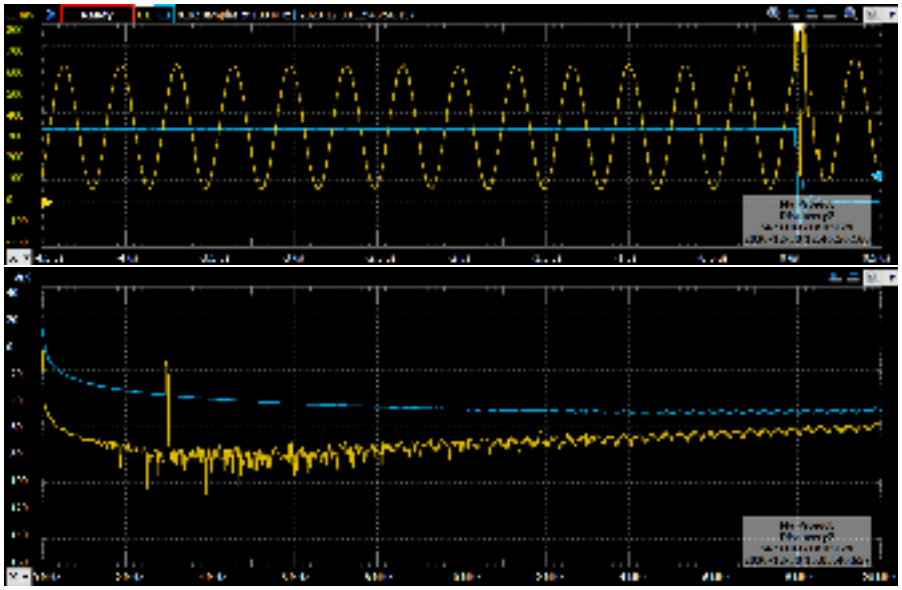


Figure 3.16 – **Top**: the waveform generated at the end of the sweep. **Bottom**: the FFT of the signal. The frequency is 3MHz as expected and the signal doesn't show signs of distortion, this is confirmed the FFT which shows a single peak at 3MHz. The amplitude is about 580 mV while the offset has increased to about 60 mV.

3.6.2 Voltage at the output of the amplifier: V_{amp}

The amplifier is a very important stage because it could impair the functionality of the circuit, if it is not properly designed. Too much gain could bring the signal out of the range of the ADC and the limitations of the OpAmp could introduce distortion and destroy the gain, as was explained in section 3.2.3.

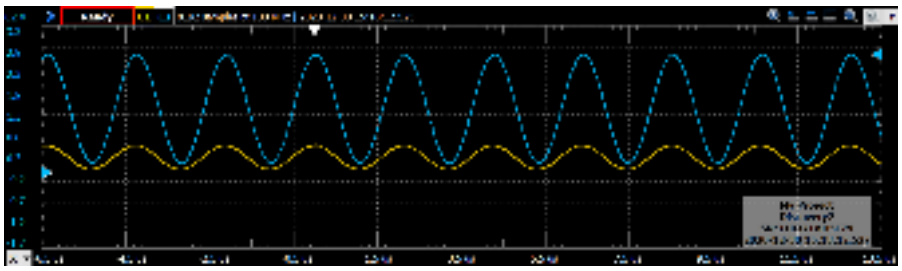


Figure 3.17 – **Top**: the signal before (yellow) and after (blue) amplification at 250 kHz. The scope capture shows the signal before and after amplification at the beginning of the sweep, when the frequency is relatively low. Notice that the two signals are in phase and the gain is about 4.3 (we predicted 4.2 in section 3.2.3)

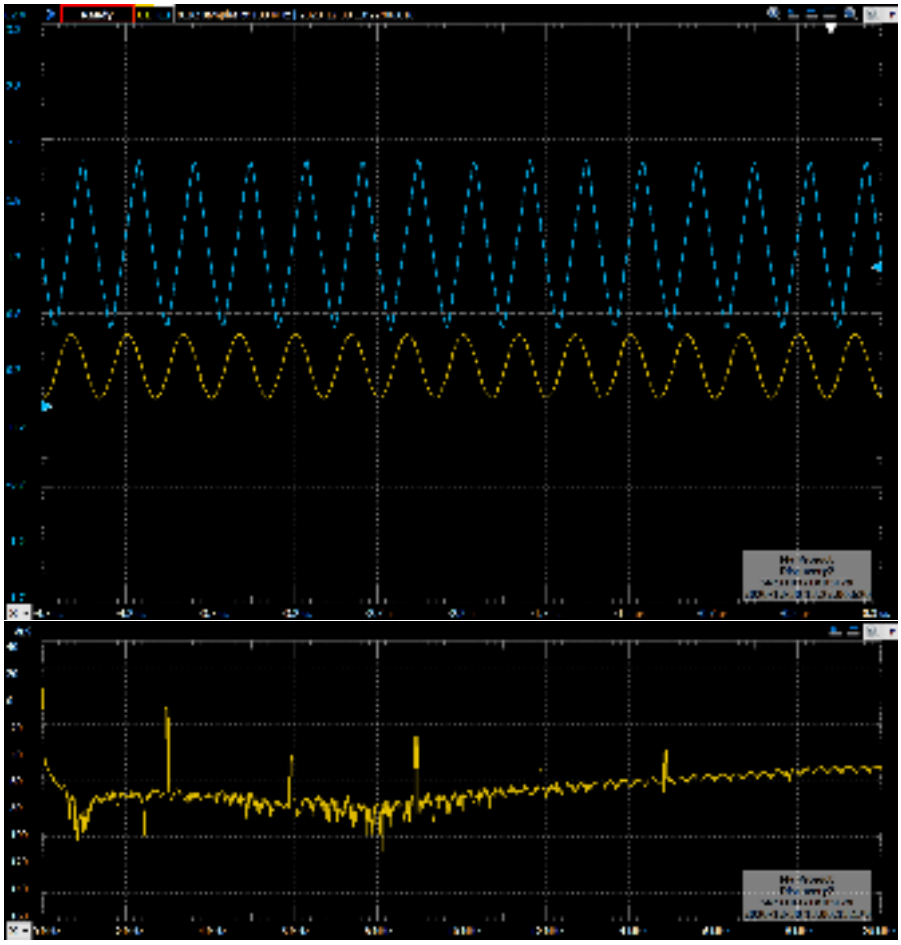


Figure 3.18 – **Top**: the signal before (yellow) and after (blue) amplification at 3MHz. **Bottom**: the FFT of the **blue** signal. At high frequency the amplifier stage shows signs of distortion and the gain is reduced to about 2.76 (in section 3.2.3 we predicted 2.74). The distortion is due to the slew-rate limitation of the OPA322, which is $10\text{V}/\mu s$. Unfortunately we noticed too late that we overlooked this requirement, in fact the required slew-rate was $20\text{V}/\mu s$ minimum (minimum slew rate $= 3MHz \cdot 2\pi$). The distortion is evident in the FFT of the signal, which shows peaks at the harmonics. Luckily, the subsequent stage (the LC filter) is selective enough to filter out the high frequency harmonics and the circuit was able to function properly.

3.6.3 Voltage after filtering: V_{filter}

The next stage is the **high-pass filter**, whose purpose is to reduce the 50 Hz noise and more importantly to filter the DC component of the signal.

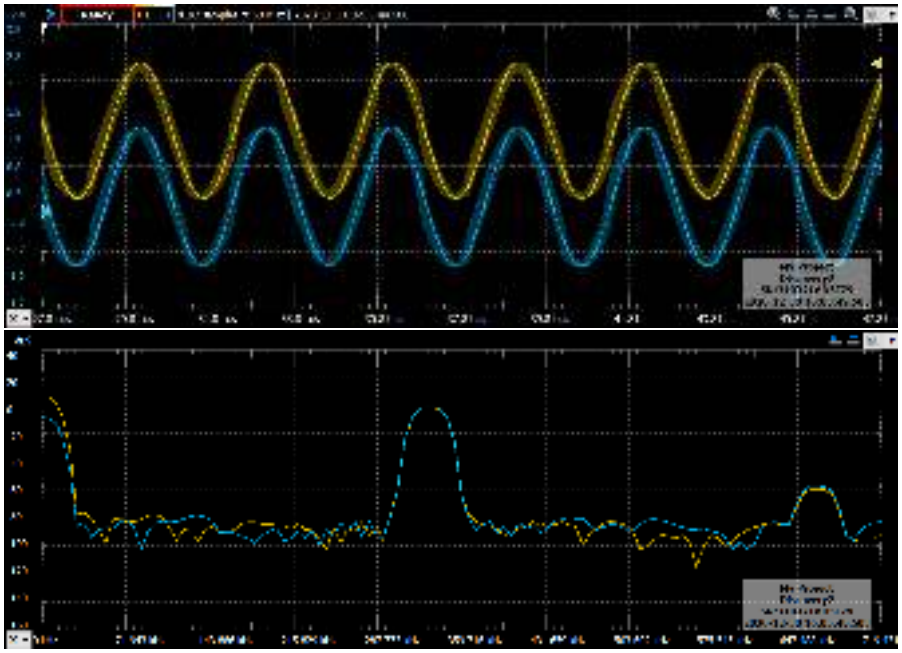


Figure 3.19 – **Top**: the signal before (yellow) and after (blue) filtering at about 300 kHz. **Bottom**: the FFT of the signals. Notice that the DC component of the signal was filtered and the waveform now oscillates around 25 mV (approximately). This can also be seen in the FFT, where the peak at 0 Hz was attenuated of about 20 Db, while the high frequency component haven't changed.

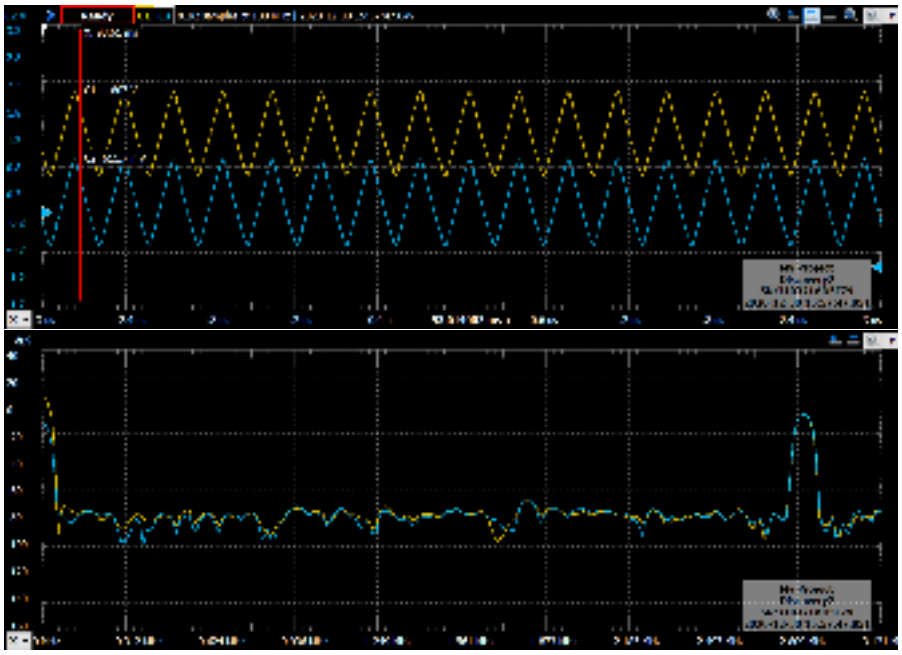


Figure 3.20 – **Top**: the signal before (yellow) and after (blue) filtering at 3MHz. **Bottom**: the FFT of the signals. The same observations made before apply here.

3.6.4 Voltage after and before the buffer: $V_{buf,in}$ and $V_{buf,out}$

Finally we analyzed the buffer stage, measuring the voltage before and after the buffer.

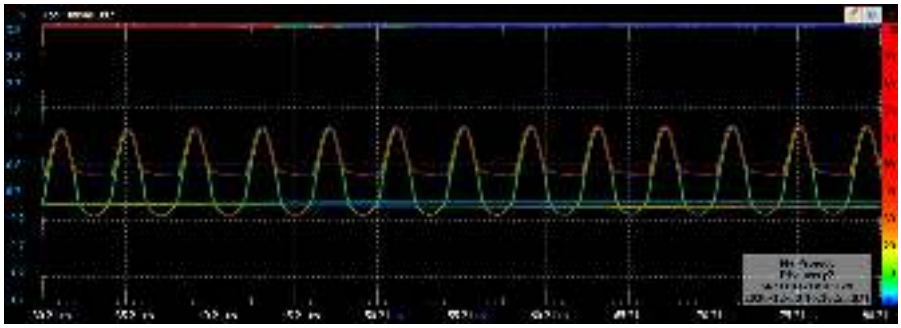


Figure 3.21 – **Green**: the signal before the buffer **Red**: the signal afterwards. Note that the buffer output follows precisely the input only when the input signal is not too close to the power rail of the buffer, as was explained in section 3.2.4. However, we are only interested in the maximum value of the signal because the next stage is the peak detector, and this limitation of the buffer doesn't cause any harm to the circuit functionality.

3.6.5 Output of the sensor

Finally we can show some examples of the output of the sensor when it is connected to an object and when the user interacts with it. For this test we used a metallic object shaped like a hand and a pair of earphones, we experimented with different types of interactions and we plotted the results below:

Figure 3.22 – One of the objects that we used for this particular experiment.

Figure 3.23 – The earphones that we used for our experiments.

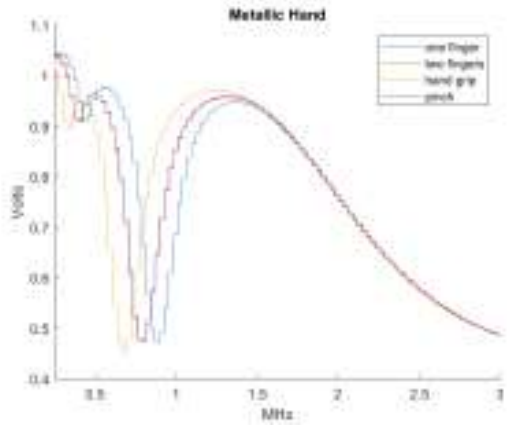


Figure 3.24 – The response of the sensor to a possible set of interactions with the metallic hand. Note that it is possible to discrimate between gestures easily.

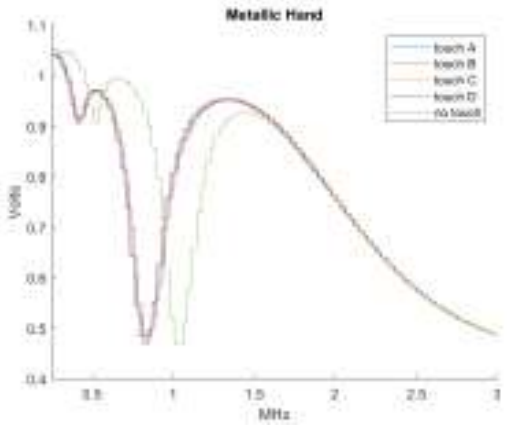


Figure 3.25 – Another set of interactions. Note that the discrimination capability is quite low in this case, because the response of the sensor is very similar.

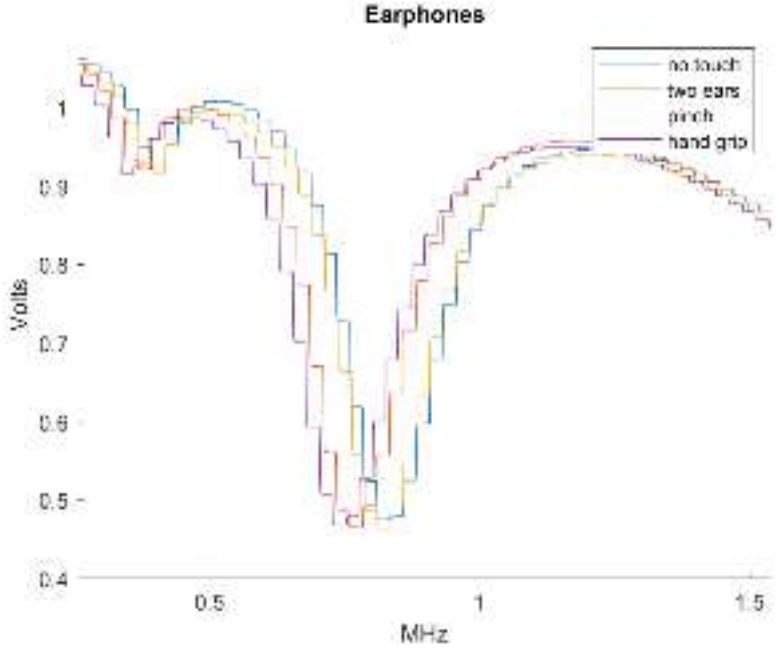


Figure 3.26 – We also tested with some earphones and we obtained the results above. Note that to the naked eye it may seem that there is little difference in the responses, however the resonant peak shifts of about 200kHz, which is more than enough to be detected and classified (at least in our controlled tests).

4. Software Work

Last semester, we programmed **graphic art generation** using **Processing**. As this semester the focus was given on design of a better sensor and since we are not working anymore with a graphic designer, we decided not to improve the graphic generation anymore (*of course, the previous code done last semester is working*). This section will presents the software work which was conducted on top of what we had last semester. All the code is accessible on the github repository of our team.

Designing the electrical sensing circuit was one part of the project, another part consisted into **creating the software environment** to have a **user-friendly classifier**. This involves the following steps:

— Handle the communication with the sensors from both side of the server (the server, that is the host computer, and the client, that is the ESP8266).
— Managing a database of the training dataset used by the classifier.
— Training a machine learning classifier which can predict the type of interaction received from the sensor, after that a training dataset was generated.
— Creating a user-interface to visualize this database, modify it, and test new points with the classifier.

Here is a diagram representing the main elements and their functionalities for our software.

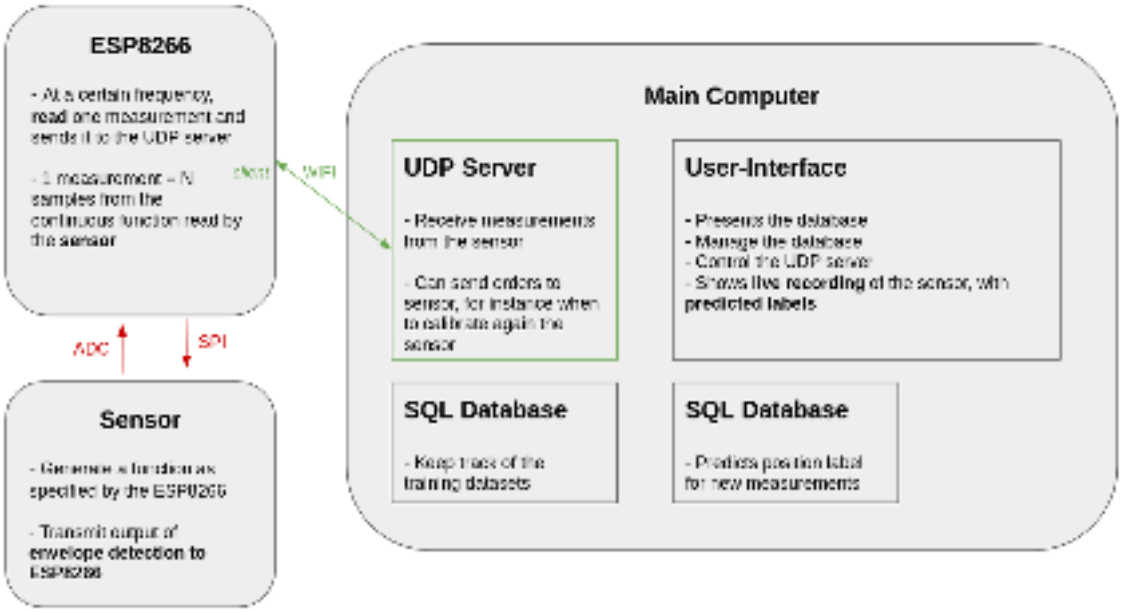


Figure 4.1 – Software Architecture

4.1 SQLite Database

Our database is implemented with the **relational database** framework **SQLite**. We decided to use SQLite over other choices (as MySQL, for instance) for its **simplicity** of use and for an easy integration with our user interface. The database is accessible using **SQL queries**, since this framework follows most of the **SQL standards** and there is a binding to the **Python** language, which was even integrated into the core code of Python. The Python libraries is accessible at this link.

4.1.1 Description of the database

As SQLite is a relational database, we can represent the **tables** with a **class diagram**.

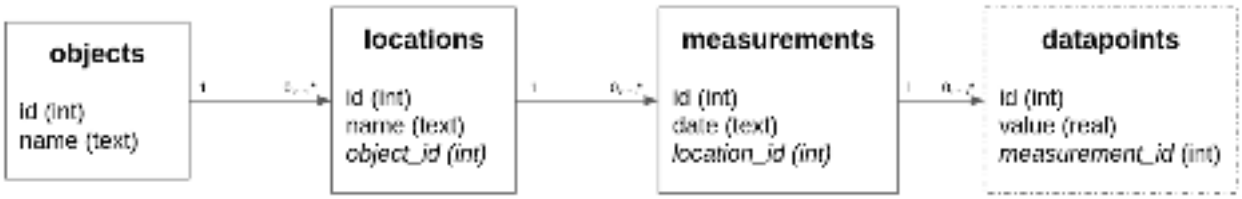


Figure 4.2 – Class Diagram of the SQLite database. The table **datapoints** is dashed because it is a fictive one, in the sense that it doesn't exist to the outside of the API.

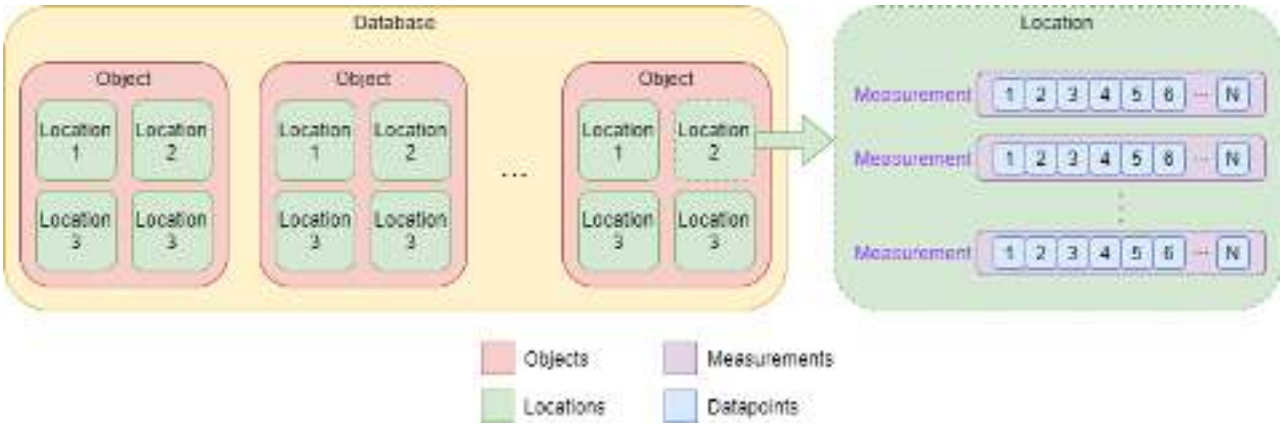


Figure 4.3 – A graphical representation of the database.

Our database contains 4 tables. To the 'outside world', only 3 of them matter:

1. **Objects**: contain the objects saved, for instance **one plant** located somewhere in someone's room.
2. **Locations**: each object must have several 'locations', i.e. points, where someone can touch. For instance, it can be **the leaf at the top** or it could be **a leaf at the bottom**.
3. **Measurements**: each location must contain several measurements for the **training** process of our **classifier**.
4. *Data-points*: this object is **fictive** in the sense that it doesn't matter to the outside world. A measurement is an array of **real** valued numbers, and to represent this in SQLite, one must create a table of real-valued rows linking to a specific measurement.

4.1.2 How to access the database using SQLite

One of the most powerful way to work with SQLite is through the **command line** software **"sqlite3"**, which allow a quick inspection of the content of the database, as well as modifications if required. In this section, we present to the reader how one can do this with a set of different commands to inspect the database.

— Create the connection to the database (i.e. open sqlite3) with

```
sqlite3 chic.db
```

— Show all the tables of the database

```
.tables
```

— Show the fields of one table

```
PRAGMA table_info(locations);
```

— Perform some queries over the dataset (here are several examples)

```
select * from locations
select id from locations where object_id = 2
select * from datapoints where measurement_id in (select id from measurements
    where location_id == 3);
```

As one can see on the last line of this example, it is possible to perform really complex queries with a single line of code.

4.1.3 Python API for better integration with our software

Working with SQLite in the command line is a handy tool, however it is not suited for integration with the rest of our software. Since the **user-interface** was written in **Python** (see next section), we created an **API** that can be used in python. Here is an example on how to load the API from any python file, with an example.

```
# 1. Import the API
import database_gestion as database

# 2. Create the connection
conn = database.create_connection("chic.db")

3. Example of a usage of the database
# reset all entries
database.reset_db(conn)
# create 2 objects
database.create_object(conn, 'obj1')
database.create_object(conn, 'obj2')
print(database.get_all_objects(conn))
# add a new object
database.rename_object(conn, 2, 'new name')
print(database.get_all_objects(conn))
```

The list of publicly available function is given here.

— **create_connection()**: it returns a connection to the database.
— **create_tables()**: it initializes the database creating the empty tables described above.
— **create_object**: it is the function to call when creating a new **object**. It will also create 4 locations (points of the object) associated to this object, with one of them being the "rest-state". It will return the ID of the object, and the IDs of each locations created.
— **get_all_objects**: returns all the saved objects (values are returned as tuples). This function must be used by the UI when it is asked to display the list of Objects, while keeping track of the id of each one of them.

— **rename_object**: rename the given object.
— **delete_object**: delete the object and all associated records, using a cascade rule for deleting.
— **save_points**: will add an array of points (a measurement) to a given location (location is provided using its unique id).
— **reset_db**: reset the entire database.
— **get_measurements_for_locations**: given a location, will return all the measurements that were done. This function is meant to be used by the classifier
— **get_locations_id_for_object()**: it returns the IDs of the Locations that belong to an object.
— **delete_measurements_from_location()**: it resets a location by deleting all of its measurements.

This API is used in a set of limited scenarios and was designed for them. Here are when the API is called.

1. As the database is made to save the **training measurements** for our classifier, its API can be called to save new training datapoints with the method **save_points**.
2. Once enough datapoints have been added to the database, the **classifier** can load all of them and train itself over them.

4.2 UDP Server in Python

User Datagram Protocol (UDP) server is the element which allows a communication between the main computer (and its user interface) with the potentially several sensors. UDP is a protocol part of the Internet Protocol Suite. Last semester, we implemented a UDP server using a **Java** library which still can be used with Processing. However, since we added elements of the software that are in Python, it was handy to have the UDP gestion in python as well.

The code to create a UDP server is really easy in Python. The code presented now shows how to create our UDP server which is waiting for **arrays** being sent as strings from the ESP8266.

```
import socket

# CONSTS and VARS for the logic
UDP_IP = "192.168.153.120"
UDP_PORT = 5005

if __name__ == "__main__":
    # 1.  create the UDP connection
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP
    sock.bind((UDP_IP, UDP_PORT))
    self.sock.settimeout(0.5)

    # 2. loop reading received data (inside a thread)
    while True:
            try:
                if (self.is_running == False):
                    print("DEBUG: raising UDP server stopped exception. (in 'UDPServer.run()')")
                    raise UDPServerStoppedException

                data, addr = self.sock.recvfrom(1024) # buffer size is 1024 bytes
                data = data.decode()

                if data == "s":
                    self.msg_start_received = True
                elif data == "e":
                    self.signals.measurementReceived.emit(self.received_data)
                else:
```

```
            self.received_data = data

        # capture exceptions
        except UDPServerStoppedException:
            print("DEBUG: UDP server stopped. (in 'UDPServer.run()')")
            return
        except socket.timeout:
            print ("DEBUG: Socket timed out. (in 'UDPServer.run()')")
```

A careful reader would see the line **while True** which is often associated with bad coding practice. Indeed this code gets sucked in an infinite loop, and therefore it must be inserted into a **thread** of its own. The thread can be stopped by setting the attribute *"is_running"* to *false*. Note also that if the socket doesn't receive any data for more than a predefined time, it raises an exception and stops waiting. This avoids that the code gets stuck if the sensor stops working for some reason.

4.3 Machine Learning Classification

The task of the **classifier** is to given a new **measurement** (*that is not part of the training dataset*) predict the **class label** of the measurement, that is the **position** of where the measurement was taken. As mentioned above, this task involves a **training dataset** of measurements with known positions that the classifier must use to predict for unseen data. Classification algorithms are a class of **supervised learning**. When it comes to classification, the work involves the selection of a classifier, the tuning of its parameters, and finally the validation of the selected model.

4.3.1 Classification Inputs

In order to select a good model, one must first observe the inputs of the classification task. In our case, we talking about electrical measurements done at several frequency.

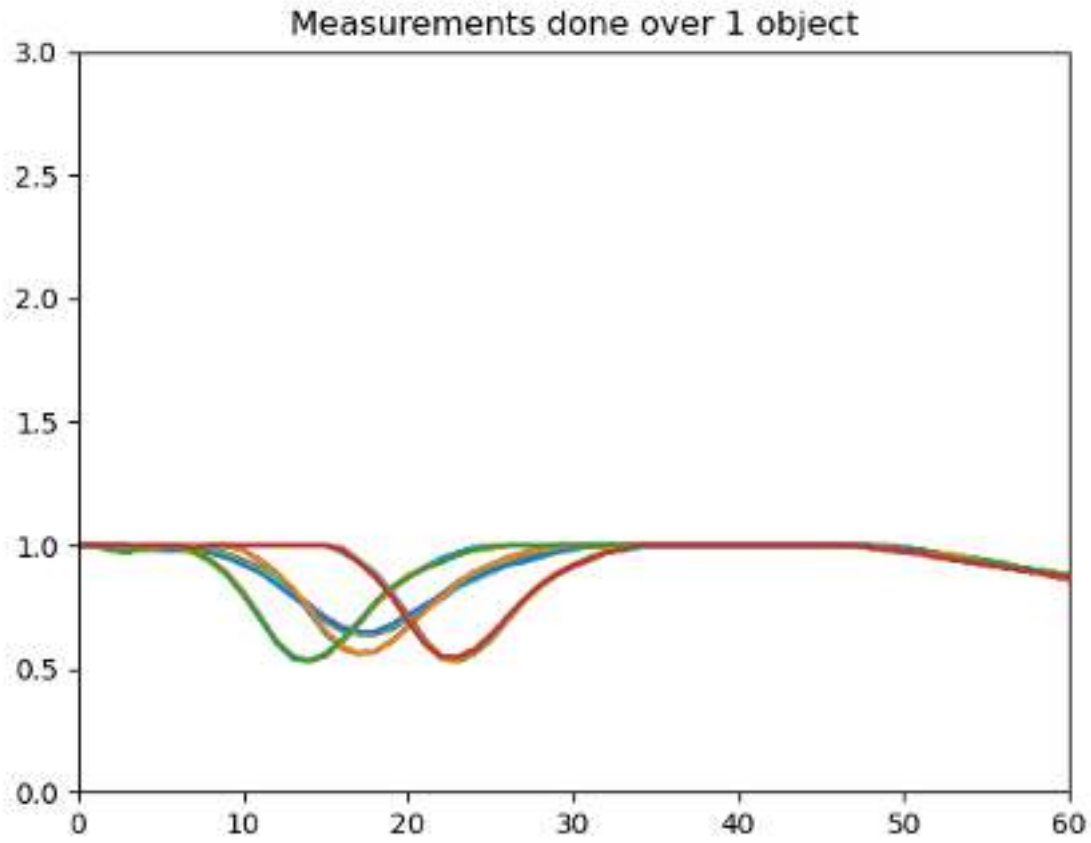


Figure 4.4 – Inputs of the classifier are sensing measurements done on object. This figure show several measurements done on the same object at 4 different positions.

More precisely, our inputs are **arrays of points** representing a function of the form $\mathbb{R}^1 \to \mathbb{R}^1$. As one can observe on the Figure 4.4, the job seems not so hard (easy enough so that humans can do it). There are two possible approaches: (i) to first extract features from measurements and then to use them as inputs to the classifier, or (ii) to use the raw datapoints in the classifier. Generally, easier approach to classifier involves some **features extraction** and harder approaches (such as some neuron network, e.g. Deep Learning) don't.

In our case, it seems like there is an apparent and rather **obvious** feature, being the **first minimum** of the curve. Therefore we decided to go for the method (i).

The outputs of the **features extraction** are the **normalized first minimum (and arg-minimum) value** of the array of points, which is - in a more physical approach - the *first and normalized* minimum voltage output and frequency at which it was recorded.

Why normalization is required ? It can seems strange to apply a normalization to physical features, in the sense that after being normalized, features loose their real meaning. The reason is that to the eye of the classifier, the input data is just **data**. Since the ranges of the 2 variables that matter (the frequency and voltage of the voltage minimum) are very different, it is important to apply a normalization of the form

$x_i = (x_i - mean(X))/std(X)$

where $mean(X)$ and $std(X)$ are the mean and standard deviations of features over the training dataset.

4.3.2 Selection of the classifier

As the input data is not high-dimensional, we wanted to try to perform classification with a simple classifier. One of the simplest classifier is the **KNN** classifier (K-nearest neighbors) which has the advantage of being extremely easy to understand. Classification is a **majority vote** performed on the class label of the K nearest neighbors of the new query point in the training set. KNN is a type of non-generalizing learning: it doesn't try to learn something from the dataset by training model, but simply stores instances of the training data.

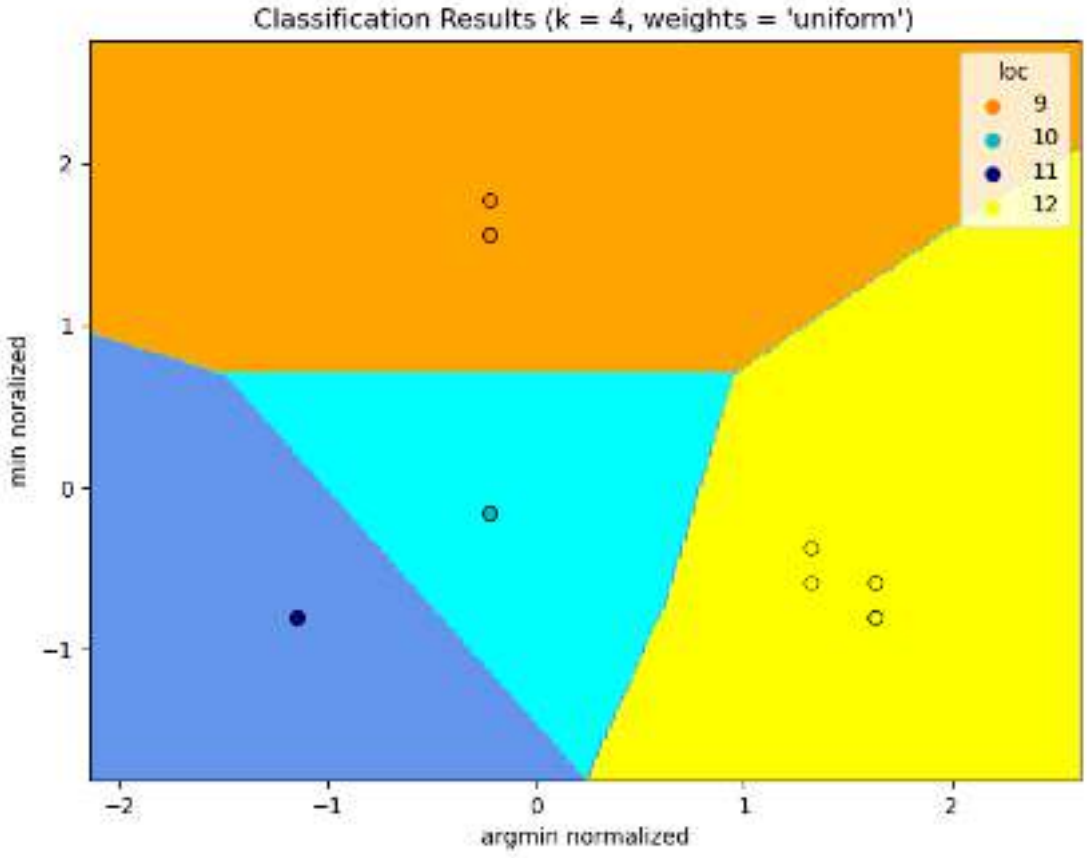


Figure 4.5 – KNN classification Results

The figure 4.5 shows the classification result for the same object presented above as a 2D function of the features. The classification boundaries are represented with colors. One can see that there is a **clear** and **precise** definition of the boundaries which, in the case of the training dataset, works really really with an accuracy of 100 %. The parameter $k = 4$ was selected to not be too high. Indeed, the training sets of this case are not necessarily big and a large k would lead to miss-classification because boundaries would be less distinct. At the same time, $k = 4$ is also large enough to get ride of 'noise' datapoints within the training set, if there are some.

It makes sense at this point to try and verify those results using a different results. The following plots recap the results obtained and tries to do the same with another object, where the **points were harder to distinguish**.

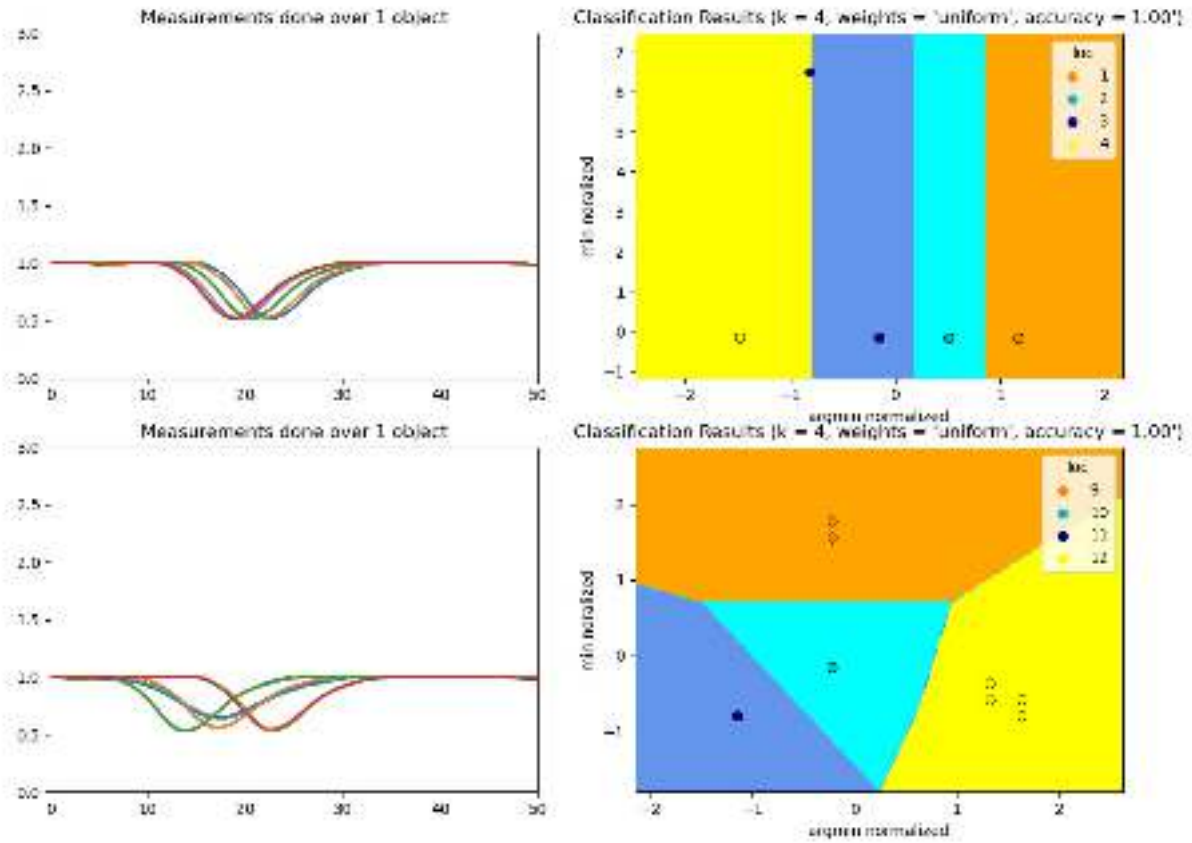


Figure 4.6 – KNN classification Results

One can observe on Figure **??** the training dataset on the left column, and on the right column the classification results obtained with our classifier. The first row represents an **headphone** touched at 4 different places, and the second rows is a **metallic hand**. Even for the first object, which seems to be harder to represents (as the voltage minimum is pretty much always the same), the classifiers performs really well and stays at an **accuracy of 100 %**. Since it is the case, we decide to select this model **KNN(k=4, weigths='uniform')**. The next step in the **machine-learning** pipeline is the **model validation** with an **unseen** testing set.

4.3.3 Validation of the model

Using the Python API (see next section), it was possible to integrate the classification task into the user-interface and to test the performances with the software. We achieved a really accurate classification.

4.3.4 The Python API

The classifier is meant to be used within the user-interface. Therefore, we wrote an API to make the **integration** of the classifier very easy for external Python Code, which contains a unique function: 'get_classifiers' that returns the classifiers Python Object for each of the objects present in the database. Here is a demonstration of how to use this API paired with the database API.

```
import classifier
import database_gestion as database

# create connection
conn = database.create_connection("roots.db")

# get the classifiers for this training set
classifiers = classifier.get_classifiers(conn)

# select one classifier (depending on the object)
object_id, classifier = classifiers[2]

# try to predict something given a new measurement
measurements = [...]

# and make the prediction
if classifier is not None:
    predicted_location_id = classifier(measurements)
    print("Predicted location : ", predicted_location_id)
```

4.4 User-Interface

We wanted to make our work **easily understandable**, so we decided to build a **graphical interface**, which can be used to demonstrate the capability of the sensor. Developing a comfortable user interface was a **time-consuming** task, mainly because we wanted our app to have a finished look. Tweaking the details costed us a lot of effort but the results were **very rewarding**, also because the app really helps highlighting the work that was done on the sensor. The current version of the app provides a **standard user interface** which allows to save/open files and implements the basic functionality needed to use our sensor. It can also detect if the sensor is disconnected or if it is not working properly and prompt the user to notify that something is wrong.

4.4.1 App design

The app is very simple: the user can create a **list of objects** to which he/she wants to attach the sensor. For example the user may create the objects "Lamp" and "Metallic hand".

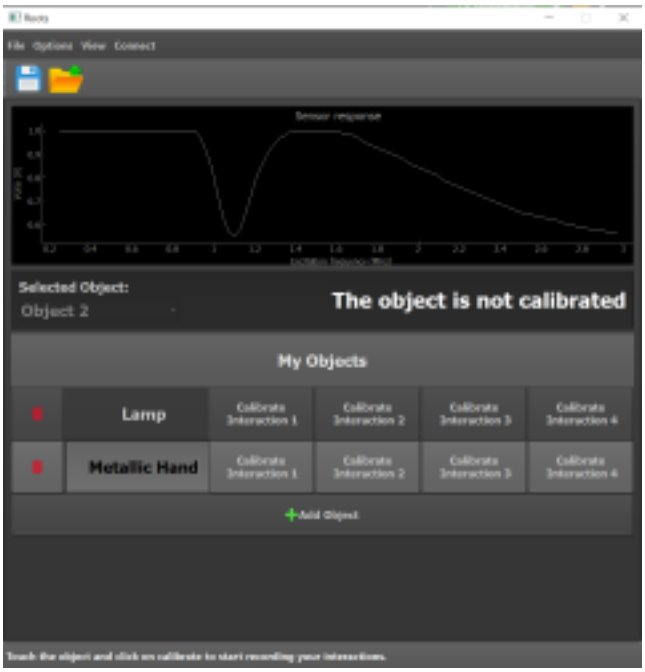


Figure 4.7

The objects are created by clicking on the "Add Object" button, and their name can be changed by double-clicking on it.

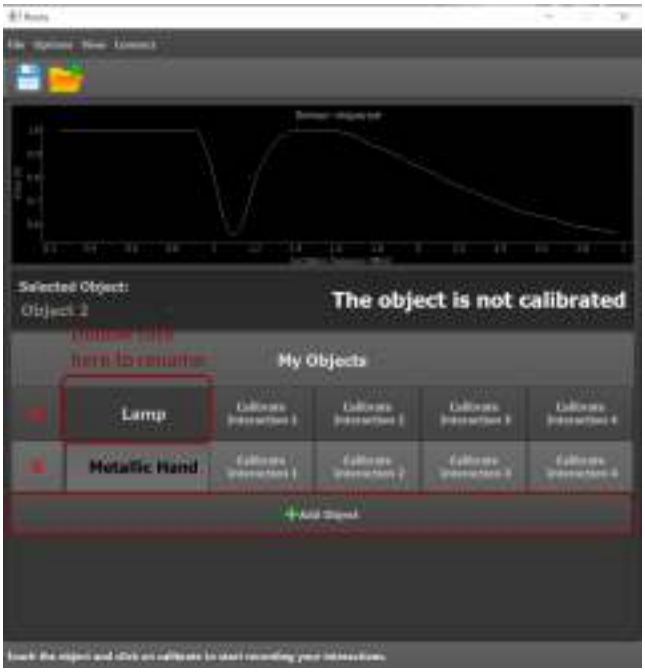


Figure 4.8

Each object has 4 **calibration points**, this means that we can calibrate the software to recognize **4 different interactions** with the user. To calibrate one interaction, one must **touch** the object and press on the **calibrate button**. Once calibration is completed, the button will turn green. Calibration, in a more machine-learning way, means **creating the training dataset**.

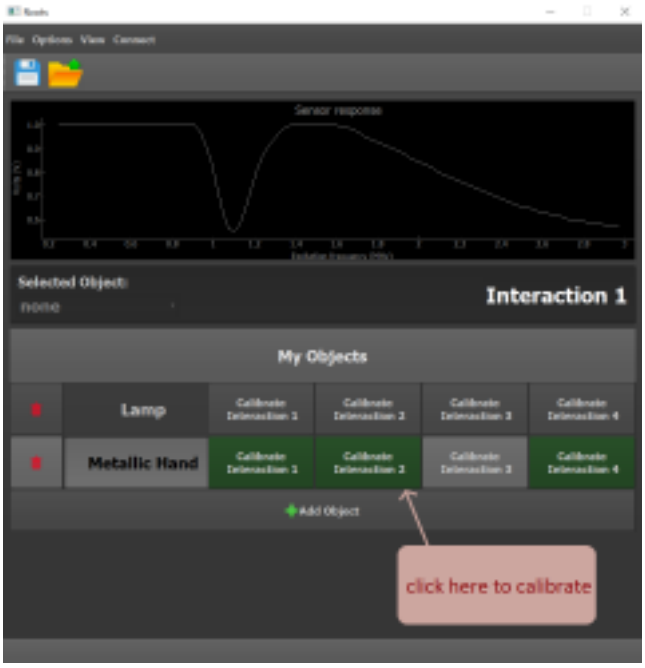


Figure 4.9

The user should then select the current **active object** by clicking on its name, or selecting it on the list on the left.

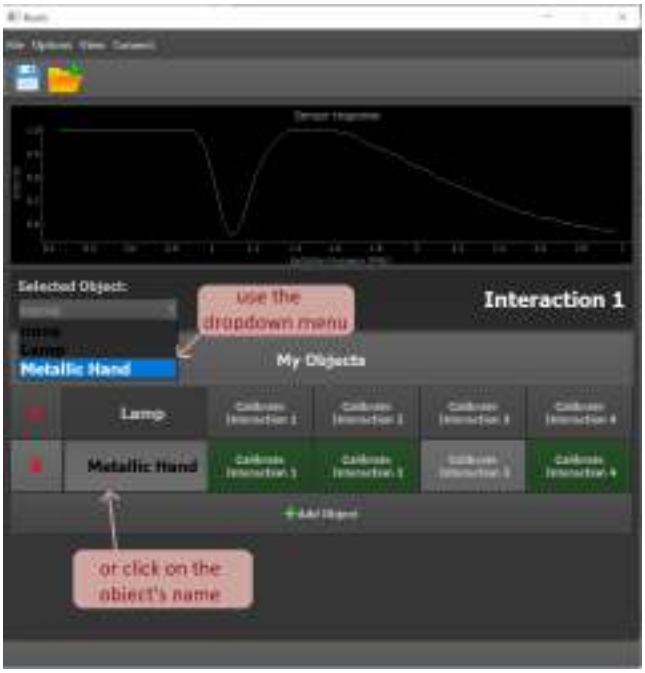


Figure 4.10

The application will try to guess the interaction using the current output of the sensor and the training data and it will show the name of the interaction that is **closest to what it observes**. For this reason it is a good idea to train one of the 4 calibration points as "no touch", so the app will be able to detect this situation.

Figure 4.11

Figure 4.12

Note that the application shows in **real time** the output of the sensor, and the user can zoom, pad, and even export the image shown on the graph. Being able to see the response of the sensor is very useful when calibrating the object, because interactions that yield very similar responses can be hard to discriminate from each other and should be avoided. In the future it may be interesting to add a functionality to tell the user if there is enough difference to correctly calibrate the point he/she is touching.

4.4.2 Technical Implementation

We decided to use the Qt framework, which is widely used and **cross-platform**. It is natively written in C++ but there exist 2 **Python** modules that basically wrap the C++ library and provide a one-to-one correspondence with the native functions. Since we wrote our machine learning algorithm in **Python** it was our natural choice to write the graphical interface also in Python. At the time of writing, these modules are **PyQt5** and **PySide2**, the last one being the one we used in this project. Note however, that they are almost 100% compatible and differ only in some minor syntax, so choosing one over the other is mainly a matter of preference.

4.4.3 Simplified State Machine

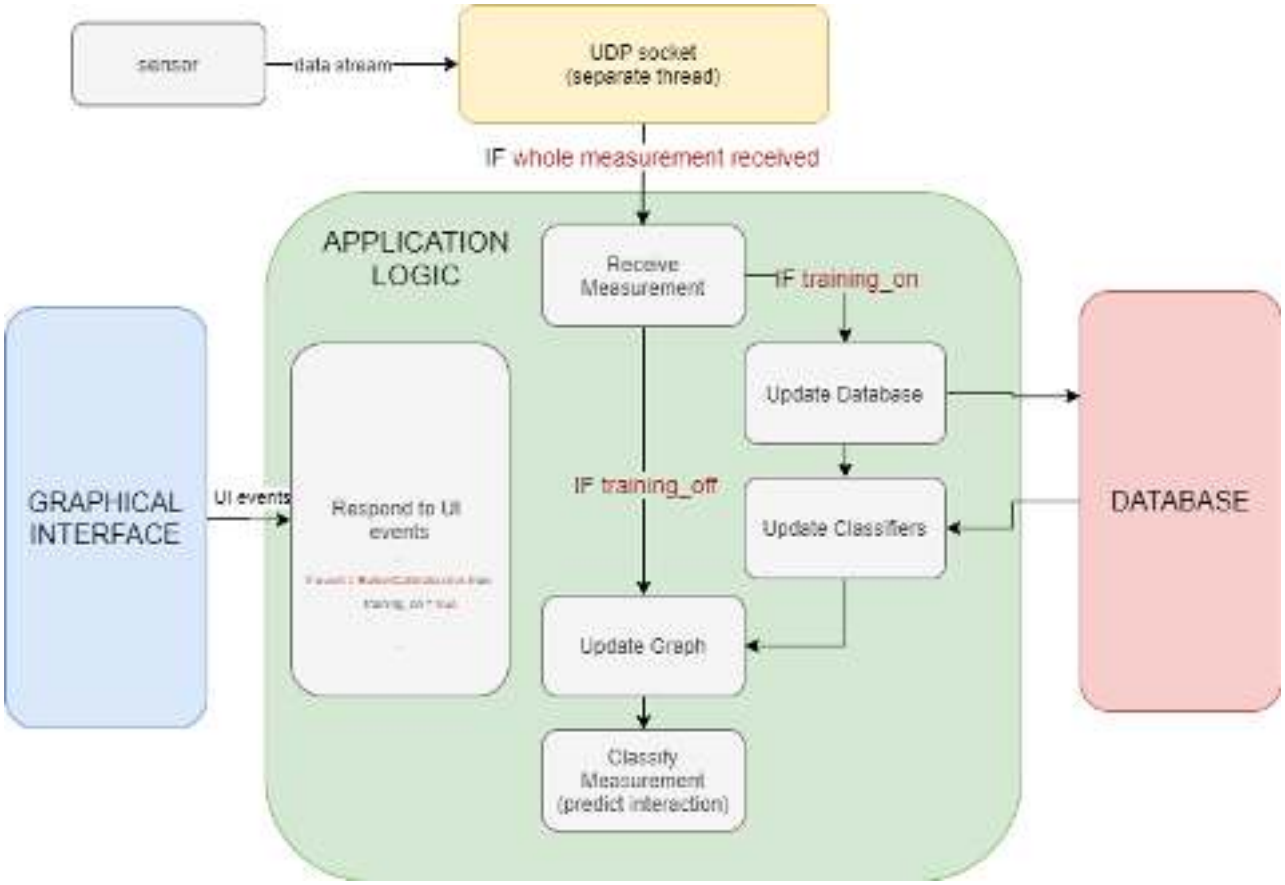


Figure 4.13

The diagram above captures the essence of the logic of our application. A separate **thread** constantly listens to the data sent by the sensor and transmits it to the application when a whole measurement is received (i.e. when N datapoints are received, which corresponds to a full frequency sweep). The application constantly updates its graphical interface with the incoming data, in other words it **redraws the plot** and tries to

predict where the user touched the object using the current training data. If training mode is on, the application updates the database, retrains the classifiers and updates the graphical interface. Note that this representation **abstracts** from all the details that we had to implement in the real application and its purpose is merely to illustrate the **general concept**.

4.4.4 UML diagram

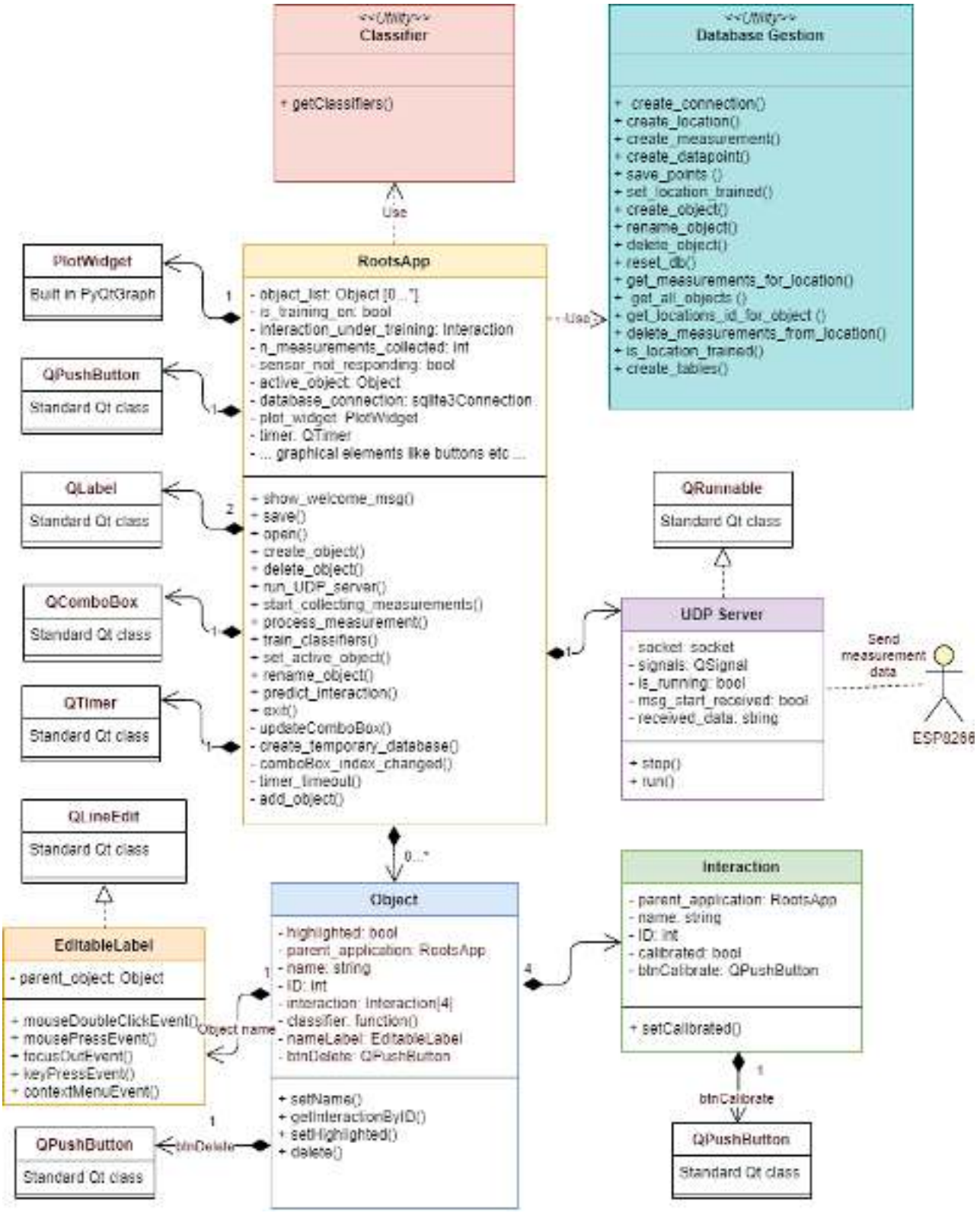


Figure 4.14

4.4.5 Data Structure

The application is organized as follows: there is a main class (RootsApp) which contains the **logic** of the application and which is responsible of **communicating with the database**. It also defines and creates the **window**, and it contains all those **graphical elements** that **do not change** during execution, as for example the menus, the drop-down list, the graph, or the output label.

On the other hand, the information relative to the objects, as well as their graphical elements, are **encapsulated** within the class "Object". The main class "RootsApp" holds a **list of "Objects"** that is dynamically updated when the user creates or deletes objects. Note that "RootsApp" doesn't need to know about the graphical interface of each object, which is responsible for its own interface. This encapsulation principle makes it **easier to modify** the application and to handle the graphic elements in a dynamic context. For example, when the "RootsApp" class creates a new object, all the corresponding buttons and labels are automatically created by the constructor of the "Object". Similarly, each "Object" instance contains four "Interaction" objects, which automatically manage the "Calibrate" buttons.

Since the logic of the application resides in the "RootsApp" class, all **events** captured by the graphical elements **are processed by this latter**. For example, clicking the calibrate button triggers a function in the "RootsApp" class, which then tells the objects to update their graphical interface. This ensures the **coherence** of the application and most importantly of the **database**.

4.4.6 Database

The database contains data about the objects, such as their name, ID and most importantly the **sensor data** associated to each object. To avoid duplicating code we wrote a set of utility functions that allow to write and retrieve data from the database easily, and which are listed in the UML diagram above (blue box). The purpose of the database is to **store** and **organize** the sensor data that will be used to train the classifiers and to **record the state of the application** when the user saves the current workspace.

4.4.7 Classification

Classification refers to the task of guessing the user interaction based on the sensor data, using machine learning algorithms. We decided to **abstract** the classification algorithm from the core of the app, to allow future modifications without the need to redesign the application. This is done in the following way: each object contains a **classifier**, which is nothing more than a pointer to a function. This function takes as argument the last measurement received from the sensor and is defined inside the *"Classifier"* utility class. For this reason, it is independent from the rest of the code, and we might imagine that in a second version of the app the user will be able to select the desired classification algorithm among a list of choices.

4.4.8 Communication

The incoming data received from the sensor is received by a **separate thread**, which is called *"UDPServer"*. This ensures that the user interface **doesn't freeze** while the application is processing the data in the background. The thread is started at the beginning and killed when the application is destroyed. It constantly retrieves and analyze the incoming data, and a simple messaging mechanism allows it to send the sensor data to the user interface whenever a whole measurement is received.

4.5 ESP8266 code

The ESP8266 is used to **collect data** from the sensor and send it to the host computer. Its goal is to program the AD5932 chip (on the sensor module) to generate a **frequency sweep** and then measure the response of the sensor as a function of frequency. The ESP8266 takes a number N of measures and sends the collected data to the host computer through the UDP connection.

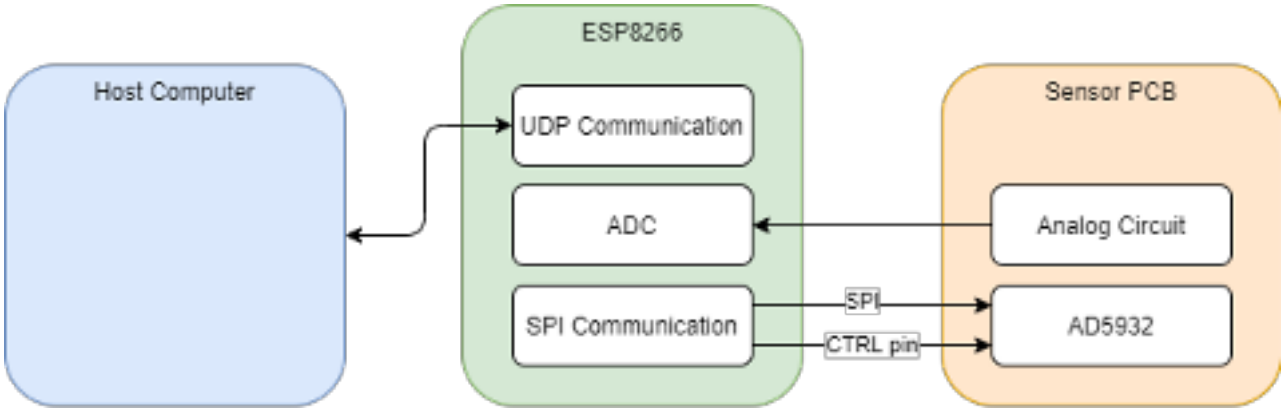


Figure 4.15 – A schematic representation of the structure of the ESP8266 code.

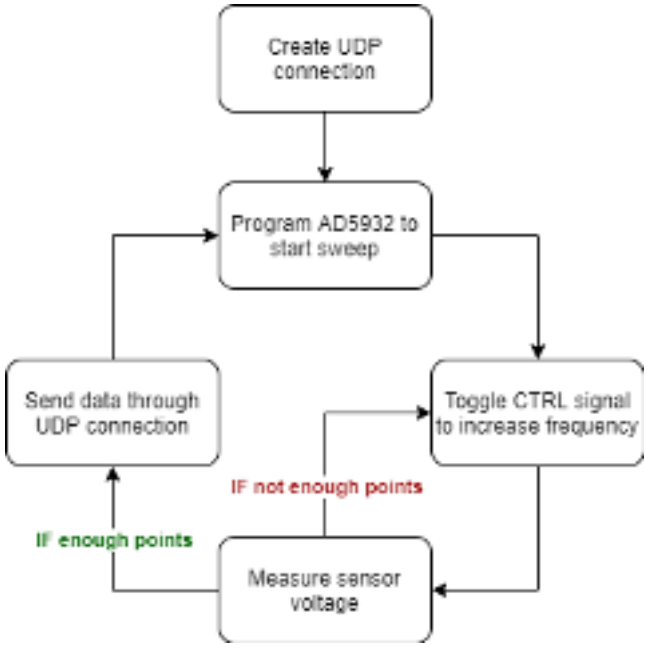


Figure 4.16 – The state machine of the ESP8266 code.

The AD5932 contains several registers that need to be programmed through the **SPI interface** to set the properties of the sweep such as the starting frequency, the number of steps and the final frequency. The communication requires **3 connections** to the following pins of the AD5932: SDATA, SCLK and FSYNC (which should be held low to signal that a transmission has started). The sweep can be programmed to be of 2 types: automatic or manual. In the first case the circuit automatically increases the frequency according to a pre-programmed time interval, in the second case the frequency increment must be triggered by toggling the CTRL pin of the AD5932. We chose to use the second option because it gives us more control over the circuit, even if it comes at the price of an additional connection.

4.5.1 AD5932 Arduino library

In a first phase of the project we programmed the registers of the AD5932 bit per bit, however this turned out to be very cumbersome, so we wrote a **small library** which can be deployed on an Arduino to help program the sweep and control the AD5932. Basically the library is composed of a **class** which holds all the information

needed to program the sweep (in other words the desired settings of the AD5932) and a few methods that format this information and send it to the AD5932 through the SPI interface. In the picture below is shown the UML diagram of this class:

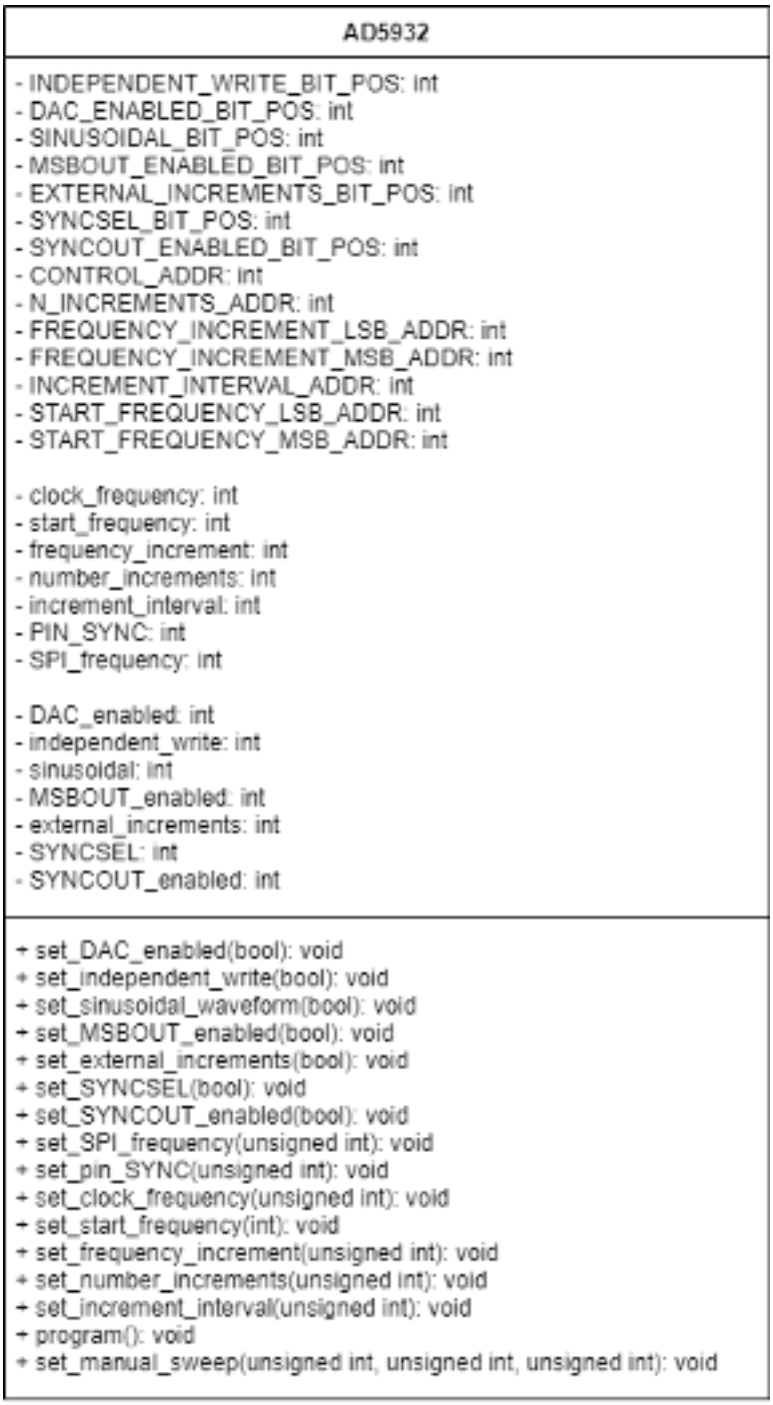


Figure 4.17 – The UML diagram of the simple library we wrote to program the AD5932: the first variables in capital letters are constants values that hold the adresses used to program the AD5932. The other class variables hold the settings with which the user wants to program the AD5932. The methods allow the user to change the settings and to program the AD5932 with the selected options.

Our library makes use of the **standard Arduino *"SPI"* library** which uses the pins 12 (MISO), 13 (MOSI) and 14 (CLOCK) of the ESP-F module for the SPI communication (note that the ESP-F is one of many module available containing the ESP8266 https://fccid.io/2AL3B-ESP-F/User-Manual/User-Manual-3387875). However, the **MISO pin is useless** in our case because the communication is unidirectional. For this reason we chose to use pin 12 to control the FSYNC pin of the AD5932, which **signals the start** of a new SPI transmission, but from a software point of view any other output pin could be used instead. All these low level details are taken care of by our library and the user only needs to specify which pin is used to control the FSYNC signal, note however that the SCLK and MOSI pins can't be changed.

4.6 Optional IP generation for Raspberry Pi

In order to connect to the server, sensors need (*i*) to be connected to the same wifi network as the main computer (in other words, they need to know the wifi's ssid and its password, written in the code of the device) and they need (*ii*) to know the main computer's IP address.

The first requirement is not really a problem: once a wifi password is set, most likely is that it won't change (and if it does, it is not a common situation). This constraint the user to **upload once** the code once he/she receives the PCB. Since we are targeting the **maker** community and that it has become really easy to upload code, it is not a big problem.

However the second requirement is a lot more problematic: it is quite often that computers change their IP address automatically, and it is surely more annoying to re-upload code with the newest IP address of the UDP server every-time the user wants to use the sensor. The most likely reason for the change of IP address is that the lease time for the IP address given by the Dynamic Host Configuration Protocol (DHCP is a protocol used to assign an IP address) has expired.

One possible solution is to change the settings of the DHCP to increase the lease time, but this solution is not the best since it also influences other devices connected to the network. This is why we have come up with a better solution for this: **impose that the main computer always have the same IP address**, by creating a new wifi network only for the purpose of connecting sensors to the main computer.

This solution was developed on tested for one **raspberry pi**. Indeed, our product was initially designed to work in pair with a raspberry pi equipped with Processing. An important note is to be made here: most of what we did this semester doesn't need a raspberry pi to work. The **classifier**, the **user-interface**, the **database** and the **UDP server** will all work on **any computer**. However, the steps explained in this section are designed to work on the raspberry pi and more generally on any Linux machine.

It is possible, using the raspberry pi's wifi card, to create a **wireless access point** which is like a **secondary network**. If raspberry pi is connected to Ethernet, it will extend the network to wireless devices. And if raspberry pi is not connected to internet, it will just allow for the communication between connected devices and the computer. This procedure requires 2 software: **hostapd** and **dnsmasq** and involve changing the network configurations of the computer. The procedure is explained on this link.

4.7 Acknowledgments

A special thanks to professor Beuchat for his precious help in debugging and designing the PCBs, as well as his remarkable willingness to help and to forgive our delays. We hope we didn't bore you too much with this very long report!