Politecnico di Milano

School of Industrial and Information Engineering
Department of Mechanical Engineering

Master of Science in
Mechatronics and Robotics

Design and implementation of real-time edge computing algorithm for tool wear forecasting in industrial machinery

Supervisor:
Prof. Ing. Alfredo Cigada

Co-Supervisor:
Ing. Leonardo Iacussi

MSc Thesis by:
Gabriele Maisano
matr. 977527

Academic Year 2022/2023

Abstract

The following paper will explain in detail a project of predictive maintenance applied on a punching machine.

The project consisted of three main parts:

1. Define the machine most critical components;
2. Find the right sensor for the application;
3. Develop an algorithm that takes the sensor's data and discriminates whether the machine is operating in a normal or faulty condition.

In order to define the most critical components, it's important to explain how the machine is made and how mechanical parts interacts with each other to obtain the punched sheet metal.

The definition of the sensor was made considering mainly the minimum sampling rate to avoid aliasing and the minimum full scale to avoid saturation.
Additionally, spacial and electrical constraints are taken into consideration, too.

At the end, the selection of the most suitable algorithm was done comparing a peak recognition algorithm, different machine learning models and a neural network approach.
The final algorithm was developed considering that it need to be deployed to a sensor node that can perform edge computing.

Keywords predictive maintenance; punching machine; artificial intelligence; machine learning; real-time edge computing; tool wear forecasting

Acknowledgements

It's difficult for me to enumerate all the people that led me during this extraordinary journey.
This thesis is only a very small part of my university career and i want to thank every friend, every professor, and every colleague that sustain me during this experience.

Let me do the acknowledgements in my mother language so that i can be more spontaneous.

La prima persona che vorrei ringraziare è la mia fidanzata Francesca, veramente per tantissime cose, ma soprattutto per la pazienza che ci mette a ricordare tutto quello che io dimentico.

Poi naturalmente un gigante rigraziamento va a tutta la mia famiglia, che in ogni momento mi ha supportato e mi ha sempre dato la carica nei momenti un po' più difficili.

Inoltre, desidero ringraziare il professor Cigada e il suo dottorando ing. Iacussi per il loro prezioso contributo, i consigli preziosi e il costante sostegno che mi hanno fornito durante il percorso di tesi.

Ci tengo anche a dire grazie a tutti i miei colleghi e amici che in qualche modo fanno tutti parte di questo elaborato.
In particolare ad Alessandro, Andrea, Marco, Alberto e Bruno, che sono stati dei colleghi con cui stare assieme è sempre stato un piacere, e con loro ringrazio anche tutta la KONE Industrial che mi ha dato l'opportunità non solo di effettuare il lavoro di tesi al suo interno, ma anche di conoscere un gruppo così coeso e affiatato.

Un gigante ringraziamento anche a tutti i miei amici di sempre: Aurora, Sofia, Elisa, Camilla, Tatjana, Beatrice e Giulia per gli innumerevoli momenti indimenticabili passati assieme.
Poi Matteo, Valerio, Davide B., Davide S. e Luca per la disponibilità, pazienza e affetto che hanno sempre dimostrato nei miei confronti.

Impossibile dimenticare Valentina e Alberto, che sono sempre stati esempio di costanza e ambizione per me.
E Alan e Giulia che con la loro tenerezza e curiosità mi hanno insegnato a vedere sempre il bello delle cose.
Infine, ma non per importanza, i miei ex compagni del liceo Riccardo, Daniele, Matteo, Lorenzo, Alessia, Virginia, Beatrice e Rebecca con i quali conservo tantissimi ricordi veramente preziosi e, ogni tanto, imbarazzanti.

Contents

List of Figures

1 Introduction

1.1 Context

Before you get to the heart of the project, I want to explain the context in which this paper takes shape.

The whole project work was carried out during an internship at KONE Industrial, a well-known company that produces and installs elevators and escalators worldwide.

The company was founded in 1910 in Finland and has grown to become one of the leader manufacturers in its field; at the moment, KONE has approximately 60'000 employees worldwide.
I worked in the headquarters of Pero (MI), Italy, that is the division responsible for the production of the standard cabins and doors of an elevator. In particular, in this company unit the final product is completely developed from a basic sheet of metal. It is punched, then pressed and shaped to become a wall of an elevator cabin, then to improve its structural integrity, additional reinforcements derived from another basic coil of metal are incorporated; furthermore, to minimize noise and vibrations, an anti-noise material is attached to the back of the cabin wall.
Similar manufacturing processes are employed for producing elevator doors starting from a basic sheet of metal as well.
Overall, in the factory there are five production lines characterized by a high level of automation, and each line can incorporate a punching machine, a bending machine and anthropomorphic robots that are responsible for fixing the reinforcements to the wall using specialized adhesive and rivets.

In Pero my desk was in the maintenance office, that is conveniently close to the punching machine on which I needed to put sensors and other hardware components.

The goal of this project was to provide a predictive maintenance solution to a punching machine, namely the Salvagnini S4 with the serial number

1262. This machine is quite old, as it was produced in 2009, and it does not have any available for consultation analog sensors inside it.
However, in order to implement a predictive maintenance solution, many data of the machine have to be collected, so at least one sensor must be installed.

The sensor can be attached to the machine in a permanent, semi-permanent, or removable way. We decided that the permanent way was the best one, since it guarantee a continuous monitoring of the state of health of the machine. The selection of the most suitable sensor was the most critical point of the whole project, so a more detailed discussion will be engaged later on.

After the sensor data have been collected, they need to be processed and evaluated in a central processing unit (CPU) that can detect the abnormal working condition of the operating machine.
Different algorithms were compared for this purpose, with particular emphasis placed on Artificial Intelligence algorithms.

The structure of the thesis is outlined below, providing an overview of the guidelines that led the progress of the project.

1.2 Thesis outline

The paper is structured in the following way:

- In chapter 2, it's explained what predictive maintenance is and why it is convenient to adopt it.
- In chapter 3, the machine and the operations performed are described in details.
- In chapter 4, we discuss about the sensor and the hardware components that need to be implemented in order to carry out the data collection and analysis.
- In chapter 5, we present our solution for the data analysis, considering different approaches.
- Finally, in chapter 7, we present our conclusions and possible future paths toward which our work could be extended.

2 What is predictive maintenance?

2.1 Different maintenance solutions

The maintenance of the machine is a fundamental requirement to guarantee continuous and correct operation of the mechanical machine.
Depending on the maintenance approach that the enterprise want to undertake, the costs of the repair and of the preventive actions can change a lot.

The main and most used maintenance strategies are the following:

- Run-to-failure (breakdown maintenance);
- Preventive (scheduled) maintenance (PM);
- Predictive maintenance (PdM).

Run-to-failure strategy

The run-to-failure approach (also called reactive maintenance) is the most simple approach. In fact the machine is let operating without any maintenance operation until one or more components fails and need to be substituted.
It may seem that this approach is a bad and ineffective strategy, but this approach is used often for non-critical components, which don't produce a dangerous or expensive damage in the system.

Preventive maintenance strategy

As opposed to the reactive maintenance, the preventive maintenance approach is largely used when we are dealing with critical or expensive components.
This strategy consists in changing a piece on a regular basis, which depends either on operating cycles or operating time. For example a rubber gasket need to be replaced after some time, because it lose its plastic properties;

while a cutting insert need to be replaced after some cycles because it lose its cutting capacity.

The scheduling of the maintenance bring the company to reduce the downtime costs and to maintain a good level of standard quality in the finished products. The drawback is that the scheduling process need some time and money in addition to the ones spent to substitute the worn components of the machine.

In KONE this approach is largely used for the punching tools, the fluid dynamic pumps and other critical components.

Predictive maintenance strategy

The most complex approach is predictive maintenance, and it is an optimization of the preventive approach.
It consists of getting data from the system, analyzing them and define a maintenance schedule based on those data.
In particular, it uses condition monitoring tools to detect various deterioration signs, anomalies, or equipment performance issues.
This strategy offers the advantage of fully utilizing components throughout their lifespan: in fact, the maintenance on the component happens only when data estimate that a piece of equipment is near to failure, avoiding unnecessary replacement of components that are still in their useful life and, at the same time, changing them before failure happens.

In synthesis, the major pros are reduced maintenance time and cost, longer asset lifespan, reduced risks relating to safety, environment, and quality; but the drawbacks are large investments in hardware, software, expertise, and staff training.
In fact, in order to implement a predictive maintenance strategy, machine learning, real-time data, and inter-connectivity must be used.

2.1.1 The best approach for the punching machine

In our case study, the best approach is the predictive maintenance one. That's because the punching machine is very articulated and expensive, and because we want to reduce the downtime cost as much as possible.

For those reasons, our goal is to schedule activities based on constant condition monitoring. So, once unhealthy trends are identified, damaged parts are repaired or replaced to avoid more costly failures.

In particular, our application aims to analyze the acceleration amplitude, which refers to the maximum rate of change of velocity experienced by a structure during its vibrational motion. We choose to go in that direction, because accelerations are key parameters in evaluating the intensity and severity of vibrations.

3 Preliminaries and State of the Art

3.1 The punching machine

In this section, a detailed description of the punching machine will be given. In particular, the layout, the main components, and an example of its operation will be presented.

3.1.1 The Salvagnini S4 - 1262

The punching machine has a conveyor belt for input and another one for output. During the operating condition, it is provided that a metal sheet enters the machine from an automated vertical warehouse, it is punched and sheared, and then it exits the Salvagnini S4 to go towards the bending machine to be transformed into the final product.

The figure below shows a synthetic machine scheme:

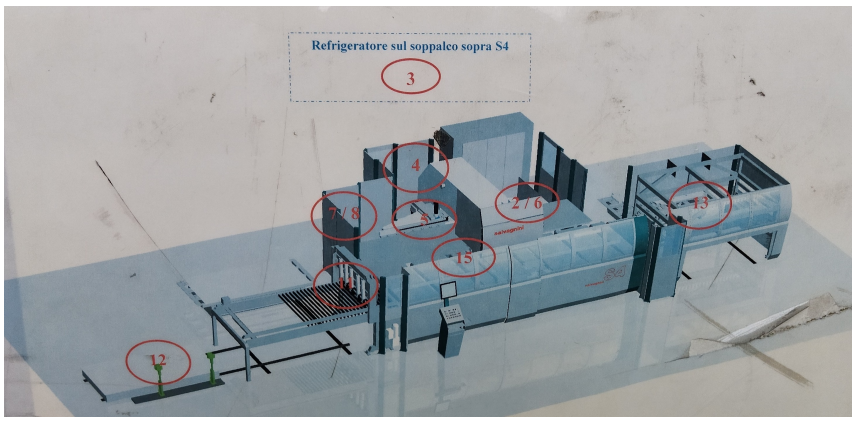


Figure 3.1: Layout of the punching machine

As you can see from the figure, the machine has the following main components:

- 2/6 : shearing and punching tools;

- 3 : refrigerator;
- 4 : lubrication grease tank;
- 5 : manipulator;
- 7/8 : scraps tanks, that are under the machine;
- 11/12 : input conveyor, that moves raw sheets;
- 13 : output conveyor, that moves machined sheets;
- 15 : working floors.

3.1.2 The main components of the machine

In this subsection, a brief description of all the main components listed before will be presented.

Shearing and punching tools

The shears are used to remove the excess of external material around the raw metal sheet, in order to give the right dimension to the sheet of metal and to prepare the bending process. The punching tools are used to make some holes in the metal sheet, in order to make couplings between the finished walls for the elevator or between them and other external components (cables, push-button panel, etc.)
The following figure (figure 3.2) shows the different kind of holes that the punching tools can produce.
Notice that a more detailed description of the machine head that includes the shearing and punching tools will be discussed in the section 4.6.1.

Manipulator

The manipulator is used to move the metal sheet during the punching and shearing operations. In fact, the punching and shearing tools can only move vertically, while every movement in the plane parallel to the ground (over the x and y axis) is addressed to the manipulator.
Two motors operate on the y axis by means of a ball screw driven transmission; while another two motors operate on the x axis by means of a rack and pinion transmission.
The manipulator has some clamps in the end in order to receive the metal sheet and keep it attached to the manipulator itself.

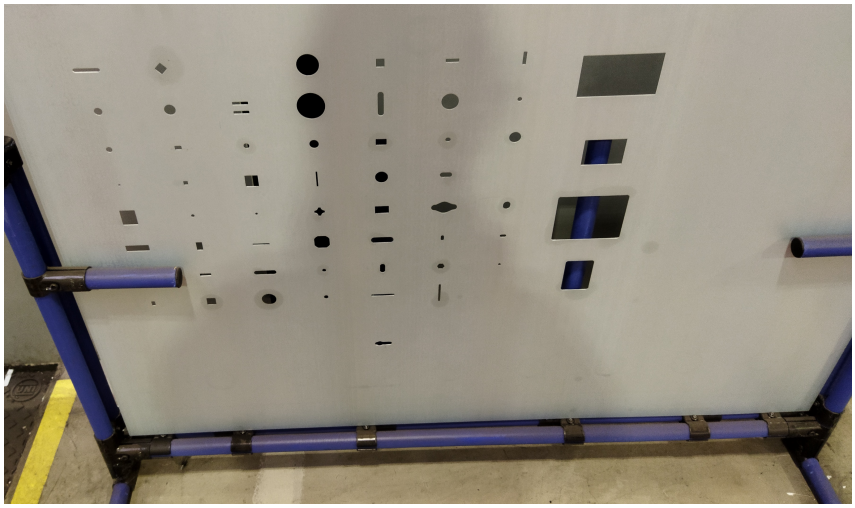


Figure 3.2: A punched metal sheet

Working floors

The working floors consists of both moving and fixed floors. The moving ones are used to receive the metal sheet from the warehouse, to place it within the clamps of the manipulator, and to move the punched sheet out of the machine towards the bending machine.
The following figure (figure 3.3) shows the manipulator from the lateral side and a part of the working floors.

Hydraulic pumps

The pumps are the actuators that use oil to govern the descent and ascent of the punching and shearing tools. In particular, a high-pressure pump operates on the punching tools during the active descent movement: the oil is sent to the tool at a pressure of up to 400 bar. Moreover, another low-pressure pump is used to manage the passive ascent movement of the tools, and it operates at 125 bar.
The pumping system is shown in the figures 3.4 and 3.5.

3.1.3 The operations of the machine

In this paragraph, all the operations of the machine will be explained in detail. Specifically, we will focus on the path followed by the metal sheet.

At the beginning, the sheet is taken out of the warehouse and led into the punching machine by means of a belt-driven conveyor.
While the conveyor is still running, a bar rises up to align the metal sheet in the x direction. At this point the conveyor stops, and a new bar rises up and

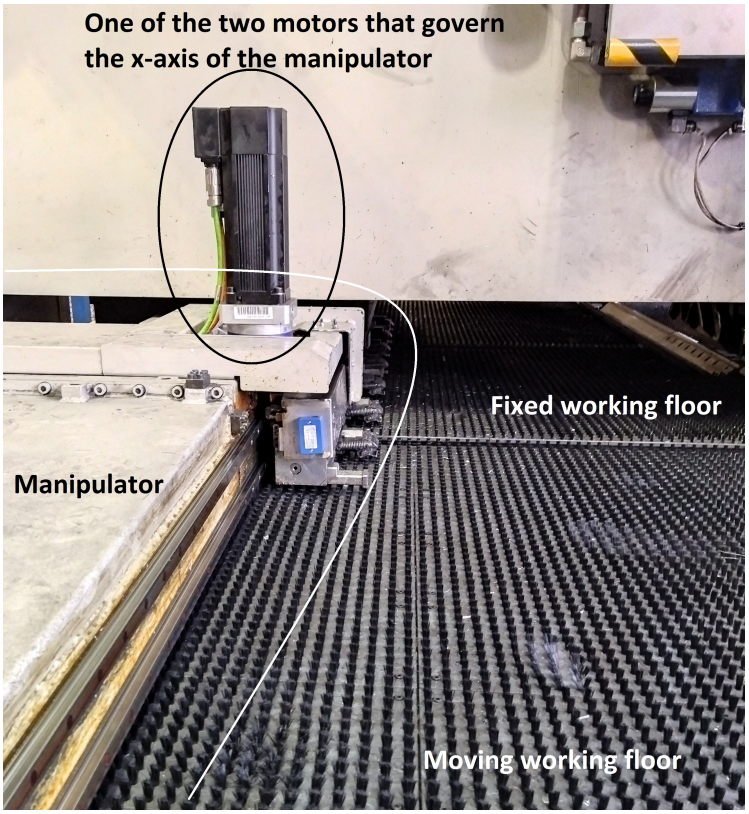


Figure 3.3: Manipulator and working floors visualization

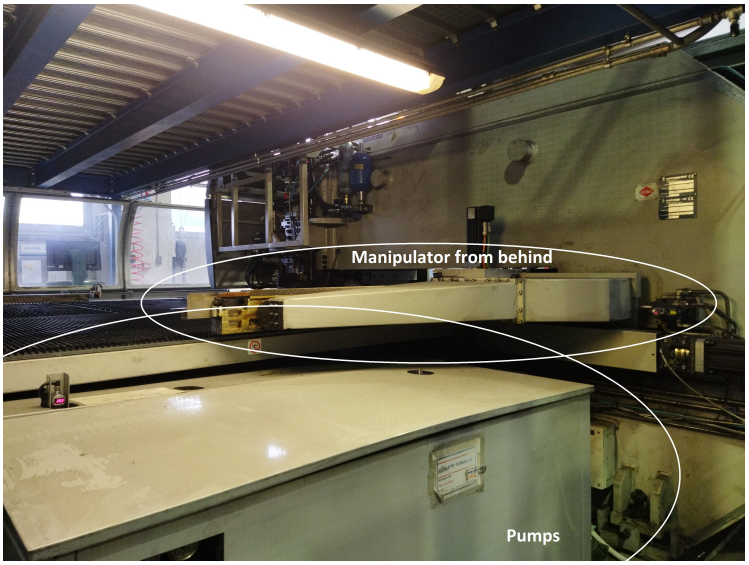


Figure 3.4: The pumping system

moves the metal sheet in the y direction to align it in that direction, too. Once the metal sheet is correctly positioned, the manipulator starts moving, and, with its clamps, grabs the metal sheet.

The metal sheet is now machined: the manipulator moves the sheet, and

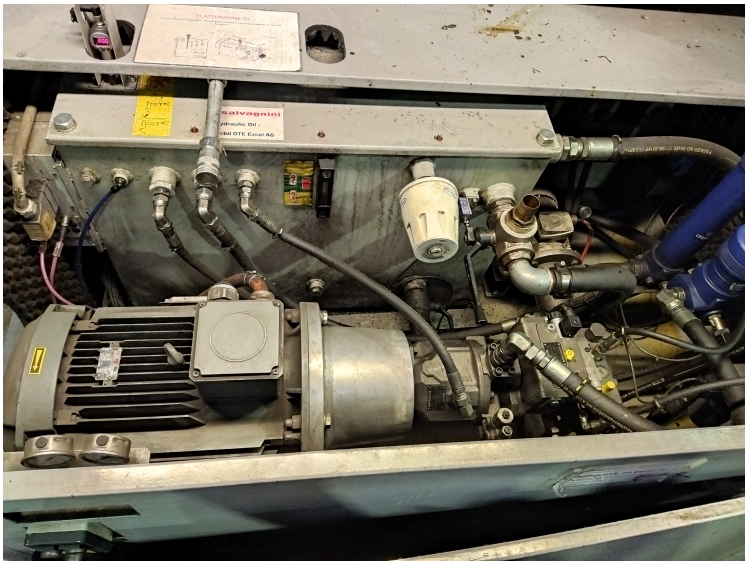


Figure 3.5: A closer look at the pumping system

the punching and shearing tools operate; the sheet is then rotated by 180 degree in order to machine it also on the side that was previously grabbed by the manipulator.

The machined metal sheet is now ready to exit the punching machine: the clamps release the metal sheet on another conveyor belt, that starts running, and let the sheet move towards the bending machine.

Now the bending process is starting, but we won't go into details with this process, because the case study is focused on the predictive maintenance applied to the punching machine.

After the bending process, the sheet is processed by a robot that sticks stiffeners to the metal sheet by means of industrial adhesives and rivets.

At the end, they are stocked into a buffer warehouse, ready to be packed and shipped to the construction site.

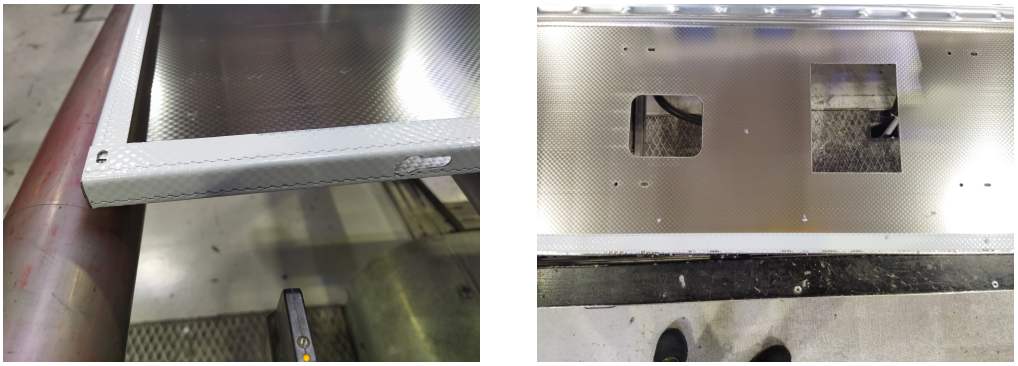
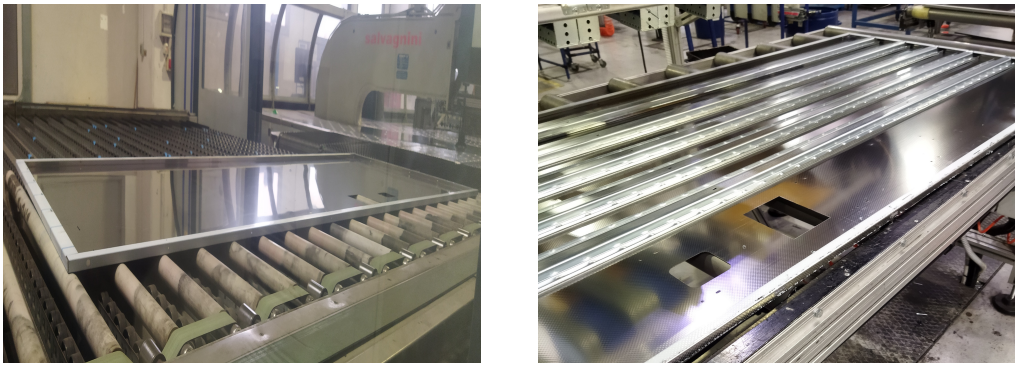


Figure 3.6: Details of holes in the punched metal sheets



(a) Metal sheet after bending (b) Metal sheet with stiffeners applied

Figure 3.7: Last operations to reach the final product

3.2 Places where sensors will be installed

In order to implement the predictive maintenance strategy, many data are needed, so some sensors need to be installed.

The selection of the sensor was quite critical, and it will be treated in detail in the chapter 4.
Anyway, the sensors chosen are accelerometers, in order to base the detection of anomalies in the machine's behaviour on the vibrations that the machine generates during its operating condition.

The decision on where to put sensors was quite difficult as well. In the beginning, we opted to install one sensor on the manipulator, in order to get data that can detect anomalies on the rack-pinion transmission, and on the four motors that control the manipulator; but, in the end, that place was inconvenient for the reasons that will be discussed in 4, so the predictive maintenance was focused on the machine head, and in particular on the shearing tool.

In addition to the accelerometer, we intend to install other sensors to check how much clean the oil is, how much lubricating grease there is on the rack, or to monitor vibrations on other critical machine components, such as the pumps.

4 Hardware components installed

4.1 Goals and constraints

The aim of the whole project is to monitor the punching machine just described. In particular, the sensor will be initially installed on the manipulator.

Unfortunately it's evident that the manipulator arm moves during the operational stage, so the first constraint appears: we need a component that is able to follow the manipulator movements, and the most suitable solution is to rely on wireless sensors that can greatly simplify the installation of the sensor, avoiding using cables and carrying them from the manipulator to the central processing unit.
For this reason, we opted for an embedded system, that can communicate by means of a Bluetooth or WiFi connection and that can take data and analyze them directly on the board.

4.2 First solution: STWIN

The first solution was identified in an embedded system from STMicroelectronics called STWIN. According to the manufacturer, this electronic board is made for industrial application.
The board is composed of the following components:

- CPU: ARM Cortex-M4 at 120MHz with 2 MB of flash memory (STM32L4R9);
- some MEMS sensors (listed further ahead);
- Bluetooth connectivity;
- optional WiFi connectivity;
- lithium battery;
- ON/OFF button and user button;

- SD card slot;
- plastic box as carter.

The sensors available inside the board are:

- ultra-wide bandwidth (up to 6 kHz), low-noise, 3-axis digital vibration sensor (IIS3DWB);
- 3D accelerometer + 3D Gyro iNEMO inertial measurement unit (ISM330DHCX) with machine learning core;
- ultra-low-power high performance MEMS motion sensor (IIS2DH);
- ultra-low-power 3-axis magnetometer (IIS2MDC);
- digital absolute pressure sensor (LPS22HH);
- relative humidity and temperature sensor (HTS221);
- low-voltage digital local temperature sensor (STTS751);
- industrial grade digital MEMS microphone (IMP34DT05);
- analog MEMS microphone with frequency response up to 80 kHz (IMP23ABSU).

The figure below (figure 4.1) shows an image of the STWIN board.

Figure 4.1: STWIN board

Despite the many sensors that are included in the board, the only sensor interested in the predictive maintenance project is the 3-axis accelerometer

and 3-axis gyroscope called *ISM330DHCX*. So we will focus only on its features.
In particular, the accelerometer has a full scale of 16g, which is equal to a full scale of almost 160 m/s^2, and a maximum sampling frequency of 6667 Hz.
It is important to notice that those features are very poor compared to those of a traditional piezo-accelerometer. In fact, a standard piezo-accelerometer can have 100g of full scale and 15 KHz of sampling frequency. In fact, the main advantage of a MEMS sensor is not its performance, but it is its low cost: it costs only a few tens of euros, while the worst piezo-accelerometer starts at a cost of a few hundred of euros.

Unfortunately, the global political context didn't help. Both the COVID pandemic and the terrible war in Ukraine have had a heavy impact on the micro-conductor industry, and the STWIN board was not available for months.
The only way to proceed with the project was to choose another electronic board available on the market.

4.3 Fallback solution: SensorTile.box

The market's constraints forced us to discard our first option: the STWIN electronic board. So, after meticulous research, we selected the most similar solution available on the market: the SensorTile.box.

The features of this electronic board can be summarized in the following list:

- CPU: ARM Cortex-M4 at 120MHz with 2 MB of flash memory (STM32L4R9);
- some MEMS sensors (listed further ahead);
- Bluetooth connectivity;
- lithium battery;
- SD card slot;
- plastic box as carter.

The sensors available inside the board are:

- Digital temperature sensor (STTS751);

- 6-axis inertial measurement unit (LSM6DSOX);
- 3-axis accelerometers (LIS2DW12 and LIS3DHH);
- 3-axis magnetometer (LIS2MDL);
- Altimeter / pressure sensor (LPS22HH);
- Microphone / audio sensor (MP23ABS1);
- Humidity sensor (HTS221).

The figure below (figure 4.2) shows an image of the SensorTile.box board.

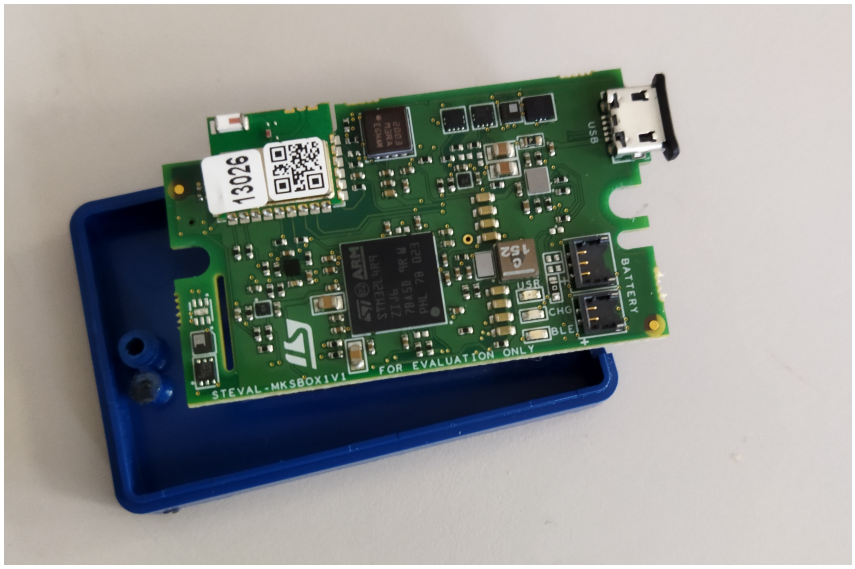


Figure 4.2: SensorTile.box board

The features of this board are comparable with those of the STWIN board; the only differences are in the absence of WiFi connectivity and the ON/OFF button.
Those differences are not substantial, so the chosen board was purchased and tested practically.

Data logging

The first step in analyzing data and implementing an anomaly detection algorithm is to log and visualize the data.

Data logging is fortunately easy with the SensorTile.box thanks to a ready-to-use firmware available on the manufacturer's website.

Data logging firmware loading

The firmware already installed on the SensorTile.box board is a default firmware that can exploit some algorithms of artificial intelligence as illustrative examples.
The Bluetooth connection is the key to visualize and control the firmware. For that purpose, an application called *ST BLE Sensor* must be installed on a smartphone.

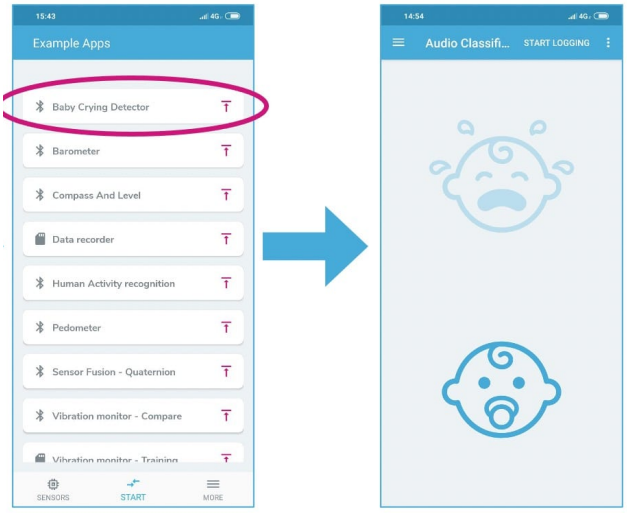


Figure 4.3: An example of pre-loaded app in the default firmware

In order to load into the electronic board the Datalog firmware, a debugger is needed, together with the PC program *STM32CubeIDE* to compile the firmware onto the SensorTile.box.

Once the *Datalog* firmware was installed in the electronic board, the board was connected to the battery, switched on, and connected to the smartphone app via Bluetooth.
Now we were able to collect data from the machine.

Data collection

The data collection was made by attaching the board to the manipulator in a rigid way, by means of self tapping screws. The figure below (figure 4.5) shows in detail how the board was connected, and the reference systems of both the manipulator and the 3-axis accelerometer, included in the electronic

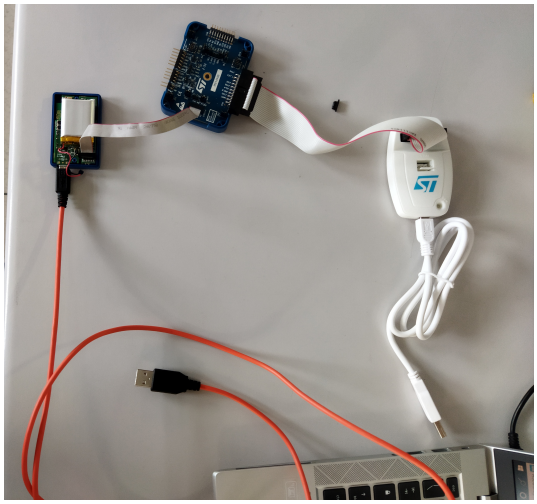


Figure 4.4: The setup of the SensorTile.box connected to the debugger

board, are shown.

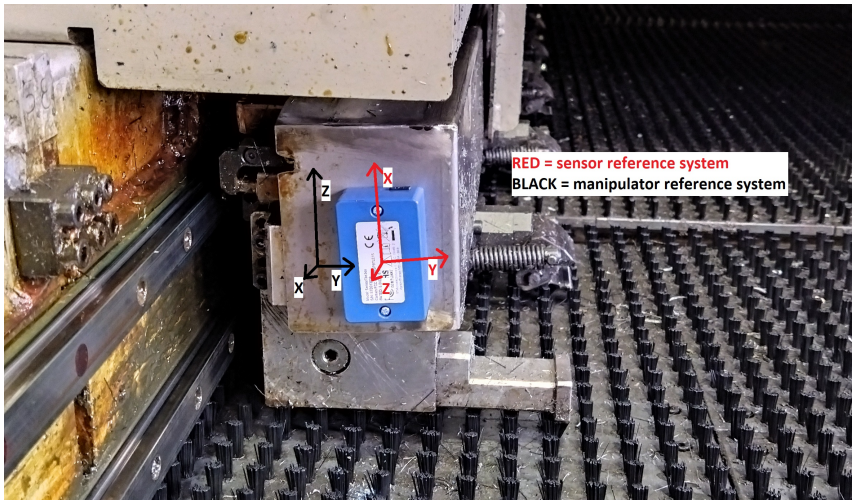


Figure 4.5: The SensorTile.box mounted on the manipulator arm

Once the sensor was mounted, the machine was switched on, and the *Datalog* firmware could start.
Notice that the features of the sensor were set as follow:

- sampling frequency = 6667 Hz;
- full scale = 16 g ~ 157 m/s^2

After the *start* button in the smartphone is clicked, the recording takes place. All the data extracted are saved in the SD card inserted in the SensorTile.box. Notice that the SD card must meet some requirements of writing speed, so

a *class10* SDHC card was used.

Now, we are ready to begin analyzing the extracted acceleration data.

Analysis of the acquired signals

The data are analyzed through jupyter notebook, that is a compiler of Python 3.

By means of the library *HSDatalog* provided by the manufacturer, it's possible to get the data from the *.csv* file saved in the SD card during the data collection.
The data that has been plotted exhibits the pattern in the figure 4.6.

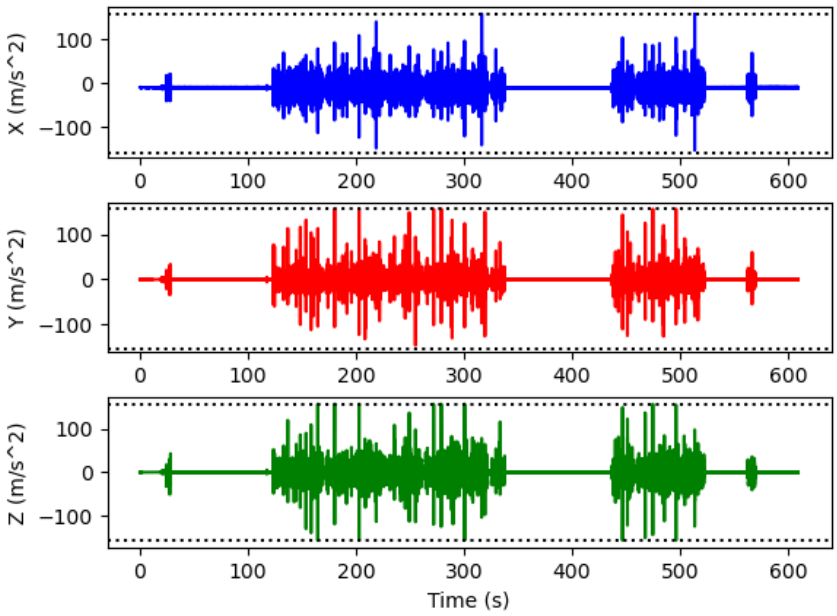


Figure 4.6: The plot of the accelerations collected

In the figure 4.6, the dashed black lines represent the maximum full scale available in the board. So every collected sample which reaches the dashed lines corresponds to a saturated point, that is not a significant point in the analysis and must be discarded.
That's actually a quite big problem due to the poor characteristics of the sensor.
But the possibility to work in wireless mode is a such big advantage that we kept going on using the SensorTile.box.

After the data are acquired, they are divided into many windows (as done in the code presented in the appendix A), and for each window a full dataset of features is extracted, as done in the section 5.3.4.
The power spectra of the acquired signal, already divided, is shown in the figure 4.7, with each line corresponding to a window of the signal.

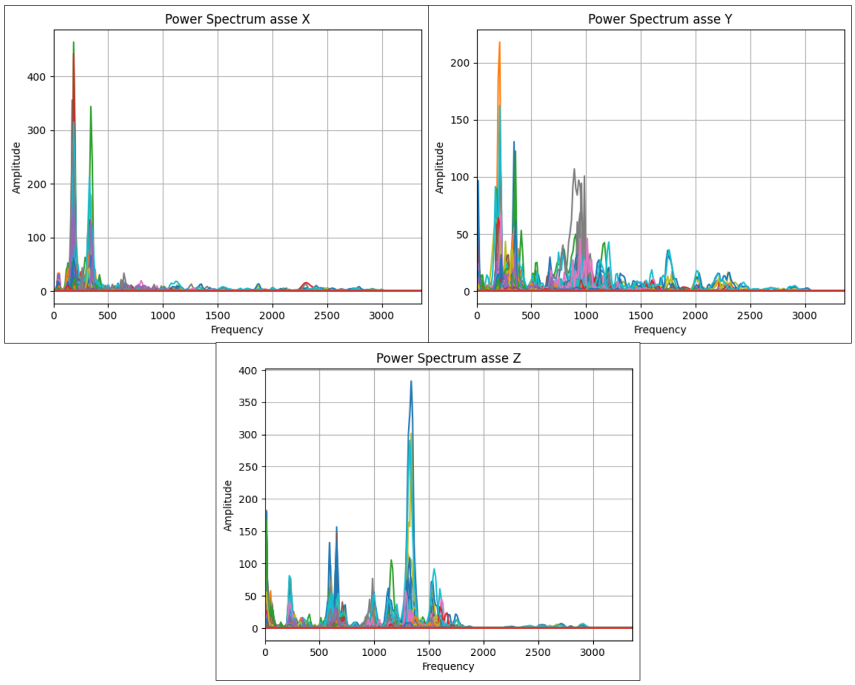


Figure 4.7: Power spectra of the three axis of the acquired acceleration signal

Therefore, the extracted features are reduced using a PCA algorithm and implemented in several clustering models in order to find relationships between the data points. The results of the PCA transformation is shown in the figure 4.8.

Unfortunately, it is difficult to detect a clear relationship between the data points, and therefore the clustering process is quite difficult to be implemented.
The only clustering that is possible to perform is between two macro-conditions of the punching machine: punching machine working on a metal sheet or punching machine still that wait for a new piece to be processed.

In fact, using a K-Means machine learning algorithm we can determine these two conditions, as it is shown in the figure 4.9.

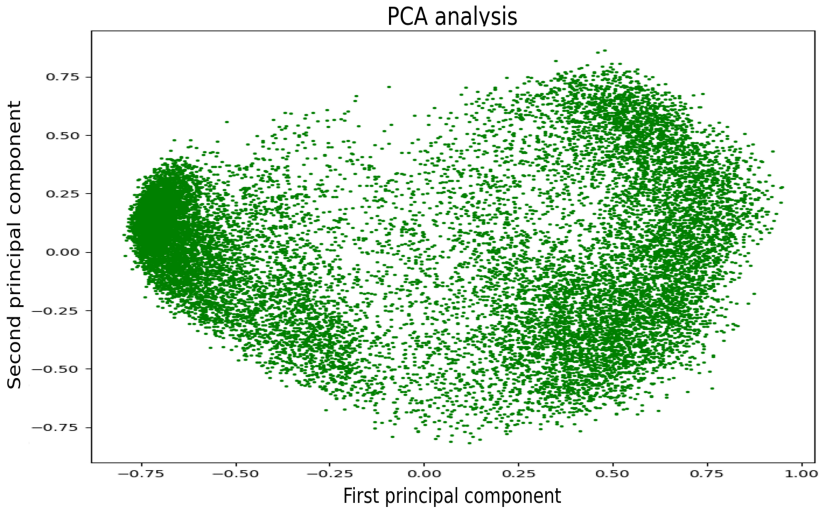


Figure 4.8: PCA analysis on the acquired signal by the SensorTile.box

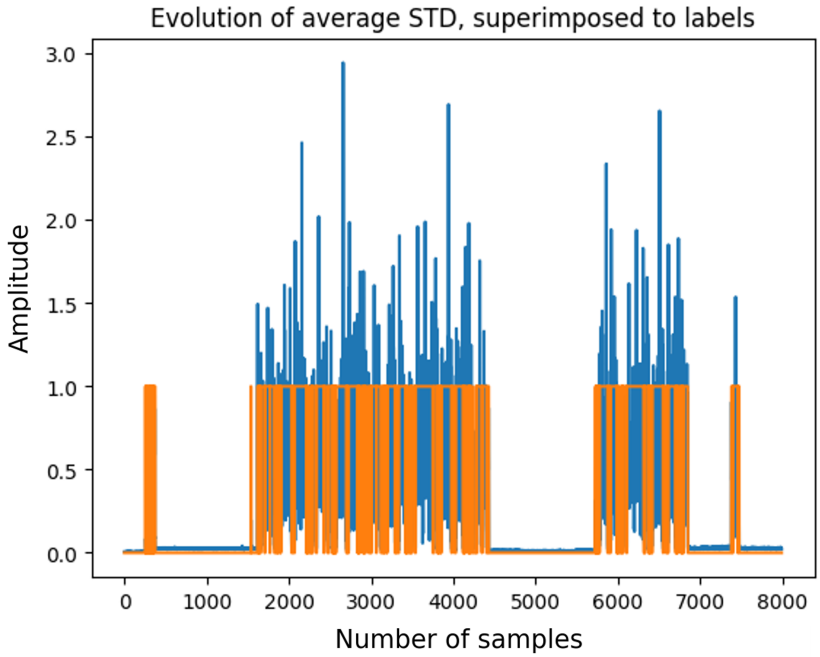


Figure 4.9: STD of a portion of the signal compared with the labels given by K-Means algorithm

In order to go on with the project and perform a machine learning algorithm on the acquired data that can define if the manipulator is working in a healthy condition or in a faulty one, we cannot rely on the clustering process, but we have to use a different machine learning algorithm. In fact, there are also some algorithms which aim is not to classify the data into groups (as the machine learning algorithms listed in the section 5.3.4), but their objective is to detect anomalies. They are called **one class machine learning algorithms**.

In particular, among the machine learning algorithms developed for anomaly detection purposes, we utilized the Elliptic Envelope algorithm and the Mahalanobis distance to the centroid of a reference acquisitions in order to define if the new acquired data are related to a healthy or faulty state of the manipulator.

Mahalanobis distance

Mahalanobis distance is a measure used to quantify the distance between a point and a distribution of points in multivariate statistical analysis.

Unlike Euclidean distance, which assumes variables are uncorrelated and have equal variances, Mahalanobis distance considers the covariance matrix of the variables, allowing for a more accurate assessment of the dissimilarity between data points. In particular, its algorithm measures how many standard deviations away a point is from the mean of the distribution, taking into account the shape and orientation of the distribution.

Elliptic Envelope

The Elliptic Envelope is an algorithm that can identify outliers in a dataset, assuming that the normal data points follow an elliptical distribution, while the outliers are placed outside that ellipse.

The algorithm works by fitting an elliptical shape to the data by means of a covariance estimation method. In particular, it estimates the center of the ellipse and its shape by considering the Mahalanobis distance, that is computed considering the correlations between the different features of the data.

Therefore, the algorithm assigns an anomaly score to each data point based on its distance from the center of the ellipse.

The limitations of this algorithm is that it is very good at recognizing outliers, but it's difficult to evaluate if the gradual wear of the components can be detected with this method. This can be a problem, because we don't have the permission or the possibility to simulate a faulty conditions, and we need to rely only on the healthy ones.
This limitation bring us to the impossibility of testing and certificating this method.

The post-processing of the acquired data using Elliptic Envelope

The idea of the predictive maintenance algorithm is to define a reference dataset that describes the distribution of the data points when the machine is in healthy condition. Therefore we can acquire other data in real-time and compare its centroid with the distribution computed by means of the reference dataset.
The complete algorithm used to define the distribution corresponding to the healthy operating condition can be divided into several pieces.

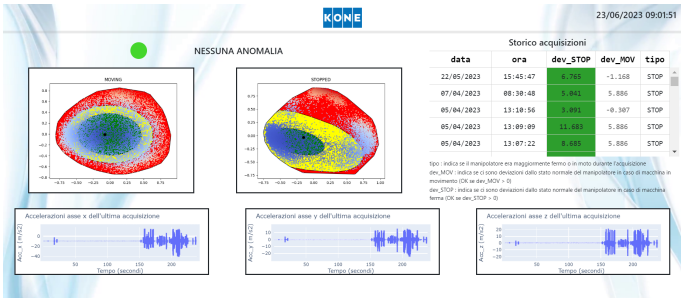
- Acquiring the data;
- cleaning the data, that is removing the corrupted data;
- extracting some statistical features (listed in the following paragraph 5.3.4);
- exploiting the PCA algorithm to reduce the features matrix dimension (a more detailed description of this algorithm will be discussed in the following paragraph 5.3.4);
- exploiting the Elliptic Envelope algorithm and define three areas (healthy, less healthy, near to failure) through the Mahalanobis distance.

Once the areas describing the level of health of the manipulator are defined, we can perform the just described algorithm on the new inference data and perform the mean of the highest Mahalanobis distances computed, in order to compare the centroid of the new incoming data with the pre-defined areas of the reference dataset.
In fact, if an anomalous condition is detected, the mean of the highest Mahalanobis distances of the inference data will be displaced far away from the centroid.

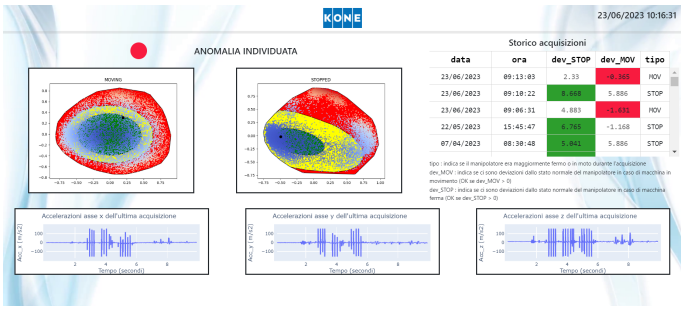
In the appendix B the complete algorithm developed in Python is shown.
The results are instead shown below (figure 4.10).

As the results shown, the generated dashboard presents distinct sections: one dedicated to the distribution figures of the reduced features through PCA algorithm; another displaying the plot of the accelerations, and a third section showing the history of the acquired signals.

In particular, the plot of the data distribution is divided in sections considering the 33%, 66% and 100% of the data that are closer to the centroid.



(a) Dashboard when machine is waiting - healthy condition is test-bench simulated



(b) Dashboard when machine is operating - anomaly is test-bench simulated

Figure 4.10: Dashboard of SensorTile.box project in conditions of waiting machine and operating machine

If the inference data is near to the centroid, the signal is considered healthy; otherwise, it indicates an anomaly.
Furthermore, it is important to notice that the algorithm is performed twice, considering both the condition of moving manipulator and of stopped manipulator. The new incoming data is then classified as either *moving* or *stopped* and it is compared with the corresponding reference distribution.

Regarding the history of the acquired data, the classified condition of the inference data is taken into consideration and only its result is highlighted: it becomes green if we detect a healthy condition, while it becomes red if a faulty condition is detected.

Specifically, we consider the faulty condition as the case in which the mean of the largest 10% of Mahalanobis distances in the inference set is higher than the mean of the largest 1% of Mahalanobis distances in the reference set.

Unfortunately this project cannot be validated, because we do not have the possibility to simulate a faulty condition in the S4 1262 punching-shearing center.

Moreover, there are additional hardware issues related to the use of the SensorTile.box, that will be discussed in detail in the next section.

4.4 The turning point in hardware definition

From the hardware point of view, some restrictions come to light, such as the requirement of a very long and strong USB cable, in order to let the board run in a cabled mode and collect the output analyzed data that declare the healthy status of the monitored components.

That problem can nullify one of the big advantages of the board, that is the cost-effectiveness. Thus, it is necessary to employ the wireless Bluetooth mode.

At this point, the battery life of the sensor need to be identified, because the application is not compatible with a device that need to be recharged too often.
In fact, since it is required to maintain constant control over the machine's health, the sensor and CPU must operate continuously.
Moreover, the battery change operation can generate some issues: the machine must be stopped first in order to have access the sensor safely, and then some minutes must be spent to dismount the sensor and change its battery.

After some research on the web about the battery life of the SensorTile.Box, it appears that the board can work continuously for a few tens of hours before requiring recharging. Unfortunately those performance levels are unacceptable for the application.

In addition to the hardware issues, it was necessary to learn from scratch the programming language of the *STM32CubeIDE*.

In the meanwhile, my colleagues suggested me an alternative solution, that is extensively used in the company: **the PLC system**, known also as Programmable Logic Controller.
In the face of the advice received, I decided to look into this option as well.

4.5 The PLC system

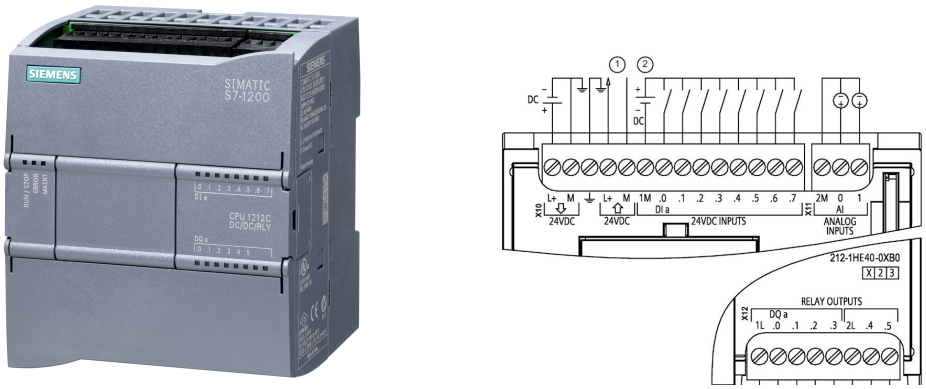
PLC stands for Programmable Logic Controller, as anticipated in the previous section, and it is an electronic device used to control the automation lines. Due to its reliability and robustness, it is quite common in companies with an high level of automation.
Moreover, the opportunity to incorporate additional modular devices makes the PLC system a great candidate for the application in the automated factory supply chain.

Up to now we have seen the most evident benefit of using this tool. However, there are also some drawbacks, such as the high cost of its components and the need to know the programming language, that is often a proprietary language.
The diffusion of this kind of tool led me to choose to investigate in further detail this system, considering also that acquiring some knowledge about this system can be maybe useful in my future projects.

In particular, the most used PLC systems in the company are the SIEMENS ones: many automatic machines or industrial robots in the company are controlled with a SIMATIC S7-1200, or SIMATIC S7-1500, or SIMATIC ET200.

4.5.1 The chosen PLC

For economic reasons I chose a *CPU 1212C DC/DC/RLY* from the SIEMENS S7-1200 series.



(a) External view (b) Wiring scheme

Figure 4.11: PLC SIEMENS S7-1200, CPU 1212C DC/DC/RLY

This CPU is compact, in order to be integrated easily into existing electrical panels, and it features 8 digital inputs, 2 analog inputs and 4 relay outputs for controlling external devices.
It is powered by a 24V DC power supply and it can be programmed using the proprietary SIEMENS IDE called TIA PORTAL over the PROFINET communication port.

In order to take the acceleration data, it's necessary to implement an accelerometer, exploiting the analog input port of the CPU.

4.5.2 The project implementation into the PLC

As already said, the idea behind the project is to define the degradation level of some machine components by analyzing the accelerometer data into the time and frequency domain. Therefore, some mathematical tools are necessary in order to extract the features of the signal and exploit them to detect anomalies or find the remaining useful life.

Unfortunately, TIA PORTAL, that is the software used to program the Siemens PLC, does not include a ready-to-use function for the Fast Fourier Transform, which is a mathematical tool at the basis of the frequency domain analysis.
So, for the sake of simplicity, it was decided to extract the accelerometer data obtained through the PLC and transmit them into a Python script, because it is a simpler and more effective tool for data analysis.

Even if the data analysis will occur into a Python script, it was necessary to program the PLC using TIA PORTAL. In particular it was needed to download the hardware configuration to the device and also a simple software program that can calibrate the input value, changing its quantity from its bit value into the physics one.
In fact, through the 10bit analog input port, the PLC can acquire a voltage or current value in form of a bit value, that is a value between 0 and $2^{10} = 1024$.

However, before starting to investigate the analog input signal, I wanted to start working with some digital input signals, that are much simpler to manipulate.

The first thing to do is to prepare the hardware configuration in TIA PORTAL and download it into the PLC.
Using the wiring scheme given by the constructor (Figure 4.12), it's possible to connect a proximity sensor for metals, that is a digital output sensor.

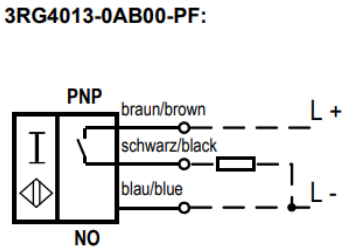


Figure 4.12: Proximity sensor scheme

Figure 4.13: Proximity sensor connection to PLC

Then it is necessary to move on the software side: the IP address of the PLC has to be set, then the PLC becomes reachable from the computer and the software can be downloaded to the device.
In order to set the IP address, we need to use a free-to-use Siemens software called *Siemens PRONETA*. In the highlighted box in figure 4.14 the IP address of the PLC can be defined.

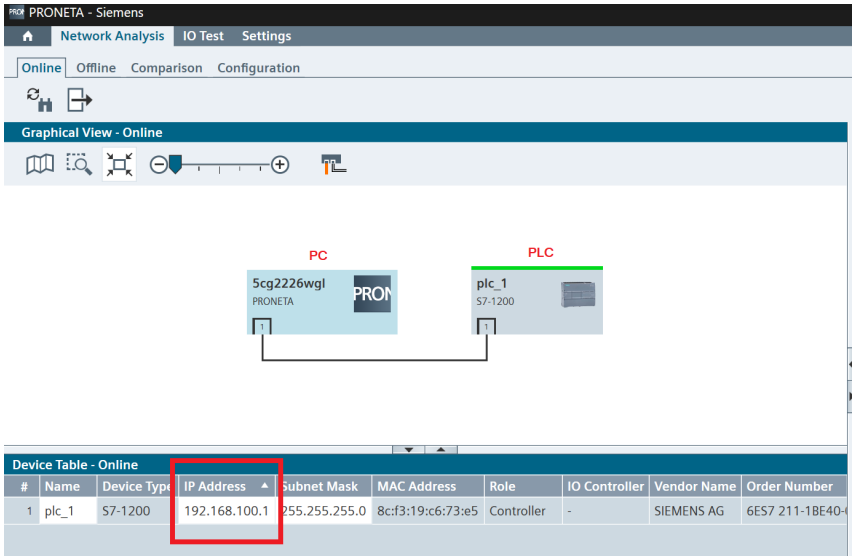


Figure 4.14: Siemens PRONETA user interface

Once the IP address is set, the PLC can be reached, and though TIA PORTAL, it's possible to go online and monitor the memory, input and output variables.
In the figure 4.15, you can see what are the results of bringing a piece of metal near to the proximity sensor.

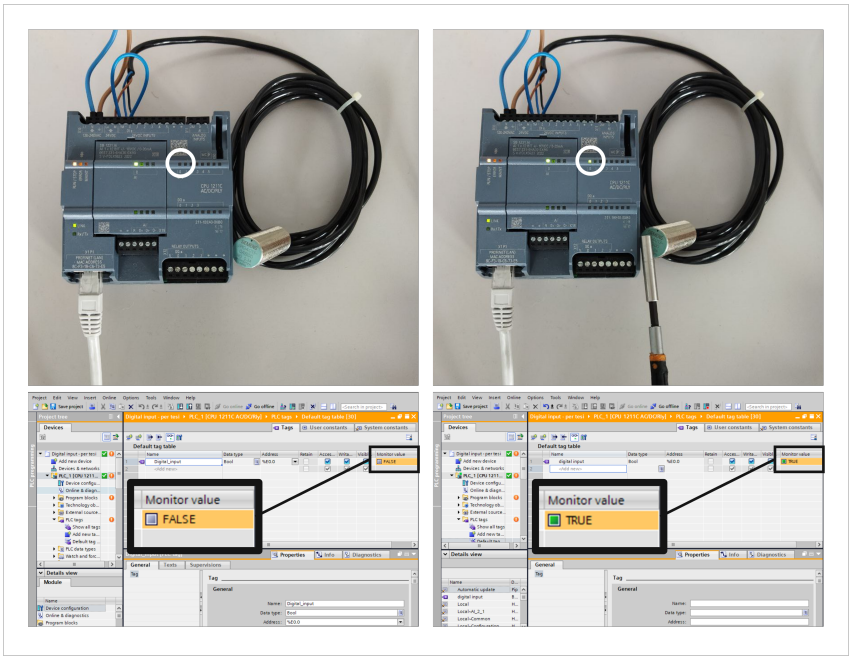


Figure 4.15: Hardware and software response to digital input stimulation

The python script to analyze the data extracted with the PLC

In order to extract data from the PLC and transmit them to the PC, a Python script is needed. In particular a library to enable the communication between the PC and the PLC is available from the web. This library is called *Snap7*, and the documentation is open for consultation at `https://snap7.sourceforge.net/`.

A sample code to connect the PLC to the PC is shown below.

Code 1

```
import snap7
import struct
IP = '192.168.100.1'
RACK = 0
SLOT = 1
plc = snap7.client.Client()
plc.connect(IP, RACK, SLOT)
```

Then, it's possible to add another piece of code to extract data from the PLC and visualize them into a webpage.
In our case, we can visualize the proximity sensor output with the piece of code described in the appendix C.
The steps of this algorithm are listed below:

- Connect with PLC S7 1200;
- Prepare server importing the necessary libraries;
- Define where to get data from, that is a database in the PLC's memory;
- Open web-page;
- Define server and dashboard on web-page.

The result is shown in the figure 4.16.

After getting some confidence with the PLC system, it's possible to switch from digital inputs to analog ones. However, some preliminary requirements must be met: in particular a good sampling frequency is needed.
So, we proceed by taking as many data as possible in a certain period of time and computing the sampling frequency.

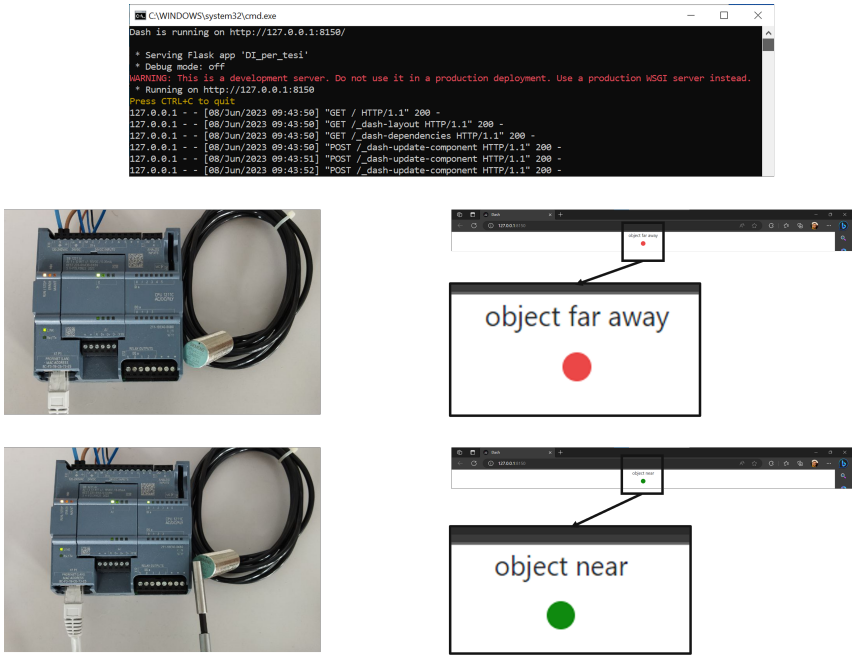


Figure 4.16: Python output for the code shown above

4.5.3 Another obstacle: the communication time

The sampling frequency is defined by two main components:

- The CPU performance of the PLC;
- The communication time between the PLC and the Python script.

For what concern the first component, it's possible to monitor the cycle time of the PLC by means of TIA PORTAL. And we can notice that the cycle time, which correspond to the updating period of the PLC variables, is around 1 millisecond (please refer to figure 4.17).

Unfortunately, getting one sample per millisecond correspond to a sampling frequency of $1/10^{-3} = 1000$ Hz. Therefore, according to the *Nyquist-Shannon theorem*, the maximum frequency of the power spectrum is equal to 500 Hz.
In fact, the Shannon-Nyquist theorem provides guidelines for accurately sampling analog signals in order to preserve all the information contained within them, and declares that a continuous-time signal can be completely and uniquely reconstructed without samples distortion or loss of information if the sampling frequency is greater than or equal to twice the maximum frequency component in the signal. In other words, the sampling rate must be greater than or equal to twice the bandwidth of the signal.

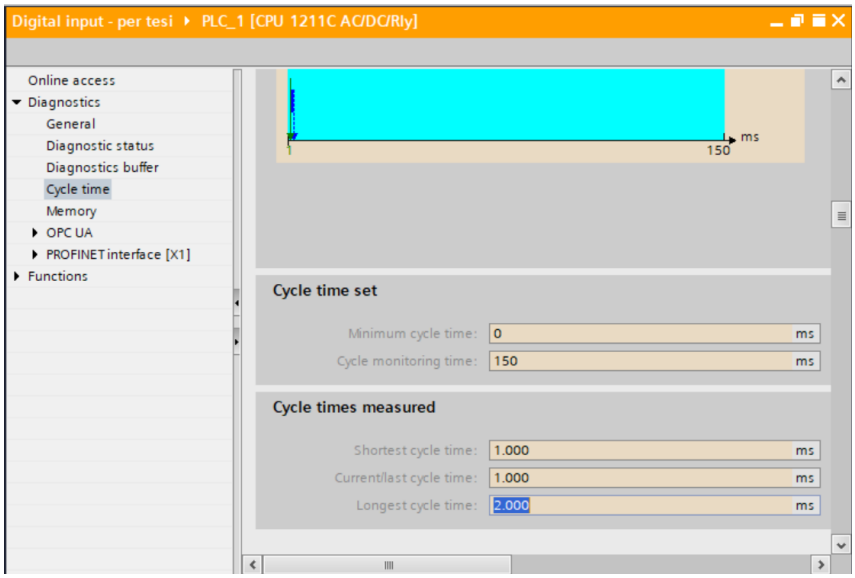


Figure 4.17: TIA Portal screen to monitor cycle time (one sample per cycle)

Thus, the performance of the PLC are not enough for our application, that need to analyze accelerations with frequency components up to 1200Hz (figure 4.7).

In order to optimize the performance of the PLC, it's possible to stack the incoming input values into an array with few lines of code using SCL programming language. In the figure 4.18 the PLC code is shown.

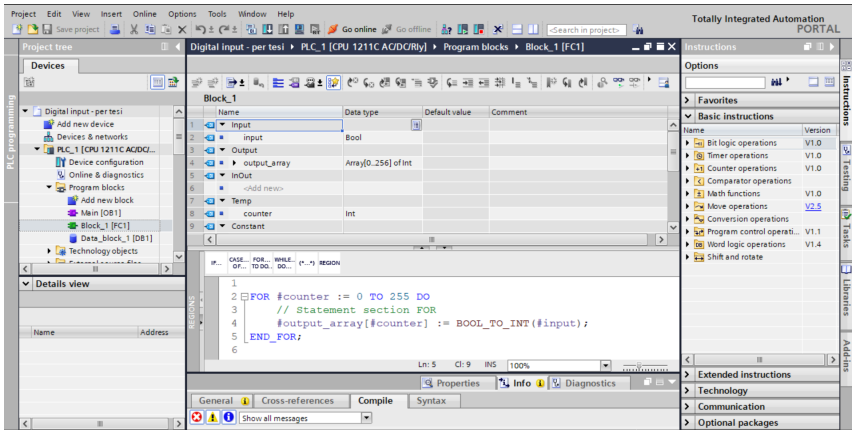


Figure 4.18: SCL function that stacks input signal into an array of 256 samples)

The CPU performance is not that different: in fact the mean cycle time is again 1 millisecond (Figure 4.19).

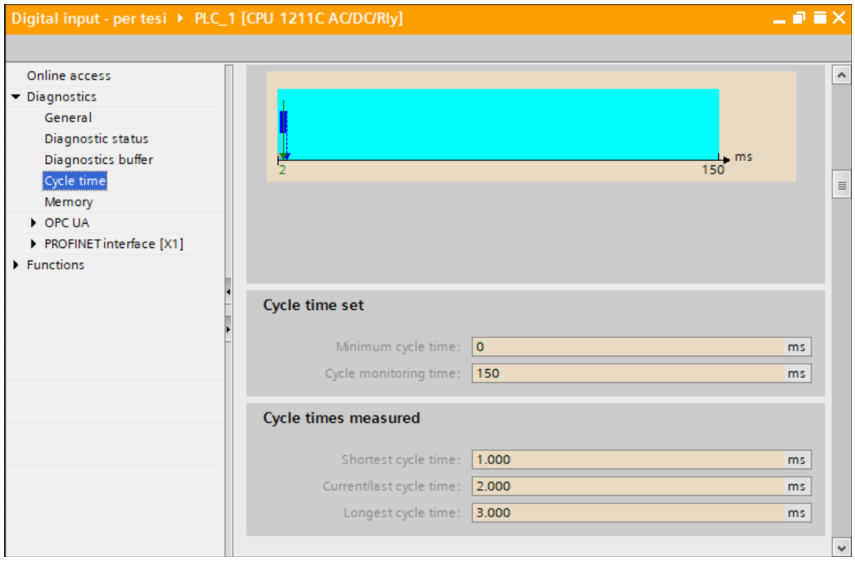


Figure 4.19: TIA Portal screen to monitor cycle time (256 samples per cycle)

Now the sampling frequency is equal to approximately $256/(2 \cdot 10^{-3}) = 128000$ Hz, that is higher than the one of the SensorTile.box and certainly more than enough for our application.

Once the first issue is solved, we can move on and evaluate if the communication time between the PLC and the Python script can raise some problems.

The Python script that connects with the PLC and extract the array of 256 samples is shown below:

Code 2

```
#### CONNECT WITH PLC S7 1200 ####
import snap7
import struct
from datetime import datetime
IP = '192.168.100.1'
RACK = 0
SLOT = 1
DB_NUMBER = 1
START_ADDRESS = 0
SIZE = 514
plc = snap7.client.Client()
```

```
plc.connect(IP, RACK, SLOT)
time_in = datetime.timestamp( datetime.now() )
db = plc.db_read(DB_NUMBER, START_ADDRESS, SIZE)
DI = []
for i in range( int(SIZE/2)-1 ):
     DI.append( int.from_bytes(db[0:2], "big") )
time_out = datetime.timestamp( datetime.now() )
```

The communication time in order to extract 256 samples from the PLC is of about 33 milliseconds, as it's shown in the results in the figure 4.20. So the sampling frequency is about $256/(33 \cdot 10^{-3}) \sim 7760$ Hz.

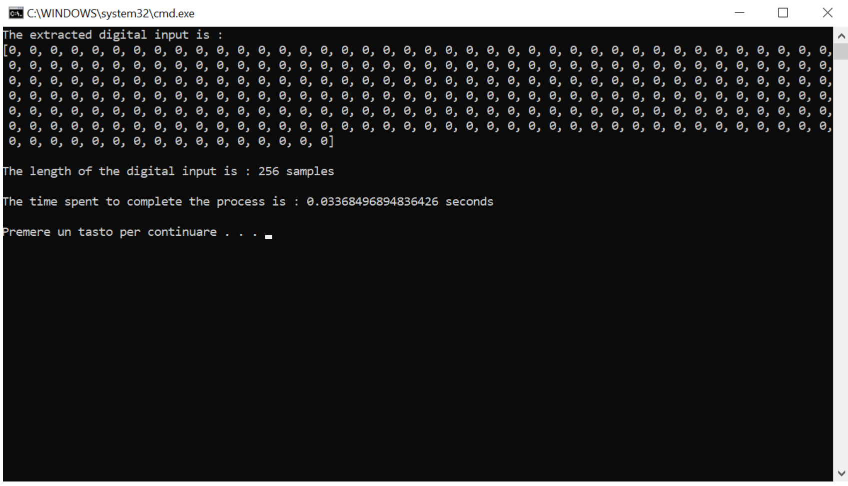


Figure 4.20: Python output for the piece of code shown above

Up to now, the performance of the overall system is quite good. But things get worse when we try to run the python script to acquire more than one array, because it get some time between one Python call and the following one.
In particular, we can plot the timestamp of the PLC, got with the python script in a similar way to what was done for the digital input, and the results shows that many data were lost during the Python call to the PLC, as shown in the figure 4.21.

Moreover, we need to consider that the signal evaluate till now was a digital one, so lighter than the real acceleration value that we need to acquire.

After these considerations, the overall performance is not sufficient in order to collect a representative timeserie of the actual phenomenon.

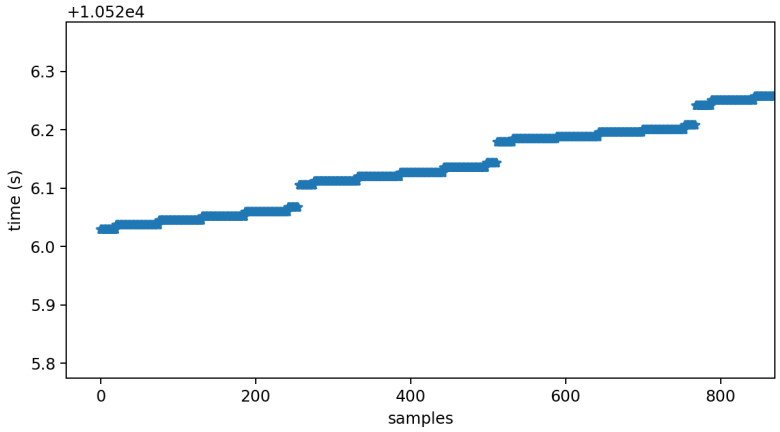


Figure 4.21: Timestamp of the PLC, extracted with the Python script

Unfortunately, even contacting the SIEMENS customer service, it was no solution for this issue.
Therefore, the PLC system is not suitable for our application, and it was discarded.

The only solution to proceed with the predictive maintenance project was to take a step backward and focus again on the ST Microelectronics board.

4.6 The definitive solution

While we are working on the PLC, a new version of the STWIN comes out on the market: the STWIN.box.
A picture of this board is shown at figure 4.22.

Its main features are the following ones:

- CPU: ARM Cortex-M33 at 160MHz with 2 MB of flash memory (STM32U585AI);
- many MEMS sensors:
  - ultra-wide bandwidth (up to 6 kHz), low-noise, 3-axis digital vibration sensor (IIS3DWB);
  - 3D accelerometer + 3D Gyro iNEMO inertial measurement unit (ISM330DHCX) with machine learning core;
  - ultra-low-power high performance MEMS motion sensor (IIS2DH);
  - ultra-low-power 3-axis magnetometer (IIS2MDC);

Figure 4.22: STWIN.box

  - high-resolution, low-power, 2-axis digital inclinometer (IIS2ICLX) with machine learning core;
  - 4 bar absolute digital output barometer (ILPS22QS);
  - relative humidity and temperature sensor (HTS221);
  - low-voltage,0.5°C accuracy temperature sensor (STTS22H);
  - industrial grade digital MEMS microphone (IMP34DT05);
  - analog MEMS microphone with frequency response up to 80 kHz (IMP23ABSU);

- Bluetooth, WiFi and NFC connectivity;
- lithium battery;
- ON/OFF button and user button;
- SD card slot;
- expandable via a 34-pin FPC connector.

The performance of the *ISM330DHCX* sensor are listed below:

- sampling frequency = 6667 Hz;
- full scale = 16 g ~ 157 m/s^2

The WiFi connectivity can be exploited to send the data read at the manipulator and classified on the edge as nominal or faulty to a database or dashboard.
However the analysis of the manipulator movements and vibrations can still raise some problems about the battery charging.
Moreover, it is better to avoid fixing the STWIN.box to the manipulator because the board is quite big with respect to the attachment area and would have been fixed to a housing connected to the manipulator. Since the housing is just a small sheet of metal, there is a chance that we monitor the housing behaviour rather than the manipulator one.

In order to avoid this quite serious issues, we decided to analyze a different machine component, that is as critical as the manipulator for a correct machine behaviour: the machine head.

4.6.1 The machine head

The heart of the Salvagnini S4 is a multi-press head placed at the center of the machine (figure 4.23). It has up to 96 tools that operates individually and are always available.

The head can accommodate many punching tools, one shearing tool and one rotator clamp.
An image of the head from below is shown in the figure 4.24 and the layout of the operating head is presented in the figure 4.25.

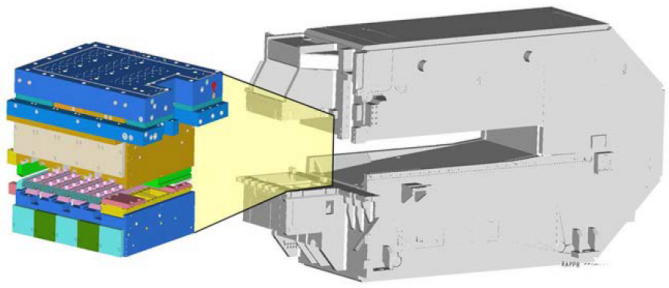


Figure 4.23: S4 operating head position

The aim of the project became to monitor the shearing process and define whether the shearing tool is working fine or not. The use of an accelerometer for that purpose can be useful to analyze the machine head structure

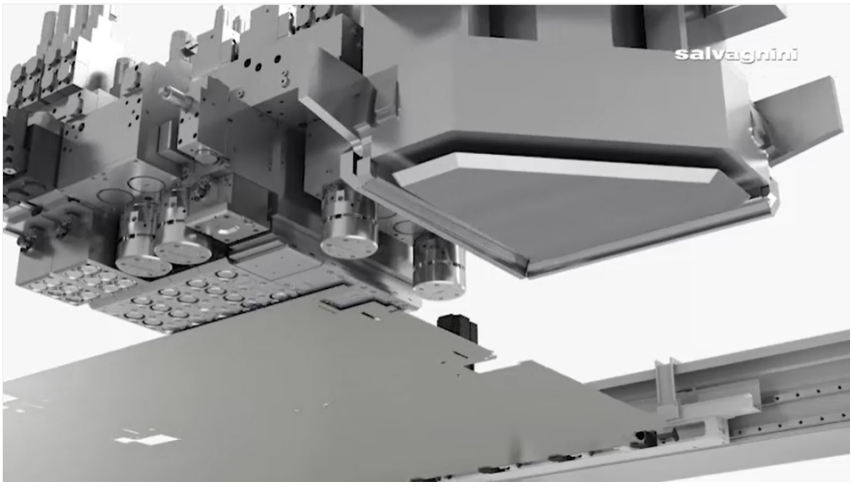


Figure 4.24: S4 operating head from below

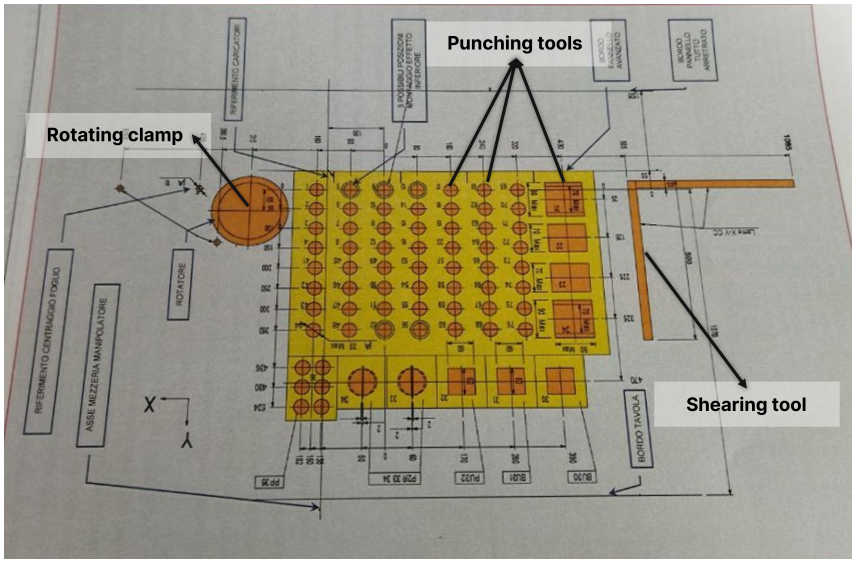


Figure 4.25: S4 operating head layout

response to the shear, that produces an impulse excitation, that is a sudden and short-duration force to the structures, leading to their subsequent vibrational response. By analyzing the acceleration amplitudes, we can observe how the structures interact with the impulse and identify any variations in their dynamic behavior.

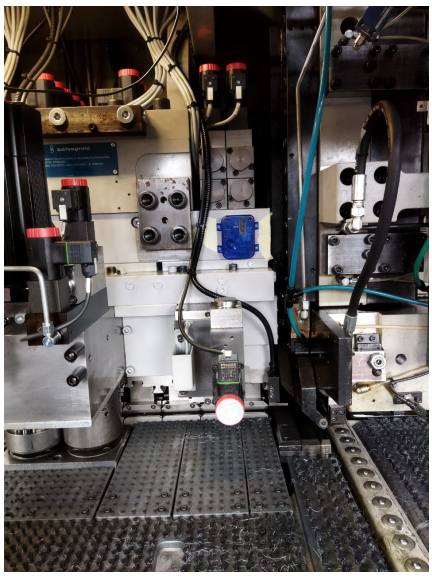
In fact, the wear of the shearing tool can affect the power spectrum of the accelerations: its cutting edges may become deformed, leading to changes in the cutting process, or its geometry and surface condition may change, causing fluctuations in the forces and vibrations transmitted to the surrounding structures.

From the predictive maintenance point of view, the shearing tool is a quite critical machine component, because the manufacturer guarantee that its lifespan is at least of 1 000 000 shears, that can be equivalently expressed as approximately one year of work, but it can be useful to schedule the maintenance operation just before failure happens due to the high cost of the component and to its high changing time.
In fact, exploiting the shearing tool at its maximum capability, we can save money directly (less shearing tools used over some years) and indirectly (minimizing the downtime of the punching machine for tool changing and optimizing the scheduling to reduce production losses).

In order to analyze the machine vibrations, the STWIN.box was installed in two different point of the machine head structure by means of a strong double-sided tape.

Then the two different acquisitions were compared and the best sensor position was defined.
In particular, the sensor was installed over the machine head (figure 4.26a), and on one of its sides (figure 4.26b).

(a) STWIN.box over the head



(b) STWIN.box on one side of the head

Figure 4.26: The different installation points of the STWIN.box

In order to acquire the accelerometer data from the STWIN.box board a datalogging firmware must be compiled into the board.
Fortunately there is a ready-to-use version available on the ST website, called *FP-SNS-DATALOG2*.

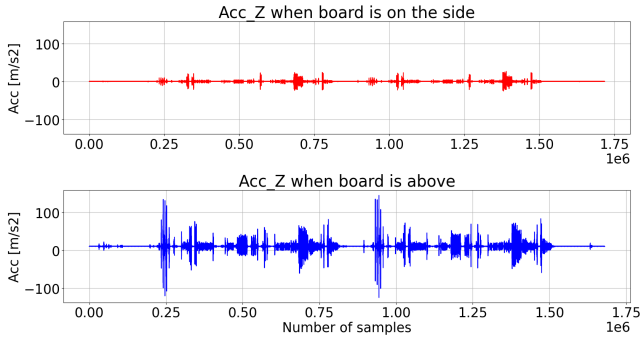
After downloading the firmware into the ST board, we can acquire the data directly into a computer, using a USB cable to connect physically with the board and launching the Python script shown in the appendix D.

The output of the script is a list of *.dat* files containing the data acquisition of all the active sensors.
In order to decrypt and analyze them, the library HSDatalog, provided by ST, can be exploited.
In particular we can have visualize the plot (figure 4.27a) using the following simple Python script :

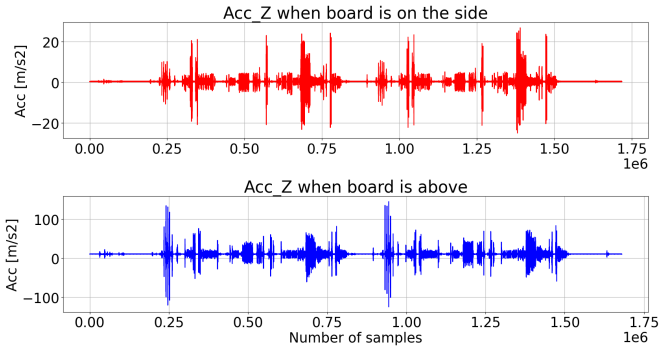
Code 3 : comparing different sensor positions

```
from HSD.HSDatalog import HSDatalog

def upload_acquisition(acq_folder):
    hsd = HSDatalog().create_hsd(acq_folder)
    sensor = HSDatalog().get_sensor(hsd, "ism330dhcx_acc")
    acc_df = HSDatalog().get_dataframe(hsd, sensor)[0] * g
    acc_df['Time'] = np.array(acc_df)[:,0] / g
    return acc_df

# upload the first acquisition (no-load cycles)
acq_folder = "..\\no-load cycles - side"
df_1 = upload_acquisition( acq_folder )
# upload the second acquisition (no-load cycles)
acq_folder = "...\\no-load cycles - above"
df_2 = upload_acquisition( acq_folder )
```

It seems that the magnitude of the accelerations is higher when the data are collected from above; nevertheless, if we don't use the `sharey` property in the plot, the differences in the shapes of the two collected acceleration data are quite few.



(a) sharey = True



(b) sharey = False

Figure 4.27: Comparison between different STWIN.box positions

Moreover, as it's better explained in the section 5.3.2, the collected data must be filtered in order to highlight the shears and to perform a peak recognition process to find and isolate them.

The piece of code (code 4) and the plot of the filtered signals (figure 4.28) are shown as follow.
Notice that the functions in the code will be declared and described in the chapter 5.
Notice also that some data in the `df_2` dataframe are corrupted, so in the code we have to discard them: a better explanation of this phenomenon will be shown in the chapter 5.

Code 4 : comparing different sensor positions using filtered data

```
def fsampfinder(df_ACC):
    ...
    return fsamp
def filter_mine(df_ACC, fsamp):
```

```
    ...
    return filtered_acc_z
df_2 = df_2.iloc[0:1680000] # remove corrupted data
fsamp_1 = fsampfinder(df_1)
fsamp_2 = fsampfinder(df_2)
filtered_1 = filter_mine(df_1, fsamp_1)
filtered_2 = filter_mine(df_2, fsamp_2)
```

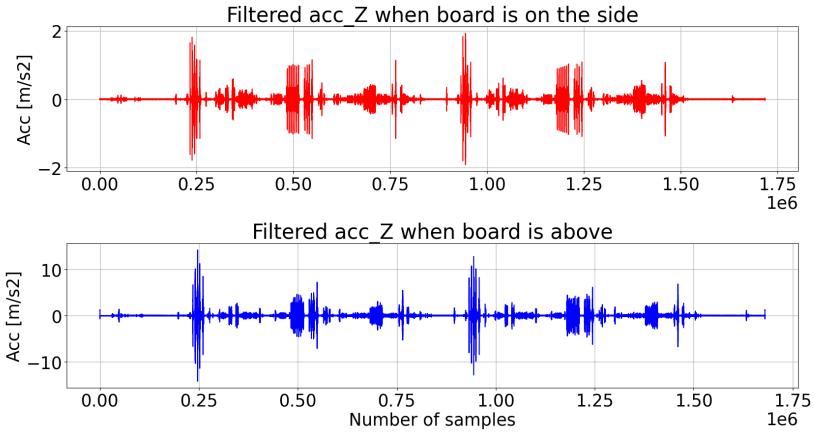


Figure 4.28: Filtered acceleration data from above and from the side

So, from the filtered data, we can see that the shears are better recognizable when the sensor is set on one side of the head.
Moreover, that position is convenient also from a maintenance point of view, because is more at hand.
In the end, it's better to set the sensor in that position also because the microphone can better acquire the sound of the shears.

Now that the hardware part is completely defined, we can concentrate on the software part, where the aim becomes to recognize the shears and find a way to monitor the health of the shearing tool.

5 Design and Implementation

5.1 The acquisitions collected by means of STWIN.box

As it was explained in the previous chapter (chapter 4), the best position for the STWIN.box is on the side of the operating head of the Salvagnini S4 machine.

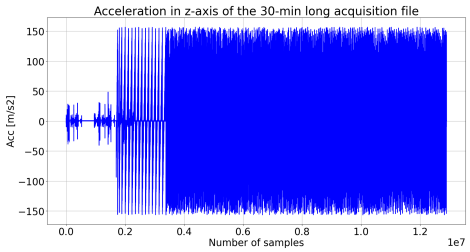
The aim of the project is to identify the shears and distinguish them from the other acceleration data, therefore the health of the shearing tool can be determined. Therefore some data have to be recorded.

The most important thing in data analysis is the quality of the data itself, and we need to ensure that the sensor is put in a position to work at best.
So, the screws that fixes the plastic case of the ST board are fixed very well, and the board was stuck at the well cleaned head surface by means of a strong industrial double-sided tape.
From that moment on, the screws were never removed or loosen by us.

The fixing of the screws is crucial for defining the quality of the extracted data, because since the sensor is installed over a plastic board, we can easily get inaccurate data acquiring the board oscillations more than the actual oscillations made by the structure of the punching head.

Another crucial aspect for data analysis is obviously the quantity of data collected. So, the first thing to do is to acquire as many data as possible.
In particular, the first idea was to leave the PC running with the Python script presented in the code 8 and modify only the line `time.sleep(240)` into `time.sleep(1800)` in order to let the acquisition run for 30 minutes straight. Unfortunately, the board generated a *runtime_error*, and all the acquired data above 4 minutes of recording were corrupted (as you can see in the figure 5.1a).

So the only way to acquire the data is to use exactly the same piece of code shown in code 32 in the appendix D and create many *.dat* files, that will



(a) Acceleration in z-axis of the 30-min acquisition

```
In [7]: df_1
Out[7]:
                    Time  A_x [m/s2]  A_y [m/s2]  A_z [m/s2]
0           1.790000e-01   -9.600004   -0.167498    0.244068
1           1.791396e-01   -9.700503   -0.172283    0.177069
2           1.792793e-01   -9.547362   -0.263210    0.167498
3           1.794189e-01   -9.600004   -0.205783    0.200997
4           1.795585e-01   -9.657432   -0.124427    0.162712
...                  ...         ...         ...         ...
12902995 -2.536565e+270   -9.662217   -0.224925    0.267996
12902996 -2.531150e+270   -9.714860   -0.267996    0.267996
12902997 -2.525735e+270   -9.724431   -0.224925    0.272782
12902998 -2.520320e+270   -9.743573   -0.253639    0.272782
12902999 -2.514905e+270   -9.710074   -0.301496    0.229711

[12903000 rows x 4 columns]
```

(b) Dataframe from the 30-min acquisition

Figure 5.1: The results of the 30-min acquisition

need to be uploaded to the analysis Python script one at a time.

In the end, using the 4 minutes acquisition script, many acceleration data are collected:

- n. **2** *.dat* files of no-duty cycles with the sensor positioned above the operating head;
- n. **2** *.dat* files of no-duty cycles with the sensor positioned on the side of the operating head;
- n. **30** *.dat* files of duty cycles with the sensor positioned on the side of the operating head, and shearing tools used for about 250'000 times;
- n. **8** *.dat* files of duty cycles with the sensor positioned on the side of the operating head of a newer Salvagnini punching machine with shearing tools used for about 1'000'000 times, that is the expected end of life by the manufacturer;
- n. **6** *.dat* files of duty cycles with the sensor positioned on the side of the operating head, and shearing tools used for about 450'000 times;

Notice that only the 2 acquisition files of no-duty cycles with the sensor on the side of the head and the 30 acquisition of duty cycles are considered from now on. The 2 acquisitions with the sensor above the head are discarded due to the considerations given in the chapter 4; the other acquisitions will be instead analyzed in the following sections.

The uploading script is similar to the script in code 3 but also the data acquired by the *imp34dt05_mic* microphone will be uploaded in order to find the shears starting from the audio, as it will be explained more in detail in

the next section.

In order to simplify the computations in the Python analysis script, it's possible to keep the accelerations over only one of the three axis: the most effective one is the z-axis as it's noticeable from the figure 5.2 that compares the accelerations over the three axis collected in the first no-duty cycle acquisition file.

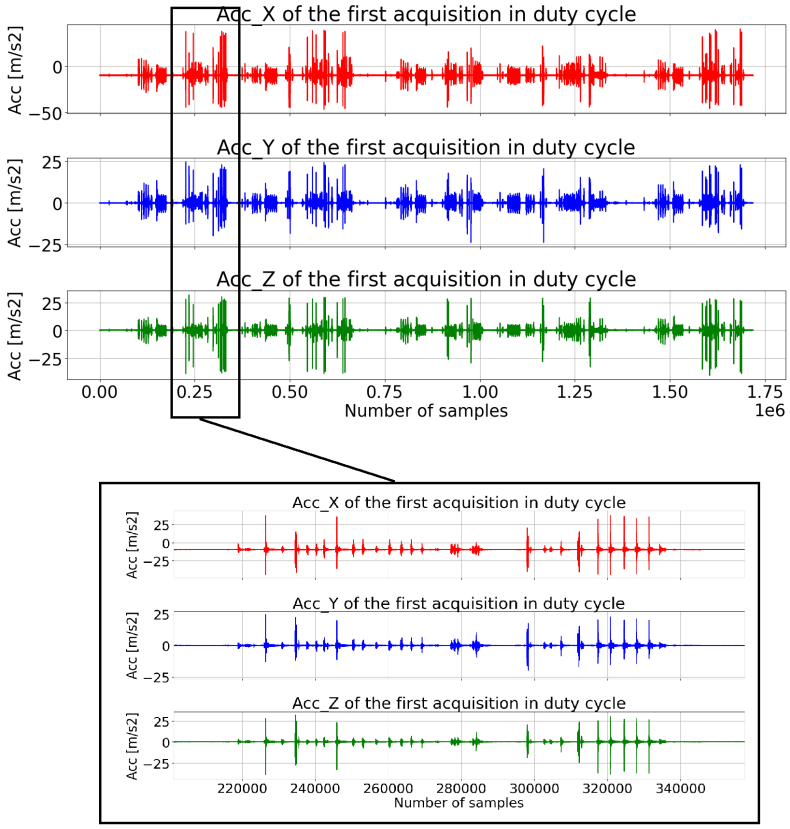


Figure 5.2: Comparison between acceleration in x-axis, y-axis and z-axis

Since no big differences are visible, we can choose an arbitrary axis to work with, and the selection goes to the z-axis, that is the one outgoing from the STWIN.box.

Moreover, we can notice that the maximum acceleration amplitude is around $25 m/s^2$ and it is way below the maximum amplitude of the MEMS accelerometer, that is equal to $16 \cdot g \sim 157 m/s^2$.

5.2 Shears recognition using the microphone acquisitions

As anticipated in the previous section, the first attempt to recognise the shears was made exploiting also the microphone data.

The used microphone included in the STWIN.box board is called *IMP34DT05*, that is a MEMS omnidirectional digital microphone for industrial applications.

In the figure 5.3 the plot of the microphone and acceleration data of only one acquisition file of a duty cycle operation are shown.

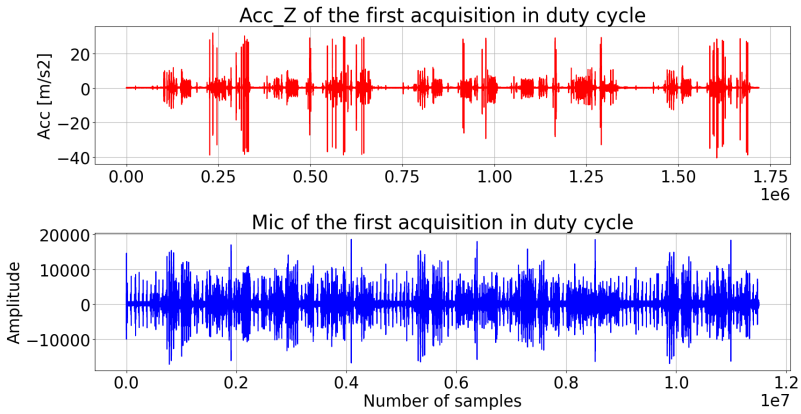


Figure 5.3: Comparison between acceleration in z-axis and microphone signal

Unfortunately, data corresponding to the loudest audio, so to the shears, are not so different from the other and we cannot distinguish the shears using the raw microphone signal directly.

Therefore, a filtering process on the microphone is needed, in order to highlight the more noisy shearing signals.
A low-pass filter that keeps only the low frequency components can be implemented on the microphone signal: in particular a 2000Hz low-pass filter is used.
Below the Python script (code 5) to implement the filter and the figure (figure 5.4) that compares accelerations in z-axis and filtered microphone signal are shown.

Code 5 : filtering the microphone acquisition

```
from scipy import signal
cutoff_freq_MIC = 2000.0 #Hz
filter_order_MIC = 2
peak_timeserie_MIC = mic_26.T.to_numpy()[1]
b_MIC, a_MIC = signal.butter(filter_order_MIC,
    cutoff_freq_MIC, 'lowpass', fs=fsamp_MIC, analog=False)
filt_timeserie_MIC = signal.filtfilt(b_MIC, a_MIC,
    peak_timeserie_MIC, method="gust") #Low Pass Filtered
```

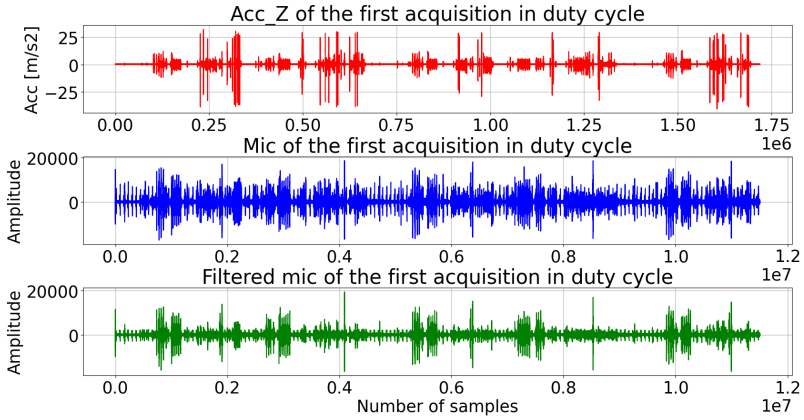


Figure 5.4: Comparison between acceleration in z-axis and filtered microphone signal

Looking at the filtered microphone acquisition the shears are quite evident, so this path can be the correct one in order to detect them.

Unfortunately, the sampling frequency of the accelerometer and of the microphone are quite different, as highlighted by the following Python function (code 6).

Code 6 : sampling frequency of the duty cycle acquisitions

```
mic_list = [mic_1, mic_2, ...  , mic_31, mic_32]
df_ISM330DHCX_MIC = pd.concat(mic_list, axis=0,
    ignore_index=True)
df_list = [df_1, df_2, ...  , df_31, df_32]
df_ISM330DHCX_ACC = pd.concat(df_list, axis=0,
    ignore_index=True)
```

```
def fsampfinder(df_ACC):
    n_samples = df_ACC.shape[0]
    time = np.matrix( df_ACC )[:,0]
    t_max = time[n_samples-1]
    t_min = time[0]
    dt = (t_max-t_min)/n_samples
    fsamp = 1/dt
    return fsamp

fsamp_acc = fsampfinder(df_ISM330DHCX_ACC)
fsamp_mic = fsampfinder(df_ISM330DHCX_MIC)
```

It results that $fsamp\_acc$ = 7166.197 Hz and $fsamp\_mic$ = 47963.387 Hz.

In the end, the concept of using the microphone data consists in identifying shears through microphone acquisition and subsequently transfer the interval of samples containing the shear to the acceleration signal; so that it can be analyzed in the frequency domain of the accelerometer acquisition.
The problem is that the interval of samples considered in the audio signal must be scaled in order to be comparable with the samples in the acceleration signal.
In practice, if the sampling frequency of the audio acquisition is higher, it results in a larger interval containing the shear compared to the corresponding interval in the acceleration acquisition.

Before entering more in the detail about signal scaling, we need to recognize the microphone peaks that correspond to the shears: in order to do that, the Python library *scipy* is exploited, and in particular the function *scipy.signal.find_peaks*.

Additionally, considering that the final objective is to work with the code in real time, the script was written to simulate the real-time behaviour of the ST board.
It is important to note that a more detailed examination of this phenomenon will be addressed in the following sections (section 5.3.1).

The Python script to emulate the real-time microphone acquisition consists of few steps, which are described below.

- Filter the microphone signal. The piece of code is shown in code 35 in the appendix E;

- Define the window and overlap dimensions;

Code 7

```
overlap = 512
win_dim = 2048
```

The simulated real-time signal consists of a series of 2048 samples long buckets that has an overlap of 512 samples with the subsequent ones.
Please be aware that these values are specific to the acceleration signal and need to be adjusted in the microphone signal because of the different sampling frequency.

- Relate the window dimension of the microphone with the acceleration one;

Using the variables defined in the code 7, it's possible to divide the acceleration signal into *len_df* = 30773 buckets, that simulate the real-time operating conditions.

It results that the microphone need *win_dim_mic* = 13710 and *overlap_mic* = 3428 to make the audio signal comparable with the accelerometer one, which has *win_dim* = 2048 and *overlap* = 512.

- Define the dimensions and the number of the microphone buckets to analyze. The piece of code is shown in codes 37 and 38 in the appendix E;

It results that the number of the necessary buckets is *len_df_for_iterations* = 30773, that is the same as *len_df* as expected.

- Perform the real-time peak recognition of the pre-filtered microphone signal. The piece of code is shown in code 39 in the appendix E;

The results for a specific segment of the signal containing shear occurrences are shown in the figure below (figure 5.5).

Given that peak recognition was conducted on the microphone signal using a bucket of samples with a length of *win_dim*, and that the algorithm recognize only if there is a peak within the bucket or there is not, it's possible to subsequently reduce the dimension of the bucket only for the *peaks_list_mic* in order to make the list dimension equal to the one of the accelerometer acquisition.
In fact, the resulting *peaks_list_divided_mic* is a list of arrays: the length of the

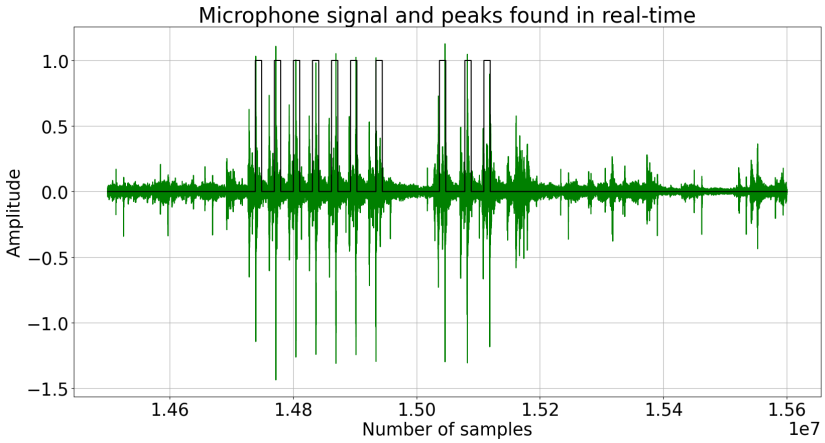


Figure 5.5: Peaks recognized by the real-time algorithm over the microphone signal

list is 30773 array, each long 13710 units. The array can contain only zeros or ones as defined by the algorithm.
So, we can reduce the array dimension simply exploiting the following script (code 8).

Code 8

```
peaks_mic1 = peaks_list_divided_mic[0][:win_dim] # reduce
    dimension of first array only
peaks_mic2 = [x[:win_dim-overlap] for x in
    peaks_list_divided_mic[1:]]  # reduce the other arrays
peaks_mic2.insert(0, peaks_mic1) # concatenate
peaks_mic_to_acc = np.hstack(peaks_mic2)
```

Now the *peaks_mic_to_acc* list consists of 30 773 arrays, each with a length of *win_dim - overlap* units, except for the first array, which has a length of *win_dim* units.
This operation is useful to adjust the peaks found by means of the microphone signal into the accelerometer acquisitions.
The results are shown in the figure below (figure 5.6).

It's possible to notice that some of the peaks detected are shifted with respect to the actual shear. We can also observe that this issue is due to the microphone adjustment of the buckets dimensions.
The figure below (figure 5.7) shows that quite well.

Considering that the algorithm that adjust the microphone peaks to the acceleration one has been developed by me, it can be wrong.

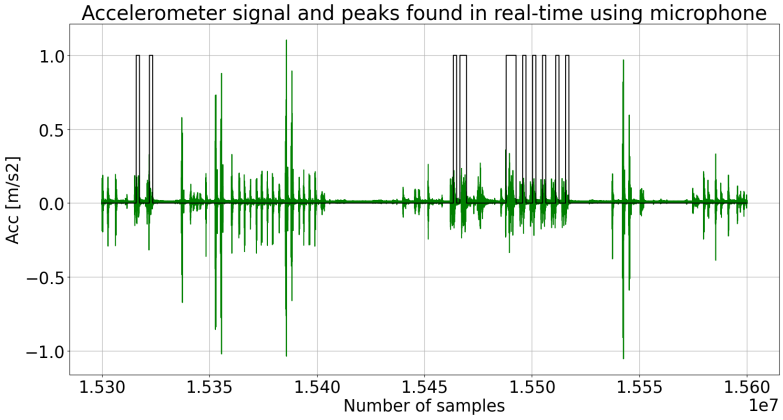


Figure 5.6: Peaks shifted from the microphone signal to the acceleration one

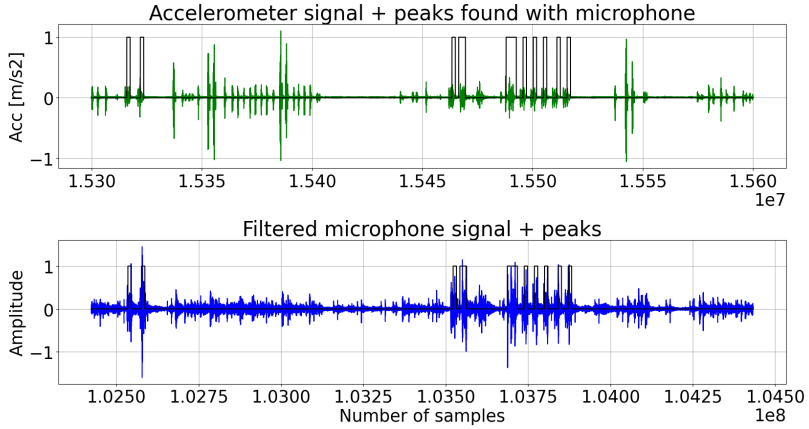


Figure 5.7: Peaks of the microphone signal over itself and over the acceleration one

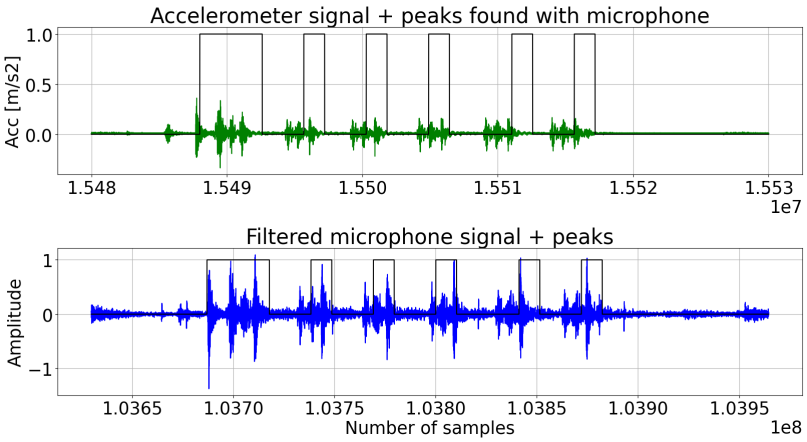


Figure 5.8: Zoom on some shears of figure 5.7

So, an alternative solution comes into play: the exploiting of the library *scipy*,

and in particular of its function *signal.resample*, which is a function that make a downsampling of the microphone signal to bring its sampling frequency similar to the one of the accelerometer.
The Python implementation of the downsampling process is provided in the code 9. The peak recognition in real-time is computed as described in the code 39.

Code 9

```
from scipy import signal
filtered_mic_downsampled = signal.resample(
    filtered_df_ISM330DHCX_MIC, len(df_ISM330DHCX_ACC))

extra_samples = ( len(filtered_mic_downsampled) - win_dim )
    % (win_dim-overlap)
df_ISM330DHCX_MIC_trim = pd.DataFrame(
    filtered_mic_downsampled[:len(
    filtered_mic_downsampled)-extra_samples] )
len_df = ( (len(df_ISM330DHCX_MIC_trim)-win_dim) /
    (win_dim-overlap) )+1
```

Unfortunately the results are not so different from the previous one, as it's shown in the figures 5.9 and 5.10.
Moreover the utilization of the microphone may give rise to potential issues on the CPU utilization due to its demanding requirements in terms of sampling frequency and subsequent data analysis of the acquired signal.

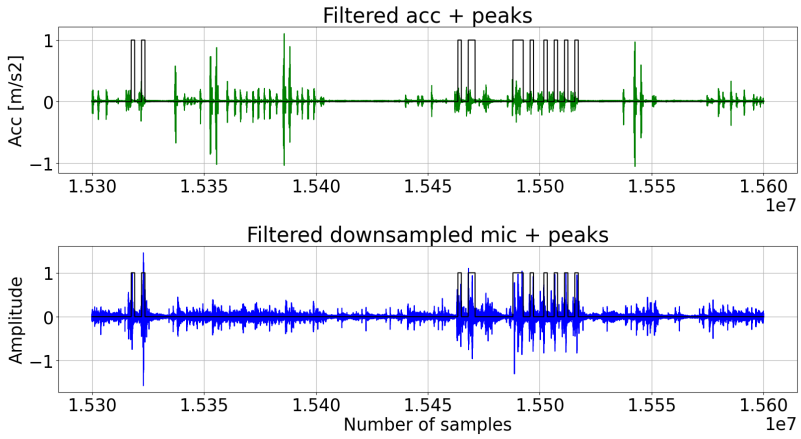


Figure 5.9: Peaks of the downsampled microphone signal over itself and over the acceleration one

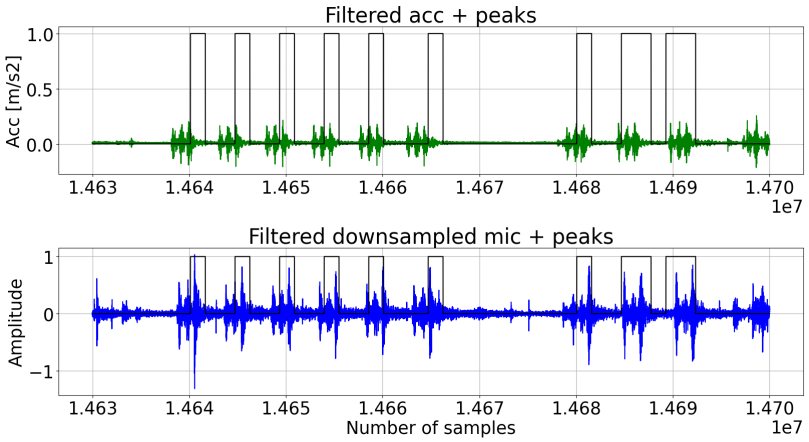


Figure 5.10: Zoom on some shears of figure 5.9

In conclusion, the microphone signal can be useful in order to detect the shears in the acceleration signal, as expected from the significant noise generated by the shear.
Despite it is possible to operate using that audio sensor, there are some difficulties that can be avoided: the main problems are the non optimal performance of the script and the CPU computational effort demanded from the microphone sensor.
It's better to avoid an high computational effort on the edge, because it may lead to less accuracy in the algorithm due to hardware limitations.

In order to avoid all of that, a new and simpler strategy was adopted: analyzing the accelerometer signal directly, and attempting the peak recognition algorithm on it without incorporating the audio sensor.

5.3 Shears recognition directly on the accelerometer signal

As seen in the previous section, using a microphone sensor can introduce several drawbacks due to the fact that the sampling frequency and data analysis of the acquired audio signal can be computationally demanding, leading to increased CPU utilization.
Moreover, the microphone may introduce challenges in synchronizing the detected peaks with the accelerometer.

To overcome these inconveniences, an alternative strategy comes into play: focusing only on the accelerometer signal.

In fact, by directly analyzing the accelerometer data and performing the peak recognition algorithm on it, the drawbacks associated with the microphone sensor can be avoided.

In this section, we will focus on:

- what "real-time application" means;
- filter the accelerometer acquisitions;
- implement the peak recognition function using both a built-in function from the *scipy* library and a custom function;
- define what is machine learning and implement some algorithms;
- implement a neural network algorithm;

5.3.1 Real-time application

Real-time applications are those application where it's important to provide instantaneous or near-instantaneous responses, often with minimal delay or latency. These applications are commonly used in robotics or control systems, where prompt and accurate responses are critical.

The STWIN.box is a board specifically designed for the real-time application, enabling simultaneous data analysis and continuous data collection. The operating condition in which will be used is referred to as FIFO mode, an acronym for "First In, First Out." In this way, the new collected sample is stacked in the end of the acquired array, while the last collected sample is discarded from the array, as shown in the figure 5.11

Before compiling the firmware in the STMicroelectronics IDE, called *STM32CubeIDE*, and deploying it to the STWIN.box board, we need to simulate through Python the behaviour of the board under real-time operating condition and check the effectiveness of every algorithm used.
Detailed discussions and explanations of these Python scripts will be presented in the subsequent subsections.

The Python simulation of the real-time operating condition involves dividing the entire acquired array into multiple buckets, each corresponding to the small array that is collected within the board in FIFO mode.

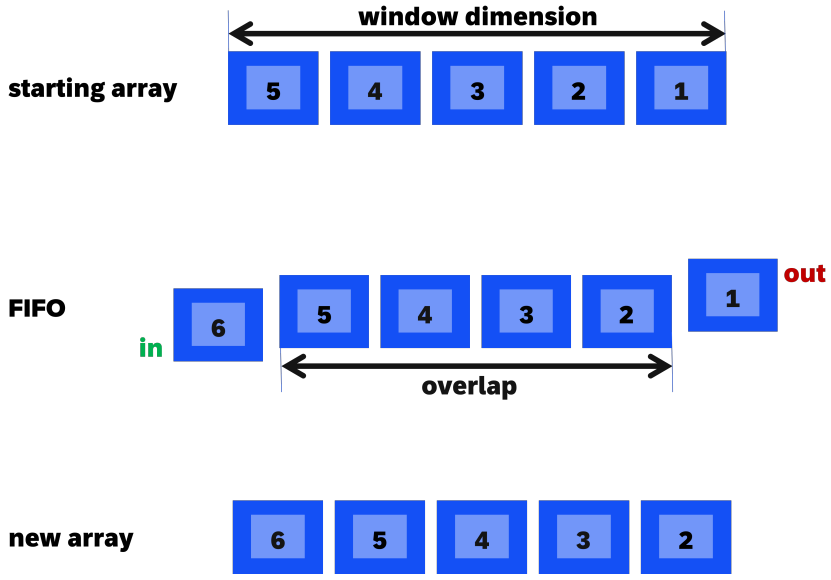


Figure 5.11: FIFO mode of operation

The code implementation is shown below (code 10).

Code 10

```
overlap = 2048-512 # 512 samples will change based on FIFO
win_dim = 2048
# win_dim is the real-time array length
# remove last incomplete bucket from the whole acquisition
extra_samples = ( len(df_ACC_450) - win_dim ) %
    (win_dim-overlap)
df_ACC_450_trim = pd.DataFrame( df_ACC_450.to_numpy()[
    :len(df_ACC_450)-extra_samples,:]  )
len_df = ( (len(df_ACC_450_trim)-win_dim) /
    (win_dim-overlap) )+1

for i in range(int(len_df)):
    if i == 0 :
        # don't use overlap for the starting array
        begin = i
        end = begin + win_dim
        df_realtime = df_ACC_450_trim.iloc[begin:end]
        ### real-time data analysis here ###
    else :
        # use overlap for every array but the first one
```

```
        begin = end - overlap
        end = begin + win_dim
        df_realtime = df_ACC_450_trim.iloc[begin:end]
        ### real-time data analysis here ###
```

The script involves the use of an `if` condition because the starting array can't overlap with the previous array, because it does not exist. So, the starting array is as long as *win_dim* and correspond to the first *win_dim* samples of the whole acquisition; while the second array will be correspondent to the interval of samples between the 512*th* and the 2048 + 512 = 2560*th* samples of the whole acquisition and so on for the remaining arrays.

Issues arising from the real-time operating conditions

The issues that arises working with short real-time arrays in a Python script can be of several types:

- Limited data resolution, especially in the frequency domain;
- limited context and representation: short arrays may not capture the complete context of the data.

The limits on the context of the data is solved using a big overlap between one array and the following one. In fact 3/4 of the array is preserved into the following array, and only 1/4 of it is discarded.

For what concern the limited data resolution, no solution is available and the only thing to do is keeping the window dimension as big as possible, in order to perform the Fast Fourier Transform using as many samples per segment as possible.
Moreover, the window dimension have to be preferably at the power of 2 and not too large, in order to optimize the memory of the STWIN.box CPU and not to lose some data.
So, we choose a window dimension of 2^{11} = 2048 samples and a overlap of 2^9 = 512 samples.

The following script (code 11) shows that, during the FFT computation, it's possible to set the *nperseg* parameter that is responsible for the frequency

resolution and filtering degree of the power spectrum.
Considering that the higher value for the *nperseg* parameter correspond to *win_dim*, we need to keep *win_dim* as large as possible to combine a good frequency resolution with the possibility to filter the spectrum.

Code 11

```
import scipy as sp
[freq_PS, PS]= sp.signal.welch(df, nperseg=512,
    scaling='spectrum')
freq_PS = freq_PS * fsamp # fsamp = fsampfinder(df)
delta_freq = np.mean( np.diff(freq_PS) )
```

In order to demonstrate that, it's possible to use for example as *df* the whole acquired dataframe *df_ISM330DHCX_ACC.T.to_numpy()[3]* and find out that the results of the frequency resolution are different based on *nperseg* parameter:

- if *nperseg = 512 => delta_freq = 13,9964* Hz;
- if *nperseg = 2048 => delta_freq = 3,4991* Hz.

After establishing the limitations imposed by utilizing a real-time application, we can move on and analyze the incoming data simulating the real-time operating conditions.

5.3.2 Filtering the signal

The filtering operation of the signal is the first thing to do, in order to highlight the data samples corresponding to the shears.

From the previous analysis on the microphone signal, performed in the section 5.2, it's possible to define where the shears are, so we can perform the filtering process in order to emphasize these shears. The filtering of the signal involves using a band pass filter that selectively includes only the frequency components associated with the shearing process.

So a trial and error operation was performed on the extreme values of the frequency range in order to find the most appropriate ones. At the end, the best result is provided by using a band pass filter between 550Hz and 650Hz.

In particular the piece of Python code that filter the signal is presented below (code 12), and the results are shown in the figure 5.12.

Code 12

```
def filter_mine(df_ACC, fsamp):  # work with z-axis only
    from scipy import signal
    cutoff_freq = 550.0 # High pass filter at 550 Hz
    filter_order = 8
    peak_timeserie_z = df_ACC.T.to_numpy()[3]
    b, a = signal.butter(filter_order, cutoff_freq,
        'highpass', fs=fsamp)
    hpf_peak_timeserie_z = signal.filtfilt(b, a,
        peak_timeserie_z, method="gust")
    cutoff_freq = 650.0 # Low pass filter at 650 Hz
    filter_order = 8
    b, a = signal.butter(filter_order, cutoff_freq,
        'lowpass', fs=fsamp)
    filtered_peak_timeserie_z = signal.filtfilt(b, a,
        hpf_peak_timeserie_z, method="gust")
    return filtered_peak_timeserie_z
```

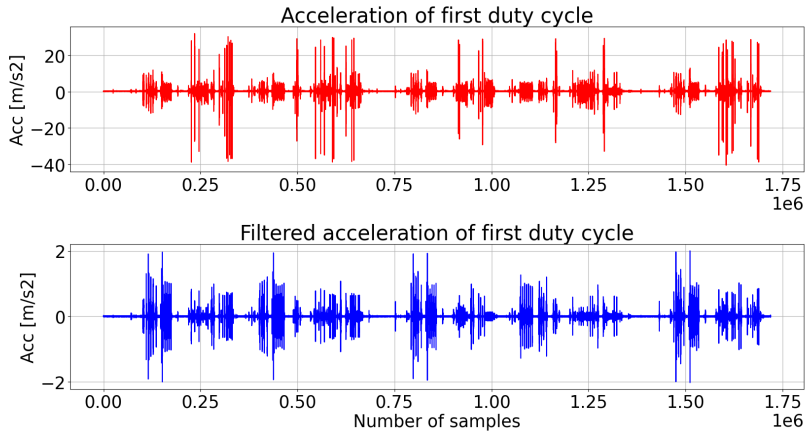


Figure 5.12: Comparison between filtered and non-filtered acceleration of the first duty cycle

The obtained results are quite satisfactory as the shears are effectively accentuated, making the peak recognition approach feasible.

Before we go on and discuss about the peak recognition function, it's important to notice a few things about the function used, in particular about

the use of *scipy.signal.butter* and *scipy.signal.filtfilt*.

Scipy.signal.butter is the built-in python function that exploit the library *scipy* to define the filter parameter got using the Butterworth method.
The Butterworth filter is a digital Infinite Impulse Filter that provides a smooth frequency response in the passband region, with minimal ripples in both the passband and stopband.
In particular, Elliptic and Chebyshev-based filters have constant ripple across their pass bands; while Bessel and Butterworth derived filters have no ripple in their pass band responses.
The Butterworth filter is defined by two parameter: the cutoff frequency, which determines the point at which the filter starts attenuating the input signal, and the filter order, that determines the drop steepness after the cutoff frequency (as shown in the figure 5.13 from the book *Digital Signal Processing* [20]) . Overall, this kind of filter is quite common due to its good performance and simplicity.

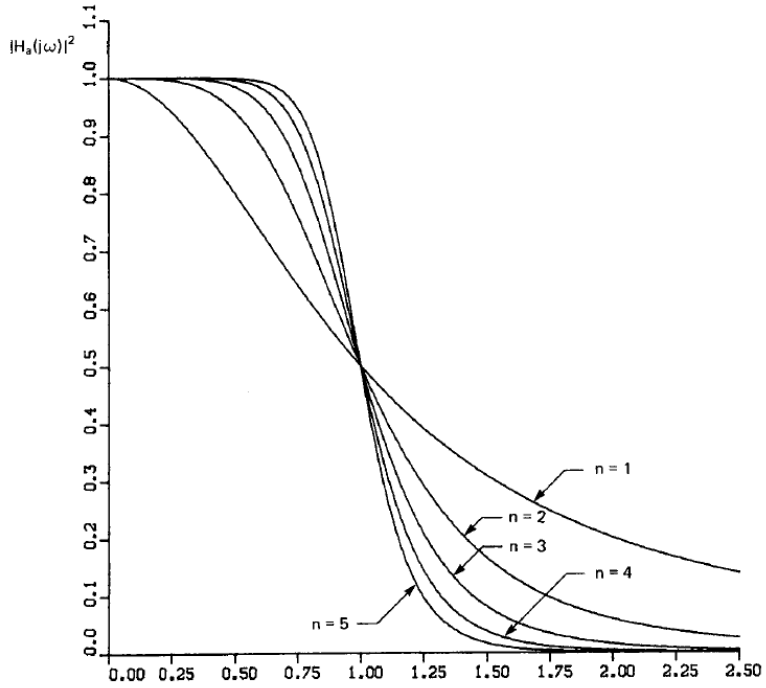


Figure 5.13: Butterworth filter at different filter orders
Source: Douglas F. Elliott. Handbook of Digital Signal Processing Book (1987)

scipy.signal.filtfilt is a filtering method that apply a zero-phase digital filter to a signal. In our case the filter used is the Butterworth one.
In general the filtering process is useful in removing unwanted noise or arti-

facts from the signal, the *filtfilt* method has the advantage of not introducing delay or phase shift to the filtered signal.

In fact, it operates by applying a digital filter twice: once in the forward direction and then in the reverse direction. This bidirectional filtering cancels out any phase distortion introduced during the initial filtering process. As a result, the filtered signal closely aligns with the original signal in terms of phase, making the real-time application really effective.

One particular parameter of the *filtfilt* algorithm is the *method* property. It can be *"gust"* or *"pad"*:

- Using the *"gust"* method, also known as the Gustafsson method, we aim to achieve zero-phase filtering;
- using the *"pad"* method, our aim is to minimize edge effects. It is possible using a padding strategy that extends the input signal by adding extra data points before and after the original signal, so to reduce edge effects due to the filtering process.

The real-time condition is considered to be more important than the padding one, because we can reduce the edge effects removing the first and last samples that can be corrupted and not significant due to the inertia of the filter before it goes to regime.

In fact, we used the *gust* method and we removed the first and last 12 samples from the computation of the peak recognition algorithm, as shown in the code 40 in the appendix F. In this way, we can maintain either a good real-time peak recognition and reduced edge effects. Moreover, that operation of removing samples will not affect the performance of the peak recognition algorithm, because we removed a little amount of samples and we can exploit the FIFO mode to detect the shear in the neighboring samples.

In the next section a more detailed analysis of the peak recognition algorithm will be discussed.

5.3.3 Peak recognition function

The peak recognition process was performed using two different approaches. Initially, a built-in function called *find_peaks* from the Python library *scipy.signal* was utilized, because it is convenient. But, considering the real-time application and the need of deploy the firmware on the board, a hand written

custom function was developed. In fact, the firmware had to be written in C++, which does not have a convenient library or built-in functions for peak detection.

SCIPY.SIGNAL.FIND_PEAKS

The documentation about this built-in function is available for consultation at [7].
The function requires several inputs, such as the signal itself, the minimum and maximum amplitude in which the function need to look for peaks, and the minimum distance between two adjacent peaks.

In order to identify the peaks in the acquired data, the following script (code 13) is executed.

Code 13

```
from scipy.signal import find_peaks
peaks_pyth_ACC_shears = find_peaks(
    filtered_df_ISM330DHCX_ACC, height=0.8, distance=2048)
```

The settings for the parameter height as 0.8 is based on the observation on the filtered signal in the figure 5.12: in fact we can notice that the shears signal has been highlighted from the filtering process and exhibits an amplitude higher than 0.8. On the other hand, no punches or noise in the signal reach such heights.

For what concern the distance parameter, it was set at 2048 to in order to exclude the decaying peaks of the shear from being considered a different shear: this ensures that every shear is associated with only one reference peaks, specifically the highest one, which correspond to the actual instant in which the shear impacts the sheet of metal to cut its excess of material.

The results of the algorithm are shown in the figure 5.14.

Nevertheless, the shown results do not align with the requirements of real-time application. In particular, there is the risk that a single peak is identified multiple times due to the overlapping of consecutive arrays. To address this issue, modification on how the algorithm will identify the peaks must be done.

The real-time application of the *find_peaks* function need to exclude every peak that has already been identified. The modified algorithm is provided

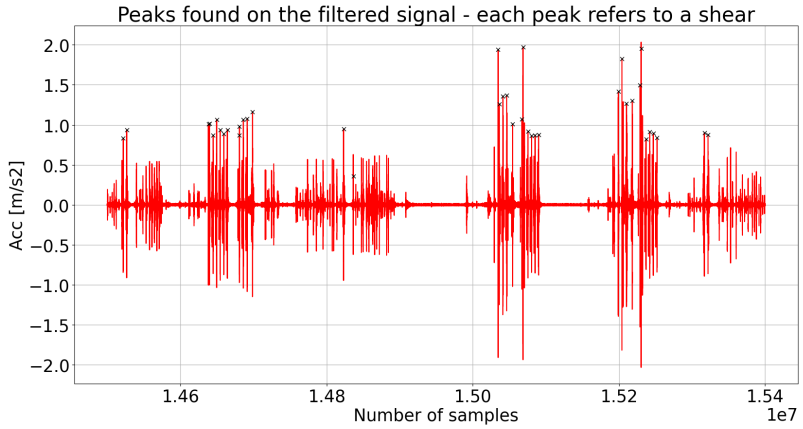


Figure 5.14: Peaks found over the filtered acceleration in z-axis

in the Appendix G, as it is lengthy and cannot be included here.

PEAKFINDER_MINE

Considering that the *C++* programming language used for writing the firmware of the STWIN.box does not have the built-in function *find_peaks*, it was necessary the development of a custom implementation.

Following best practices, the function was initially written in a simpler form and later extended in order to incorporate also the parameters of height and distance, similarly to the built-in *find_peaks* function.
The initial implementation of the code involves a simple iteration in a *for* cycle that compares all the values of the signal with the previous and the following one; if the current value was found to be higher than both its neighbors, it is identified as s peak and its index was appended to a list. Finally, the function returned the list of indexes where peaks were detected (code 27).

Afterwards, additional features were incorporated into the algorithm to account for the height and distance parameters.
The intermediate steps are presented in the code 27 and 28 and 29 in the appendix H, while here (code 14) only the final version of the custom function is shown.

Code 14

```
def peakfinder_mine(ar, lim_min=-100000, lim_max=100000,
    action_range=1):
```

```
    peaks = [] # initialize peaks list
    if lim_min == -100000:
        lim_min = ar.mean() # use mean of the signal as
            lower limit by default
    i = 0 # i identify the current value under analysis
    for num in ar:
        if num > lim_min and num < lim_max:
            if i<action_range and i+action_range>len(ar)-1:
                action_range_arr = ar.copy()
            elif i < action_range:
                action_range_arr = ar[:i+action_range+1]
            elif i+action_range>len(ar)-1 :
                action_range_arr = ar[i-action_range:]
            else:
                action_range_arr = ar[i-action_range:
                    i+action_range+1]

            if num >= max(action_range_arr):
                peaks.extend([i])
        i = i+1 # move on to the next value
    return peaks
```

Then, the modified version of the function was used in the code 26 (code in the appendix G). The piece of code (code 15) which originally exploited the *find_peaks* function, was replaced with the a different algorithm (code 16) that implement the custom peak detection algorithm.

Code 15

```
...
filtered_peak_timeserie_z_realtime, peaks =
    filter_and_peakfinder(df_realtime, fsamp)
...
```

Code 16

```
...
filtered_peak_timeserie_z_realtime = filter_mine(
    df_realtime, fsamp)
peaks_shears = peakfinder_mine(
    filtered_peak_timeserie_z_realtime, lim_min=0.8,
```

```
    action_range=2048)
...
```

The comparison between the two approaches are shown below (figure 5.15).

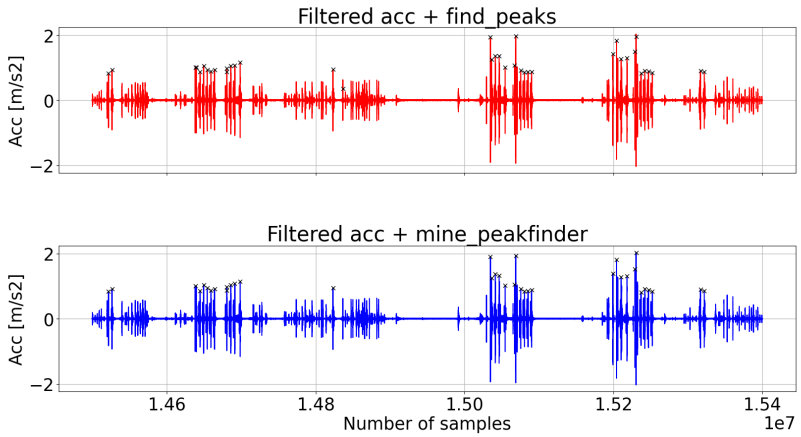


Figure 5.15: Comparison between built-in function *find_peaks* and custom function

The complete code that consider the real-time condition with the custom functions *filter_mine* and *peakfinder_mine* are presented in the appendix F.

In order to improve the accuracy of the shears recognition, the implementation of a machine learning algorithm is taken into consideration. A detailed discussion about this topic will be explored in the following section.

5.3.4 Machine learning implementation

Even if the peak recognition function works quite well in detecting the shears, we want to implement a Machine Learning algorithm in order to improve the accuracy of the output, and the robustness and flexibility of the algorithm.

Starting from the fundamentals, in this section we will explore some aspects of the wide topic of machine learning in order to discriminate which can be the most suitable one for our application.

What is machine learning?

Machine learning is a field of artificial intelligence that focuses on developing algorithms and models that enable computers to learn from data and make predictions or decisions without being explicitly programmed. It involves the use of algorithms to analyze and interpret patterns and relationships in data.

In particular, the aim of machine learning is to analyze data and classify them into groups, based on some provided features.
It is evident that certain features can be very significant in order to distinguish two different groups, while others may be less influential.
For example, we can consider flowers data and exploit a machine learning algorithm in order to classify them. Maybe the color of the petals or their number can be distinguishing features, but the presence of petals is not informative for sure, since all flowers have at least one petal.
This discussion about of how much important is a feature in classification tasks will be treated in the following sections.

The figure 5.16 shows the relationship between machine learning, artificial intelligence, and deep learning. In particular, deep learning focuses on the development of artificial neural networks, which are designed to simulate the structure of the human brain in order to get them to autonomously learn and make decisions.

supervised vs unsupervised

One of the primary classifications of machine learning algorithms is based on whether they fall into the category of supervised or unsupervised learning.

Those two kind of algorithms have some differences both on the needed input and on the objectives.
In particular:

- Supervised Learning: these algorithms are designed to learn from labeled input data, so a target label must be associated to each data point. This algorithms aim to generalize the patterns and relationships learned from the labeled training data and make accurate predictions or classifications on new incoming data.

- Unsupervised Learning: these algorithms operate on unlabeled train-

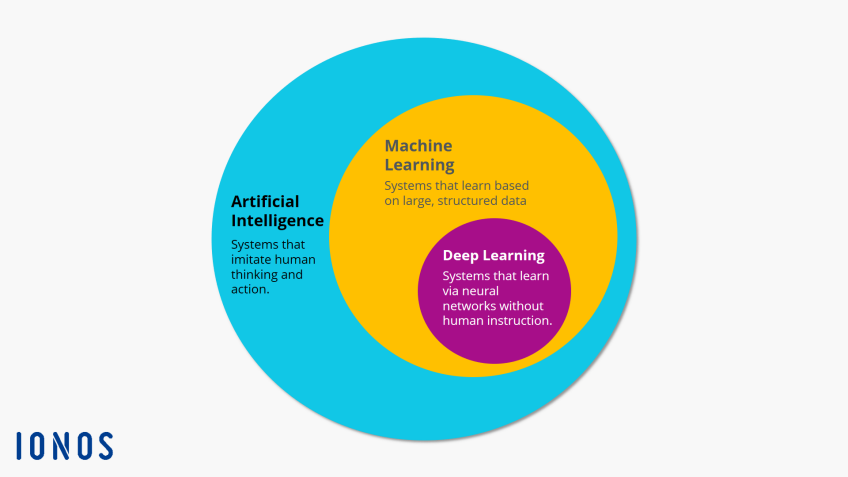


Figure 5.16: Relationship between machine learning, artificial intelligence and deep learning
Source: https://www.ionos.com/digitalguide/online-marketing/search-engine-marketing/deep-learning-vs-machine-learning

ing data. As for the supervised learning, these algorithms aim to find similarities within the data, but the unsupervised algorithm cannot lean on the correct output.

A third group that is between those exists and it is called Semi-supervised Learning, and typically is a kind of unsupervised learning with only a few data that are labeled and can help the machine learning algorithm to find patterns on the input data.
Nevertheless, this category is of secondary importance, so it will not be explored in detail.

Overall, the supervised learning algorithms are typically more accurate, but need for the labeled data, that means needing for more knowledge about the data. So, it is evident that there is a trade-off between the level of knowledge about the data and the accuracy of the output returned.
This consideration is crucial in several situation and, in particular, when the features of the input data are not distinctive and similar features can correspond to different output. In fact, the unsupervised learning can fail its task in this condition, as noticeable in the figure 5.17.

The figure 5.17 illustrates the different approach of the two types of algorithms based on whether the correct output labels are given or not. In particular, *(a)* and *(c)* are related to unsupervised learning, while *(b)* and *(d)* are related to supervised learning.

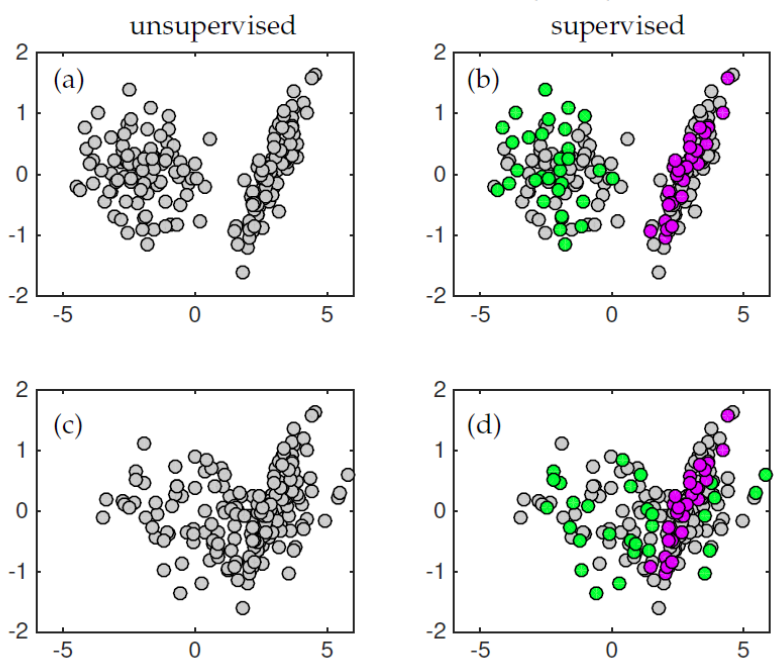


Figure 5.17: Machine learning performance on different kind of data
Source: Steven L. Brunton and J. Nathan Kutz. Data Driven Science and Engineering

In the figure two different conditions are presented: well separated data (*(a)* and *(b)*) , for which both the supervised and unsupervised algorithms return the output with good accuracy; and overlapping data (*(c)* and *(d)*), where the unsupervised learning often fails to assign the right class to the input data, and where the supervised learning have some difficulties to find patterns in the training data.

In conclusion, we can make two important observations.
The first one is that, for well separated data without labeled output, unsupervised algorithms can struggle with classifying the data: they may effectively group the data, but they may also assign the incorrect labels to the groups. In fact, running the algorithm multiple times, the group on the right could be arbitrarily assigned either to the group "0" or to the group "1".
While, the second, and more important observation is that the condition of well separated data must be met in order to guarantee a good performance of the algorithm.

So, the features extracted need to be as many as possible in order to define at least two or three distinctive features that get the data to be well separated.

As discussed in the previous paragraphs, the features are crucial in order to guarantee good performances on the machine learning algorithm.

So the extraction of the features from the data need to consider as many features as possible, in order to have the chance to select the most distinctive ones and provide a well separated set of data features to the machine learning algorithm.

The features extracted correspond to several statistical characteristics. In order to simulate the real-time operating condition, it's important to define a single feature for each array with length *samples_per_window* that enters the board.
For that purpose a custom function called *divide_into_windows* was developed. Its definition is shown in the Appendix A and its application is presented below (code 17).

Code 17

```
input_dataframe = df_ISM330DHCX_ACC
samples_per_window = 512
frames_reduced = divide_into_windows(input_dataframe,
    samples_per_window)
```

Therefore, it's possible to extract the features from the time domain acquisitions in all the three axis. In more detail, here are the extracted features:

- **Mean**, that is the mean of the input
- **STD**, that is the standard deviation of the signal with respect to the mean
- **RMS**, that is the Root Mean Square of the input
- **Median**, that is the median of the signal, less sensitive to outliers
- **Max**, that is the maximum value of the input
- **Min**, that is the minimum value of the input
- **Energy**, that is the area under the squared magnitude of the considered signal

- **q25, q75**, that are the first and third quartiles
- **ARCoeff**, that are the parameters for the autoregressive model that make predictions about the following values
- **skewness**, that is the parameter for the asymmetry of the input
- **kurtosis**, that is the parameter for how often outliers occur
- **CF**, that is the crest factor of the input signal and it is defined by `Max/RMS`
- **SNR**, that is the signal to noise ratio
- **corrCoeff_xy, corrCoeff_xz, corrCoeff_yz**, that are the correlation coefficient between the acceleration in the different axis

The features extracted from the frequency domain acquisition are similar to those extracted from the time domain, but the following additional features are also considered:

- **fEntropy**, that is the spectral entropy of the signal and it's important to discriminate highly likely event occurs from the unlikely ones;
- **fTHD** (Total Harmonic Distortion of the power spectrum) is defined as the ratio of the sum of the powers of all harmonic components to the power of the fundamental frequency;
- **fPS_maxInd**, that is the index of the maximum power spectrum amplitude;
- **bands_energy**, that is the area under the power spectrum. In particular we divided the power spectrum in 14 equal regions and got a value of bands_energy for each region.

Up to now, many features of the signal have been extracted and put in an organized dataframe. Therefore, the new goal become the definition of the most distinguishing features among all the extracted ones.
Some different approaches can be explored:

- brute analysis of all the features, looking at them one compared to another. In this way, about 10'000 images are generated and quickly analyzed in order to detect which features can generate a well separated set of data;

- Principal Components Analysis, or PCA, that is a technique used in order to reduce the dimension of the features matrix;
- *SelectKBest* Python built-in function, that create a hierarchical structure among the features, ranking them based on their distinctiveness.

The first approach was the most demanding one in terms of time, because the algorithm needs to generate many images and the supervisor needs to analyze each of them. Some of the generated images are shown in the figure 5.18.

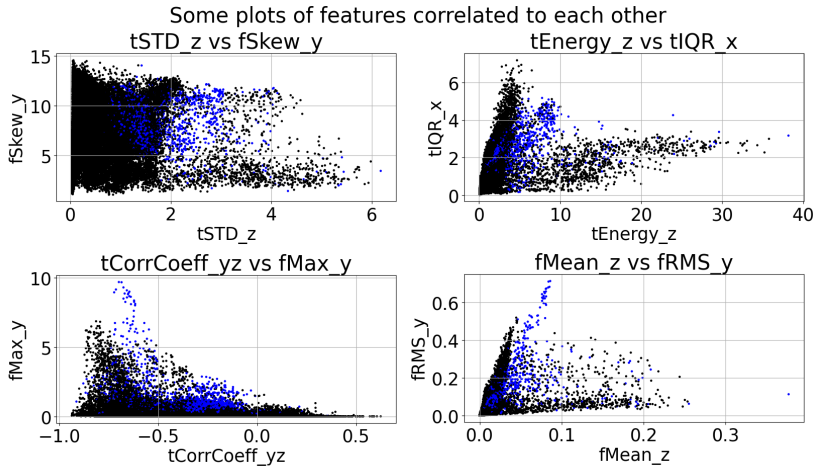


Figure 5.18: Some plots of features, one correlated with each other

It is possible to notice that, even if the shown features in figure 5.18 are the most significant ones among all the generated images, none of those shows a clear well separation between the shear data (blue dots) and other ones (black dots).

However, given the importance of finding features that effectively separate the data, we tried another solution: *SelectKBest*.
The algorithm is implemented easily, thanks to the library *sklearn* (code 18).

Code 18

```
from sklearn.feature_selection import SelectKBest, f_classif
selector = SelectKBest(f_classif, k=10)
selector.fit(features_df, labels_supervised_short)
cols_idxs = np.flip( selector.get_support(indices=True) )
print(features_df.columns[cols_idxs]) # visualize name of
    kbest features
```

The result is a list of ten elements (defined by *k=10* in *SelectKBest*) that enumerate the most distinguishing features.
They are **[bandsEnergy_z_2, bandsEnergy_y_2, bandsEnergy_y_1, fARCoeff1_y, fMean_y, tIQR_z, tIQR_y, tEnergy_y, tRMS_y, tSTD_y]**.
Unfortunately, even with this function, the defined features are not distinguishing enough, as it is shown in the figure 5.19.

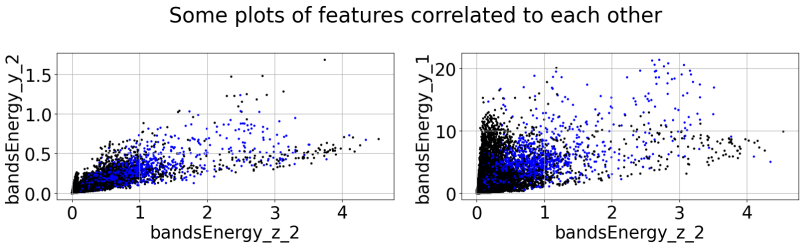


Figure 5.19: Plots of features identified with SelectKBest, one correlated with each other

The last chance to get the desired well separated data is the use of the PCA algorithm (code 19).

In detail, **PCA** means Principal Component Analysis and it is a dimensional reduction technique that exploit an unsupervised learning algorithm in order to transform a data-set of possibly correlated variables into a new set of uncorrelated variables called principal components.
The idea is to identify the directions where the data shows more variations and project the data onto these directions. In this way, only the more distinguishing features are kept.
The main drawback of this approach is that it is very difficult to map the generated matrix of principal components back into the original features.

Code 19

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler, normalize
X = features_df
# Scaling and normalizing X in order to make all the
features comparable
X_scaled = StandardScaler().fit_transform(X)
X_normalized = pd.DataFrame( normalize(X_scaled) )
pca = PCA(n_components = 2)
X_principal = pd.DataFrame(pca.fit_transform(X_normalized))
```

```
X_principal.columns = ['P1', 'P2']
```

Mathematically speaking the steps of PCA are subtraction of the mean from the data and computation of the covariance matrix, then the principal component is the eigenvector of B^*B corresponding to the largest eigenvalue. These steps are in the equations 5.1, 5.2, 5.3, 5.4, 5.5, taken from *Data Driven Science* [13].

$$\text{mean of each column} \qquad \bar{\mathbf{x}}_j = \frac{1}{n}\sum_{i=1}^{n} \mathbf{X}_{ij}, \tag{5.1}$$

$$\text{mean array} \qquad \bar{\mathbf{X}} = \begin{vmatrix} 1 \\ \vdots \\ 1 \end{vmatrix} \bar{x}, \tag{5.2}$$

$$\text{data with mean removed} \qquad \mathrm{B} = \mathbf{X} - \bar{\mathbf{X}}, \tag{5.3}$$

$$\text{covariance matrix} \qquad \mathbf{C} = \frac{1}{n-1} B*B, \tag{5.4}$$

$$\text{first principal component} \qquad \mathbf{u_1} = \arg\max_{\|u_1\|=1} \mathbf{u_1}^* B^* B \mathbf{u_1}. \tag{5.5}$$

Since the PCA approach is an unsupervised learning algorithm, it is impossible to generate a map that can refer the labels to the principal components, because they were generated from unlabeled features and cannot be labeled afterwards due to the struggle in reversing the algorithm.
The unlabeled components are shown in the figure 5.20, and they appear to be not well separated.

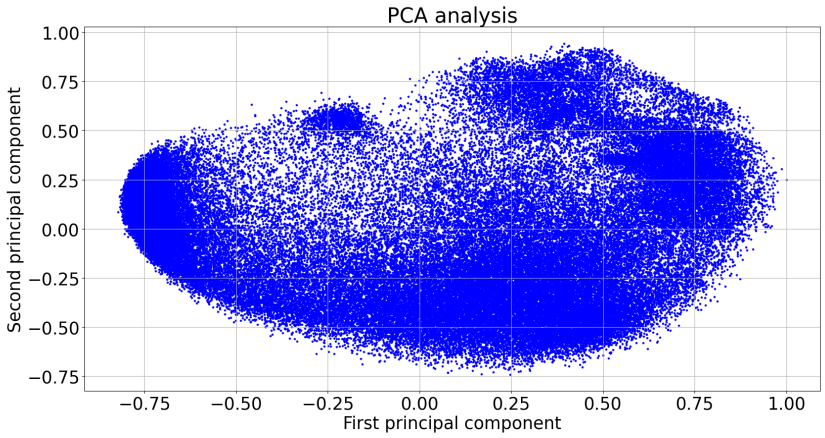


Figure 5.20: PCA analysis of the features matrix

Even without the condition of well separated data, we can try to adopt some supervised and unsupervised learning algorithm in order to check

their performance.

The different kinds of existing algorithms

In addition to the supervised or unsupervised classification, the machine learning algorithms can be grouped based on the approach they use.
The groups are listed below.

- **Hierarchical Based Methods**: those methods aim to create a hierarchical structure. They can be divided into *agglomerative clustering*, that consider each data point as a separate cluster and merges similar clusters together, and *divisive clustering*, where all data points belongs to a single cluster which is split into smaller subclusters. Those methods are useful if a more in-depth understanding of the relationships in the data is required.

- **Partitioning Methods**: the goal of these methods is to find a partitioning that maximizes the intra-cluster similarity and minimizes the inter-cluster similarity. In this category the most important algorithm is *K-means clustering*, that starts by randomly initializing K cluster centroids and assigns each data point to the nearest centroid. It then iteratively updates the centroids based on the mean of the data points assigned to each cluster until convergence.

- **Density-Based Methods**: Density-based methods identify clusters based on the density of data points in the feature space. These methods consider dense regions of data points as clusters and separate them by low-density regions. The most common algorithm is called *DBSCAN* (Density-Based Spatial Clustering of Applications with Noise): it defines clusters as high-density areas separated by low-density areas. It can discover clusters of arbitrary shapes and handle noise and outliers.

The machine learning model used

The results of the PCA algorithm in the figure 5.20 are not optimal, but we can try to use each method in order to find out if it can anyway detect the shears.

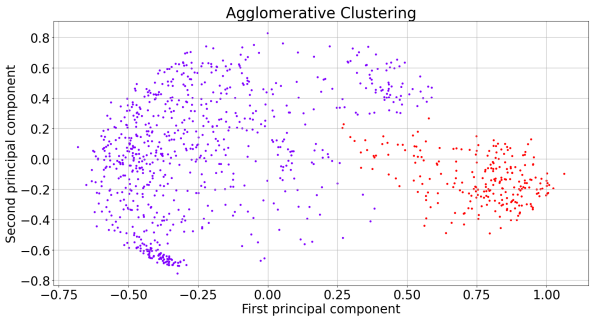
The methods are implemented in Python by means of the library *scipy* and the piece of code used is shown in code 20.

Code 20

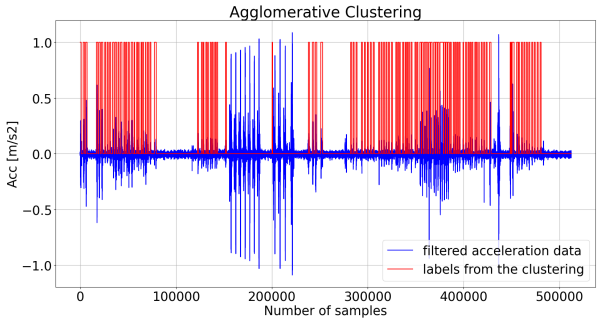
```
from sklearn.cluster import AgglomerativeClustering, KMeans,
DBSCAN
X = features_df.iloc[2000:3000] # work only in a portion of
the acquisitions
...  # find X_principal by means of PCA
ac2 = AgglomerativeClustering(n_clusters = 2)
y_ac2 = ac2.fit_predict(X_principal)
km = KMeans(n_clusters = 2, init = "k-means++")
y_kmeans = km.fit_predict(X_principal)
dbscan = DBSCAN(eps = 0.1, min_samples = 70)
y_dbscan = dbscan.fit_predict(X_principal)
```

The output of the algorithms, named as y, are a list of labels (one or zero) that defines the cluster in which each window of 512 samples lies based on its features.

The results of the application of the machine learning models are shown in the figures 5.21, 5.22 and 5.23.

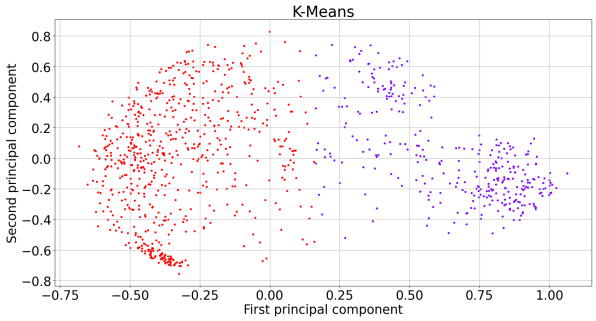


(a) PCA data points

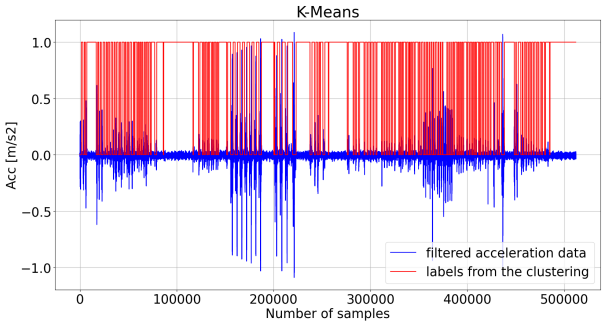


(b) Labels compared to filtered timeserie

Figure 5.21: Agglomerative Clustering results

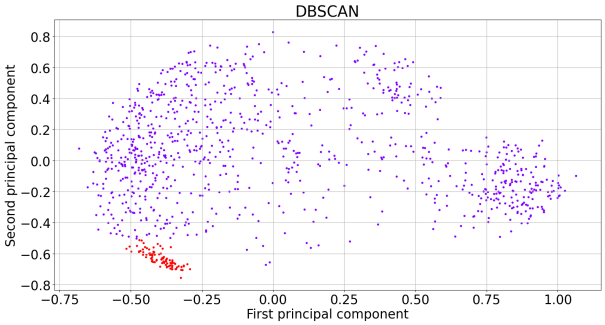


(a) PCA data points

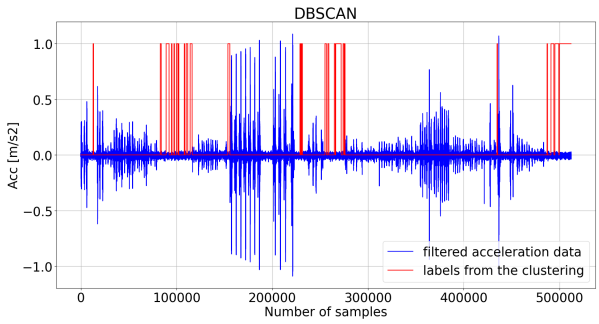


(b) Labels compared to filtered timeserie

Figure 5.22: K-Means results



(a) PCA data points



(b) Labels compared to filtered timeserie

Figure 5.23: DBSCAN results

Unfortunately, the lack of well separated data is a significant limit and the performance of each of the machine learning algorithms used are not enough for our application.

5.3.5 Discussion about the feasibility of a good neural network algorithm

The low performance of the machine learning algorithm bring us to explore a new tool that can be more powerful than the simple peak recognition function: the neural network.

Neural networks are algorithms included in the deep learning topic, that is a sub-topic of machine learning.
They are inspired by the structure and function of the human brain. It consists of interconnected nodes, called neurons, organized in layers. Each neuron receives input from multiple neurons in the previous layer, processes the information, and passes the output to neurons in the next layer.
The figure below (figure 5.24) shows the typical structure of a neural network.

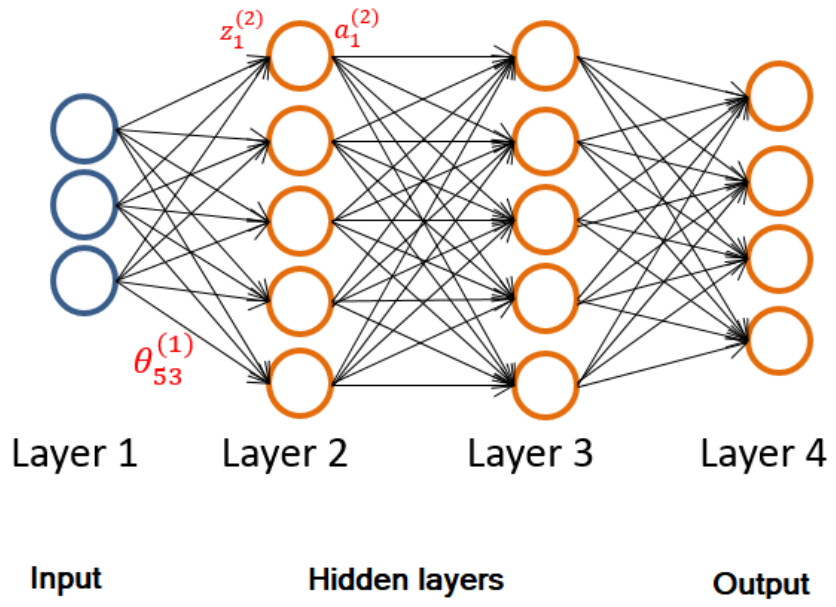


Figure 5.24: Neural network structure

In particular there are an input layer, that includes all the input variables, one or more hidden layers, that interprets the relationships between the in-

put data, and an output layer for the output variables.
The hidden layers are the heart of the neural network structure and they are composed by several nodes, also called neurons, each including a function to map the input data.

In detail, each connection between neurons correspond to a weight that the neural network give to that specific link (in the figure 5.24 the parameter $\theta^{(1)}_{53}$ is the weight of the first connection set between the input 3 and the neurons 5).

Moreover, each node correspond to an activation function that transform the input data often in a non-linear way.
The activation functions are typically of *Sigmoid* or *ReLU* shape (shown in the figure 5.25), but many other shapes exists too.

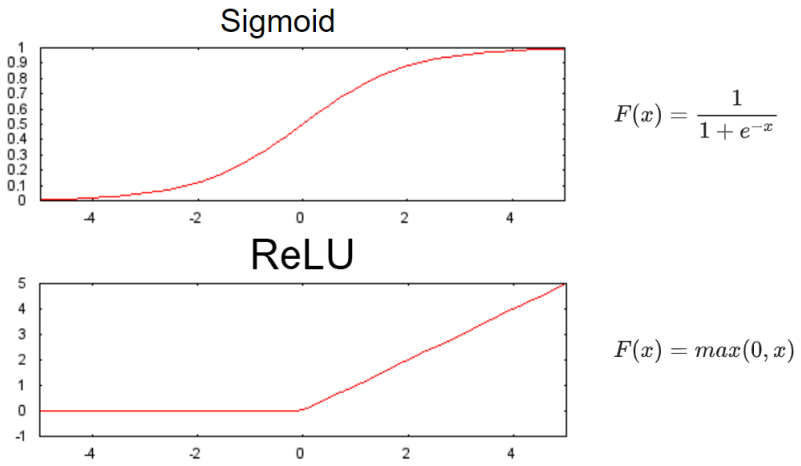


Figure 5.25: Sigmoid and ReLU non-linear activation functions

For example, using a ReLU activation function, we can determine the value exiting from the first node of the first hidden layer with the following equation:

$$z^{(2)}_1 = \theta^{(1)}_{11} \cdot input_1 + \theta^{(1)}_{12} \cdot input_2 + ...$$
$$a^{(2)}_1 = max(0, z^{(2)}_1 + b^{(2)}_1)$$

The neural network need to be trained in order to define its weights (θ) and biases (b).
The training process is done using pre-defined target output for the input data given and the neural network parameters are defined iteratively until the output converges to the correct one. This process is similar to the supervised learning one.

For our application, the code showing the algorithm implementation is shown in the code 21.

Code 21

```
import sklearn
import tensorflow as tf
labels_nn = np.zeros( len(filtered_df_ISM330DHCX_ACC) )
for x in peaks_no_overlap:
    labels_nn[x-212:x+812] = 1 # assign pre and post trigger
x = filtered_df_ISM330DHCX_ACC
y = labels_nn # y can be 0 (not shear), or 1 (shear)
X_1, X_2, y_1, y_2 = train_test_split(x, y, test_size=0.25,
    shuffle=False)
x_train_NN, y_train_NN = sklearn.utils.shuffle( X_1 , y_1 )
y_train_NN = tf.keras.utils.to_categorical(y_train_NN, len(
    np.unique(y_train_NN)))
model = tf.keras.Sequential([
              tf.keras.Input(shape = 1),
              tf.keras.layers.Dense(4, activation="relu"),
              tf.keras.layers.Dense(2, activation="relu"),
              tf.keras.layers.Dense(len(np.unique(y_train_NN,
                  axis=0)), activation="softmax") ])
model.compile(loss='categorical_crossentropy', optimizer=
    'adam', metrics=['accuracy'])
history = model.fit(x_train_NN, y_train_NN, batch_size=1024,
epochs=10, validation_split=0.25)
x_test_NN = X_2
y_test_NN = y_2
y_test_NN = tf.keras.utils.to_categorical(y_test_NN, len(
    np.unique(y_test_NN)))
pred = model.predict(x_test_NN, batch_size=1024) # test on
    filtered acceleration signal z-axis, same batch size
    used for training (1024)
```

The neural network model is the most common one: it is called Sequential and its structure is similar to the one shown in the figure 5.24, but with a single input neuron, followed by a hidden layer with 4 neurons, another hidden layer with 2 neurons, and finally, an output layer with 2 neurons, that classify signals by determining whether they belong to the shear category

or not.
The summary of the model is shown in the picture below (figure 5.26) and the results of the neural network application on a piece corresponding to the 20% of the acquired signal compared to the expected output value are shown in the figure 5.27.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 4)                 8

 dense_1 (Dense)             (None, 2)                 10

 dense_2 (Dense)             (None, 2)                 6

=================================================================
Total params: 24
Trainable params: 24
Non-trainable params: 0
```

Figure 5.26: Summary of the neural network structure

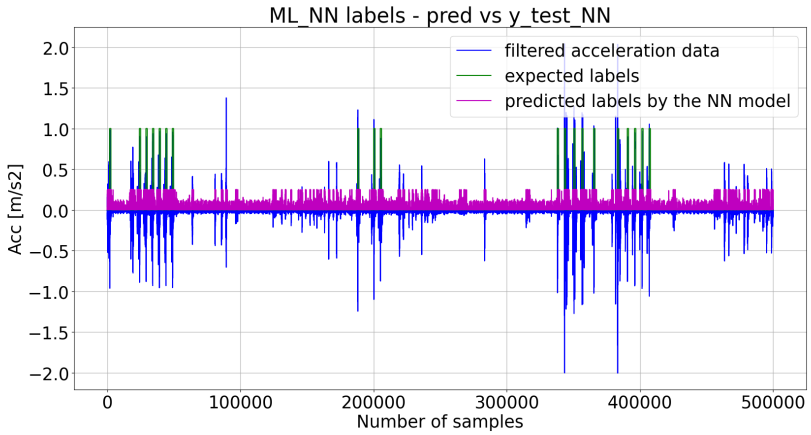


Figure 5.27: Comparison between the predicted label by the model and the expected one

The performance is quite poor, so the neural network is not suitable for shear recognition in our application. Maybe the cause of this poor performance is the difficulty in identifying distinct relationships between the data associated with punches and those associated with shears, as both types of signals exhibit impulsiveness.

Moreover, we decide not to go in further details in the neural network topic, as the peak recognition algorithm seems to work well enough for our application. Therefore, we shifted our focus towards identifying the punching operation by means of the peak recognition tool.

5.4 Punching recognition

In this section, we will analyze the possibility to identify also the punches made by the machine on the metal sheet.

The purpose of monitoring also the punching operations is not to assess the state of health of the punching tools, but to detect some anomalies in the overall operation, such as power loss or variations in the thickness of the metal sheet being punched.
In fact, we cannot monitor the health of each individual punching tools because there are many different tool placed in the machine head and it become difficult to differentiate and isolate the vibrations generated by each tool.

The detection of the punches can be performed using similar approaches to the ones used to detect shears.
Therefore, peak recognition and machine learning algorithms can be used in order to detect them. In the following paragraphs all the approaches will be treated in detail.

Machine Learning

We tried to implement a machine learning algorithm also to detect the punches and to distinguish the shears from the punches from the other machine conditions.
Unfortunately, similarly to the results shown in the section 5.3.4, no machine learning algorithm is able to separate effectively the data points belonging to the shears and those belonging to the punches.
In fact, the results are shown in the figure below (figures 5.28 and 5.29).

Considering these results, we need to discard the idea of using a machine learning algorithm in order to distinguish punches from shears.
So we implemented a new version of the peak recognition algorithm that can find also the punches.

Peak recognition

As done for the shear recognition, a suitable filtering frequency need to be identified. After a trial and error operation on the filtering frequency, we found out that there are no range of frequency where all the punches are highlighted with respect to the shears (as shown in the figure 5.30)

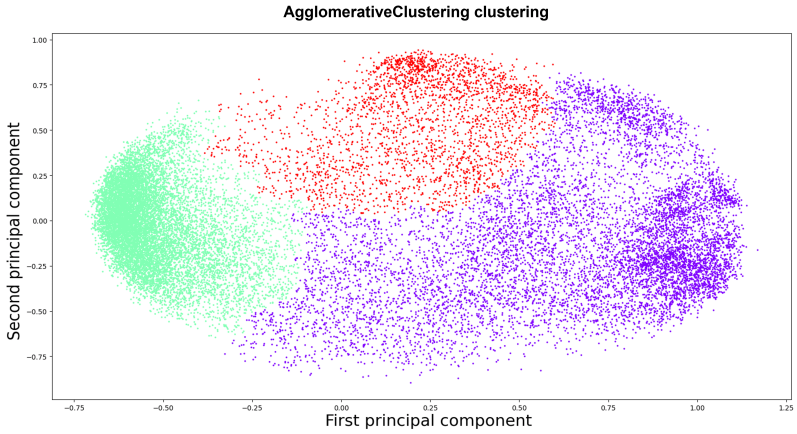


Figure 5.28: Agglomerative Clustering algorithm performed over the acquired signals - data point distribution

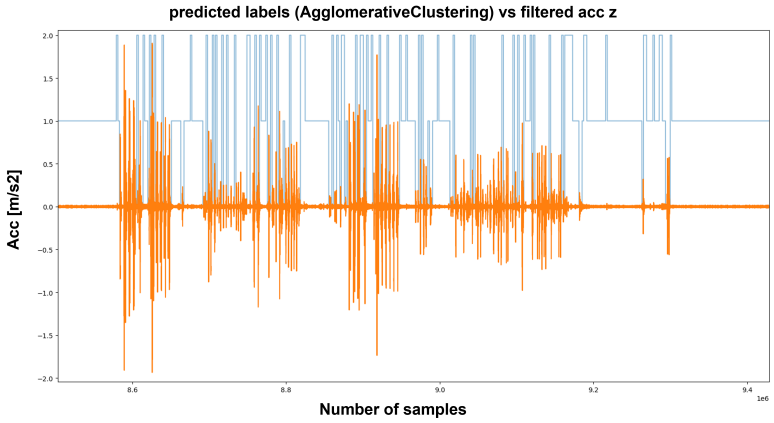


Figure 5.29: Agglomerative Clustering algorithm performed over the acquired signals - labels over the acquired signal

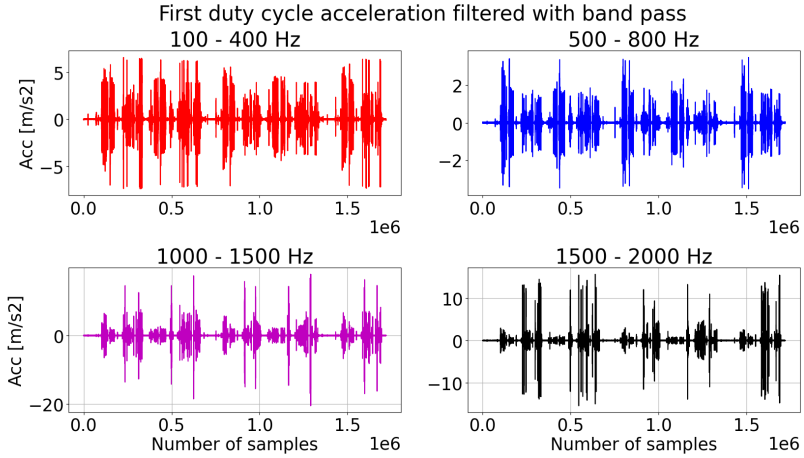


Figure 5.30: Pass band filter on the first duty acceleration acquisition

Even if some punches can be detected using the 1500-2000Hz band pass filter, we cannot detect them all. Moreover, for the real-time purpose is better to avoid performing two different filtering processes on the same acquired data, because the filtering process is CPU demanding and time consuming, so there is some risk of data loss.

In the end, the peak recognition function is done on the filtered data using a band pass filter between 550Hz and 650Hz, that is the same filtered array used for the shears recognition.
The idea is to consider only a portion of the filtered signal, exploiting the *lim_min* and *lim_max* parameters in the function *filter_mine* presented in the code 12.

The results are not satisfactory because also some peaks belonging to the shears are considered as punches. So, we need to implement a new piece of code (code 22) that excludes all the piece of signal near to the shears. It consists in removing all the peaks detected in the neighborhood of 2500 samples of the recognized shear.

Code 22

```
shears_list = [x.item() for x in peaks_list_notbinary]
punz_list = [x.item() for x in peaks_list_notbinary_punz]
shears_list_long = []
for x in peaks_list_notbinary:
    arange_created = np.arange(x-2500,x+2500)
    list_to_be_appended = [j.item() for j in arange_created]
    shears_list_long.extend(list_to_be_appended)
punz_list_noshear = []
for y in punz_list:
    if y not in shears_list_long:
        punz_list_noshear.append(y)
```

The results of the code 22 is shown below (figure 5.31) and they highlight that the punches can be well distinguished from the shears. Unfortunately the presented code is working in a non real-time condition, but it can be implemented if a multi-thread solution is selected, as we will discuss in the section 7.2.

It is important to recall that the punches recognition algorithm cannot be directly employed to determine the wear of individual punching tools, because the machine head includes multiples punching tools and we are not

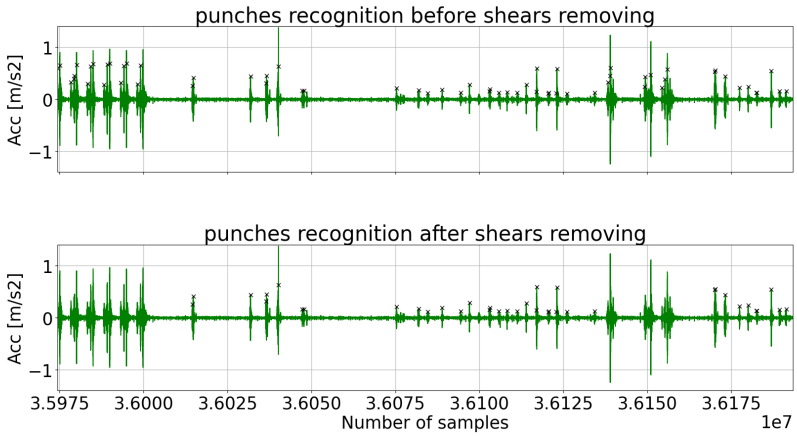


Figure 5.31: Results of the operation of removing shears from punches recognition

able to differentiate the signal corresponding to each specific tool. However, this punches recognition technique can be used in order to detect some anomalies in the overall punching operation.

In order to do that, the power spectrum of the signal corresponding to the punches can be analyzed. In particular, as shown in the figure 5.32 we can expect a different behaviour in the frequency region around 1000Hz and 1800Hz if a faulty punching operation is detected.

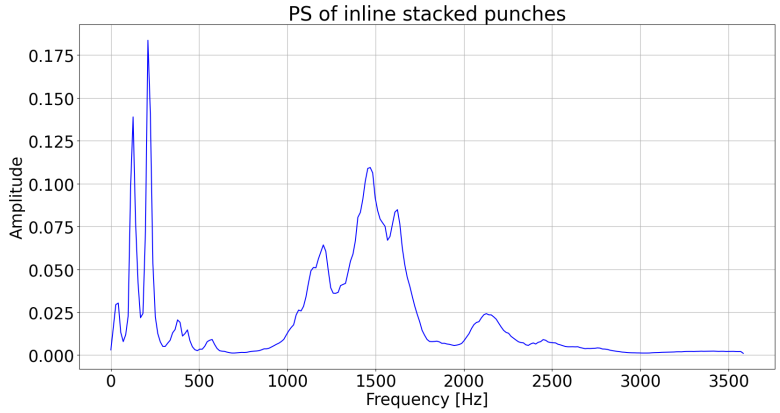


Figure 5.32: Power spectrum of the stacked recognized punches

Unfortunately, there was no time to investigate further more this part of the project, because we wanted to go more in the detail about the definition of the wear level of the shearing tool.

5.5 Comparison with a newer punching machine, operating with a shearing tool at the end of its service life

Since the shears are recognized well by the peak recognition algorithm, it's time to get more into detail about the definition of a good and bad state of health of the shearing tool.

All the acquisitions used to detect the shears were recorded around the beginning of April and the shearing tool of the punching-shearing center under exam (S4 1262) had performed about 250'000 shears (figure 5.33).

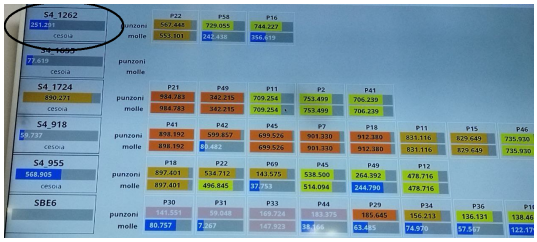


Figure 5.33: Dashboard of the state of health of the punching and shearing tool of several punching-shearing center as of 12th April

The maximum useful life for the shearing tool given by the manufacturer is 1'000'000 shears, so the recorded shearing tool was used for approximately one-fourth of its expected lifespan. Unfortunately, the shearing tool reaches its end of life after approximately one year of usage, and, due to the eight-month duration of my stay at KONE, it was not possible to monitor the entire lifespan of the shearing tool.

In order to compare the actual state of health of the shearing tool with a different one, we considered the option of recording the shearing tool of another punching-shearing center that has a shearing tool at its end of useful life.
In particular, the S4 1724 punching machine was taken into consideration because its shearing tool was over 1'100'000 shears old.

Unfortunately the machine is 10 years younger than the S4 1262, so it is a very different machine.
In detail, we need to check if the collected acquisitions can be useful for a comparison between the two punching centers.
The collected acquisitions of the accelerations over the machine head of the punching machine S4 1724 are made with a wear level of the shearing tool

as shown in the figure 5.34.

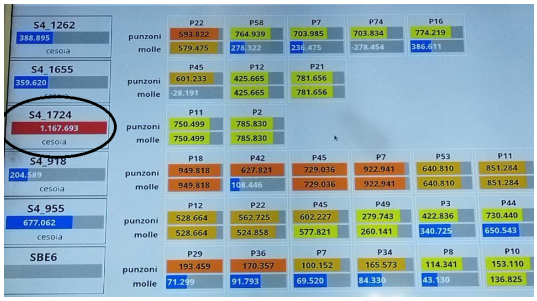


Figure 5.34: Dashboard of the state of health of the punching and shearing tool of several punching-shearing center as of 22th May

The time domain acquisitions of the accelerations of the S4 1724 in comparison with the ones of the S4 1262 are shown in the figure 5.35.
Moreover, they were filtered with a pass band filter between 550Hz and 650 Hz and the results are shown in the figure 5.36: it results that the amplitude of vibrations of the different machines are quite similar, so the machines are not extremely different between each other and we can proceed with the analysis.

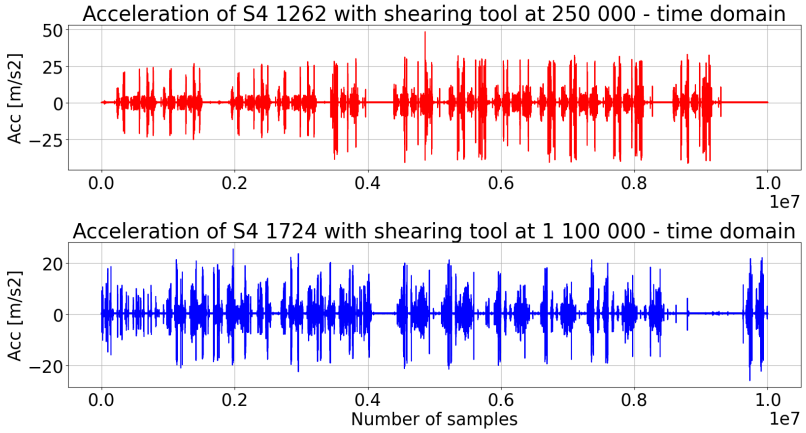


Figure 5.35: Comparison between the raw accelerations of the S4 1724 and those of the S4 1262

Going ahead, it's possible to detect the shears and analyze them. In particular, the *peakfinder_mine* function was exploited.
The identified shears are shown in the figure 5.37.

In order to analyze the response of the machine head to the shearing impulse, a frequency analysis is performed. In particular two different approaches are considered: compute the power spectrum of the whole acquired signal (figure 5.38) and compute the power spectrum of only the shear

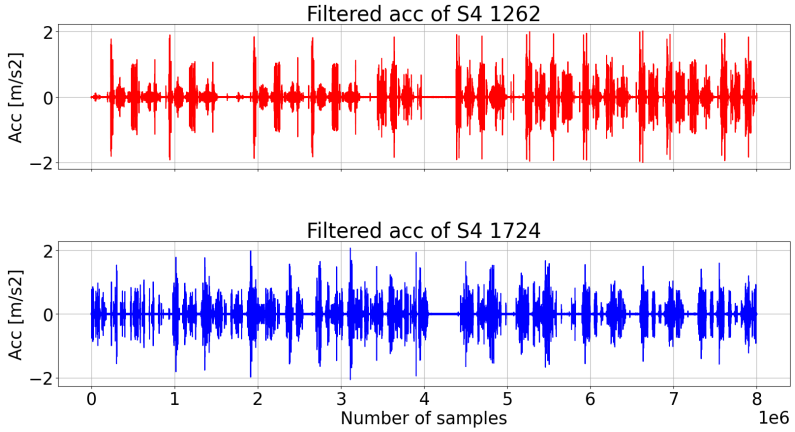


Figure 5.36: Comparison between the filtered accelerations of the S4 1724 and those of the S4 1262

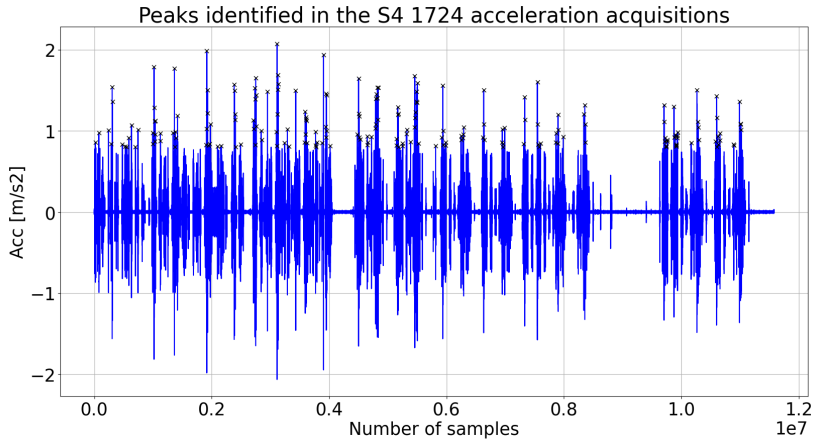


Figure 5.37: Shears identified in the S4 1724 acceleration data

signals isolating them by means of the *peakfinder_mine* function and taking also the neighboring data points (figure 5.39). The algorithm exploited in order to obtain such results is shown in the code 24.

Code 23

```
# find power spectrum of the whole signal
fsamp_1724 = fsampfinder(df_1724)
from scipy.signal import welch
[freqps_1724, ps_1724] = welch(df_1724.T.to_numpy()[3],
    nperseg=10000, scaling='spectrum')
freqps_1724 = freqps_1724*fsamp_1724
fsamp_1262 = fsampfinder(df_ISM330DHCX_ACC)
```

```
[freqps_1262, ps_1262] = welch(df_ISM330DHCX_ACC.T.
    .to_numpy()[3], nperseg=10000, scaling='spectrum')
freqps_1262 = freqps_1262*fsamp_1262
```

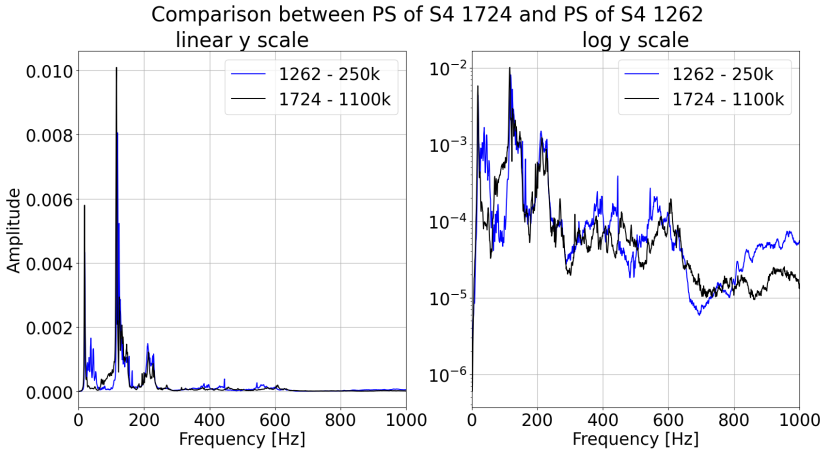


Figure 5.38: Comparison between the PS of S4 1262 and S4 the one of 1724 (whole acquired signal)

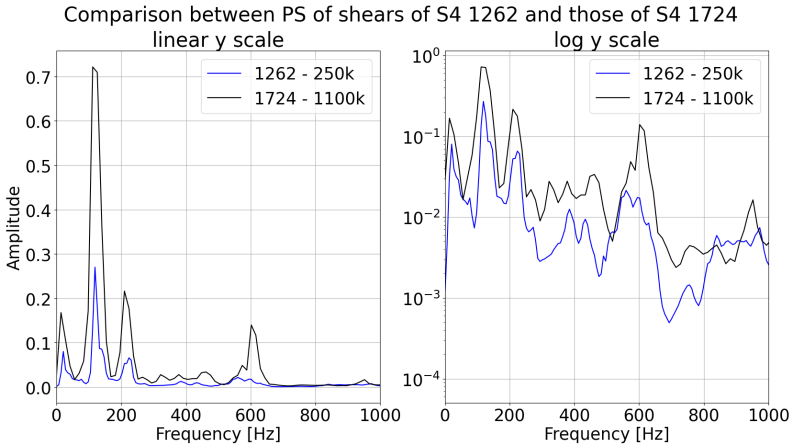


Figure 5.39: Comparison between the PS of S4 1262 and S4 the one of 1724 (considering only the signal corresponding to the shears)

Code 24

```
# create array including only the shear signals
fsamp = fsampfinder(df_ISM330DHCX_ACC)
peaks = [x.item() for x in peaks_list_notbinary_shears_rt]
ref_timeserie = np.array(filtered_df_ISM330DHCX_ACC_rt)
matrix_peaks_stacked = []
```

```
matrix_peaks_stacked_notfilt = []
for i in range( len(peaks) ) :
    matrix_peaks_stacked.append( ref_timeserie[
        peaks[i]-pre_trigger:peaks[i]+post_trigger] )
    matrix_peaks_stacked_notfilt.append(
        df_ISM330DHCX_ACC.T.to_numpy()[3][peaks[i]-
        pre_trigger:peaks[i]+post_trigger] )

array_shear = matrix_peaks_stacked.flatten()
array_shear_notfilt = matrix_peaks_stacked_notfilt.flatten()
# Then use code 23 to find the power spectra
```

The results shows that there are some differences in the power spectra of the two shear wear conditions. In particular, the region of the spectrum related to the shears, which is the one between 550 Hz and 650 Hz, is shifted to the right in the figure 5.39.

It's important to notice that the observations made up to now are sons of a comparison between two distinct machines, so it is possible that the variations in the power spectrum are due to differences in the machine structures rather than reflecting the actual response of the structure to varying levels of wear in the shearing tool.

Nevertheless, in the meanwhile some weeks has passed, and the shearing tool of the S4 1262 has accumulated additional shears throughout its useful life. So, a comparison between the two conditions can be finally developed.

5.6 Comparison between different level of shear wear on the same punching-shearing center

After some weeks, a new recording session was performed on the S4 1262 with an higher shear wear.
The previous recording session was with a shearing tool that was 250'000 shears old; the new acquisitions were made with a shearing tool that was 450'000 shears old (figure 5.40).

After we made some acquisitions at these shear wear, the data collected were analyzed and compared with the ones already analyzed.

The signal were filtered with a pass band filter between 550Hz and 650Hz as made for the other acquisitions (figure 5.41), and the custom peak recog-

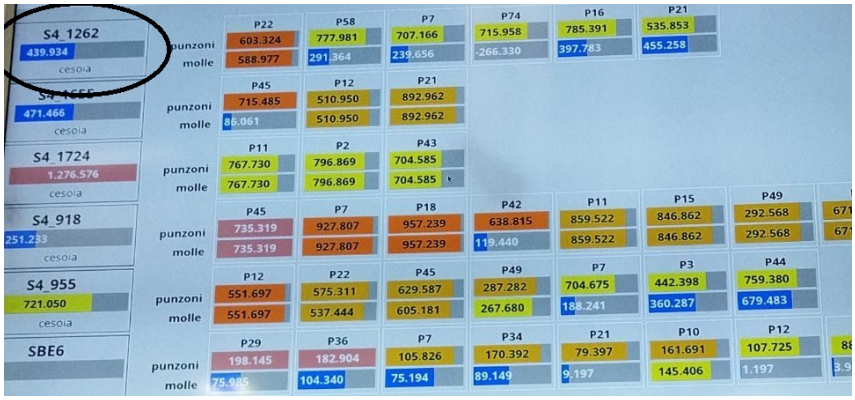


Figure 5.40: Dashboard of the state of health of the punching and shearing tool of several punching-shearing center as of 06th June

nition algorithm has been utilized (figure 5.42).

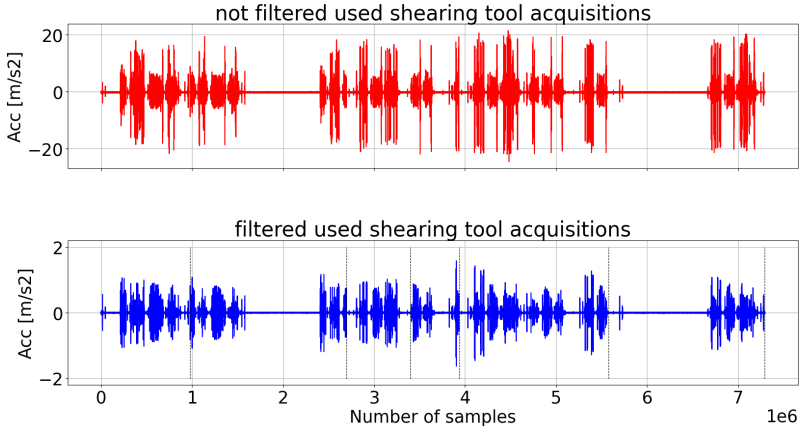


Figure 5.41: Comparison between filtered used shearing tool signal and non filtered one

Once the shear are recognized, it's possible to isolate them and analyze only the signal correspond to them. The piece of code used is similar to the one discussed in the code 24.

Then, as done for the S4 1724 signal in the code 23, the whole array of shears can be analyzed in the frequency domain. The resulting spectra are shown in the figures 5.43 and 5.44.

The results presented in figure 5.44 demonstrate that the wear of the shearing tool can cause some differences in the amplitude of the signal in the frequency range between 300Hz and 500Hz of the power spectrum and a slightly shift on the left of the frequency range between 500Hz and 700 Hz. Unfortunately, this finding contradicts the analysis conducted on the signal

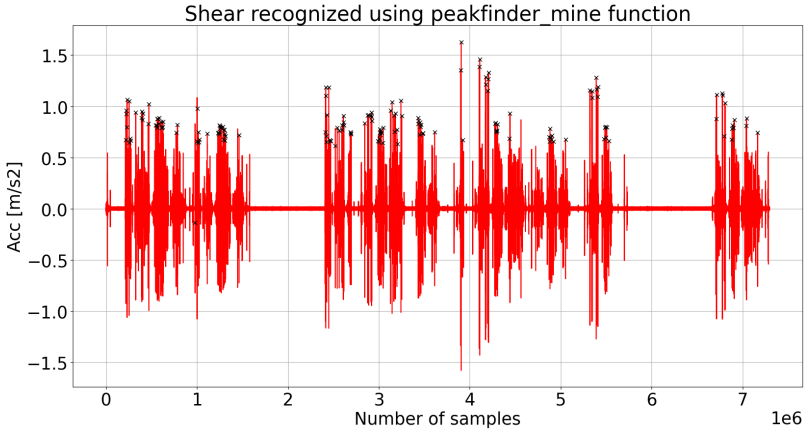


Figure 5.42: Shears recognized by the peakfinder_mine function

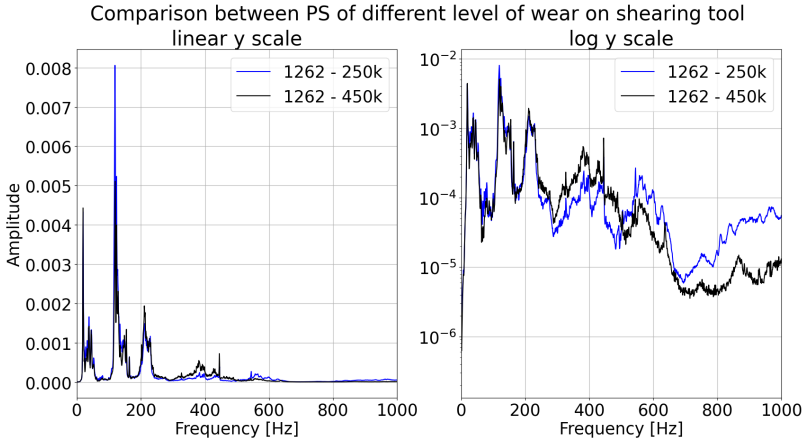


Figure 5.43: Power spectrum on the whole signals

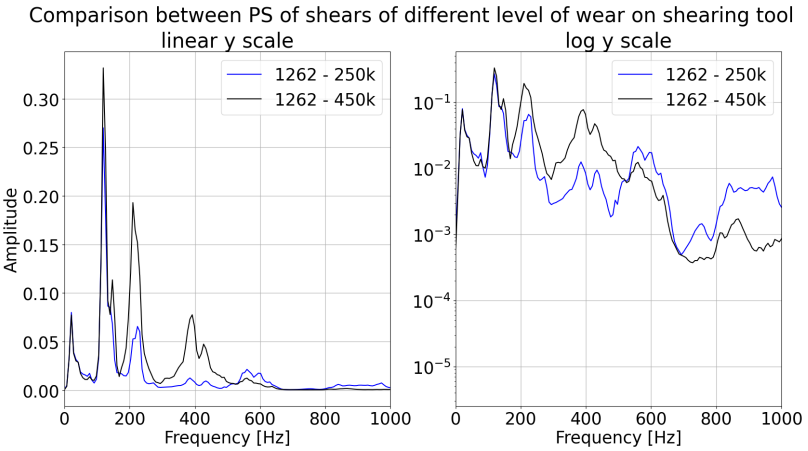


Figure 5.44: Power spectrum on the shear signals only

of the S4 1724 punching machine.

Anyway, the results of the comparison between different wear level on

the same shearing tool is more significant since the analysis is performed on the same machine structure. So, the observations made in the paragraph 5.5 were discarded and we proceeded with these evidences.

In order to transfer the algorithm into a real-time condition approach, an analysis of the shears is made using ten shears at the time instead of using the whole array of stacked shears one after another.
The Python script is here described (code 25) and the results are shown in the figure 5.45.

Code 25

```
### S4 1262 - 250k ###
pre_trigger = 128
post_trigger = 512-pre_trigger
fsamp = fsampfinder(df_ISM330DHCX_ACC)
peaks = [x.item() for x in peaks_list_notbinary_shears_rt]
ref_timeserie = np.array(filtered_df_ISM330DHCX_ACC_rt)
peaks_stacked = []
peaks_stacked_notfilt = []
for i in range( 10 ) :
    peaks_stacked.append( ref_timeserie[
        peaks[i]-pre_trigger:peaks[i]+post_trigger] )
    peaks_stacked_notfilt.append(
        df_ISM330DHCX_ACC.T.to_numpy()[3][
        peaks[i]-pre_trigger:peaks[i]+post_trigger] )
array_shear_250 = peaks_stacked.
    .flatten()
array_shear_notfilt_250 = peaks_stacked_notfilt.flatten()
# Similar script for the "S4 1262 - 450k" case
```

The results are similar to the ones discussed before as expected. Unfortunately my eight-month stay in KONE is not enough to monitor the machine head during its whole useful life, so we cannot validate the obtained results using a more worn shearing tool on the S4 1262.

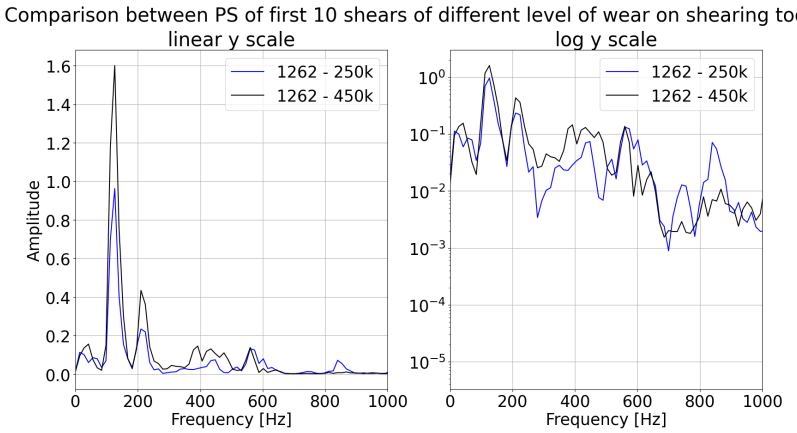


Figure 5.45: Power spectrum on the first 10 shear signals only

6 Side Projects

The main predictive maintenance project led us to explore some additional collateral projects. Moreover, we carried out some minor supplementary work for the company, all within the maintenance field.
In particular, there are three side projects that deserves a more detailed investigation:

- PLC project;
- Main failures on the S4 1262 punching machine;
- TPM smartphone app.

PLC project

While working on the predictive maintenance project based on peak recognition, a side project takes shape.
It consisted in making a simpler predictive maintenance project using the PLC and some sensors that do not requires demanding performance in terms of sampling frequency.

Therefore a dashboard similar to the one generated for the visualization of the anomalies in the SensorTile.box project is created. Its interface is shown in the figure 6.1.

It is incomplete and the idea is to add some other sensors to the PLC, such as other temperature sensors, oil level sensors, oil quality sensors that exploit the particle counter technology, mass flow sensors and others.
Up to now the hardware components are set in a temporary way in the electrical panel (as shown in the figure 6.3) and only the temperature sensor is correctly installed, following the wiring scheme presented in the figure 6.2), and it measures the electrical panel internal temperature. This data is then processed by means of the *snap7* library and shown in the dashboard (figure 6.1).

Overall, this side project taught me many electrical concept and let me learn something about how to wire a temperature sensor to the PLC system.

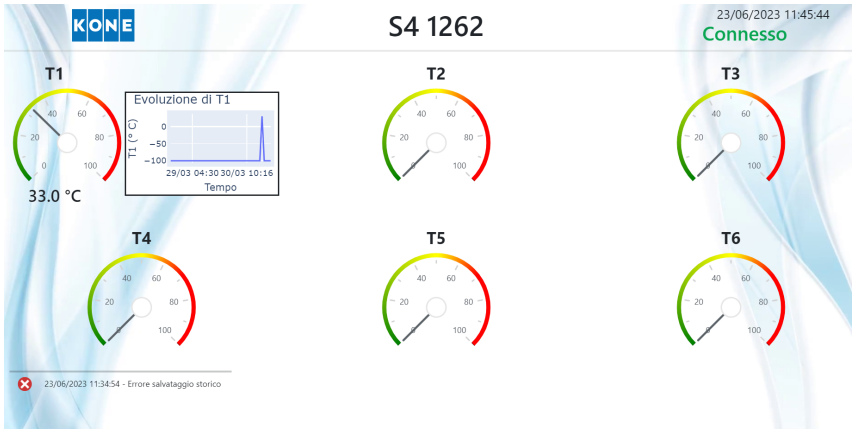


Figure 6.1: Dashboard related to the data acquired by the PLC system

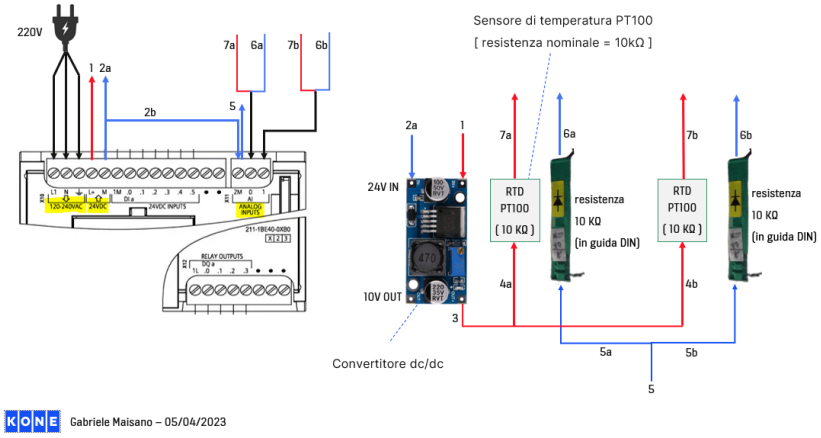


Figure 6.2: Electrical scheme to install the PT100 temperature sensor into the PLC system

Main failures on the S4 1262 punching machine

In order to prioritize maintenance efforts and identify components that require more attention due to their susceptibility to breakdowns, a breakdown analysis was conducted using Microsoft Excel©.

The breakdown data, obtained from the database, were organized based on the frequency of breakdowns and the corresponding downtime they caused.
This analysis provided valuable insights about the criticality of several components and helped in determining which require more attentions.

The results of the analysis are shown below (figure 6.4).

As you can see, the most critical components of the production line were

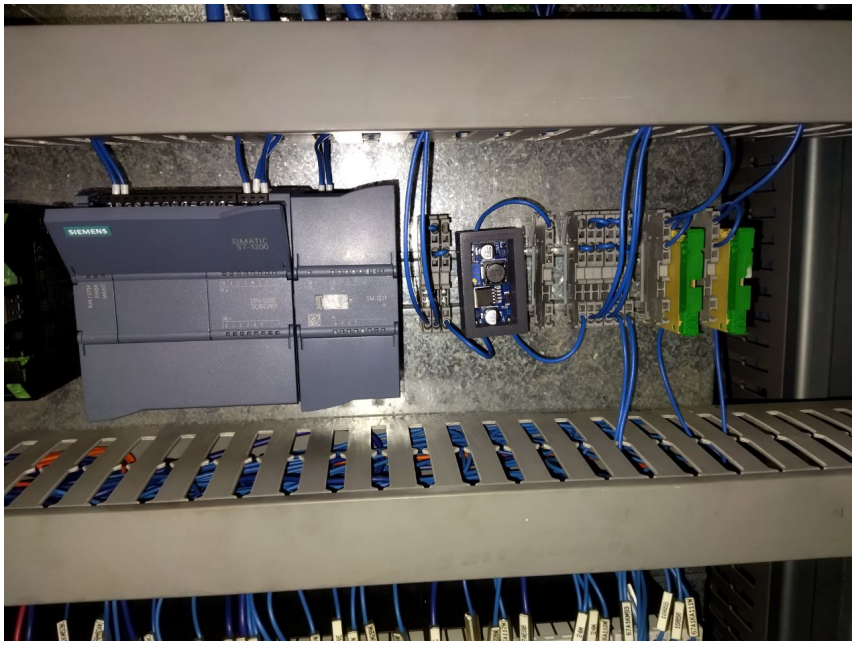
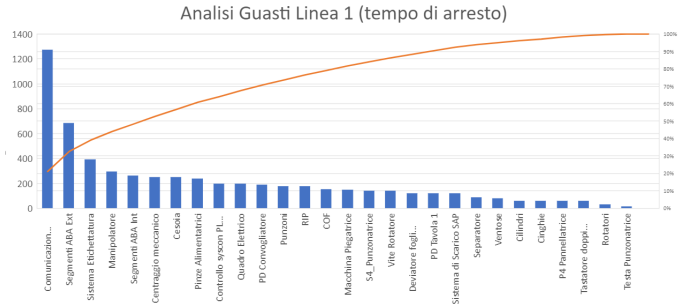
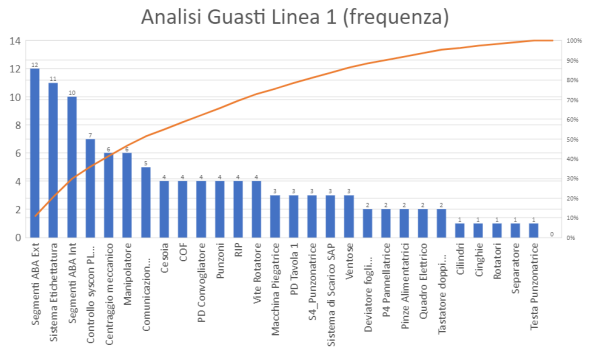


Figure 6.3: The PLC system installed in the electrical panel



(a) Downtime caused



(b) Frequency of the breakdown

Figure 6.4: Pareto diagrams from the breakdown analysis

the bending tools of the P4 machine, the labeling system and some software issues. The main problems concerning the punching-shearing machine were the manipulator and the shearing tool.

The main predictive maintenance project, in fact, analyzed in particular those components of the S4 1262 machine.

TPM smartphone app

I spent some days in my stay in KONE to develop a simple smartphone application that can collect the informations of an evaluation of the TPM work.
Specifically, TPM stands for Total Productive Maintenance, and it consists in a maintenance approach that aims to maximize the efficiency and effectiveness of production equipment. It focuses on preventive maintenance practices and it involves the entire organization, including also the production operators in the maintenance process.
In fact, an important part of the TPM work is to instruct the operators to do some autonomous maintenance. Nevertheless, a supervision of the maintenance team is needed, in order to reach a better result.
After that, this routine maintenance tasks have to be evaluated and the evaluation data need to be stored in a tidy way.
The evaluation task had been made in a paper form, with a difficult and untidy storing of the results.

So I made a simple app, using the Microsoft Powerapps© tool in order to collect all the evaluation data and save them in a database called Microsoft Sharepoint©.
The figure below (figure 6.5) shows the pages of the smartphone app and in the figure 6.6 you can observe how the data are collected in the database.

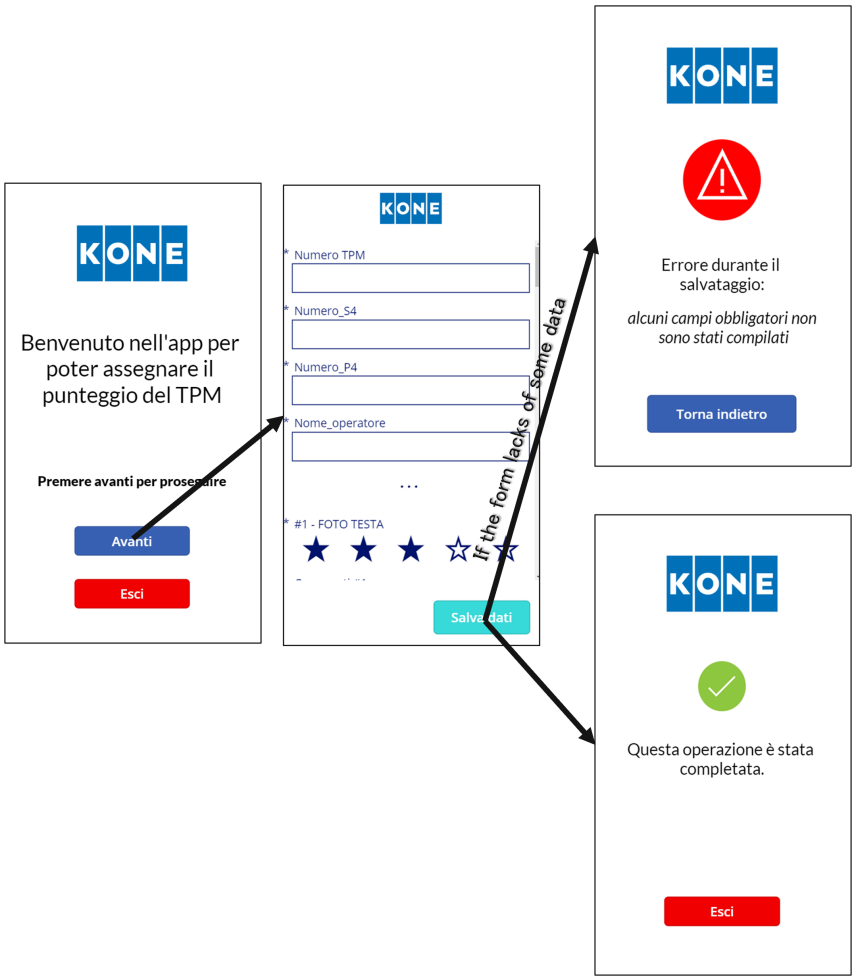


Figure 6.5: The different smartphone app screens

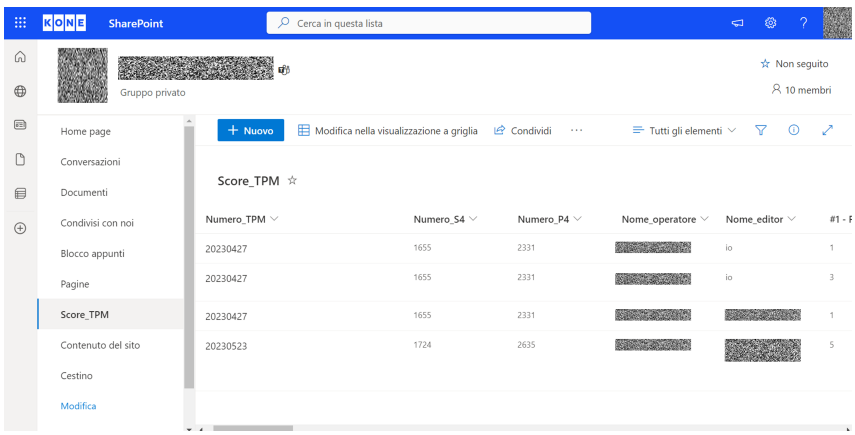


Figure 6.6: The database which include all the evaluation results

7 Conclusions and Future Works

In this chapter we can summarize all the obtained results of the project.

7.1 Conclusions

The predictive maintenance project has produced some results concerning several themes.
The results are listed below, and each of them will be discussed in more detail afterwards.

- S4 1262 most critical components;
- PLC system and its performance;
- optimal locations for conducting predictive maintenance;
- Machine Learning algorithms for peak recognition;
- wear level definition using the power spectrum analysis.

S4 1262 most critical components

In the Side Project chapter (chapter 6) we have discussed about the performed analysis on the most critical components of the line where the punching machine is placed. It resulted that the most problematic components of the entire line in the year 2022 are the bending tools of the Salvagnini P4 bending machine, the labeling system and some issues on communications between the PC and the operating machines.
Therefore, it's possible to notice that overall the punching machine is performing quite well.

However, going more in detail into the analysis of the punching machine alone, the components that have the greatest impact on the number of

maintenance interventions and downtime due to maintenance are the manipulator and the shearing tool.

Furthermore, it is important to consider that there are highly critical components that would require weeks of machine downtime and significant production losses if they would fail, such as the oil pumps or the machine head.
In fact, the project focused in particular on the machine head vibration analysis.

PLC system and its performance

We found out what a PLC system is and its limitations. In particular, we discovered that the PLC operates very well for robotic and automation purposes, working mainly with digital signals that can derive from photocells or other digital proximity sensors like the one used in the section 4.5.2.

Unfortunately it is not capable of performing complex mathematical operation, such as the Fast Fourier Transform, on its own. Additionally, the data transfer speed to a Python script is not enough for the analysis and processing of signals that exhibit high-frequency components in the power spectrum, like acceleration signals.

In conclusion, the PLC system is not suitable for our application, but it can be exploited using its modular capability and adding many sensors that can indicate the temperature of some components, such as electric motors, or the oil level in the tanks or, again, the level of waste bin filling using a weight sensor.

Optimal locations for conducting predictive maintenance

In order to perform a significant predictive maintenance projects, two factors need to be taken into consideration: the feasibility of placing the sensor in specific positions, and the identification of the most critical components in terms of cost and repair time.

So, we discovered the difficulty to put a sensor on the manipulator component and that the three main components to be monitored in the S4 1262

punching-shearing center are the manipulator, the machine head and the oil pumps.

Machine Learning algorithms for peak recognition

The first concept that I linked with the idea of predictive maintenance was the Machine Learning concept. Honestly, I thought that the Machine Learning world was simpler, at the contrary it is a large world with many different approaches that can be applied on the data in order to get the model to find relationships among the data.

Moreover, the application of a Machine Learning algorithm was not possible for our application, due to the impulsive nature of the vibrations.

In fact, a peak recognition algorithm was used and an analysis of the power spectrum of the vibrations generated by the shears was performed.

Wear level definition using the power spectrum analysis

The main concrete results of the project is the definition of the wear level based on the power spectrum of the response of the machine head structure to a shear impulse.

In particular, the results obtained shows that if the wear of the shearing tool increase, we can notice an increment of the amplitude in the frequency range between 300Hz and 500Hz and an inverse trend in the frequency range between 500Hz and 700Hz, as shown in the figure 5.44.

Unfortunately, the available dataset is relatively small, limiting the reliability of the observations done. Nevertheless it was impossible to collect a complete dataset due to the long useful lifetime of the shearing tool.

7.2 Future works

There are several ongoing areas of focus, especially the development of the predictive maintenance project for the S4 1262 punching-shearing center. In fact, the observations on the change in the power spectrum based on the level of wear of the shearing tool need to be validated. Moreover, the algorithm

have to be deployed on the STWIN.box sensor for real-time monitoring of shear wear conditions, exploiting the board's CPU for edge computing.

Additionally, the PLC project remains open for further development, with the potential to implement additional sensors to increase the dashboard's completeness and to make it provide more informative insights on the performance of every machine components.

Furthermore, the SensorTile.box project also requires complete development and improvement.
Specifically, it is necessary to search for and buy a durable USB cable of the Olflex© type, capable of sustaining the continuous stresses caused by manipulator movement.
Additionally, it is possible to study and implement a multi-threaded Python approach for simultaneous data collection and processing. However, it should be noted that data processing is time-consuming, and even the multi-threading solution may not be entirely satisfactory. As an alternative, we can consider deploying a dedicated firmware on the sensor itself for Mahalanobis distance computation; so to use Python only to create a dashboard and compare the inference Mahalanobis distances found in real-time with the reference dataset.

Algorithm that divide the acquired signal into many batches without overlapping

A

Code 26

```
def divide_into_windows(input_dataframe,samples_per_window):
    n_samples = input_dataframe.shape[0]
    time = input_dataframe['Time']
    N = int(time.size/samples_per_window)
    frames = [input_dataframe.iloc[ i*samples_per_window:
        (i+1)*samples_per_window].copy() for i in
        range(N+1) ]
    # remove last incomplete element
    frames_reduced = frames[:-1]
    return frames_reduced
```

B

SensorTile.box final script to define dashboard

Code 27 : dashboard generation

```
import pandas as pd
import numpy as np
import dash
from dash import dcc
from dash import html
from dash import dash_table
import plotly.express as px
import dash_bootstrap_components as dbc
import dash_daq as daq
import base64

### DEFINE TIMESTAMP ###
import datetime
ora = datetime.datetime.now()
timestamp_string = ora.strftime("%d/%m/%Y %H:%M:%S")

### OPEN WEBPAGE ###
import webbrowser
webbrowser.open('http://127.0.0.1:8000', new=2)

### DEFINE TXT FROM WHICH I NEED TO TAKE CLASSIF DATAS ###
txt_name = "...\\ Classificazione STwin.txt"
df_csv = pd.read_csv(txt_name, sep=" ")
df = pd.DataFrame({'data':  np.array(df_csv)[:,0],
     'ora':  np.array(df_csv)[:,1],
     'dev_STOP': np.array(df_csv['md_STOP_ref'] -
         df_csv['md_STOP']),
     'dev_MOV': np.array(df_csv['md_MOV_ref'] -
```

```
        df_csv['md_MOV']),
    'tipo':  np.array(df_csv)[:,6]})

### DEFINE TXT WHERE I TAKE ACC DATAS ###
txt_name_acc = "...  \\ Acquisition files STwin
        \\ism330dhcx_acc.txt"
df_acc = pd.read_csv(txt_name_acc, sep="\t")
df_time = np.array(df_acc)[1:,0]
df_time_label = np.array(df_acc)[0,0]
df_accx = np.array(df_acc)[1:,1]
df_accx_label = np.array(df_acc)[0,1]
df_accy = np.array(df_acc)[1:,2]
df_accy_label = np.array(df_acc)[0,2]
df_accz = np.array(df_acc)[1:,3]
df_accz_label = np.array(df_acc)[0,3]

### DEFINE ACCELERATION PLOT ###
fig_accx = px.line(x=df_time, y=df_accx, title='Acc_x ultima
acquisizione')
fig_accx.update_layout(
    xaxis_title='Tempo (sec)', yaxis_title='Acc_x (m\s2)',
    margin=dict(l=5, r=20, b=10, t=50, pad=5),
    font=dict(size=8), )
fig_accy = px.line(x=df_time, y=df_accy, title='Acc_y ultima
acquisizione')
fig_accy.update_layout(
    xaxis_title='Tempo (sec)', yaxis_title='Acc_y (m\s2)',
    margin=dict(l=5, r=20, b=10, t=50, pad=5),
    font=dict(size=8), )
fig_accz = px.line(x=df_time, y=df_accz, title='Acc_z ultima
acquisizione')
fig_accz.update_layout(
    xaxis_title='Tempo (sec)', yaxis_title='Acc_z (m\s2)',
    margin=dict(l=5, r=20, b=10, t=50, pad=5),
    font=dict(size=8), )

### DEFINE SERVER ###
app1 = dash.Dash(external_stylesheets=[dbc.themes.BOOTSTRAP],
```

```
    update_title=None)
app1.title = "ServerMicroSensoreST"
app1._favicon = ("sensor.ico")
app1.layout = html.Div([
dbc.Row([ # the dashboard includes 3 rows
    dbc.Row([ # first row include KONE logo
    dbc.Col([
    dbc.Row([ # Create timestamp
        dcc.Interval('timestamp-update',
            interval = 500, n_intervals = 0),
        html.Div(id='timestamp'), # Import "logo_kone"
        html.Div([
        html.Img(
            src="...url...",
            title="KONE Logo",
            alt="KONE Logo",
            width=70,
            style={'margin':'auto', 'display':'block'}),
        ], style={'width':'100%'}),
    ]), html.Hr(style={'color':'darkgrey'}), ]), ]),
dbc.Row([ # second row includes plots and history
    dcc.Interval('graph-update',
            interval = 2000, n_intervals = 0),
    html.Div([
    dbc.Row([
    dbc.Col([
    # Create circular indicator (green\red)
        daq.Indicator(
            value=True, # initialize as True (green)
            color = '#44D62C',
            id='classif_indicator1',
            size=30,)
    ]),
    dbc.Col([ # Create text box
        html.Div(id='anomaly_string'), ]),
]),
dbc.Row([
    html.Div([
```

```
    # Create image "Plot_md_MOV"
        html.Img( id="img_Plot_md_MOV" ),
    ], style={'width':'50%'}),
    html.Div([
    # Create image "Plot_md_STOP"
        html.Img( id="img_Plot_md_STOP"),
    ], style={'width':'50%'}),
]),
], style={'width':'60%'}), # Close Div"
html.Div([
# Create header "storico acquisizioni"
    html.H6('Storico acquisizioni'),
    # Create table
    dash_table.DataTable(id = 'table',
        data = df[::-1].to_dict('records'),
        columns=["name": i, "id": i for i in df.columns],
        fixed_rows='headers': True,
], ),
dbc.Row([ # third row includes acceleration plots
html.Div([
    dcc.Graph(id='accx_plot', figure=fig_accx,),
], style={'width':'30%', 'margin':'auto'}),
html.Div([
    dcc.Graph(id='accy_plot', figure=fig_accy,),
], style={'width':'30%', 'margin':'auto'}),
html.Div([
    dcc.Graph(id='accz_plot', figure=fig_accz,),
], style={'width':'30%', 'margin':'auto'}),
]),
]),
], style={'background-image': 'url(...url...)'})

# CALLBACK TO UPDATE TABLE "STORICO ACQUISIZIONI"
@app1.callback(
dash.dependencies.Output('table','data'),
[dash.dependencies.Input('graph-update', 'n_intervals')])
def updateTable(n):
    df_csv = pd.read_csv(txt_name, sep=" ")
```

```
    df = pd.DataFrame({'data':  np.array(df_csv)[:,0],
        'ora':  np.array(df_csv)[:,1],
        'dev_STOP': np.array( round( df_csv['md_STOP_ref']
- df_csv['md_STOP'] , 3) ),
        'dev_MOV': np.array( round( df_csv['md_MOV_ref'] -
df_csv['md_MOV'] , 3) ),
        'tipo':  np.array(df_csv)[:,6]})
    return df[::-1].to_dict('records')

# CALLBACK TO UPDATE IMAGES Plot_md_MOV E Plot_md_STOP
@app1.callback(
dash.dependencies.Output('img_Plot_md_MOV', 'src'),
dash.dependencies.Output('img_Plot_md_STOP', 'src'),
[dash.dependencies.Input('graph-update', 'n_intervals')])
def updateMOVSTOPimg(n):
    encoded_img_MOV = base64.b64encode(open(
        "...\\ Acquisition files STwin\\ Plot_md_MOV.png",
'rb')
        .read())
    encoded_img_STOP = base64.b64encode(open(
        "...\\ Acquisition files STwin\\ Plot_md_STOP.png",
'rb')
        .read())
return encoded_img_MOV.decode(),encoded_img_STOP.decode()

# CALLBACK TO UPDATE ACC PLOTS
@app1.callback(
dash.dependencies.Output('accx_plot', 'figure'),
dash.dependencies.Output('accy_plot', 'figure'),
dash.dependencies.Output('accz_plot', 'figure'),
[dash.dependencies.Input('graph-update', 'n_intervals')])
def updateAccGraph(n):
    df_acc = pd.read_csv(txt_name_acc, sep="")
    df_time = np.array(df_acc)[1:,0]
    df_accx = np.array(df_acc)[1:,1]
    fig_accx = px.line(x=df_time, y=df_accx)
    fig_accx.update_layout(
        margin=dict(l=5, r=20, b=10, t=50, pad=5),
```

```
            font=dict(size=8),
            pad=dict(l=30)) )
        df_accy = np.array(df_acc)[1:,2]
        fig_accy = px.line(x=df_time, y=df_accy)
        fig_accy.update_layout(
            margin=dict(l=5, r=20, b=10, t=50, pad=5),
            font=dict(size=8),
            pad=dict(l=30)) )
        df_accz = np.array(df_acc)[1:,3]
        fig_accz = px.line(x=df_time, y=df_accz)
        fig_accz.update_layout(
            margin=dict(l=5, r=20, b=10, t=50, pad=5),
            font=dict(size=8),
            pad=dict(l=30)) )
        return fig_accx, fig_accy, fig_accz

app1.run_server(port=8000, debug=False)
```

Code 28 : analysis of the SensorTile.box data

```
from HSD_link.HSDLink import HSDLink
import time, os
from threading import Thread, Event, main_thread
while True:
    hsd_link = HSDLink("stwin_hsd") # OPEN COMMUNICATION
    hsd_link.start_log(0)
    class SensorAcquisitionThread(Thread):
    def __init__(self, event, d_id, s_id, ss_id,
    sensor_data_file):
        Thread.__init__(self)
        self.stopped = event
        self.d_id = d_id
        self.s_id = s_id
        self.ss_id = ss_id
        self.sensor_data_file = sensor_data_file
        self.data_counter = 0
    def run(self):
        while not self.stopped.wait(0.1):
```

```
            sensor_data = hsd_link.get_sensor_data(
                self.d_id, self.s_id, self.ss_id)
        if sensor_data is not None:
            res = self.sensor_data_file.write(sensor_data[1])
            self.data_counter += sensor_data[0]
sensor_list = hsd_link.get_sub_sensors(d_id = 0)
threads_stop_flags = []
sensor_data_files = []
fold_input = hsd_link.get_acquisition_folder()
path_input=os.path.join("...\\Micro_Sensore_ST",fold_input)
for sensor in sensor_list:
    for i, ssd in enumerate(sensor.sensor_descriptor.
    .sub_sensor_descriptor):
    if sensor.sensor_status.sub_sensor_status[i].is_active:
        sensor_data_file_path = os.path.join(fold_input,(
            "_.dat".format(sensor.name,ssd.sensor_type)))
        sensor_data_file=open(sensor_data_file_path,"wb+")
        sensor_data_files.append(sensor_data_file)
        stopFlag = Event()
        threads_stop_flags.append(stopFlag)
        thread = SensorAcquisitionThread(stopFlag, 0,
            sensor.id, ssd.id, sensor_data_file)
        thread.start()
    else:
        print("-> (!)  Sensor _ is not active".format(
            sensor.name,ssd.sensor_type))
time.sleep(10) # ACQUIRE FOR 10 SECONDS
for sf in threads_stop_flags:
    sf.set()
for f in sensor_data_files:
    f.close()
hsd_link.stop_log(0)
del hsd_link # CLOSE COMMUNICATION

# MOVE ACQUISITIONS INTO A SPECIFIC DIRECTORY
import shutil, os
path_output = os.path.join("...\\Micro_Sensore_ST",
    "Acquisition files")
```

```
file_names = os.listdir(path_input)
for file_name in file_names:
    shutil.copy(os.path.join(path_input, file_name),
        path_output)
shutil.rmtree(path_input)
# ANALIZE ACQUIRED DATA
# calibrate them
import numpy as np
from HSD.HSDatalog import HSDatalog
hsd = HSDatalog(acquisition_folder=path_output)
g = 9.80665 # standard acceleration of gravity
df_LSM6DSOX_ACC = hsd.get_dataframe("LSM6DSOX", "ACC") * g
df_LSM6DSOX_ACC['Time'] = np.array(df_LSM6DSOX_ACC)[:,0] / g
# extract time and frequency details
n_samples = df_LSM6DSOX_ACC.shape[0]
time = df_LSM6DSOX_ACC['Time']
t_max = time[n_samples-1]
t_min = time[0]
dt = (t_max-t_min)/n_samples
fsamp = 1/dt
# add absolute acceleration as the column of dataset
import pandas as pd
abs_acc = np.sqrt(np.sum(np.square(
    np.array(df_LSM6DSOX_ACC)[:,1:]), axis=1))
A_abs = pd.DataFrame({'A_abs [m\s2]':  abs_acc})
df_LSM6DSOX_ACC = df_LSM6DSOX_ACC.join(A_abs)
# divide the dataset in batches of 512 samples each
from FunzioniDataAnalysis import divide_into_windows
input_dataframe = df_LSM6DSOX_ACC
samples_per_window = 512
frames_reduced = divide_into_windows(input_dataframe,
    samples_per_window)
# extract time features of the signal
from FunzioniDataAnalysis import time_feature_extraction
import time
Burg = 4
[time_features_df, Mean, STD, RMS,
Median, Max, Min, Energy, mag,
```

```
sma, q25, q75, iqr, Entropy,
ARCoeff, skewness, kurtosis, CF,
SNR, corrCoeff_xy, corrCoeff_xz, corrCoeff_yz] =
    time_feature_extraction(frames_reduced, Burg)

# find powerspectra of the signal
from FunzioniDataAnalysis import powerspectrum
samples_per_segment = samples_per_window # small nperseg ->
    better noise rejection BUT lower frequency resolution
x_lim = [0,fsamp/2]
[SpectralCentroid, freq_PS, PS, PSD] = powerspectrum(
    frames_reduced, samples_per_segment, fsamp, x_lim)
spectralCentroid_df = pd.DataFrame({SpectralCentroid})
# find frequency features of the signal
from FunzioniDataAnalysis import freq_feature_extraction
[freq_features_df, fMean, fSTD, fRMS,
fMedian, fMax, fMin, fEnergy, fq75,
fq25, fiqr, fEntropy, fARCoeff, fSkewness,
fKurtosis, fCF, fTHD, fSNR, fPS_maxInd] =
    freq_feature_extraction(frames_reduced,freq_PS,PS,Burg)
# find bandsEnergy of the powerspectra
from FunzioniDataAnalysis import bandsEnergy
band_number = 14      # number of section in which I divide
the frequencies of the power spectrum
bandsEnergy_df = bandsEnergy(frames_reduced,PS,band_number)
final_sec_f = time.time()

# get total dataframe (time+freq features)
fin_feature_df = time_features_df.join(spectralCentroid_df)
fina_feature_df = fin_feature_df.join(freq_features_df)
final_features_df = fina_feature_df.join(bandsEnergy_df)

# PCA and definition of moving_labels
from sklearn.preprocessing import StandardScaler, normalize
from sklearn.decomposition import PCA
# get features dataframe of the pre-recorder files
final_features_df_Dic_Feb = pd.read_csv(
    '...\\features_dataframe_ACC.txt', sep='\t')
```

```
# define moving_labels according to the STD value
moving_labels = np.where(final_features_df_Dic_Feb[
    'tSTD_x'] > 0.5, 1, 0)
moving_labels_df_Dic_Feb = pd.DataFrame({moving_labels})
features_df_KBEST = features_df.
    join(moving_labels_df)
# based on moving_labels, define the PCA of the pre-recorded
data
if features_df_KBEST['moving'].value_counts().get(1) != None:
    features_df_MOV = features_df[features_df_KBEST[
        'moving']==1] # working with mov
    X_scaled_MOV = StandardScaler().fit_transform(
        features_df_MOV)
    X_normalized_MOV=pd.DataFrame(normalize(X_scaled_MOV))
    reduced_data_MOV = PCA(n_components=2).fit_transform(
        X_normalized_MOV)
else:
    reduced_data_MOV = [[0],[0]]
if features_df_KBEST['moving'].value_counts().get(0) != None:
    features_df_STOP = features_df[features_df_KBEST[
        'moving']==0] # working with stop
    X_scaled_STOP = StandardScaler().fit_transform(
        features_df_STOP)
    X_normalized_STOP=pd.DataFrame(normalize(X_scaled_STOP
)
    reduced_data_STOP = PCA(n_components=2).fit_transform(
        X_normalized_STOP)
else:
    reduced_data_STOP = [[0],[0]]
    X_scaled = StandardScaler().fit_transform(features_df)
# working with the whole database
X_normalized = pd.DataFrame( normalize(X_scaled) )
reduced_data=PCA(n_components=2).fit_transform(X_normalized)
# ACTUAL ACQUISITION
# define moving_labels
moving_labels = np.where(features_df['tSTD_x'] > 0.4, 1, 0)
moving_labels_df = pd.DataFrame({'moving':  moving_labels})
features_df_KBEST = features_df.join(moving_labels_df)
```

```
# define the PCA of the pre-recorded data
    based on moving_labels
if 0 in features_df_KBEST['moving'].values:
    features_df_STOP = features_df[features_df_KBEST[
        'moving']==0]
    if len(features_df_STOP) > 1:
        X_scaled_STOP = StandardScaler().fit_transform(
            features_df_STOP)
    else:
        X_scaled_STOP = np.vstack((np.arange(
            features_df_STOP.shape[1])+1,np.array(
            features_df_STOP)))
    X_normalized_STOP=pd.DataFrame(normalize(X_scaled_STOP))
    reduced_data_STOP = PCA(n_components=2).fit_transform(
        X_normalized_STOP)
else:
    reduced_data_STOP = np.array( [[0,0],[0,0]] )
if 1 in features_df_KBEST['moving'].values:
    features_df_MOV = features_df[features_df_KBEST[
        'moving']==1]
    if len(features_df_MOV) > 1:
            X_scaled_MOV = StandardScaler().fit_transform(
                features_df_MOV)
        else:
            X_scaled_MOV = np.vstack((np.arange(
                features_df_MOV.shape[1])+1,np.array(
                features_df_MOV)))
        X_normalized_MOV=pd.DataFrame(normalize(
            X_scaled_MOV))
        reduced_data_MOV = PCA(n_components=2).
            fit_transform(X_normalized_MOV)
    else:
        reduced_data_MOV = np.array( [[0,0],[0,0]] )

if len(reduced_data_STOP)>len(reduced_data_MOV): tipo="STOP"
else:  tipo="MOV"

# Define the ELLIPTIC ENVELOPE
```

```
# MOV database
from sklearn.covariance import EllipticEnvelope
import math
el = EllipticEnvelope(store_precision=True,
     assume_centered=False,support_fraction=None,
     contamination=0.0075,random_state=0) # contamination
     is the quantity of outliers in the reference database
d_MOV = pd.DataFrame(reduced_data_MOV)
el.fit(d_MOV)
el_labels = pd.DataFrame({'Anomaly Y/N': el.predict(d_MOV)})
d_el_MOV = d_MOV.join(el_labels)
# Define the MAHALANOBIS DISTANCE
md_labels_MOV = pd.DataFrame({el.mahalanobis(d_MOV)})
d_md_MOV = d_MOV.join(md_labels_MOV)
# Compare Mahalanobis Dist of inference set with reference
d_inference_MOV = pd.DataFrame(reduced_data_MOV)
md_labels_inference_MOV=pd.DataFrame({el.mahalanobis(
     d_inference_MOV)})
d_md_inference_MOV = d_inference_MOV.join(
     md_labels_inference_MOV)
# get mean of the 10% of the max values of inference set
max_md_inference_MOV = np.flip(np.sort(el.mahalanobis(d_
     _inference_MOV)))[0:math.ceil(len(d_inference_MOV)/100)]
mean_md_inference_MOV = np.mean(max_md_inference_MOV)
# get mean of the 1% of the max values of reference set
max_md_MOV = np.flip( np.sort(el.mahalanobis(d_MOV)) )[
     0:math.ceil(len(d_MOV)/100)]
mean_md_MOV = np.mean(max_md_MOV)

# STOP database
el = EllipticEnvelope(store_precision=True,
     assume_centered=False,support_fraction=None,
     contamination=0.0075, random_state=0)
d_STOP = pd.DataFrame(reduced_data_STOP)
el.fit(d_STOP)
el_labels=pd.DataFrame({'Anomaly Y/N': el.predict(d_STOP)})
d_el_STOP = d_STOP.join(el_labels)
# Define the MAHALANOBIS DISTANCE
```

```
md_labels_STOP = pd.DataFrame({el.mahalanobis(d_STOP)})
d_md_STOP = d_STOP.join(md_labels_STOP)
# Compare Mahalanobis Dist of inference set with reference
d_inference_STOP = pd.DataFrame(reduced_data_STOP)
md_labels_inference_STOP = pd.DataFrame(
    {el.mahalanobis(d_inference_STOP)})
d_md_inference_STOP = d_inference_STOP.join(
    md_labels_inference_STOP)
# get mean of the 10% of the max values of inference set
max_md_inference_STOP = np.flip( np.sort(el.mahalanobis(d_
    _inference_STOP)))[0:ceil(len(d_inference_STOP)/100)]
mean_md_inference_STOP = np.mean(max_md_inference_STOP)
# get mean of the 1% of the max values of reference set
max_md_STOP = np.flip( np.sort(el.mahalanobis(d_STOP)) )[
    0:math.ceil(len(d_STOP)/100)]
mean_md_STOP = np.mean(max_md_STOP)

# SET GREEN-YELLOW-RED AREAS BASED ON 33% OF MOST
    DISTANT/NEAR POINTS IN REFERENCE DATABASE
from scipy.spatial import ConvexHull
import matplotlib.pyplot as plt

def encircle(x,y, ax=None, **kw):
    if not ax:  ax=plt.gca()
    p = np.c_[x,y]
    hull = ConvexHull(p)
    poly = plt.Polygon(p[hull.vertices,:], **kw)
# STOP database
d_flip = np.array( d_md_STOP )
df_100perc_STOP = d_flip
df_66perc_STOP = d_flip[(d_flip[:,2]>np.percentile(
        d_flip[:,2], 0)) & (d_flip[:,2]<=np.percentile(
        d_flip[:,2], 66))]
    df_33perc_STOP = d_flip[(d_flip[:,2]>np.percentile(
        d_flip[:,2], 0)) & (d_flip[:,2]<=np.percentile(
        d_flip[:,2], 33))]
    plt.figure()
    encircle(np.array(df_100perc_STOP)[:,0], np.array(
```

```
        df_100perc_STOP)[:,1], ec="k", fc="red", alpha=1)
    encircle(np.array(df_66perc_STOP)[:,0], np.array(
        df_66perc_STOP)[:,1], ec="k", fc="yellow", alpha=1)
    encircle(np.array(df_33perc_STOP)[:,0], np.array(
        df_33perc_STOP)[:,1], ec="k", fc="green", alpha=1)
    plt.scatter(np.array(d_md_STOP)[:,0], np.array(
        d_md_STOP)[:,1],c=d_md_STOP['Mahalanobis Distance'],
        cmap = 'coolwarm', s=2)
    plt.scatter(d_inference_STOP[0:math.ceil(len(
        d_inference_STOP)/10)].mean()[0], d_inference_STOP[
        0:math.ceil(len(d_inference_STOP)/10)].mean()[1],
        color='k', s=50)
    plt.title("STOPPED")
    plt.savefig("...\\Acquisition files\\Plot_md_STOP.png")

# MOV database
d_flip = np.array( d_md_MOV )
df_100perc_MOV = d_flip
df_66perc_MOV = d_flip[(d_flip[:,2]>np.percentile(
        d_flip[:,2], 0)) & (d_flip[:,2]<=np.percentile(
        d_flip[:,2], 66))]
df_33perc_MOV = d_flip[(d_flip[:,2]>np.percentile(
        d_flip[:,2], 0)) & (d_flip[:,2]<=np.percentile(
        d_flip[:,2], 33))]
plt.figure()
encircle(np.array(df_100perc_MOV)[:,0], np.array(
    df_100perc_MOV)[:,1], ec="k", fc="red", alpha=1)
encircle(np.array(df_66perc_MOV)[:,0], np.array(
    df_66perc_MOV)[:,1], ec="k", fc="yellow", alpha=1)
encircle(np.array(df_33perc_MOV)[:,0], np.array(
    df_33perc_MOV)[:,1], ec="k", fc="green", alpha=1)
plt.scatter(np.array(d_md_MOV)[:,0], np.array(d_md_MOV)[
    :,1],c=d_md_MOV['Mahalanobis Distance'],
    cmap = 'coolwarm', s=2)
plt.scatter(d_inference_MOV[0:math.ceil(len(
    d_inference_MOV)/10)].mean()[0], d_inference_MOV[0:math.
    ceil(len(d_inference_MOV)/10)].mean()[1],color='k',s=50)
plt.title("MOVING")
```

```
plt.savefig("...\\Acquisition files\\Plot_md_MOV.png")

# define the .txt file read by the datatable "Storico
    Acquisizioni" in CreateServer.py
actual_date = time.strftime("%d/%m/%Y", time.time())
actual_time = time.strftime("%H:%M:%S", time.time())
classif_df = pd.DataFrame(['data':  actual_date,
    'ora':  actual_time,
    'md_STOP': mean_md_inference_STOP,
    'md_STOP_ref':  mean_md_STOP,
    'md_MOV': mean_md_inference_MOV,
    'md_MOV_ref':  mean_md_MOV,
    'tipo':  tipo])
classif_df_file_path = '...\\Classificazione.txt'
classif_df.to_csv(classif_df_file_path, mode='a',
    index=False, header=False, sep=" ")

time.sleep(60) # wait 1 min, then restart while loop
```

C

Define webpage where the variable got with Python from the PLC is shown

Code 29

```
# Connect with PLC S7 1200
import snap7
import struct
IP = '192.168.100.1'
RACK = 0
SLOT = 1
DB_NUMBER = 1
START_ADDRESS = 0
SIZE = 2
plc = snap7.client.Client()
plc.connect(IP, RACK, SLOT)

# Prepare server importing the necessary libraries
import dash
from dash import dcc, html
import dash_bootstrap_components as dbc
import dash_daq as daq

# Define where to get data from, that is a database
    in the PLC's memory
db = plc.db_read(DB_NUMBER, START_ADDRESS, SIZE)
DI = struct.unpack('?',db[0:1])[0]

# Open webpage
import webbrowser
webbrowser.open('http://127.0.0.1:8150', new=1)

# Define server and dashboard on webpage
app1 = dash.Dash(external_stylesheets =
```

```
          [dbc.themes.BOOTSTRAP], update_title=None)
app1.layout = html.Div([
     # declare the dashboard updating time (1000ms)
     dcc.Interval('graph-update', interval = 1000,
               n_intervals = 0),
     daq.Indicator(
          id='indicator-1',
          value=DI,
          label="object far away",
          color="red",
     ), ])

# CALLBACK THAT UPDATE INDICATOR
@app1.callback([
     dash.dependencies.Output('indicator-1', 'label'),
     dash.dependencies.Output('indicator-1', 'color')],
     [dash.dependencies.Input('graph-update',
          'n_intervals')])
def updateTable(n):
     db = plc.db_read(DB_NUMBER, START_ADDRESS, SIZE)
     DI = struct.unpack('?', db[0:1])[0]
     if DI == 1:
          label = "object near"
          color = "green"
     else:
          label = "object far away"
          color = "red"
     return label, color
app1.run_server(port=8150, debug=False)
```

Datalog directly on the PC by means of Python and a USB cable

D

Code 30

```
from HSD_link.HSDLink import HSDLink
from HSD.HSDatalog import HSDatalog
import time, os

while True:
    print("--------\n")
    print(" Trying to connect with the STWIN.box\n")
    hsd_link = HSDLink().create_hsd_link() ### COMMUNICATION
WITH THE BOARD OPENED
    fold_input = hsd_link.get_acquisition_folder()
    print("--------\n")
    print(" Successfully connected with the sensor \n")
    threads_stop_flags = []
    sensor_data_files = []
    sensor_list = HSDLink.get_sensor_list(hsd_link,
0, only_active=True)
    HSDLink.start_log(hsd_link, 0) ### START THE ACQUISITION
    print("--------\n")
    print(" Data acquisition started \n")
    for s in sensor_list:
        HSDLink.start_sensor_acquisition_thread(hsd_link,
0, s, threads_stop_flags, sensor_data_files,
print_data_cnt = False)
    time.sleep(240) ### ACQUIRING FOR 4 MINUTES
    for sf in threads_stop_flags:
        sf.set()
    for f in sensor_data_files:
        f.close()
    HSDLink.stop_log(hsd_link, 0) ### STOP THE ACQUISITION
```

```
    time.sleep(1) # sleep useful to avoid bug
    del hsd_link ### COMMUNICATION WITH THE BOARD CLOSED
    print("--------\n")
    print(" Data acquisition finished \n")
```

Real time peak recognition exploiting the microphone acquired signal

E

Code 31 : filter the microphone signal

```
cutoff_freq = 2000.0 #Hz
filter_order = 2
fsamp_mic = fsampfinder(df_ISM330DHCX_MIC)
b, a = signal.butter(filter_order, cutoff_freq,
    btype='lowpass', fs=fsamp_mic)
filtered_df_ISM330DHCX_MIC = signal.filtfilt(b, a,
    df_ISM330DHCX_MIC.T.to_numpy()[1], method="gust")
```

Code 32 : trimming the acceleration signal to discard the last incomplete bucket

```
extra_samples = ( len(df_ISM330DHCX_ACC) - win_dim ) %
    (win_dim-overlap)
df_ISM330DHCX_ACC_trim = pd.DataFrame(
    df_ISM330DHCX_ACC.to_numpy()[:len(df_ISM330DHCX_ACC) -
    extra_samples,:]  )
len_df = ( (len(df_ISM330DHCX_ACC_trim)-win_dim) /
    (win_dim-overlap) )+1
```

Code 33 : scale win_dim in order to have same number of buckets in the microphone signal

```
win_dim_MIC_float = win_dim*len(filtered_df_ISM330DHCX_MIC)
    / len(df_ISM330DHCX_ACC.T.to_numpy()[3])
overlap_MIC_float = overlap*len(filtered_df_ISM330DHCX_MIC)
    / len(df_ISM330DHCX_ACC.T.to_numpy()[3])
win_dim_mic = int(win_dim_MIC_float)
overlap_mic = int(overlap_MIC_float)
filtered_df_ISM330DHCX_MIC_new = filtered_df_ISM330DHCX_MIC
    [0:int( len(df_ISM330DHCX_ACC.T.to_numpy()[3])
```

```
    * win_dim_MIC_float )]

extra_samples_mic = ( len(filtered_df_ISM330DHCX_MIC) -
    win_dim_mic ) % (win_dim_mic-overlap_mic)
df_ISM330DHCX_MIC_trim = pd.DataFrame(
    filtered_df_ISM330DHCX_MIC[:len(
    filtered_df_ISM330DHCX_MIC)-extra_samples_mic] )
len_df_mic = ( (len(df_ISM330DHCX_MIC_trim)-win_dim_mic)
    / (win_dim_mic-overlap_mic) )+1

win_dim_mic = round( win_dim_MIC_float*len_df_mic/len_df )
overlap_mic = round( overlap_MIC_float*len_df_mic/len_df )+1
```

Code 34 : define number of buckets that need to be analyzed

```
filtered_df_ISM330DHCX_MIC_new = filtered_df_ISM330DHCX_MIC[
    0:int( len(df_ISM330DHCX_ACC.T.to_numpy()[3]) *
    win_dim_MIC_float )]

extra_samples_mic_new = ( len(
    filtered_df_ISM330DHCX_MIC_new) - win_dim_mic ) %
    (win_dim_mic-overlap_mic)
df_ISM330DHCX_MIC_trim_new = pd.DataFrame(
    filtered_df_ISM330DHCX_MIC_new[:len(
    filtered_df_ISM330DHCX_MIC_new) -
    extra_samples_mic_new] )
len_df_mic_new = ( (len(df_ISM330DHCX_MIC_trim_new) -
    win_dim_mic) / (win_dim_mic-overlap_mic) )+1
len_df_for_iterations = round( len_df * len_df_mic_new
    / len_df_mic )
```

Code 35 : real-time peak recognition of the pre-filtered microphone signal

```
from scipy.signal import find_peaks
peaks_list_divided_mic = []
peaks_list_mic = []
for i in range(int(len_df_for_iterations)):
    if i == 0 :  # overlap unused in the first bucket
```

```
            begin = i
            end = begin + win_dim_mic # simulate real-time
                bucket with a length equal to win_dim_mic
            df_realtime_mic = df_ISM330DHCX_MIC_trim
                [begin:end].T.to_numpy()[0]
            peaks_mic, _ = find_peaks( df_realtime_mic,
                height=6500, distance=256 ) # peaks_mic are
                the indexes of the peaks found in the bucket
            if len(peaks_mic) > 0 :
                peaks_list_mic.extend(1*np.ones(win_dim_mic))
                peaks_list_divided_mic.append(1*np.ones(
                    win_dim_mic).astype(int))
            else :
                peaks_list_mic.extend(0*np.ones(win_dim_mic))
                peaks_list_divided_mic.append(0*np.ones(
                    win_dim_mic).astype(int))
        else :
            begin = end - overlap_mic
            end = begin + win_dim_mic
            df_realtime_mic = df_ISM330DHCX_MIC_trim[
                begin:end].T.to_numpy()[0]
            peaks_mic, _ = find_peaks( df_realtime_mic,
                height=6500, distance=256 )
            peaks_mic = [x for x in peaks_mic if x >
                overlap_mic] # remove peaks in the overlap
                (already taken in the previous bucket)
            if len(peaks_mic) > 0 :
                peaks_list_mic.extend(1*np.ones(
                    win_dim_mic-overlap_mic))
                peaks_list_divided_mic.append(1*np.ones(
                    win_dim_mic-overlap_mic).astype(int))
            else :
                peaks_list_mic.extend(0*np.ones(
                    win_dim_mic-overlap_mic))
                peaks_list_divided_mic.append(0*np.ones(
                    win_dim_mic-overlap_mic).astype(int))
```

Definitive real time peak recognition algorithm

F

Code 36

```
shears_rt = []
filtered_df_ACC_rt = []
for i in range(int(len_df)):
    if i == 0 :
        begin = i
        end = begin + win_dim
        df_realtime = df_ACC_trim.iloc[begin:end]
        fsamp = fsampfinder(df_realtime)
        filtered_timeserie_z_rt = filter_mine(df_realtime,
            fsamp)
        filtered_df_ACC_rt.extend(filtered_timeserie_z_rt)
        peaks_shears = peakfinder_mine(
            filtered_timeserie_z_rt, lim_min=0.8,
            action_range=2048)
        if len(peaks_shears) > 0 :
            shears_rt.extend(peaks_shears)
    else :
        # define overlap with previous window
        begin = end - overlap
        end = begin + win_dim
        df_realtime = df_ACC_trim.iloc[begin:end]
        fsamp = fsampfinder(df_realtime)
        filtered_timeserie_z_rt = filter_mine(df_realtime,
            fsamp)
        filtered_timeserie_z_rt = filtered_timeserie_z_rt[
            12:-12] # remove corrupted samples due to
            filtering process
        filtered_df_ACC_rt.extend(filtered_timeserie_z_rt)
```

```
            peaks_shears = peakfinder_mine(
                filtered_timeserie_z_rt, lim_min=0.8,
                action_range=2048)
            peaks_shears=[x if x in peaks_shears if x>overlap]
            if len(peaks_shears) > 0 :
                shears_rt.extend([x+begin+12 for x in
                    peaks_shears]) # add the removed samples
                    in order to synchronize recognized shears
                    with the signal
```

Real-time application code using find_peaks

G

Code 37

```
overlap = 2048-512 # with this value, only one-fourth of
    the array undergoes changes after each cycle.
win_dim = 2048 # bucket dimension

# trim the last incomplete bucket of df_ISM330DHCX_ACC
extra_samples = ( len(df_ISM330DHCX_ACC) - win_dim ) %
    (win_dim-overlap)
df_ISM330DHCX_ACC_trim = pd.DataFrame(
    df_ISM330DHCX_ACC.to_numpy()[:len(df_ISM330DHCX_ACC) -
    extra_samples,:]  )
len_df = ( (len(df_ISM330DHCX_ACC_trim)-win_dim) /
    (win_dim-overlap) )+1

def fsampfinder(df_ACC):
    n_samples = df_ACC.shape[0]
    time = np.matrix( df_ACC )[:,0]
    t_max = time[n_samples-1]
    t_min = time[0]
    dt = (t_max-t_min)/n_samples
    fsamp = 1/dt
    df = 1/t_max
    return fsamp

def filter_and_peakfinder(df_ACC, fsamp):
    from scipy import signal
    cutoff_freq = 550.0 # High pass filter at 550Hz
    filter_order = 8
    peak_timeserie_z = np.array(df_ACC)[:,3]
    b, a = signal.butter(filter_order, cutoff_freq,
        'highpass', fs=fsamp, analog=False)
    hpf_peak_timeserie_z = signal.filtfilt(b, a,
```

```
        peak_timeserie_z, method="gust")
    cutoff_freq = 650.0 # Low pass filter at 650Hz
    filter_order = 8
    b, a = signal.butter(filter_order, cutoff_freq,
        btype='lowpass', fs=fsamp)
    filtered_peak_timeserie_z = signal.filtfilt(b, a,
        hpf_peak_timeserie_z, method="gust")
    from scipy.signal import find_peaks
    ref_timeserie = -filtered_peak_timeserie_z # the first
        acceleration of the shear is downward,
        so i invert the signal to get it
    peaks, _ = find_peaks( ref_timeserie, height=0.8,
        distance=2048 )
    return ref_timeserie, peaks

    peaks_list_divided = []
    peaks_list = []
    peaks_list_punz = []
    peaks_list_notbinary = []
    peaks_list_notbinary_punz = []
    filtered_df_ISM330DHCX_ACC = []
    for i in range(int(len_df)):
        if i == 0 :
            # don't use overlap for the first bucket
            begin = i
            end = begin + win_dim
            # simulate real-time acquisition with
                win_dim=2048 and overlap=512
            df_realtime = df_ISM330DHCX_ACC_trim.iloc[
                begin:end]
            # find fsamp
            fsamp = fsampfinder(df_realtime)
            # filter and peak detection
            filtered_peak_timeserie_z_realtime, peaks,
                = filter_and_peakfinder(df_realtime,fsamp)
            filtered_df_ISM330DHCX_ACC.extend(
                filtered_peak_timeserie_z_realtime)
            # if a peak is found, 1*np.ones(win_dim) is
```

```
                added to the peaks list
            if len(peaks) > 0 :
                peaks_list.extend(1*np.ones(win_dim))
                peaks_list_divided.append(1*np.ones(
                    win_dim).astype(int))
                peaks_list_notbinary.extend(peaks)
            else :
                peaks_list.extend(0*np.ones(win_dim))
                peaks_list_divided.append(0*np.ones(
                    win_dim).astype(int))
        else :
            # add overlap with previous window
            begin = end - overlap
            end = begin + win_dim
            df_realtime = df_ISM330DHCX_ACC_trim.iloc[
                begin:end]
            fsamp = fsampfinder(df_realtime)
            filtered_peak_timeserie_z_realtime, peaks
                = filter_and_peakfinder(df_realtime,fsamp)
            filtered_df_ISM330DHCX_ACC.extend(filtered_
                _peak_timeserie_z_realtime[overlap:])
            # remove peaks in overlap, I picked them in
                the previous window
            peaks = [x for x in peaks if x > overlap]
            if len(peaks) > 0 :
                peaks_list.extend(1*np.ones(
                    win_dim-overlap))
                peaks_list_divided.append(1*np.ones(
                    win_dim-overlap).astype(int))
                peaks_list_notbinary.extend( [x+begin for
                    x in peaks] )
            else :
                peaks_list.extend(0*np.ones(
                    win_dim-overlap))
                peaks_list_divided.append(0*np.ones(
                    win_dim-overlap).astype(int))
```

Steps to define peakfinder_mine

H

- Step 1 : simple code that compares current value with neighbors

Code 38

```
def mine_peakfinder(ar):
    peaks = [] # initialize peaks list
    i = 0 # i correspond to the current value
    for num in ar:
        if i == 0:  # analyze first sample
            if num > ar[i+1]:
                peaks.extend([i])
        elif i == len(ar)-1:  # analyze last sample
            if num > ar[i-1]:
                peaks.extend([i])
        else :
            if num > ar[i+1] and num > ar[i-1]:
                peaks.extend([i])
        i = i+1 # move on to the next value
    return peaks
```

- Step 2 : add minimum limit, below which every local peak is discarded

Code 39

```
def mine_peakfinder(ar, lim_min=0):
    peaks = []
    i = 0
    for num in ar:
        if num > lim_min:  # add minimum limit
            if i == 0:  # analyze first sample
                if num > ar[i+1]:
                    peaks.extend([i])
            elif i == len(ar)-1:  # analyze last sample
                if num > ar[i-1]:
                    peaks.extend([i])
```

```
            else :
                if num > ar[i+1] and num > ar[i-1]:
                    peaks.extend([i])
        i = i+1
    return peaks
```

- Step 3 : add action range, that defines the interval in which i can obtain only one local peak

Code 40

```
def peakfinder_mine(ar, lim_min=-100000, lim_max=100000,
    action_range=1):
    peaks = []
    if lim_min == -100000:  # use mean as lim_min by default
        lim_min = ar.mean()
    i = 0
    for num in ar:
        # work within minimum and maximum limit
        if num > lim_min and num < lim_max:
            if i < action_range:
                action_range_arr = ar[:  i+action_range+1]
            elif i+action_range>len(ar)-1:
                action_range_arr = ar[i-action_range:]
            else:
                action_range_arr = ar[i-action_range:
                    i+action_range+1]
            # compare the current value with the whole
                action_range array
            if num >= max(action_range_arr):
                peaks.extend([i])
        i = i+1
    return peaks
```

Bibliography

[1] How can i communicate between a siemens s7-1200 and python? https://stackoverflow.com/questions/10355953/how-can-i-communicate-between-a-siemens-s7-1200-and-python/27466845#27466845, December 2015.

[2] Neural networks with gradient descent. https://blog.csdn.net/liuheng0111/article/details/52521050, September 2016.

[3] Classification algorithms for imbalanced datasets. https://www.blockgeni.com/classification-algorithms-for-imbalanced-datasets/, July 2020.

[4] How to implement a high-pass filter in python. https://dsp.stackexchange.com/questions/41184/high-pass-filter-in-python-scipy, November 2020.

[5] Comprehensive list of activation functions in neural networks. https://qastack.it/stats/115258/comprehensive-list-of-activation-functions-in-neural-networks-with-pros-cons, 2021.

[6] Arm education media's machine learning course. https://learning.edx.org/course/course-v1:ArmEducationX+EDARMXML.6x+2T2022/home, 2022.

[7] Scipy guide on scipy.signal.find_peaks. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.find_peaks.html, 2023.

[8] AltexSoft. Predictive maintenance: The complete guide. https://www.altexsoft.com/blog/predictive-maintenance/, March 2020.

[9] Saumya Awasthi. Unsupervised learning algorithms. https://dataaspirant.com/unsupervised-learning-algorithms/, January 2021.

[10] Richard Baraniuk. Signals and systems. https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Signal_Processing_

and_Modeling/Signals_and_Systems_(Baraniuk_et_al.), May 2023.

[11] Tim Brown. Predictive maintenance sensors and implementation. https://www.themanufacturer.com/articles/predictive-maintenance-sensors-and-implementation-a-solution-overview-from-dell/, January 2017.

[12] Jason Brownlee. One-class classification algorithms for imbalanced datasets. https://machinelearningmastery.com/one-class-classification-algorithms/, March 2020.

[13] Steven L. Brunton and J. Nathan Kutz. Data driven science and engineering machine learning, dynamical systems, and control, 2017.

[14] Nikhil Buduma and Nicholas Locascio. Fundamentals of deep learning - designing next-generation machine intelligence algorithms, May 2017.

[15] Lester Cardoz. Fault detection for predictive maintenance in industry 4.0. https://github.com/lestercardoz11/fault-detection-for-predictive-maintenance-in-industry-4.0, August 2020.

[16] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. Anomaly detection using mahalanobis distance. https://arxiv.org/abs/1802.06360v2, December 2021.

[17] Limble CMMS. A complete guide to predictive maintenance. https://limblecmms.com/predictive-maintenance/, March 2023.

[18] ST Community. Does anyone know how long the battery lasts? https://community.st.com/s/question/0D53W00001JAja9SAD/im-working-on-a-sensortilebox-for-the-first-time-id-like-to-create-an-app-that-records-steps-activity-type-and-axis-information-gyro-for-three-days-straight-would-appreciate-any-insight-for-a-newbie-here-questions-below, January 2022.

[19] SensiML Corporation. St sensortile box firmware download guide. https://sensiml.com/documentation/firmware/st-sensortile-box/st-sensortile-box.html, August 2020.

[20] Douglas F. Elliott. Handbook of digital signal processing, 1987.

[21] Simone Ferri. Sensors in predictive maintenance for industry 4.0. https://www.eletimes.com/sensors-in-predictive-maintenance-for-industry-4-0, March 2022.

[22] KONE Corporation Finland. Kone in brief. https://www.kone.com/en/company/, March 2023.

[23] Google. Introduction to neural networks. https://developers.google.com/machine-learning/crash-course/introduction-to-neural-networks/anatomy?hl=it, 2021.

[24] SPD Group. Predictive maintenance using machine learning. https://spd.group/machine-learning/predictive-maintenance/, July 2021.

[25] Growthbotics. Predictive maintenance using python and machine learning. https://growthbotics.medium.com/predictive-maintenance-using-python-and-b92261d3593b, November 2020.

[26] The MathWorks Inc. Predictive maintenance with matlab. https://www.mathworks.com/content/dam/mathworks/ebook/gated/predictive-maintenance-ebook-all-chapters.pdf, 2020.

[27] The MathWorks Inc. predictive maintenance tech talk series. https://fr.mathworks.com/videos/series/predictive-maintenance-tech-talk-series.html, September 2022.

[28] The MathWorks Inc. Spectal entropy of signal. https://fr.mathworks.com/help/signal/ref/pentropy.html, 2023.

[29] Romeo Kienzler. Create neural network anomaly detection. https://developer.ibm.com/learningpaths/iot-anomaly-detection-deep-learning/create-neural-network-anomaly-detection/, September 2022.

[30] Egor Korneev. Lstm neural networks for anomaly detection. https://medium.datadriveninvestor.com/lstm-neural-networks-for-anomaly-detection-4328cb9b6e27, January 2022.

[31] Chris McCormick. Mahalanobis distance. https://mccormickml.com/2014/07/22/mahalanobis-distance/, July 2014.

[32] Alan S. Morris. Measurement and instrumentation principles, 2001.

[33] Chris Murphy. Choosing the most suitable predictive maintenance sensor. https://www.analog.com/en/technical-articles/choosing-the-most-suitable-predictive-maintenance-sensor.html, February 2023.

[34] Pepperl+Fuchs. Inductive proximity switch data sheet. https://files.pepperl-fuchs.com/webcat/navi/productInfo/edb/mdoc4008b.pdf?v=20200316232212, September 2017.

[35] Plotly. Gauge charts in python with plotly. https://plotly.com/python/gauge-charts/, August 2020.

[36] Everything RF. What are pass band ripples in a filter? https://www.everythingrf.com/community/what-are-pass-band-ripples-in-a-filter, April 2021.

[37] Seonghan Ryu, Sangjun Koo, Hwanjo Yu, and Gary Lee. Out-of-domain detection based on generative adversarial network. https://aclanthology.org/D18-1077.pdf, November 2021.

[38] Salvagnini. Salvagnini s4 punching centers. https://www.salvagnini.com/en/product/punching-centers/s4, June 2015.

[39] Salvagnini. Operating manual - integrated punching and shearing system, January 2018.

[40] Siemens. Simatic et 200s im 151-8 pn/dp cpu operating instructions. https://cache.industry.siemens.com/dl/files/312/47409312/att_78874/v1/et200s_im151_8_pn_dp_cpu_operating_instructions_en-US_en-US.pdf, April 2022.

[41] Snap7. Snap7 - step7 ethernet communication suite. https://snap7.sourceforge.net/, September 2013.

[42] STMicroelectronics. St sensortile box. https://www.st.com/en/evaluation-tools/steval-mksbox1v1.html#documentation, December 2019.

[43] STMicroelectronics. Ism330dhcx - inemo inertial module. https://www.st.com/en/mems-and-sensors/ism330dhcx.html, November 2020.

[44] STMicroelectronics. St sensortile box data brief. https://www.st.com/resource/en/data_brief/steval-mksbox1v1.pdf, August 2020.

[45] STMicroelectronics. St sensortile box schematic. https://www.st.com/content/ccc/resource/technical/layouts_and_diagrams/schematic_pack/group1/bd/37/07/03/6d/87/40/cd/STEVAL-MKSBOX1V1_SCHEMATIC/files/steval-mksbox1v1_schematic.pdf/jcr:content/translations/en.steval-mksbox1v1_schematic.pdf, December 2020.

[46] STMicroelectronics. How to use the wireless multi-sensor development kit with customizable app for iot and wearable sensor applications. https://www.st.com/resource/en/user_manual/um2580-how-to-use-the-wireless-multi-sensor-development-kit-with-customizable-app-for-iot-and-wearable-sensor-applications-stmicroelectronics.pdf, September 2021.

[47] STMicroelectronics. St sensortile box quick start guide. https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/product_presentation/group0/5c/4e/96/c2/a6/98/4a/7f/FP-SNS-STBOX1_Quick_Start_Guide/files/FP-SNS-STBOX1_Quick_Start_Guide.pdf/jcr:content/translations/en.FP-SNS-STBOX1_Quick_Start_Guide.pdf, April 2021.

[48] STMicroelectronics. Stm32 ode function pack: Fp-sns-datalog1. https://www.st.com/content/st_com/en/products/embedded-software/mcu-mpu-embedded-software/stm32-embedded-software/stm32-ode-function-pack-sw/fp-sns-datalog1.html, February 2021.

[49] STMicroelectronics. Getting started with the stm32cube high-speed datalog function pack for stwin evaluation kits. https://www.st.com/resource/en/user_manual/dm00685314-getting-started-with-the-stm32cube-high-speed-datalog-function-pack-for-stwin-evaluation-kits-stmicroelectronics.pdf, December 2022.

[50] STMicroelectronics. High-speed datalog for stwin and sensortile.box. https://content.st.com/high-speed-datalog-for-stwin-and-sensortilebox-webinar.html, August 2022.

[51] STMicroelectronics. Vibration detection with lsm6dsox - part 1. https://community.st.com/s/article/Vibration-detection-with-LSM6DSOX---Part1, March 2022.

[52] STMicroelectronics. Getting started with the steval-stwinbx1 sensortile wireless industrial node development kit. https://www.st.com/en/evaluation-tools/steval-stwinbx1.html?rt=um&id=UM2965, March 2023.

[53] David Tax and Robert Duin. Support vector domain description. http://rduin.nl/papers/prl_99_svdd.pdf, February 2014.

[54] Nitish Kumar Thakur. Anomaly detection in python - part 1: Basics, code, and standard algorithms. https://medium.com/analytics-vidhya/anomaly-detection-in-python-part-1-basics-code-and-standard-algorithms-37d022cdbcff, May 2021.

[55] Venelin Valkov. Time series anomaly detection using lstm autoencoder with pytorch in python. https://curiousily.com/posts/time-series-anomaly-detection-using-lstm-autoencoder-with-pytorch-in-python/, March 2020.

[56] A. Vitali. Sensortile box hands-on. https://www.st.com/content/dam/AME/2019/technology-tour-2019/minneapolis/presentations/T3S1_Minneapolis_SensorTileBox_HandsOn_A.Vitali.pdf, September 2021.

[57] Elena Vodovatova. Unsupervised machine learning. https://theappsolutions.com/blog/development/unsupervised-machine-learning/, May 2023.

[58] Abhishek Wasnik. Principal component analysis (pca) example. https://www.askpython.com/python/examples/principal-component-analysis, October 2020.