POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Development of a low-power embedded solution for the real-time detection of structural cracks through acoustic emission

Supervisors

Prof. Stefano DI CARLO

Eng. Alessio CARPEGNA

Eng. Jonathan MELCHIORRE

Prof. Amedeo MANUELLO BERTETTO

Prof. Giuseppe MARANO

Candidate

Bálint BUJTOR

Graduation session of April 2025

Abstract

This thesis discusses the development of a low-power embedded system capable of performing real-time structural crack detection with acoustic emissions. Firstly, a panoramic analysis of structural health monitoring and acoustic emissions is given, highlighting the most important notions to understand the field. Alongside this study, the crucial and necessary civil engineering, and electronic concepts and components employed in this work are introduced. Building on this foundation the thesis embellishes the most transformative and decisive proceedings in the structural health monitoring field to understand the current state of the art, the most interesting areas of research, and future directions. Thereafter, a detailed description of the embedded solution's evolution is given, showcasing the hardware development's caveats, and the solutions for sampling the acoustic emission signal. During this phase, an analog amplifier circuit is designed to condition the raw signal for easier processing by the selected embedded microcontroller. Afterward, it details the firmware development that is capable of capturing the sensors' acoustic emissions and detecting the structural cracks. The constructed firmware transforms the analog signal into the digital domain, detects structural cracks, and transmits the obtained data through a serial connection to a Python client. Then, the Python script for receiving the information from the embedded platform and visualizing the data is introduced and examined. After the detailed presentation of the methodology, the proof of concepts and the results obtained with the system are presented. The developed embedded solution detects structural cracks generated on a marble block with the industry-standard pencil lead break test. It successfully amplifies, processes, transmits, and visualizes four concurrent acoustic emission signals from 4 sensors in real-time. A performance analysis is given with overall system accuracy and potential real-life applications. Finally, a conclusion is drawn and future development ideas are debated.

Acknowledgements

Ezt a szakdolgozatot a családomnak és a barátnőmnek szeretném ajánlani, különösen a nagyapáimnak, akik már nem lehetnek itt velem. Szeretném megköszönni a fáradhatatlan támogatásotokat, szereteteteket és útmutatásotokat. Köszönöm, hogy mindig számíthatok rátok és hogy mindig fordulhatok hozzátok tanácsért, akármibe is szeretnék kezdeni. Anya, szeretnék megköszönni mindent, amit értem tettél, és az áldozatokat, amiket hoztál, hogy most itt lehessek.

Chciałbym szczególnie podziękować mojej dziewczynie, za to, że była przy mnie przez te dwa lata, wspierała mnie, pomagała mi i znosiła moje niedogodności. Dziękuję za cierpliwość i czekam na nasz kolejny rozdział. Kocham cię, nie dałbym rady bez ciebie.

I sincerely appreciate the invaluable guidance and support of my supervisors and professors throughout my thesis. Their expertise and encouragement have been instrumental in shaping this work, and I am truly grateful for their contributions.

Infine, desidero esprimere la mia sincera gratitudine agli assistenti di laboratorio e ai responsabili del dipartimento LED per il loro prezioso supporto, la disponibilità e l'aiuto durante il mio lavoro.

"Talent is a divine blessing, but without incredible will and humility, it is worth nothing."
Steven Gerrard

Table of Contents

List of Tables

List of Figures

Acronyms

AC
alternating current

ADC
analog-to-digital converter

AE
Acoustic Emission

AIC
Akaike Information Criterion

AI
artificial intelligence

CNN
convolutional neural network

CMRR
common-mode rejection ratio

CWT
Continuous Wavelet Transform

DC
direct current

DL
deep learning

DMA

Direct Memory Access

ESD

electrostatic discharge

FRA

Frequency Response Analyzer

FW

firmware

GND

ground (reference point in electronic circuits - usually 0)

HPF

high-pass filter

HW

hardware

I2C

Inter Integrated Circuit Protocol

IC

integrated circuit

IDE

Integrated Development Environment

INA

Instrumentation amplifier

LPF

low-pass filter

ML

machine learning

MSPS

Mega Samples Per Second

NN

neural network

NDT

non-destructive testing

PLB

Pencil Lead Break

SAR

successive-approximation

SHM

structural health monitoring

SNR

Signal-Noise Ratio

SPI

Serial Peripheral Interface

SW

software

TOA

Time Of Arrival

UART

Universal Asynchronous Receiver-Transmitter

Chapter 1

Introduction

Ensuring public safety, life, and well-being is a cornerstone for communities worldwide. Hence, the requirement to spend more effort and money on preventing accidents and decreasing risks has become ever so important. In civil engineering, deteriorating structures pose a significant risk to human life, necessitating proactive measures to avoid accidents. Structural Health Monitoring (SHM) is crucial in safeguarding infrastructure such as buildings, bridges, pipelines, and aircraft to protect human life by detecting potential failures, in advance. SHM also contributes to preserving our cultural heritage by identifying fundamental flaws in historical landmarks.

Traditionally, as electronic, machine learning, and embedded solutions were not as advanced, engineers relied on techniques such as visual inspections or ultrasonic testing to assess the integrity of civil installations. Even though these methods are effective, they are time-consuming, expensive, and reactive—identifying damage only after it has occurred. As safety regulations become increasingly stringent, the need to detect microcracks, fatigue, corrosion, and other structural failures has grown.

Non-destructive testing (NDT) methods have revolutionized the field of SHM by enabling damage assessment without compromising structural integrity, thus decreasing costs and inspection time. Among these, acoustic emission (AE) stands out as an effective real-time monitoring technique. AE utilizes acoustic signals to monitor and detect structural flaws generated by physical deformation. Unlike traditional approaches, acoustic emission provides continuous, remote monitoring, allowing engineers to inspect edifices and other systems without harming the structure and respond immediately to emerging defects. This real-time capability enhances safety while reducing inspection costs by minimizing the need for frequent on-site evaluations.

This thesis develops an embedded solution for real-time structural crack detection using acoustic emission signals. Using the provided acoustic emission sensors, a

dedicated electronic circuit is designed to improve signal interpretability. An embedded microcontroller then translates the analog data to the digital domain to detect structural cracks in civil engineering structures. The proposed embedded platform offers a cost-effective alternative to the existing commercial products available on the market. The solution can also serve as a platform to generate acoustic emission data for future research applications.

The thesis begins with an overview of the necessary background knowledge in Chapter 2, covering fundamental concepts of SHM, acoustic emission, non-destructive testing methods, and electronics. Chapter 3 provides a comprehensive review of the state-of-the-art techniques in SHM, comparing different methodologies, technologies, and recent advancements in the field. Chapter 4 elaborates on the design and implementation of the embedded solution, detailing both hardware and software components, including sensor integration, signal processing techniques, and data analysis. Chapter 5 presents the experimental setup, data acquisition process, and analysis of findings, demonstrating the system's effectiveness in detecting structural cracks. Finally, Chapter 6 summarizes the key outcomes, discusses limitations, and suggests potential future improvements and research directions.

Chapter 2

Background

This chapter briefly introduces and describes the background information necessary to understand the topics mentioned in this thesis. It covers structural health monitoring, structural cracks, and acoustic emissions. From the electrical engineering side, it includes an introduction to piezoelectricity and signal amplifying. Finally, it gives an overview of the commonly used methods to detect and classify structural cracks. It also provides a solid base for understanding machine learning methods and the chosen models. The goal of this chapter, however, is not to discuss state-of-the-art techniques and technologies as a collection of those methods is given in 3.

2.1 Structural health monitoring

Structural health monitoring (SHM) is an approach for periodically monitoring physical structures, such as bridges, buildings, or airplanes [1] and reporting changes in critical parameters reliably to identify damages [2, 3, 4]. Damage can be defined as changes to the material and/or the system's geometry, negatively affecting the system's performance [5]. Doing so makes it possible to ensure that the structure maintains its functionality and prevents fatal events. Long-term monitoring is needed because systems degrade over time and use. Short-term monitoring is employed for rapid status screening when catastrophic events such as earthquakes, typhoons, or other damages happen [6]. SHM is an approach that does not aim to replace traditional health monitoring methods like inspections but to complement them [4].

To perform SHM, sensors are placed in the critical positions of the system to be monitored to report the selected metrics and variables. The SHM process consists of choosing the variables to be monitored and their locations, specifying the types and number of sensors placed on the system and designing the data

acquisition pipeline. This pipeline includes signal acquisition, signal conditioning, and preprocessing on the edge device on the system (or close to it). After this, the edge device transmits the data to the cloud or central device where the data interpretation happens. The data can be interpreted with classical or statistical models, machine learning, or deep learning methods.

One of the most common methods to extract information about the structure's health is based on correlating the monitored attributes' vibration amplitude and frequency [7]. Another popular methodology is to monitor waves propagated in the structure [3], [8].

During the evolution of SHM, several key observations have been made. Worden et al. [9] summarized these takeaways into axioms presented in 2007. Some of the most interesting are:

- Axiom II: the assessment of damage requires a comparison between two system states;
- Axiom III: Identifying the existence and location of damage can be done in an unsupervised learning mode, but identifying the type of damage present and the damage severity can generally only be done in a supervised learning mode;
- Axiom IVa: Sensors cannot measure damage. Feature extraction through signal processing and statistical classification is necessary to convert sensor data into damage information;
- Axiom IVb: Without intelligent feature extraction, the more sensitive a measurement is to damage, the more sensitive it is to changing operational and environmental conditions;
- Axiom VI: There is a trade-off between the sensitivity to damage of an algorithm and its noise rejection capability.

These axioms outline the properties such a system needs to include and key SHM design guidelines. Axiom II supports the notion that continuous monitoring is necessary, while Axiom IV states the methods to identify and classify damages.

In the context of monitoring buildings, the goal of SHM can be:

1. to detect the damage
2. to locate the damage
3. to categorize the damage's type
4. to evaluate the severity of the damage

These goals can provide vital information for engineers to decide the correct course of action, maintain the structure's health, prevent disastrous events, and protect human life.

2.2 Structural cracks

One of the common reasons for structural crack formation is fatigue. Fatigue in material science is defined as the formation and propagation of progressive damage in the material over time due to cyclical loading and use [10]. Fatiguing has different stages: crack initiation, growth, and propagation, and once the material reaches its fatigue limit, an irreversible crack forms [11, 12]. Other causes of crack formation can be the application of loads whether they are applied cyclically or not.

Cracks in materials can be classified according to different criteria. Regarding cracks in concrete, Chitte et al. [13] state that thin, medium, or wide cracks can be distinguished according to their sizes.

Classification	**Crack width**
Thin	<1mm
Medium	1mm - 2mm
Wide	>2mm

Table 2.1: Crack classification based on the crack's width, according to [13]

Another option to categorize cracks in concrete is based on their effect on the building [14]. Structural cracks stem from incorrect design, flawed construction, or overloading. They may threaten the building's safety. Non-structural cracks mostly appear due to internally induced stresses in the materials, and even though they might look unappealing, they do not endanger the safety of the construction. Causes can be various, for instance, drying shrinkage or plastic shrinkage [15]. Structural cracks are more dangerous for the structure's environment and the people around and inside it, so this type of crack requires more care. For this reason, the focus will be directed towards this type of failure during this work.

2.3 Acoustic emission

Acoustic emission (AE) is a phenomenon where energy gets released from a local source inside a solid material, producing transient elastic waves [16]. The phenomenon happens when an irreversible change occurs on the microscopic scale, for instance, a crack formation or plastic deformation due to use, aging, or external stress. Acoustic emission wavelengths are usually in the range of 1kHz to 1MHz [17].

Acoustic emissions are carried through elastic waves or stress waves. Elastic waves are mechanical waves carried in elastic or viscoelastic materials. Some highly elastic or viscoelastic materials are tendons, wood, rubber, and polymers. Ultimately, every material is somewhat elastic and, thus, can be a carrier of elastic

waves. Seismic waves are a special type of elastic waves that occur in the Earth due to an earthquake.

Acoustic emissions provide a non-destructive way to monitor the formation and status of structural cracks from the earliest stages, which gives civil and mechanical engineers a tool to monitor structures' health (more details in section 2.1). Using acoustic emissions, it is possible to determine the location of the source from one receiver out of many or from triangulation and cross reference, characterize the source mechanism [18], evaluate the carrying materials and structures, and monitor the operation of the structure.

2.4 Methods for structural crack detection

There are several ways to detect and classify structural cracks, but we can point out two main groups. Statistical methods or machine learning methods are based on selected attributes of the data that are descriptive of a crack and can be used to train a statistical model to detect when such a crack happens.

The other leading group does not use statistics but achieves the goal using formulas and equations that the industry has developed over the years [19]. This section aims to give a general overview of the possibilities one can choose from when designing an inference algorithm, while Chapter 3 highlights the latest proceedings and the state-of-the-art detection algorithms.

2.5 Crack localization

Structural crack localization using acoustic emission (AE) signals involves detecting and pinpointing the origin of stress waves emitted by growing cracks within materials. A prevalent algorithm employed for this purpose is the time difference (TD) method, which calculates the differences in arrival times of AE signals at multiple sensors to triangulate the crack's location [20]. According to the algorithm presented in the paper, the crack's location can be determined using two acoustic emission sensors. However, the paper only attempts to locate the crack along a straight line, with the propagation speed of the elastic wave being known. If one wants to locate the crack in three dimensions not knowing the speed, 4 sensors are needed. The algorithm used to determine the crack's location is as follows as per [21]. The time for a stress wave to travel from the source located at x_0, y_0, z_0 to the piezoelectric (PE) sensor located at x_A, y_A, z_A, with a given speed c can be calculated as 2.1.

$$T_A = \frac{\sqrt{(x_0 - x_A)^2 + (y_0 - y_A)^2 + (z_0 - z_A)^2}}{c} \tag{2.1}$$

This equation however cannot be used to determine the crack's location as the absolute time T_A is not known. For this reason, the time differences have to be used at each sensor, denoted as Δt_A. Equation 2.1 can be rewritten by subtracting Δt_R - the arrival time of the crack to a reference sensor. Then the equation 2.2 gives the time difference between the arrival time to sensor A and sensor R.

$$\Delta t_A = T_A - T_R = \frac{\sqrt{(x_0 - x_A)^2 + (y_0 - y_A)^2 + (z_0 - z_A)^2}}{c} - T_R \tag{2.2}$$

It can be seen that there are 4 unknowns in this equation, namely x_0, y_0, z_0 and c, confirming the need for 4 sensors.

Accurate determination of the onset time of these signals is crucial for precise localization. However, challenges arise due to background noise and the need for automated detection in continuous monitoring systems. Various techniques, including dynamic thresholding and artificial intelligence-based approaches, have been developed to enhance onset time detection and improve localization accuracy [21].

Advancements in Bayesian methodologies also offer robust frameworks for localizing AE sources in complex structures, providing probabilistic mappings that enhance the reliability of crack detection [22].

2.6 Piezoelectricity

The piezoelectric effect happens when a positive electric charge accumulates on one side of a non-conducting crystal and negative on the other when the material is subject to mechanical strain, such as compression or flexion. The phenomenon was discovered by Pierre and Jacques-Jules Curie (husband of Marie Curie) in 1880 [23].

The piezoelectric effect stems from the rearrangement of charges within the material. In a non-piezoelectric material, the overall charge within the material cancels out even if the material is subject to mechanical strains. Meanwhile, due to the unique crystal-like structure of piezoelectric materials, the overall charge distribution becomes imbalanced, and the material will resemble a dipole. Piezoelectric materials usually have a hexagonal or tetragonal crystal structure that allows them to behave in the way mentioned above. When the material is deformed, the relative position of the charges changes, tipping the balance of the net charge distribution. Hence, the material will have a measurable charge. Similarly, when the substance is subject to an electric charge, the conducting particles get pulled to the opposite pole, deforming the material. The phenomenon can be seen in Figure 2.1.

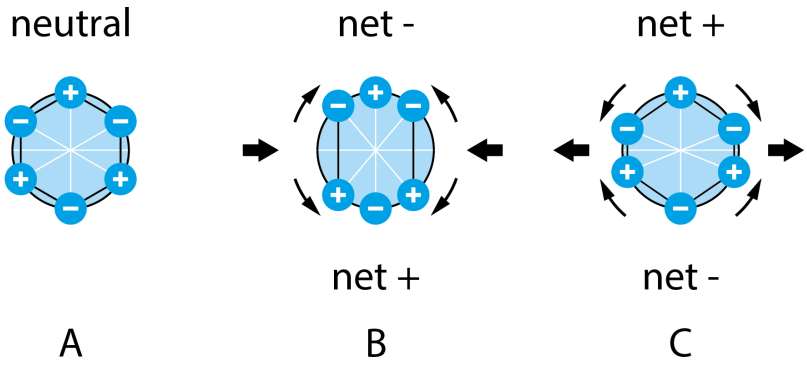


Figure 2.1: Material deformation due to the piezoelectric effect. Source: [24]

A unique property of piezoelectricity is that it is reversible, meaning that the same material can generate charge when subjected to mechanical strain and charge when electricity is channeled through it. Several materials, both natural and human-made, possess piezoelectric effects, and they can be grouped into three categories: naturally occurring crystal substrates (e.g., quartz, bones, topaz), ceramics with perovskite structure (e.g., lead zirconate titanate, barium titanate) and polymers [25].

This allows engineers to create a variety of devices based on the piezoelectric effect. Piezoelectric sensors and actuators both utilize this phenomenon. The most employed application in everyday life is the electric lighter, which employs a quartz crystal as a spark source when a spring-loaded hammer hits it. Other applications are piezoelectric sensors that measure acceleration, force, or vibration in the material they are attached to. On the other hand, piezoelectric actuators incorporate loudspeakers and piezoelectric motors.

2.7 Signal amplifiers

Signal amplifiers are analog electronic devices that multiply the input signal's magnitude by a configurable value called gain. They are one of the most commonly used building blocks in analog circuits. Amplifiers most commonly have two inputs, one output, and two additional inputs for the power supply.

Amplifiers can be controlled by different physical quantities, namely current and voltage. The controlled output property can also be either current or voltage, yielding four different types of amplification, detailed in Table 2.2. This table contains the ideal values for the input and output impedances. In real life, achieving them is impossible, however, one can construct equivalent or nearly equivalent circuits to obtain a behavior close to the ideal one.

Amplifier type	Input	Output	Input impedance	Output impedance	Gain units
Current	I	I	0	0	Unitless
Transresistance	I	V	0	∞	Ohm
Transconductance	V	I	∞	0	Siemens
Voltage	V	V	∞	∞	Unitless

Table 2.2: Amplifier circuit types

Amplifiers have several important properties to consider when designing such a device. Some of the most important ones are listed below:

- Gain: the ratio between the magnitude of the output over the input signal
- Bandwidth: the usable frequency range
- Slew rate: the output's maximum rate change

A negative feedback loop is one of the most common ways to configure a signal amplifier. This technique increases the bandwidth, reduces the signal distortion, and regulates the gain at the expense of slightly decreasing the maximum achievable gain. Essentially, due to the negative feedback, any excess signal or unwanted noise is fed back to the input with an opposite sign (unlike the input signal), eliminating it from the outputted signal. The negative feedback also stabilizes the amplifier's operating point against minor disturbances in the power supply line.

Amplifiers must include an active device that performs the signal amplification. Nowadays, this device is typically a transistor, a three-port electrical device, as opposed to vacuum tubes used in the past. The transistor is either a bipolar junction transistor (BJT) or a metal oxide semiconductor field-effect transistor (MOSFET), depending on the design needs. Both options have their advantages and downsides, but it is outside the scope of this thesis to discuss them.

Operational amplifiers typically have more than one transistor per circuit and are widely used in analog electronics as gain blocks. These devices usually have a very high loop gain and a differential output.

Differential amplifiers are usually constructed from several operational amplifiers, providing a very high voltage gain on their output. As its name suggests, a differential amplifier multiplies the voltage difference on its input ports by an adjustable value. They are frequently used to drive an ADC's input.

Amplifier circuits can be categorized based on their power supplies as well. Single-supply and dual-supply amplifiers can be distinguished, both having pros

and cons. A dual-supply amplifier requires two power supplies, $+V_{dd}$ and $-V_{dd}$, them being the positive and the negative supply voltage of the circuit.

The dual-voltage setup is desirable because it can conserve and amplify both the positive and the negative portion of a zero-centered AC signal. Such a configuration helps the designer because it can directly amplify the signal without shifting it to keep both parts of the input signal before feeding it to the amplifier stage. A downside of this setup is that the designer needs to provide both $+V_{dd}$ and $-V_{dd}$ to the circuit, which is not trivial in some use cases.

On the other hand, single-supply amplifiers only need the positive supply voltage ($+V_{dd}$) to operate. The other supply terminal is usually connected to the ground (GND), allowing the designer to place the amplifier in circuits that cannot deliver negative supply voltage. On the flip side, to preserve the negative portion of a zero-centered AC signal, it needs to be adequately shifted by a value not to cut off signal parts.

Amplifiers can be built in various configurations. Some of the most common architectures will be detailed in the following paragraphs. Furthermore, other arrangements relevant to this thesis's scope will also be introduced.

2.7.1 Inverting operational amplifier

One of the most often used amplifier arrangements is the inverting operational amplifier. An inverting amplifier circuit has two additional resistances to set the closed-loop gain: one between the input and the inverting terminal of the amplifier and the other between this terminal and the output terminal 2.2. The amplifier is called inverting because it always produces the inverse amplification of the input signal. The two resistances usually referred to as feedback resistance R_f and input resistance R_{in} determine the amplification according to Formula 2.3. As there are two tunable parameters, one can be chosen freely, and it is the designer's responsibility to select them properly. This configuration is ideal when the phase of the signal is less important. Another application of this configuration is the voltage follower when the two resistances are removed. The circuit acts as a signal stabilizer and noise remover in this case.

$$Gain(A_v) = -\frac{V_{out}}{V_{in}} = -\frac{R_f}{R_{in}} \tag{2.3}$$

2.7.2 Non-inverting operational amplifier

The other basic architecture of operational amplifiers is the non-inverting one. Unlike the previous example, this amplifier does not change the signal's phase. An essential difference to the inverting amplifier is that this construction can only

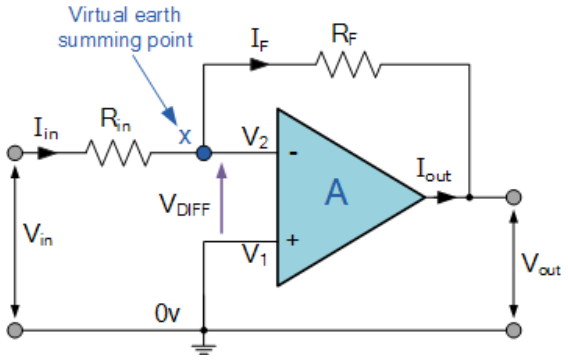


Figure 2.2: Inverting operational amplifier. Source: [26]

have a gain greater than 1. This architecture directly connects the input to the non-inverting input terminal. Two resistors are connected to the inverting terminal, the feedback resistance R_f connecting it with the amplifier's output and another resistance connecting it with the ground. An inverting amplifier design can be seen in Figure 2.3. The gain of the amplifier can be calculated with Equation 2.4. Non-inverting amplifiers are practical when it is vital to conserve the phase of the signal.

$$Gain(A_v) = \frac{V_{out}}{V_{in}} = 1 + \frac{R_f}{R_2} \tag{2.4}$$

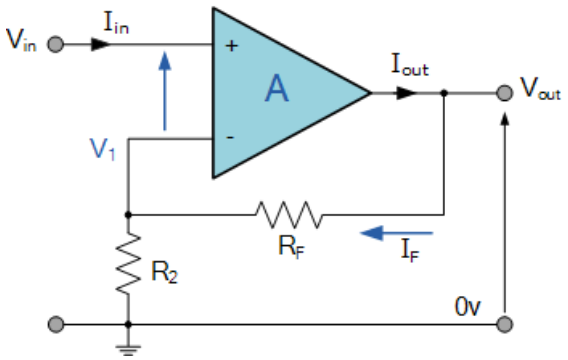


Figure 2.3: Non-inverting operational amplifier. Source: [27]

2.7.3 Summing amplifier

A summing amplifier is another popular structure used when multiple signals must be summed together. In its essence, it is most similar to an inverting amplifier, except that more input signals are fed to its inverting terminal through identical

(or, in exceptional cases, not identical) input resistances R_f. The architecture can be seen in 2.4, and the circuit's gain is specified in Equation 2.5. In this design, the output signal is proportional to the sum of the input signals. This circuit adds and amplifies more than one signal in one stage. In case every input signal is fed through identical input resistances, every signal gets amplified by the same value. However, changing these resistances to achieve different amplification for different signals before adding them together is possible.

$$V_{out} = -\frac{R_f}{R_{in}}(V_1 + V_2 + V_3) \tag{2.5}$$

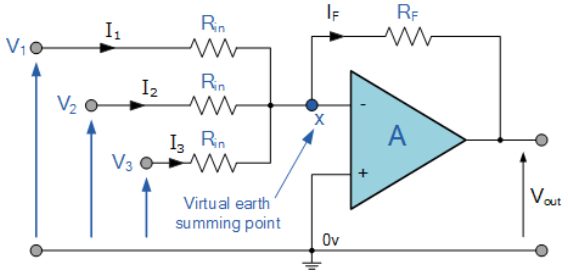


Figure 2.4: Summing operational amplifier. Source: [28]

2.7.4 Differential amplifier

Another structure to consider is the differential amplifier that produces an output proportional to the input difference. It is basically a subtractor-amplifier circuit, having four different resistances, denoted from 1-4, visible in 2.5. The output can be computed using Equation 2.6.

$$V_{out} = -V_1\frac{R_3}{R_1} + V_2\left(\frac{R_4}{R_2 + R_4}\right)\left(\frac{R_1 + R_3}{R_1}\right) \tag{2.6}$$

If resistances $R_1 = R_2$ and $R_3 = R_4$ Equation 2.6 becomes 2.7, revealing more clearly that the output is proportional to the difference of the inputs.

$$V_{out} = \frac{R_3}{R_1}(V_2 - V_1) \tag{2.7}$$

2.7.5 Instrumentation amplifier

Instrumentation amplifiers (INAs) are special differential amplifiers with high differential amplification and an outstanding common-mode rejection ratio (CMRR). These devices offer high input impedance and low output impedance and are

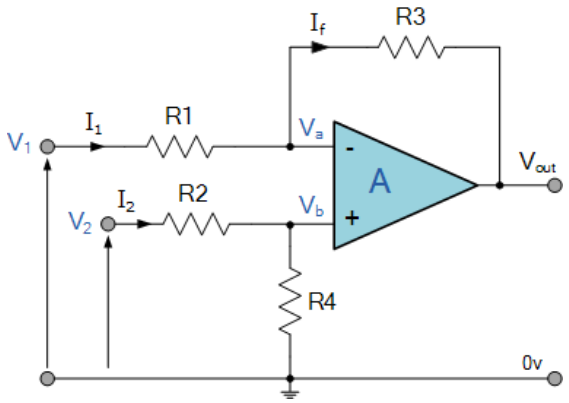


Figure 2.5: Differential operational amplifier. Source: [29]

commonly used in applications where small error margins are required. These devices can generally be constructed by three discrete operational amplifiers or bundled into one IC. In the former case, two amplifiers act as input buffer amplifiers, whilst the last is a generic differential amplifier discussed in the previous point. Moreover, the architecture usually involves an adjustable (discrete) resistance to set the circuit's gain that should be connected between two dedicated terminals. A typical architecture can be seen in 2.6. In case every resistance is equal ($R_1 = R_2 = R_3$) in the circuit except for the gain resistance (R_{gain}), the amplification of the circuit can be given with Equation 2.8.

$$Gain(A_v) = \frac{V_{out}}{V_2 - V_1} = \left(1 + \frac{2R}{R_{gain}}\right) \tag{2.8}$$

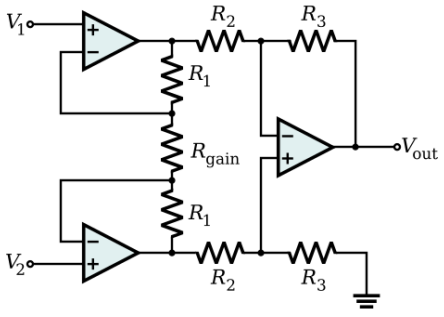


Figure 2.6: Instrumentation amplifier. Source: [30]

2.7.6 Charge amplifier

The charge amplifier is probably the most essential architecture for the scope of this thesis, as this is the topology responsible for processing the PE sensor's signal in the final circuit. A charge amplifier has an integrator design, having a capacitor C_f in the feedback loop that produces a voltage proportional to the current on its inverting terminal. It also encloses a resistor R_f in the feedback loop, parallel with the capacitor intended to prevent the amplifier from saturation by discharging it over time. Without it, the DC gain of the circuit would also be very high, so even a tiny DC offset would appear highly amplified on the output. Therefore, it is important to accurately select the feedback resistor value as it sets the circuit's lower frequency limit and the feedback capacitor according to Equation 2.9. On top of these blocks, an input resistance is also planned to protect the operational amplifier from overcurrent in case of an ESD strike. The overall architecture can be seen in Figure 2.7.

$$f_l = \frac{1}{2\pi C_f R_f} \tag{2.9}$$

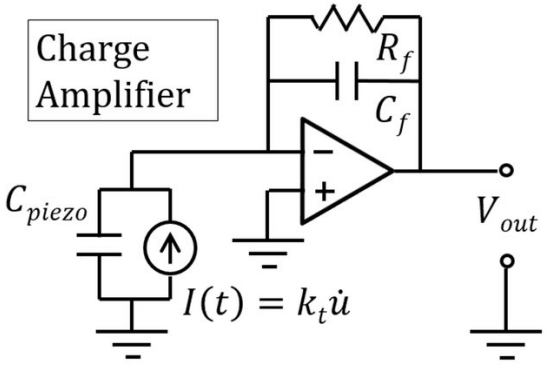


Figure 2.7: Charge amplifier architecture. Source [31]

2.7.7 Low-pass filter

The last circuit mentioned in this section is the low-pass filter with an operational amplifier. Compared to a simple RC low-pass filter it includes a unit-gain operational amplifier. This extra component results in better noise rejection and stability compared to an RC circuit without such an op-amp. The two parametrizable values of the circuit are the capacitor's value (C) and the resistor's value (R). The cutoff frequency of the amplifier can be computed using the same equation as the charge amplifier (Equation 2.9). The architecture can be seen in Figure 2.8.

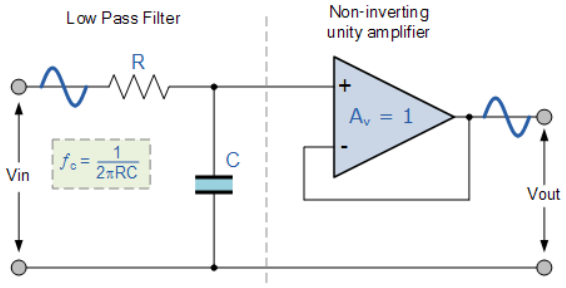


Figure 2.8: Low-pass filter with unit-gain operational amplifier. Source [32]

2.8 Machine learning

Machine learning (ML) has emerged as one of the most transformative fields in computer science, revolutionizing diverse industries and research domains. The origins of machine learning can be traced back to the mid-20th century when researchers began exploring the potential of computational systems to simulate human intelligence. Alan Turing's seminal work on the concept of a "learning machine" laid the foundation for this field [33]. Early advancements included the development of perceptrons by Rosenblatt in 1958, marking one of the first attempts at creating an artificial neural network [34]. During the following decades, the field experienced a series of peaks and valleys, with significant breakthroughs occurring in the 1980s and 1990s, such as the backpropagation algorithm for training neural networks [35].

Recent decades have witnessed unprecedented growth in machine learning capabilities, driven by advancements in computational power, data availability, and algorithmic innovation. Deep learning, a subset of machine learning characterized by multi-layered neural networks, emerged as a dominant paradigm, achieving state-of-the-art performance in image recognition, natural language processing, and other domains. Milestones such as AlexNet's success in the ImageNet competition in 2012 [36] and the development of transformer architectures [37] have further firmed ML's role in tackling complex problems. Moreover, reinforcement learning techniques have demonstrated remarkable capabilities, exemplified by systems like AlphaGo [38] and advancements in robotics.

Machine learning can be categorized into supervised, semi-supervised, unsupervised, and reinforcement learning. Supervised learning involves training models on labeled data, making it particularly effective for classification and regression tasks. On the other hand, unsupervised learning aims to find patterns or structures in unlabeled data, with clustering and dimensionality reduction being common applications. Semi-supervised learning connects these approaches by leveraging labeled and unlabeled data. Finally, reinforcement learning enables agents to

make sequential decisions by interacting with an environment to maximize rewards. These areas, supported by continuous research and innovation, form the backbone of modern machine learning.

Some of the most prominent ML algorithms relevant to this thesis are presented in the following subsections.

2.8.1 Principal Component Analysis

Principal Component Analysis (PCA) is a widely used unsupervised learning technique for dimensionality reduction, transforming high-dimensional data into a lower-dimensional representation while keeping as much variance as possible [39]. By identifying orthogonal components that capture the maximum variance in the data, PCA enables improved learning and computational efficiency, making it essential for preprocessing in fields such as computer vision and bioinformatics.

2.8.2 Support Vector Machines

Support Vector Machines (SVMs) are powerful supervised learning algorithms that are particularly effective for classification and regression tasks [40]. SVMs operate by finding an optimal hyperplane that maximally separates classes in a high-dimensional feature space, often using kernel functions to handle non-linear separations. Their versatility and effectiveness have made them a standard choice for text classification and image recognition applications.

2.8.3 Random Forest

The Random Forest classifier is an ensemble learning method that combines multiple decision trees to achieve higher predictive accuracy and robustness [41]. Aggregating the outputs of individual trees mitigates overfitting and enhances generalization. Random Forests have been successfully applied in domains ranging from healthcare to finance for disease prediction and fraud detection tasks.

2.8.4 Decision Tree

Decision Tree classifiers are intuitive and interpretable models that recursively partition data based on feature values [42]. Constructing a hierarchical tree structure, they predict by traversing paths defined by simple decision rules. While prone to overfitting in their basic form, they remain popular for their ease of implementation and transparency in decision-making processes.

2.9 Neural Networks

Building upon the foundational concepts of machine learning, neural networks represent a class of models inspired by the interconnected structure of biological neurons. These models have driven remarkable progress in processing complex data and solving challenging tasks across domains such as computer vision, speech recognition, and natural language processing.

Following the success of AlexNet in 2012, [36], another significant advancement came with the development of GoogleNet and its inception architecture [43]. GoogleNet's inception modules allowed the model to process features simultaneously at multiple scales, achieving high accuracy with fewer parameters than earlier architectures. This innovation set a new standard for efficiency in neural network design.

ResNet (Residual Network) marked another milestone, addressing the degradation problem encountered in deep networks [44]. By introducing skip connections, ResNet enabled extremely deep architectures, allowing superior performance in tasks such as image recognition and object detection. Its residual learning framework has become a cornerstone in modern neural network design.

More recently, the transformer architecture revolutionized natural language processing by introducing an attention mechanism that enables models to focus on relevant parts of input sequences [37]. Unlike recurrent models, transformers process sequences simultaneously, offering remarkable scalability and performance. This architecture serves as the basis of state-of-the-art systems like Chat-GPT and Claude.

2.9.1 U-Net

U-Net is a convolutional neural network architecture initially designed for biomedical image segmentation but has since been widely adopted in various fields, including structural health monitoring (SHM). The network features a U-shaped structure with an encoder-decoder design that captures local and global features. The encoder compresses the input data to extract high-level representations. At the same time, the decoder reconstructs spatially precise outputs by combining these representations with features from earlier layers through skip connections [45]. This architecture makes U-Net well-suited for tasks requiring pixel-wise predictions, such as damage localization and crack detection in SHM.

Chapter 3

Related Works

Structural Health Monitoring with passive sensory devices has established itself as one of the more popular choices in the industry. This technology gained momentum as a non-destructive method capable of detecting and localizing defects. This property is beneficial when SHM has to be performed on historical buildings and architectural heritage whose structures are fragile.

The goal of crack detection with acoustic emission is twofold. The first, more trivial task is to determine the onset time of the acoustic emission and localize the crack in space, while the second aims to classify the crack's type. To determine the exact location of the crack, the onset time of the crack must first be specified in the data recorder by the acoustic emission sensor.

The onset time of a crack can be determined as the first time the crack arrives at the piezoelectric sensor [46]. The precision of the onset time is crucial in building a high-quality acoustic emission pipeline. Over the years, many algorithms have localized, distinguished, and categorized structural cracks. These algorithms may rely on different triangulation techniques. The most straightforward algorithms are based on numerical techniques using optimization methods such as the Least Squares Method (LSM) [47]. Carpinteri et al. [48] and Ohtsu et al. [49] both employ a moment tensor analysis to determine the crack's orientation, direction, and modality. For Carpinteri et al.'s algorithm, six sensors are needed. These techniques relied on the manual detection of onset times, however, later on, automatic onset detection algorithms have been developed such as [50, 51].

3.1 Classical methods

One of the more popular groups of methods to determine a crack's onset time is the group of threshold methods. These methods employ static or dynamic thresholding to determine the onset crack's onset time. The basic idea of static methods is to

use a threshold to distinguish when the signal changes from noise to a crack. The most evident solution monitors the crack's amplitude to achieve this goal [52, 53]. The main obstacle to overcome in this method group is finding the proper trade-off between too-low and too-high thresholds. In the prior case, one risks setting the threshold to a moment when the signal is only noise. In the latter scenario, one is at risk of losing helpful information about the signal by determining an onset time that is later than the actual one.

To enhance the performance of static methods, dynamic threshold methods utilize an algorithm capable of adjusting the threshold in runtime. This approach allows us to adapt to the signal amplitude over time and employ the same algorithm in different real-world scenarios without touching the algorithm or calibrating it beforehand. Dynamic threshold methods include using Short Term Average and Long Term Average (STA/LTA) to achieve optimal selection [54].

In 2016, Bai et al. [50] introduced three new methods for determining onset time based on dynamic thresholds. They introduced a time-varying correlation method based on cross-correlation and a surrogate significance test. The second introduced method is a Continuous Wavelet Transform (CWT) [55] based correlation method. The third method is a CWT-based binary map that encapsulates the AE signal's time-frequency response, which is filtered through a median filter. They concluded that new methods outperformed the baseline AIC method.

The onset time can also be determined using markers different than the signal's amplitude. Hirotugu Akaike introduced a criterion named after him, the Akaike Information Criterion (AIC) [56] that can be utilized to determine a signal's onset time. The AIC is essentially a statistical method capable of determining the onset time. The AIC states that a complete solution of an autoregressive moving average model can be obtained using the Markovian representation of the process, commonly referred to as AIC picker in the literature [57, 21]. It is stated that a signal can be adequately split into a set that contains the data points before the onset time and another set that includes the data points after the onset time [51]. Later on, the AIC picker algorithm was enhanced by Carpinteri et al. [58]. Their method is based on the accuracy level of the AE signals, which can be estimated with the second derivative of the AIC function, and on another parameter based on the AE signal's propagation velocity.

Time-domain statistical methods such as the Hinkley criterion [59] can also be employed in the SHM domain. This method computes the partial energy of a time-series signal for all samples, with the global minimum of the partial energy function indicating the onset time. Furthermore, time-frequency domain approaches have also been developed. One such method involves cross-correlating the signal with a short Gaussian pulse at a specific frequency [60]. The onset time at that frequency corresponds to the peak of the correlation function. Another method of Ciampa and Meo [61] uses the Continuous Wavelet Transform (CWT) [55]. This

approach identifies the maximum squared modulus of the CWT coefficients. The time corresponding to this maximum is then used as a reference for the Time Difference of Arrival measurements, providing an alternative to traditional methods for detecting the signal's onset time.

In addition to AIC-based methods, other approaches exist to determine the onset time of a signal. For signals with low signal-to-noise ratios (SNR), a fractal dimension-based method has been introduced [62]. This algorithm analyzes variations in the fractal dimension along the signal trace, demonstrating high accuracy even in the presence of significant noise. However, it is computationally intensive, making it unsuitable for real-time applications such as the one presented in this thesis.

3.2 ML-based methods

The methods introduced in the previous paragraphs have found a vast territory of applications over the years, among which the SHM domain with AE was a proponent participant. Even though their potential and concepts were tried and proven, real-life applications of these algorithms have revealed several drawbacks and weaknesses. Many thresholding methods have expressed inaccuracies and failures when a significant amount of noise is present in the environment, especially when the noise and the signal characteristics are relatively similar. Another weakness of these algorithms is that they are often statically configured, i.e., the threshold cannot be adapted to a particular real-life scenario. Moreover, some algorithms require significant computational time, which makes them unfit for real-time processing.

Enhancements and evolutions have been designed for these algorithms to remedy the aforementioned points. With the breakthrough of machine learning algorithms, however, researchers' attention has shifted towards adapting and applying these algorithms to the SHM field.

Perhaps one of the first published examples of using a machine-learning-based method to solve an SHM task was written by Emanian et al. [63]. The technique is a two-stage process. Firstly, it eliminates background noise in the AE signal using covariance analysis, principal component analysis (PCA), and differential time delay estimation [64]. Secondly, the remaining data is processed using a self-organizing map (SOM) neural network in combination with short-time Fourier Transformation (STFT) that separates the noise and the AE signal.

Another early example of using machine learning approaches to classify acoustic emission signals was presented by Omkar et al. in 2002 [65]. They used fuzzy c-means (FCM) clustering to classify AE signals to different sources of signals. FCM can discover the clusters in the data even if the boundaries overlap because

FCM is distribution-free.

Ince et al. [66] used pairwise correlation of AE waveforms and hierarchical clustering to enhance the SNR by generating "super" AEs, which improve time-of-arrival estimation. Key features such as wavelet packets, autoregressive parameters, and Fourier coefficients are extracted to identify P-wave patterns in noisy conditions. A support vector machine (SVM) classifier with probabilistic outputs is employed to reliably detect P-wave arrivals, demonstrating the approach's effectiveness in noisy environments for AE localization.

Zhang et al. [67] compared three machine learning algorithms — extreme learning machine (ELM), decision tree classifier (DTC), random forest classifier (RFC), and deep belief network (DBN) — for classifying ambient and ultrasonic signals. They also proposed continuous wavelet transform (CWT) as preprocessing to eliminate waveform distortion, enhancing detection accuracy. Results showed that RFC outperformed DBN and DTC, providing the highest classification accuracy. The proposed method was compared with the traditional autocorrelation function and demonstrated higher accuracy and lower errors across 300 waveforms.

In 2020, Chen et al. [68] introduced a method for onset time detection using deep learning models, particularly Convolutional Neural Networks (CNN). The approach utilizes the short-time Fourier transform for feature extraction, which serves as the input for the CNN classifier.

In 2022, Zonzini et al. [69] demonstrated that deep learning models outperform traditional methods for onset time detection in boisterous environments. They compared the accuracy of a Convolutional Neural Network and a capsule neural network [70] with the classical Akaike Information Criterion (AIC). The results showed that deep learning models achieved a tenfold improvement in accuracy.

In 2023, Melchiorre et al. [21] highlighted the robustness of deep learning models for onset time detection. A Convolutional Neural Network (CNN) designed for Sound Event Detection (SED) [71] was trained on normalized seismic accelerograms and later tested on Acoustic Emission (AE) signals. The models accurately identified onset times.

In 2024, Melchiorre et al. [72] demonstrated the successful use of the U-net neural network, specialized for segmentation tasks. In their study, they used the network to determine the onset time of the AE signal as a one-dimensional segmentation task. They demonstrated the network's superior performance on PLB tests compared to traditional methods.

Another research team that used the U-net network to solve crack detection tasks was Yu et al. [73]. They improved the original U-net architecture to achieve the performance of the task at hand. The network was named Residual Linear Attention U-net (RLAU-net). They deduced that the network improves processing time while maintaining the high detection accuracy of the original architecture.

Convolutional networks are not the only deep learning algorithms that can solve

arrival time estimation tasks in the AE signal domain. In 2017, Zheng et al. [74] presented a recurrent neural network architecture to tackle arrival time estimation. They used a Long-Short-Term Memory architecture. Their experiments tested the network's performance on data under different SNR levels. They found that the network detected the arrival time comfortably, even in noisy environments.

In 2021, Nguyen et al. [75] assembled a multi-step architecture to predict the remaining useful lifetime (RUL) for concrete structures to prevent failures and extend useful life. The deterioration process is modeled through a health indicator (HI), automatically constructed using a stacked autoencoder deep neural network (SAE-DNN). The authors built a novel hit removal process employing a one-class support vector machine (OC-SVM), filtering relevant hits for training. The constructed health indicators are then used to train a long short-term memory recurrent neural network (LSTM-RNN), leveraging its ability to capture long-term dependencies. The proposed method outperforms schemes without hit removal and alternative models like GRU-RNN and standard RNNs.

The models demonstrated high accuracy in identifying onset times. Applying machine learning and deep learning techniques has proven effective in advancing the field of AE. These methods have shown remarkable robustness and the ability to enhance accuracy in detecting onset times in acoustic emission signals. However, research in this area is ongoing, as current methodologies face challenges such as the computational and storage demands of the algorithms and the reliance on data preprocessing to improve SNR. Additionally, these techniques require signal segmentation, making preprocessing essential to define suitable signal windows for classification by machine learning algorithms.

Chapter 4

Methodology

This chapter explains the details of the implementation followed during the platform's development phase. First, the acoustic emission sensor is introduced. Then, the hardware development is detailed, followed by the firmware development and the test setup.

4.1 Sensor characterization

The sensor used for this thesis is a piezoelectric sensor specifically developed to measure acoustic emissions. A piezoelectric sensor is a passive device commonly used in structural health monitoring applications [76] that produces an electric charge proportional to its measured mechanical quantity. These instruments are based on the piezoelectric effect, which generates charge in certain materials under mechanical stress, such as acceleration, pressure, temperature, or force 2.6. The detector is produced by *Lunitek Srl*, an Italian company specializing in manufacturing seismic sensors and structural health monitoring systems.

Several measurements were performed to determine the sensor's behavior and characteristics before designing a circuit capable of transforming the analog signal to digital.

Firstly, the frequency response was controlled to verify that the device could be modeled as a capacitive sensor, typical of most piezoelectric devices. The modeling is based on the following notion. As it was introduced in the background section 2.6, piezoelectric sensors generate electric charge proportional to the mechanical stress they experience. It is only sensible to model this phenomenon with a current source. The current of this source shall be proportional to the charge generated over a unit of time.

$$I = \frac{dQ}{dt}$$

As the sensor's piezoelectric layer is connected to the rest of the circuit through two electrodes, it is reasonable to model this by employing a capacitance connected in parallel to the current source. At this point, it is possible to compute the output voltage generated by the charge in this intermediate equivalent circuit with the following equation:

$$V = \frac{1}{C}\int Idt = \frac{1}{C}\int \frac{dQ}{dt} = \frac{Q}{C} \tag{4.1}$$

If a more realistic representation is desired, a final resistance can be added parallel to the previous circuit to model the leakage of the current over time. Thus, we arrive at the final equivalent circuit of a typical piezoelectric sensor 4.1.

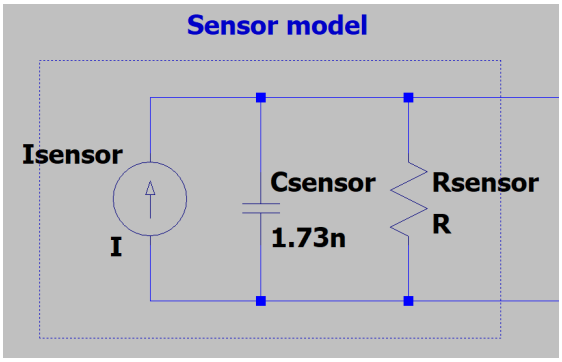


Figure 4.1: Piezoelectric equivalent circuit model

It must be verified that the circuit possesses the aforementioned capacitive characteristics. The verification was performed with a Frequency Response Analyzer (FRA) between 10 Hz and 1 kHz. The instrument confirmed the device's theoretical behavior; the results can be seen in Figure 4.2. At lower frequencies, the device behaves like an exceptionally high-impedance - an open circuit. It can be seen in the figure that as the testing frequency increases, the circuit's impedance continuously diminishes, resembling more and more of a short circuit.

Afterward, to get the approximate value of the sensor's capacitance, another measurement was conducted with an automatic passive component analyzer at 15 kHz to validate the behavior at a frequency higher than checked with the previous device. The measurement returned 1.773 nF, which was in line with the expectations. Results can be seen in Figure 4.3.

Now that a preliminary idea of the sensor's behavior was constructed, the sensor's response had to be verified against a typical PLB test. The sensor was tested by directly connecting it to an oscilloscope via a BNC-BNC cable to understand the voltage generated by the sensor in such a test. A typical response can be seen in Figure 4.4. After several experiments, the maximum peak of the sensor's response

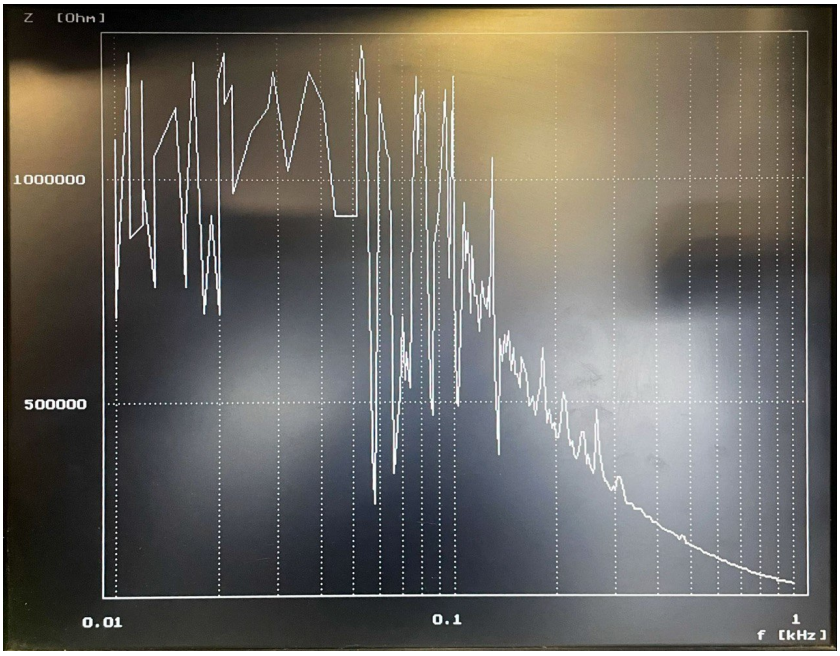


Figure 4.2: The AE sensor's frequency response

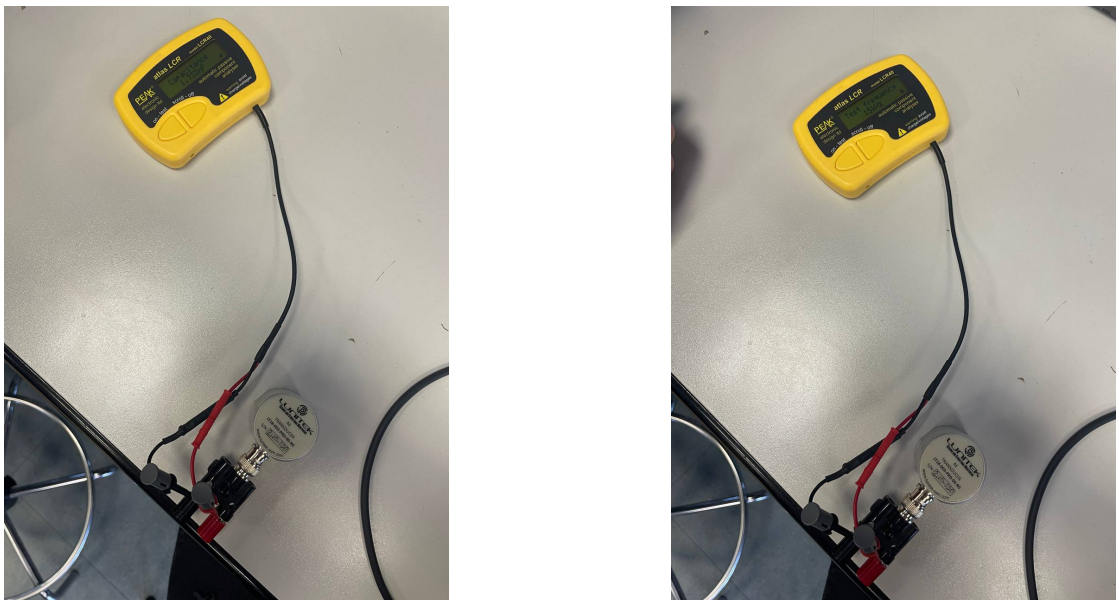


Figure 4.3: The AE sensor's capacitance at 15 kHz

to the PLB test was around 120-130 mV, meaning 240-260 mV rail-to-rail, while the typical timespan is around 15 ms.

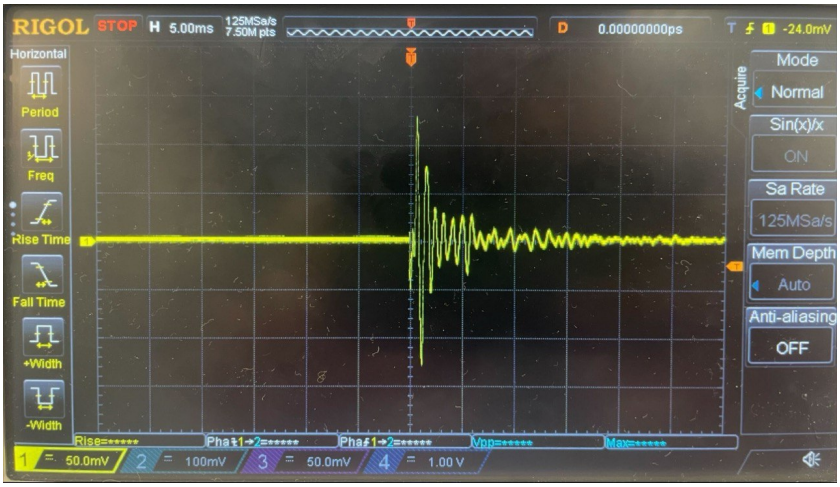


Figure 4.4: The AE sensor's typical response to a PLB test

Although this voltage range is usable, it lacks vital properties for constructing a practical sampling circuit. Had it been directly used, many details would have been lost during the digitalization process. Such a small amplitude would require a more precise selection of the onset time's threshold, and the signal's resolution would also be smaller.

During the thesis, an STM Nucleo-F446RE development board [77] was used, featuring a high-performance ARM Cortex-M4 core running at up to 180 MHz. It provides a range of peripherals, most importantly a 12-bit SAR (Successive Approximation Register) ADC with up to 16 multiplexed channels, supporting single and continuous conversion modes and features like oversampling and hardware-triggered conversions for enhanced accuracy. The ADC operates with a sampling rate of up to 2.4 MSPS (mega samples per second), making it suitable for high-speed data acquisition. The ADC input voltage range is from 0V to Vref+, where Vref+ can be at most 3.3V, defining the measurable analog signal range. Connecting the sensor directly to the ADC would produce inferior results because the total voltage range is not utilized, and this would decrease resolution. For this reason, an amplification circuit had to be designed, which will be detailed in the next section.

4.2 Hardware development

Given the small voltage range of the AE sensor, an appropriate amplifier circuit had to be designed to transform the voltage to fill out the whole range of the Nucleo board's ADC. The ADC is a configurable successive approximation (SAR) ADC between 6-12 bits, with an input voltage range of 0 and 3.3 Volts. Ideally, this circuit needs to be powered by the board with a single supply voltage between

0 and 5 Volts. Moreover, the circuit needs to have three vital attributes. Firstly, it must shift the signal to the mid-level of the usable voltage range, which is 1.8 V. Secondly, it must amplify the signal to span the ADC's entire range. Finally, the board must be protected from overvoltage, which can occur when a knock or crack is too big.

Several different amplifier circuits were considered before choosing the appropriate one [78]. Due to the sensor's piezoelectric nature, different circuit designs did not work; a detailed description can be found in Appendix A. Consequently, only the final circuit design and its development will be discussed in this chapter.

4.2.1 Charge amplifier

A different approach was sought and considered after the unsuccessful implementation of various classical amplifier circuits. Sources [79, 80, 81] mention that a charge amplifier circuit (introduced in 2.7.6) is capable of transforming a piezoelectric circuit's signal into a manageable, correctly behaving signal.

The problem with the voltage mode amplification of the previous circuits lies in the fact that the interconnect capacitances can greatly modify the voltage output of the system. To understand this, the output voltage of the piezo sensor over time has to be understood:

$$V_{out} = \frac{1}{C} \int \frac{dQ}{dt} dt = \frac{Q}{C}$$

Any other capacitance parallel to the piezoelectric sensor, such as a cable connecting the device to the amplifier, will contribute to this term. Since the sensor's capacitance is not large, these parasitic elements will significantly influence the overall system's behavior, especially if the cables are long. For this reason, it is also better to use a charge amplifier rather than a voltage mode amplifier.

Following this notion, the charge amplifier's components had to be selected to transform and amplify the sensor's signal correctly. A simple integrator circuit that consists of a negative feedback op-amp with a capacitor in the feedback loop would be adequate as an initial idea. As the information that we seek is the charge of the piezoelectric sensor, an integrator fits the task perfectly, as it returns the voltage, which is proportional to the sensor's charge:

$$V_{out} = \frac{1}{C_F} \int -I dt = -\frac{Q}{C_F}$$

The output voltage will be proportional to the current multiplied by $\frac{1}{C_F}$. Subsequently, the gain also depends on the capacitance's value.

However, a simple integrator would only work in an ideal environment. Due to the amplifier's input bias current, the capacitor in the feedback loop would

eventually charge up, causing the op-amp to saturate. To avoid this, a resistor may be placed parallel to the capacitor. With this step, we arrive at the circuit presented in chapter 2.7.6.

It is possible to create a non-zero-centered output signal by placing an offset voltage to the positive input terminal of the op-amp. This way, the sensor's signal will be centered around the offset voltage. However, the effect of a possible amplification (by a subsequent amplifier) has to be considered when selecting the amplifier's value.

Adding a resistor to the amplifier's inverting terminal can be beneficial. This resistor protects the op-amp by limiting the current generated by the piezoelectric sensors. Although the current generated by the piezoelectric sensor is mostly harmless, an unexpected fault, such as an ESD strike, can severely damage the component. Therefore, it was decided to include this component in the design.

Another important aspect of this circuit to consider is its frequency response. An idealized integrator that only includes a capacitor behaves similarly at every frequency. Nonetheless, when resistors are added to the circuit, the behavior changes both at high and low frequencies.

The feedback resistor R_f that is included in parallel with the feedback capacitor C_f together create a high-pass filter that attenuates the low-frequency signals according to equation 4.2. The value of the resistor placed in the parallel does not affect the overall gain of the circuit. Hence, it was possible to choose this considering other aspects of the circuit.

$$f_{HP} = \frac{1}{2\pi R_F C_F} \tag{4.2}$$

At this point, values for two components had to be selected. The value of the feedback capacitor and the the feedback resistor's value. Looking at Equation 4.2, it can be seen that a lower resistor value would increase the cutoff frequency, while a higher value would decrease it. Recommendations suggest an initial value of $1M\Omega$ and adjust it based on the application needs. A relatively high $6.8M\Omega$ resistor was chosen to attenuate only the very low-frequency noises. The available resistors partly influenced the choice in the laboratory.

The other component to select at this point was the capacitor's value. At this point, it was thought that the cutoff frequency of the low pass filter would not be important to the overall system behavior as the typical frequency of an acoustic emission signal is around 10-20 kHz. For this reason, a cutoff frequency of 10 Hz was chosen. It will be seen during the test phase that a better decision could have eliminated some noise from the environment. Regardless, the resistor's value and the imposed HPF frequency limit determine the capacitor's value, according to equation 4.3.

$$C_F = \frac{1}{2\pi R_F f_{HP}} = \frac{1}{2\pi \cdot 6.8 \cdot 10^6 \cdot 10} = 2.34nF \tag{4.3}$$

The closest capacitor value in the laboratory was 2.2 nF, which imposed a slightly different cutoff frequency of 10.6 Hz.

The last component that hasn't been considered yet is the input resistor R_{in}. It has been mentioned before that the choice of charge amplification eliminates the effect of parasitic capacitances. This statement changes somewhat with the inclusion of the input resistor. This resistor, combined with the sensor's internal capacitance and other parasitic capacitances, creates a high-frequency attenuation that must be considered. This cutoff frequency can be computed according to equation 4.4, where C_{all} denotes the sum of the sensor capacitance and every other capacitance.

$$f_{LP} = \frac{1}{2\pi R_{in} C_{all}} \tag{4.4}$$

For simplicity and because the cable resistances were negligible due to the short cable length (< 10 cm), $C_{all} := C_{sensor} = 1.773nF$ was selected. Due to the dependency of the sensor's capacitance on the frequency and the uncertainty of the actual cable capacitance, a much higher-than-needed cutoff frequency of 50 kHz was chosen. This frequency is 2.5 times larger than the typical frequency of an acoustic emission signal. Consequently, this choice determined the input resistor's value, according to Equation 4.5.

$$R_{in} = \frac{1}{2\pi \cdot C_{sensor} \cdot f_{LP}} = \frac{1}{2\pi \cdot 1.73 \cdot 10^{-9} \cdot 50 \cdot 10^3} = 1840\Omega \tag{4.5}$$

The circuit described above was built and tested using the sensor as input and the oscilloscope as output to verify the circuit's behavior. The circuit design can be seen in Figure 4.5, while the constructed circuit can be seen in Figure 4.6. For the op-amp, a TL082 dual-supply amplifier was chosen from Texas Instruments [82].

When the sensor was connected to the input of the circuit and a typical pencil lead break test was performed, the circuit outputted an expected characteristic acoustic emission signal (see Figure 4.7), verifying the circuit's correct functioning.

4.2.2 Dual-supply amplifier circuit

As the charge amplifier was correctly functioning, the development of the hardware continued. Now, we focus on building the whole circuit, which includes amplifying the signal and shifting it to the mid-value of the ADC's input range.

Two additional components had to be built and connected to the charge amplifier to achieve this. Firstly, a circuit that can provide the offset to shift the piezoelectric

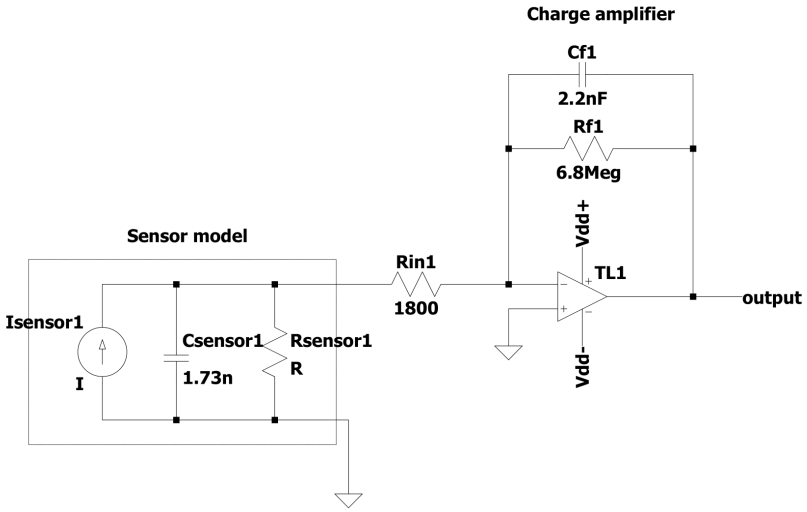


Figure 4.5: Charge amplifier circuit design

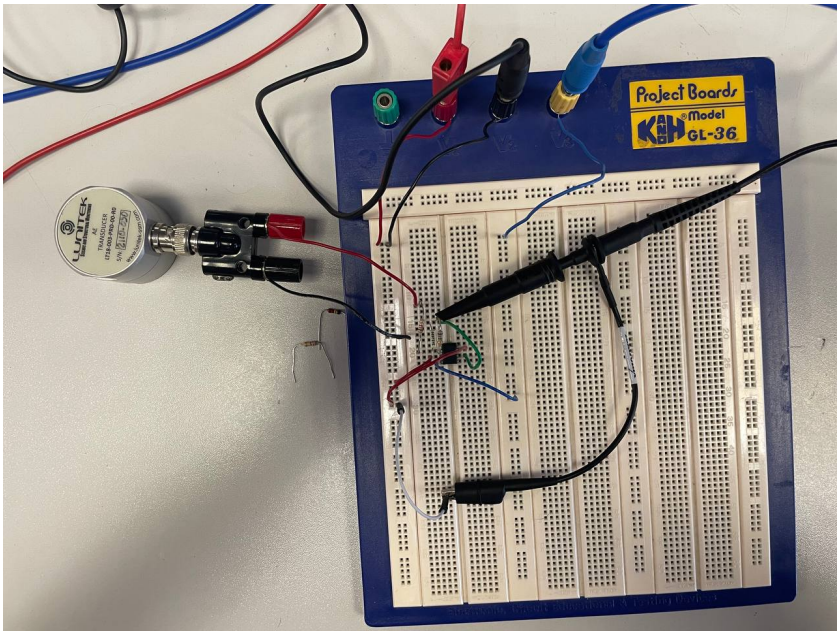


Figure 4.6: Charge amplifier circuit with the sensor as input and a probe connected to its output

sensor's signal is required. The offset was achieved using a voltage divider of two resistors combined with a unit-gain inverting op-amp. As the previously built

Figure 4.7: The charge amplifier circuit's output

charge amplifier used a dual-supply topology, the same TL082/TL081 op-amp was chosen for this stage, too. Since the selected summing amplifier had an inverting topology, the supply on the resistor ladder had to be negative to get a positive voltage on the output. Since the negative supply voltage was -5 Volts, the selected resistors had a value of $R_4 = 200k\Omega$ and $R_5 = 100k\Omega$, thus creating a voltage offset of -1.65 Volts.

Another stage is needed to add the two signals together and amplify them, achieved with a classic inverting summing amplifier. As before, the TL081/TL082 dual-supply op-amp was chosen. Other than choosing the op-amp to include, the correct resistor values had to be selected, too. No amplification was needed as the offset voltage already had the proper magnitude. A $R = 22k\Omega$ resistor was chosen for the feedback. Hence, the resistor connecting the offset with the inverting input of the summing amplifier had to be the same. On the other hand, the sensor's signal was not yet amplified. To achieve the correct amplification, a resistor of $3.3k\Omega$ was chosen, thus achieving around 6.67 fold amplification.

Finally, the circuit finished with two 1N5819 Schottky protection diodes before the whole circuit's output. The design can be seen in Figure 4.8.

After finishing the circuit design, the circuit was built and tested with classic laboratory equipment. The constructed circuit can be seen in Figure 4.9.

The first test verified the correct signal output when the sensor was connected to its output without the protection diodes. Upon examining the output on an oscilloscope, it was deduced that the circuit amplified and shifted the signal correctly. The circuit's output can be inspected in Figure 4.10.

Afterward, the Schottky protection diodes were added to the circuit to verify that they could limit the voltage spikes that an eventual too-strong input event

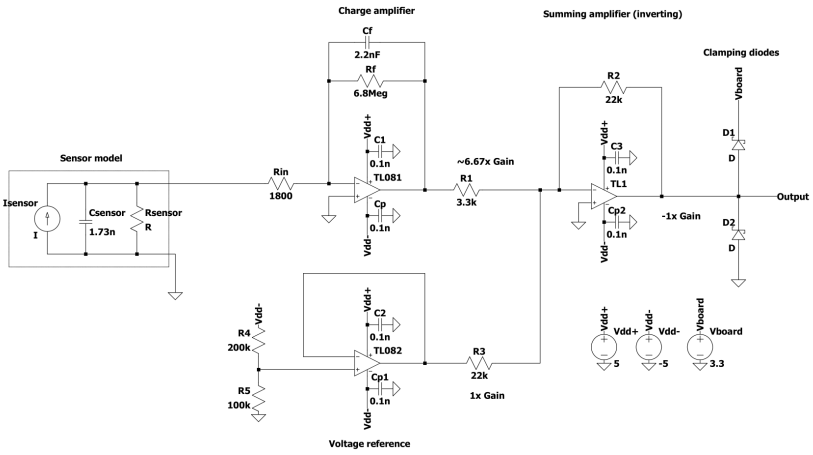


Figure 4.8: Dual supply sensor amplifier circuit

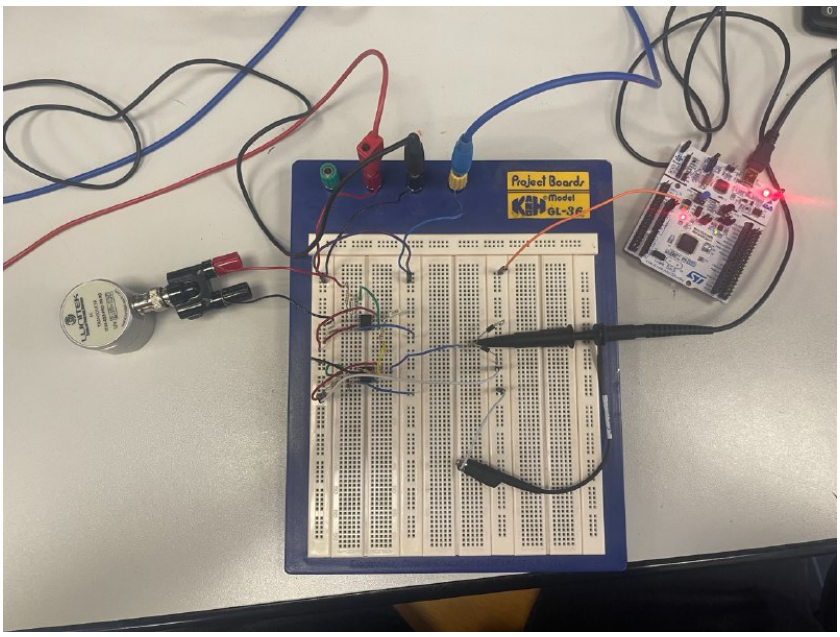


Figure 4.9: The assembled dual-supply circuit with protection diodes

would trigger. To confirm this, a significantly stronger knock was used on the sensor to simulate such an event. Capturing the output with the oscilloscope verified that the protection diodes can limit the maximum voltage on the output. Results can be witnessed in Figure 4.11.

As can be seen, the diodes correctly fulfill their purpose by limiting the maximum

Figure 4.10: The dual-supply circuit's output to a typical PLB test as input

Figure 4.11: The dual-supply circuit's response to an overvoltage signal

voltage present on the circuit's output. At this point, the circuit's correct functioning has been verified. Although the circuit is correct and functioning, certain aspects can be improved. Most importantly, it currently relies on a dual-supply topology that a simple microcontroller cannot provide. It would be beneficial to change to a single-supply solution to solve this issue so that any simple commercial microcontroller can power the circuit.

4.2.3 Single-supply amplifier circuit

In the previous section, the first functioning solution for a circuit that fulfills the initial requirements of offset and amplification was presented. This section details

the switch to a single supply topology.

Most importantly, to achieve a functioning single-supply circuit, every component that needs dual-supply has to be exchanged for another that only requires single-supply. Every op-amp had to be changed to ones requiring only single-supply for the circuit. As a single-supply op-amp, the TLC271 model was selected from Texas Instruments [83].

Another significant change was made to the circuit. Instead of using a summing amplifier and adding the offset to the amplified signal, the input signal was offset before the charge amplifier's input. This change meant that the offset was also amplified; hence, the offset's value had to be changed. Instead of having a 1.65 Volts offset, the value was set to be 0,215 V according to Equation 4.6. Consequently, one less resistor was needed than in the previous circuit.

$$V_{offset} = \frac{R_2}{R_1 + R_2} V_{in} = \frac{100k\Omega}{2.2M\Omega + 100k\Omega} \cdot 5V = 0.217V \tag{4.6}$$

Finally, the inverting amplifier was swapped for a non-inverting amplifier with a gain of 6.66, using a feedback resistor of $56k\Omega$ and an input resistor of $10k\Omega$. Thus, the circuit's final offset was around 1.5V. The final circuit design can be seen in Figure 4.12.

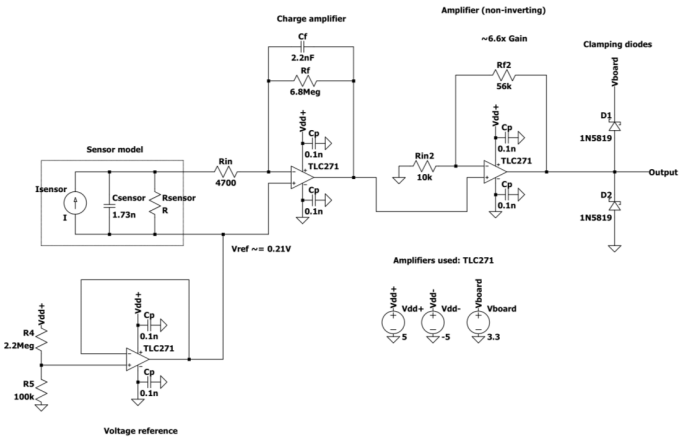


Figure 4.12: Single supply amplifier circuit

Upon finishing the circuit's design, it had to be built and tested. The assembled circuit can be seen in Figure 4.13. Similarly to the dual-supply circuit, the tests were conducted using the PE sensor and an oscilloscope to display the circuit's output.

Two tests were performed similarly to the previous circuit. The first test was intended to verify the correct output for a typical input. In contrast, the other

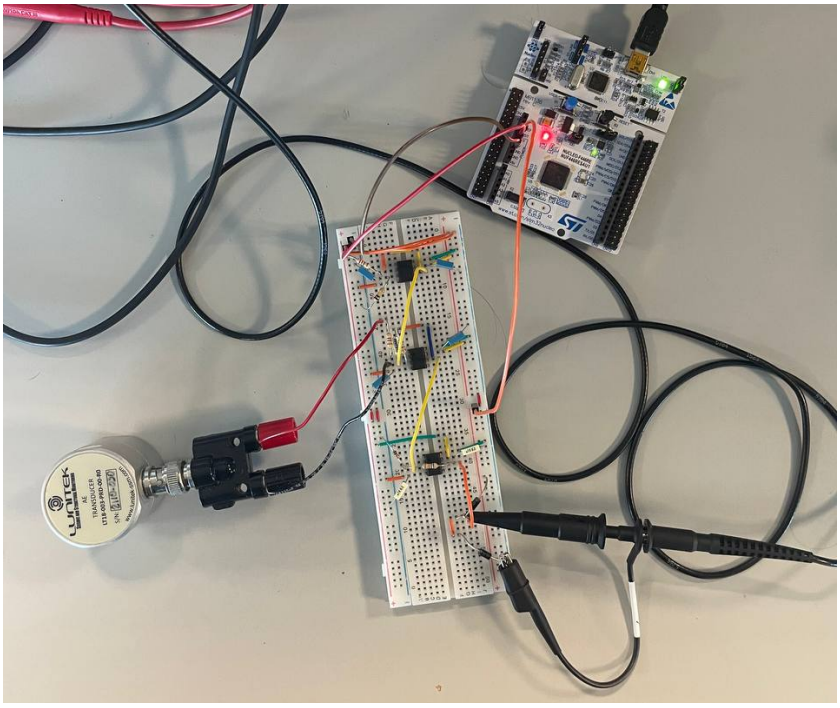


Figure 4.13: The assembled single-supply amplifier circuit

tests were aimed to verify the proper functioning of the protection diodes in case of an overvoltage on the amplifier's output.

It can be seen in Figure 4.14 that the circuit correctly amplifies a typical input signal, hence confirming the correct behavior. The offset is around 1.5 Volts in a steady state, while the signal peaks are between 0 and 3.3 Volts. Upon closer examination, it was also noticed that a small sinusoidal wave was creeping in from the supply lines. This noise was deemed negligible at this stage of the development, so it was not dealt with. Looking at Figure 4.15, a response to an overvoltage can be inspected, and how the diodes limit the maximum voltage to 3.3 Volts.

At this point, the first phase of the work - the hardware development - was finished. A correctly functioning circuit has been designed to amplify and offset a typical acoustic emission signal. Moreover, the circuit has been designed so that any commercial board with minimal power supply pins can power it. Additionally, the circuit includes protection diodes that safeguard the internals of the embedded device to which it is attached. The next step was to design and develop a firmware that could sample, interpret, and transmit the acoustic emission signals. On top of this, it should also be able to detect cracks in the signals and measure the onset times of these signals and the differences between the onset times coming from the various sensors.

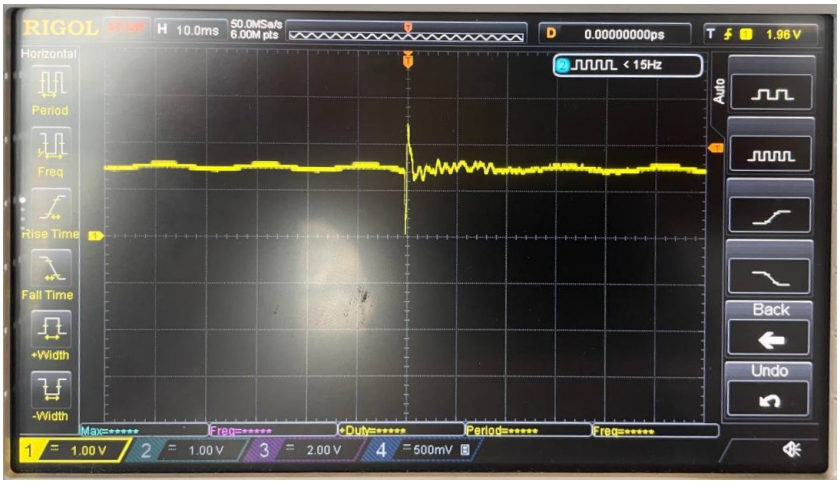


Figure 4.14: The single supply circuit's response to a typical input signal

Figure 4.15: The single supply circuit's response to a signal with overvoltage

4.3 Software development

The first step in firmware development was to design the algorithm and architecture, considering the system's imposed constraints. At this point, it was essential to understand better the peculiarities of the microcontroller used and what was possible.

The selected microcontroller is an STM Nucleo F446-RE microcontroller [77], the most popular and widespread lineup in ST's portfolio. It employs a 32-bit ARM architecture. The most relevant features of this project are the following:

it has three 12-bit, 2.4 MSPS (Mega Samples Per Second) with up to 24 total channels. It includes a general purpose 16 stream DMA (Direct Memory Access) controller with FIFOs and burst support. Moreover, the microcontroller has a vast selection of independently programmable timers, including twelve 16-bit timers and two 32-bit timers up to 180 MHz with separate IC/OC/PWM or pulse counters. On the connectivity front, it has I2C, SPI, SAI, CAN, SDIO, and USART/UART interfaces. The UART/USART interface will be the most important for the thesis's purposes. This communication protocol contains a transmission speed of up to 11.25 Mbit/s, with four USART interfaces and two UART interfaces.

Every ST microcontroller has a native IDE and debug environment developed in-house. The F446-RE is no different; it is developed through the *STM32CUBE* IDE [84], an eclipse-based environment with a plethora of additional features. For the development of the firmware version, *1.16.1* was used, the most recent at the time of the development. The IDE integrates STM32 configuration and project creation functionalities from *STM32CubeMX* to support rapid development and prototyping. Using the *STM32CubeMX*, it is possible to preconfigure the clocks, ADCs, peripherals, DMAs, and communication protocols and generate an initialization code from the configuration. Returning and modifying the configuration during the development phase and regenerating code without destroying the user-written code is also feasible. On top of these features, the IDE also sports an advanced debugger that can view CPU core registers, memories, peripheral registers, live variable watch, and fault analyzer, to mention only a few.

After familiarizing with the architecture and environment, the first step was to design the firmware architecture and an initial algorithm. To do this, a flowchart was created to help us better understand and visualize what needs to be done.

The general sequence starts with setting up every necessary register, peripheral, clock, and system component that will be used or required for the microcontroller's correct functioning. After finishing this, the ADC samples its input and saves the data in a buffer. Then, the data is checked against a preset threshold to decide if the signal is classified as the beginning of a crack. If it is not the start of a crack, the sampling and the saving continue. Otherwise, the firmware saves the next X samples into a designated buffer. Finally, after filling this buffer, the buffer is transmitted to a client via a UART interface, and the process starts again. The flowchart of the algorithm can be better inspected in figure 4.16.

4.3.1 Single ADC channel with polling

Firmware

The first firmware version validated the concept's feasibility and functioned as a springboard for subsequent versions. Because of this, it was not intended to be the most efficient or to include more advanced features, such as a DMA.

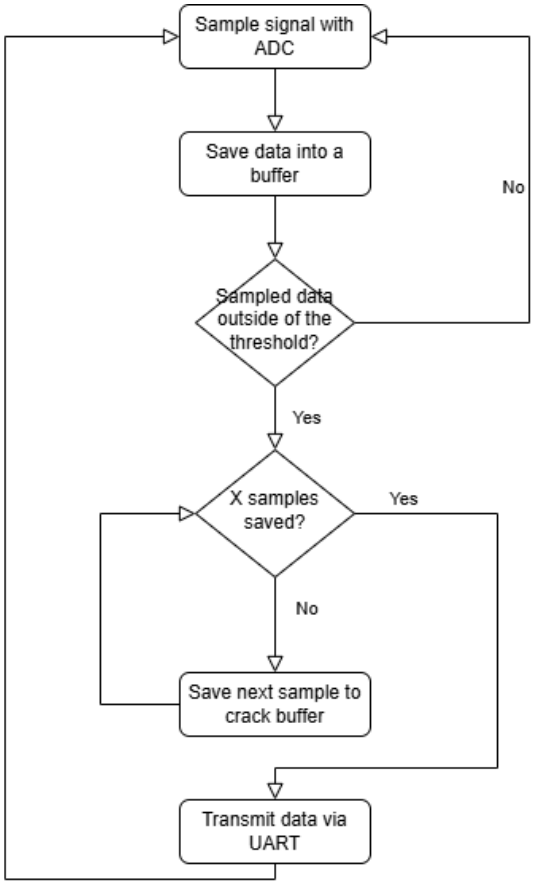


Figure 4.16: Initial flowchart detailing the sampling algorithm

This initial version used one ADC with one input channel. The ADC was configured to include the maximal 12-bit resolution, with continuous mode disabled, meaning the ADC only performed one conversion whenever it started and stopped. The start was triggered by software. The ADC created an interrupt after finishing each conversion. The only configured channel had a sampling time of three cycles. The ADC configuration can be seen in Listing 4.1. The author finds it necessary to mention that not all source codes are included, but only the most essential pieces are needed to understand the behavior.

Listing 4.1: ADC initialization code

```
1 static void MX_ADC1_Init(void)
2 {
3   ADC_ChannelConfTypeDef sConfig = {0};
4
```

```
5   /* Configure the global features of the ADC (12b Resolution, Single
      conversion, Started by SW, Interrupt at the end of every
     conversion) */
6   hadc1.Instance = ADC1;
7   hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
8   hadc1.Init.Resolution = ADC_RESOLUTION_12B;
9   hadc1.Init.ScanConvMode = DISABLE;
10  hadc1.Init.ContinuousConvMode = DISABLE;
11  hadc1.Init.DiscontinuousConvMode = DISABLE;
12  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
13  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
14  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
15  hadc1.Init.NbrOfConversion = 1;
16  hadc1.Init.DMAContinuousRequests = DISABLE;
17  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
18  if (HAL_ADC_Init(&hadc1) != HAL_OK)
19  {
20    Error_Handler();
21  }
22
23  /* Configure the selected ADC regular channel (Sample time) */
24  sConfig.Channel = ADC_CHANNEL_0;
25  sConfig.Rank = 1;
26  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
27  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
28  {
29    Error_Handler();
30  }
31 }
```

The other peripheral that was used in the first version is the UART. The peripheral was configured to use a slower baud rate of *57600*, with parity *None* and one stop bit. The UART's init section can be seen in Listing 4.2.

Listing 4.2: UART initialization code

```
1  static void MX_USART2_UART_Init(void)
2  {
3    huart2.Instance = USART2;
4    huart2.Init.BaudRate = 57600;
5    huart2.Init.WordLength = UART_WORDLENGTH_8B;
6    huart2.Init.StopBits = UART_STOPBITS_1;
7    huart2.Init.Parity = UART_PARITY_NONE;
8    huart2.Init.Mode = UART_MODE_TX_RX;
9    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
10   huart2.Init.OverSampling = UART_OVERSAMPLING_16;
11   if (HAL_UART_Init(&huart2) != HAL_OK)
12   {
13     Error_Handler();
14   }
```

```
15 }
```

The *main* function starts the ADC, then polls for conversion, and once the conversion finishes, it saves the data in a buffer. Then, as the flowchart described, the algorithm checks if the data is outside a threshold and acts accordingly. A flag is used to know if a crack is detected and where to save the data. An engaging portion is the circular buffer where the regular (not crack) data is saved, enabling a compact buffer size where only the most recent data is stored. However, this brings some difficulties to the transmission and the visualization as extra information must be transmitted to know how to reorder this buffer. Once the buffer is filled up, the UART performs three total transmissions. First, it transmits the current index so the client knows how many samples of standard data it can expect in the subsequent transmission. Secondly, the standard buffer is transmitted, followed by the crack buffer. Later, the indices and variables are reset to their initial value.

Listing 4.3: main function

```
1  int main(void)
2  {
3    // user variables
4    uint16_t raw_adc_val;
5    uint16_t normal_it = 0;
6    uint16_t crack_it = 0;
7    uint8_t is_crack_detected = 0;
8    const uint16_t threshold = 1200;
9    uint16_t normal_sample_buffer[NORMAL_ARRAY_LEN];
10   uint16_t crack_buffer[CRACK_ARRAY_LEN];
11
12   // ... component initializations omitted ...
13   while (1)
14   {
15       //Start ADC conversion
16       HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
17
18       HAL_ADC_Start(&hadc1);
19       HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
20       raw_adc_val = HAL_ADC_GetValue(&hadc1);
21
22       if(raw_adc_val > threshold ){
23           // detected a crack
24           is_crack_detected = 1;
25       }
26
27       if (is_crack_detected){
28           crack_buffer[crack_it] = raw_adc_val;
29           crack_it++;
30
31           // reached the array end, reset, wait for new crack
```

```
32              if(crack_it == (CRACK_ARRAY_LEN)){
33                  //first transmit the non-crack samples from the buffer,
     it has a varying length
34
35                  // transmit the size of the array
36                  HAL_UART_Transmit(&huart2, (uint8_t *)&normal_it,
    sizeof(normal_it), 100);
37                  HAL_Delay(10);
38
39                  // transmit the non-crack array
40                  HAL_UART_Transmit(&huart2, (uint8_t *)
    normal_sample_buffer, normal_it * sizeof(normal_sample_buffer[0]),
     100);
41                  HAL_Delay(10);
42
43                  //transfer the samples from the crack buffer
44                  HAL_UART_Transmit(&huart2, (uint8_t *)crack_buffer,
    CRACK_ARRAY_LEN * sizeof(crack_buffer[0]), 100);
45
46                  // reset the iterators and flags
47                  crack_it = 0;
48                  normal_it = 0;
49                  is_crack_detected = 0;
50              }
51          }
52          else {
53              // save it into the normal array
54              normal_sample_buffer[normal_it] = raw_adc_val;
55              normal_it = (normal_it + 1) % NORMAL_ARRAY_LEN;
56          }
57          HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
58      }
59  }
```

This solution, albeit working, was far from ideal. Even though it can sample the ADC, it does it in a blocking fashion due to the polling function call, meaning the CPU cannot perform other tasks while waiting for the conversion to end. This approach in a real-world, real-time environment is not ideal, as other interruptions could happen. Moreover, the CPU's constant use does not comply with the initial request for a low-power solution. The other area to improve is data transmission, which also happens in a blocking mode, preventing the CPU from performing different tasks in the meantime. Furthermore, all information is transmitted in 3 separate function calls with delays between them, which further decreases efficiency and performance. Despite these drawbacks, this firmware was able to validate the initial algorithm. In the following sections, the more advanced firmware solutions will be presented.

Python client

In parallel with the firmware development in C, a Python client is also created. This client is capable of receiving, saving, and visualizing the processed data. The first version of the Python client uses only one function to receive the data from the microcontroller. The *read_from_UART* has three parameters: the communication port, the baud rate, and the data length. The function attempts to read the specified data length, notifies the user if the received data's length does not match, and then transforms the data. The script can be seen in B.1. Even though the function received the data, it was somewhat unstable, especially when receiving the last data section. Moreover, this version was not yet capable of visualizing the data.

To improve the communication and check data loss, the parity bit was set to *Even* later on. This change didn't solve the communication issues, but for additional debug functionality, it was kept. Subsequently, it was evident that this script needed rework and the addition of more functionalities.

4.3.2 Single ADC channel with interrupt

After verifying the algorithm's correctness, the attention was turned towards improving efficiency, speed, and power consumption.

Firmware

One can utilize an interrupt-based ADC conversion to save energy and free the CPU from the workload. This allows the CPU to perform other tasks or go to a low-power state while the ADC converts the data.

At the end of the conversion, a high-priority interrupt is raised. When this happens, normal code execution stops, the interrupt is served, and then code execution continues. Interrupts cannot be preempted, so keeping them as short as possible is generally advisable.

As the project was not using other interrupt-based functionalities, keeping the ADC conversion completed interrupt short was not a priority. Consequently, the processing and saving functionality was moved inside the interrupt. The ADC conversion completed callback can be seen in 4.4. Essentially, what was previously performed in the *main* function was moved to the callback. An additional flag notifies the firmware to transmit the data once the buffer gets full.

Listing 4.4: ADC conversion completed callback

```
1 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
2 {
3     if(hadc->Instance == ADC1)
4     {
```

```
5           raw_adc_value = HAL_ADC_GetValue(hadc);
6
7           if (crack_capturing_flag == 1) {
8               // CAPTURING CRACK
9               // Store the current ADC value into crack buffer
10              crack_buffer[crack_it++] = raw_adc_value;
11
12              // Once 512 samples are captured, set the tx flag
13              if (crack_it >= CRACK_ARRAY_LEN) {
14                  crack_capturing_flag = 0;  // Stop capturing
15                  // Indicate that UART transmission is needed
16                  transmission_flag = 1;
17              }
18          }
19          else
20          {
21              // Update circular buffer with the most recent sample
22              normal_sample_buffer[normal_it++] = raw_adc_value;
23              if (normal_it >= NORMAL_ARRAY_LEN) {
24                  normal_it = 0;  // Wrap around circular buffer index
25              }
26          }
27      }
28  }
```

The other feature experimented with was the use of the *Analog Watchdog*. This feature enables the user to set predefined upper and lower thresholds. If the currently converted data is outside these thresholds, an interrupt is raised, and the user can perform actions inside the callback routine. The analog watchdog can be enabled inside the ADC configuration. The callback was utilized to set the flags, signaling that a crack had been detected. The corresponding code snippets are in 4.5.

Listing 4.5: ADC AWD initialization and callback

```
1  static void MX_ADC1_Init(void)
2  {
3    /* ... regular ADC init code omitted ... */
4    ADC_AnalogWDGConfTypeDef AnalogWDGConfig = {0};
5    /* ... regular ADC init code omitted ... */
6
7    /* Configure the analog watchdog */
8    AnalogWDGConfig.WatchdogMode = ADC_ANALOGWATCHDOG_SINGLE_REG;
9    AnalogWDGConfig.HighThreshold = 2500;
10   AnalogWDGConfig.LowThreshold = 1000;
11   AnalogWDGConfig.Channel = ADC_CHANNEL_0;
12   AnalogWDGConfig.ITMode = ENABLE;
13   if (HAL_ADC_AnalogWDGConfig(&hadc1, &AnalogWDGConfig) != HAL_OK)
14   {
```

```
15      Error_Handler();
16    }
17    /* ... regular ADC init code omitted ... */
18  }
19
20  void HAL_ADC_LevelOutOfWindowCallback(ADC_HandleTypeDef *hadc)
21  {
22      if (hadc->Instance == ADC1 && crack_triggered_flag == 0) {
23          // Triggered when value is outside the threshold
24          // Flag to indicate watchdog triggered
25          crack_triggered_flag = 1;
26          // Start capturing the next 512 samples
27          crack_capturing_flag = 1;
28          // Reset index for the new capture
29          crack_it = 0;
30      }
31  }
```

At this point of development, the code had some issues. First, there was a problem with the interrupt-based conversion. Initially, the idea was to use the ADC in continuous mode, only starting the ADC once at the start. However, due to an incorrect flag, it was not working. An intermediate solution was chosen: the ADC was restarted for every conversion at the beginning of the *while* loop in the *main* function.

The problem was solved by creating a blog post in ST's forum [85]. The answer pointed out that the flag *hadc2.Init.EOCSelection* was set to *ADC_EOC_SINGLE_CONV* instead of *DISABLE*, which had to be changed. The reason why this change achieves the desired outcome is relatively obscure (it is not documented well in the official materials, either). However, it did solve the problem.

More importantly, this conversation pointed out fundamental flaws in the firmware, mainly how the ADC was used until this point. So far, the ADC has been started by software, and it has had a continuous conversion and three cycles to sample the signal. It is safe to say that it was going at full speed, putting the CPU and the whole system on a huge load. The code often had a hard fault error due to the speed being too high and getting stuck in the error handler. The contributors suggested using DMA to save the ADC's data, increasing the sampling time, and implementing an external hardware trigger to start the conversion to decrease the load.

The sampling time was temporarily increased to 56 cycles to solve the hard fault error. According to the blog post, the ADC's functioning will be changed in later firmware versions.

Furthermore, the analog watchdog functionality was removed as it did not bring any particular benefits. The conversion callback had to be performed regardless of

whether the current sample was inside or outside the threshold. Hence, calling an additional interrupt handler did not bring benefits. On the contrary, it takes extra time to modify the stack, the pointers, and the other necessary parameters that come with function calls.

The UART communication was changed to use a DMA instead of a blocking transmission to improve efficiency and decrease power consumption and the CPU's load. Again, this allows the CPU to perform other tasks or go to a low-power state. The data now was being transmitted in one function call to have an effective DMA transmission.

The DMA only needed basic configuration, initialization, and modification of the UART function. Other additions weren't necessary to enable this feature. The relevant code sections are in 4.6.

Listing 4.6: Relevant DMA code snippets

```
  /* ... code omitted ... */
int main(void)
{
  /* ... code omitted ... */
  while (1)
  {
    /* ... code omitted ... */
      if(transmission_flag == 1)
      {
          data_buffer[0] = normal_it;
          // time of crack
          data_buffer[DATA_BUFFER_LEN - 1] = time_at_crack;
          // transfer the samples from the buffer
          HAL_UART_Transmit_DMA(&huart2, (uint8_t *)data_buffer,
   DATA_BUFFER_LEN * sizeof(uint16_t));
          transmission_flag = 0;
      }
  }
}
/* ... code omitted ... */
static void MX_DMA_Init(void)
{
  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();
  /* DMA interrupt init */
  HAL_NVIC_SetPriority(DMA1_Stream6_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Stream6_IRQn);
}
/* ... code omitted ... */
```

Another goal of the embedded solution is to locate the detected crack eventually. To do this, the time difference between the cracks' arrival times must be measured as precisely as possible. A high-frequency clock is used to do this. The clock is

configured using the IDE's code configuration tool. The clock's value is saved in a variable at the crack's arrival time. The timer operates with an 84 MHz peripheral clock. As the prescaler is set to *84 - 1*, the final clock frequency is 1 MHz. The 1 MHz was only an initial setting; it is possible to decrease the prescaler to 0 to achieve the maximum 84 MHz frequency and the highest measurement precision. For now, this value is adequate to validate the concept.

Listing 4.7: Timer initialization code

```
static void MX_TIM10_Init(void)
{
  htim10.Instance = TIM10;
  htim10.Init.Prescaler = 84 - 1;
  htim10.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim10.Init.Period = 65535;
  htim10.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim10.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim10) != HAL_OK)
  {
    Error_Handler();
  }
}
```

Python client

Several improvements were added to the client to make the data reception more reliable on the Python side, as it exhibited errors and data loss.

To be able to find the start and the end of the data, start (*0xC0, 0xD0*) and stop (*0xD0, 0xC0*) patterns were added to the transmitted information. Even though it was an overhead, it was hoped to fix the issues with the transmission. However, despite this effort, the transmission was still unstable when tested, so ultimately, these patterns were removed.

After digging deeper into the documentation of the *serial* package, it was discovered that the library handled the data lengths differently compared to ST's UART communication. The parity bit was active so far to better understand the transmission correctness. While the ST IDE involves the parity bit in the data bit size, the *serial* Python library does not count the parity bit as a data bit. Ultimately, the parity was set to *None* to avoid these problems.

The data was joined into one frame to enable easier DMA transmission. The first element of the frame was the index of the current last element in the standard data buffer, allowing the Python client to reorder the standard data buffer into time-series data. The following data section was the crack data. Finally, the last transmitted portion was the timer's value. A frame's format can be seen in 4.17.

0	[1, N]	[N + 1, N + 1 + C]	N + 1 + C + 1
Index	Normal data (N)	Crack data (C)	Time at crack

Figure 4.17: Transmitted data frame format

Other than this, the Python script has undergone various changes. Threading has been introduced to stop the script while the data processing happens dynamically. As its name says, the *read_fixed_length_data* attempts to read a fixed data amount from the serial connection and raises an error if the received data length does not match the expected one.

The main work happens in the *process_data* function; the threads and the serial connection are created inside this function. Thereafter, a loop starts reading data from the serial. Once data arrives, the processing happens, which includes reordering the standard data based on its index. Then, data is saved into a file, and if visualization is turned on, it is also shown in a *pyplot* plot. If the received data is incorrect, different errors are raised based on the cause. The processing goes on until the stop event is received.

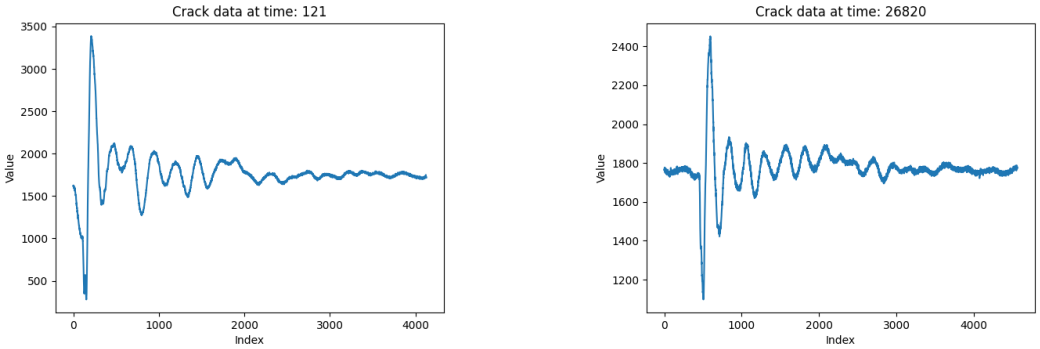
Finally, there is an option to visualize the data separately, implemented by the *visualize_data* function. The whole Python code can be seen in B.2.

Results

This point is ideal for assessing the situation. The firmware can sample, process, and store acoustic emission data from one piezoelectric sensor. It can transmit this data through a UART connection to a Python client that permanently stores and visualizes it. Moreover, a timer can measure the onset time of the signal with up to 84 MHz frequency. In summary, the solution's initial requirements are almost fulfilled.

To verify the correct functioning, the whole pipeline was tested with the piezoelectric sensor, which performed a PLB test on an arbitrary surface. Results can be seen in Figure 4.18. It can be noticed, the results behave as expected from the various tests performed with the oscilloscope and only the HW.

Nonetheless, there is further work to be completed. Currently, the embedded solution can only host one piezoelectric sensor instead of the required four. The solution still exhibits instabilities due to the software-based ADC triggering. The next section of the software development addresses these points.



(a) Result 1 **(b)** Result 2

Figure 4.18: PLB test result using one piezoelectric sensor

4.3.3 Multiple ADC channels with HW trigger, interrupt, and DMA

Considering the general structure of the firmware, several modifications were made. Looking at an updated flowchart (see 4.19) to understand the most critical updates is helpful. After every peripheral is initialized, the operation starts by converting the channels, and once every channel is converted, it checks each value to decide what to do next. If every value is inside the threshold, it is saved inside a circular buffer containing the standard data. If one of them is outside, a crack is detected, and they will be saved in a different *'crack buffer'*. Once the buffers are filled up, the conversion stops, the data is transmitted through UART communication, and the operation continues from the beginning. After a general overview is shared, more detailed explanations will follow on each element.

Firmware

Since now multiple channels were used, the variables supporting this transition had to be created. More importantly, a vital design decision also had to be made. As the firmware now hosted four different signals, it was not evident anymore how to handle when a crack was detected. Several options were available:

1. it was possible to start saving every signal as a crack once the threshold was exceeded at least in one signal.

2. Another option was to start saving each signal their respective *'crack buffer'* only when each of them transcended the threshold

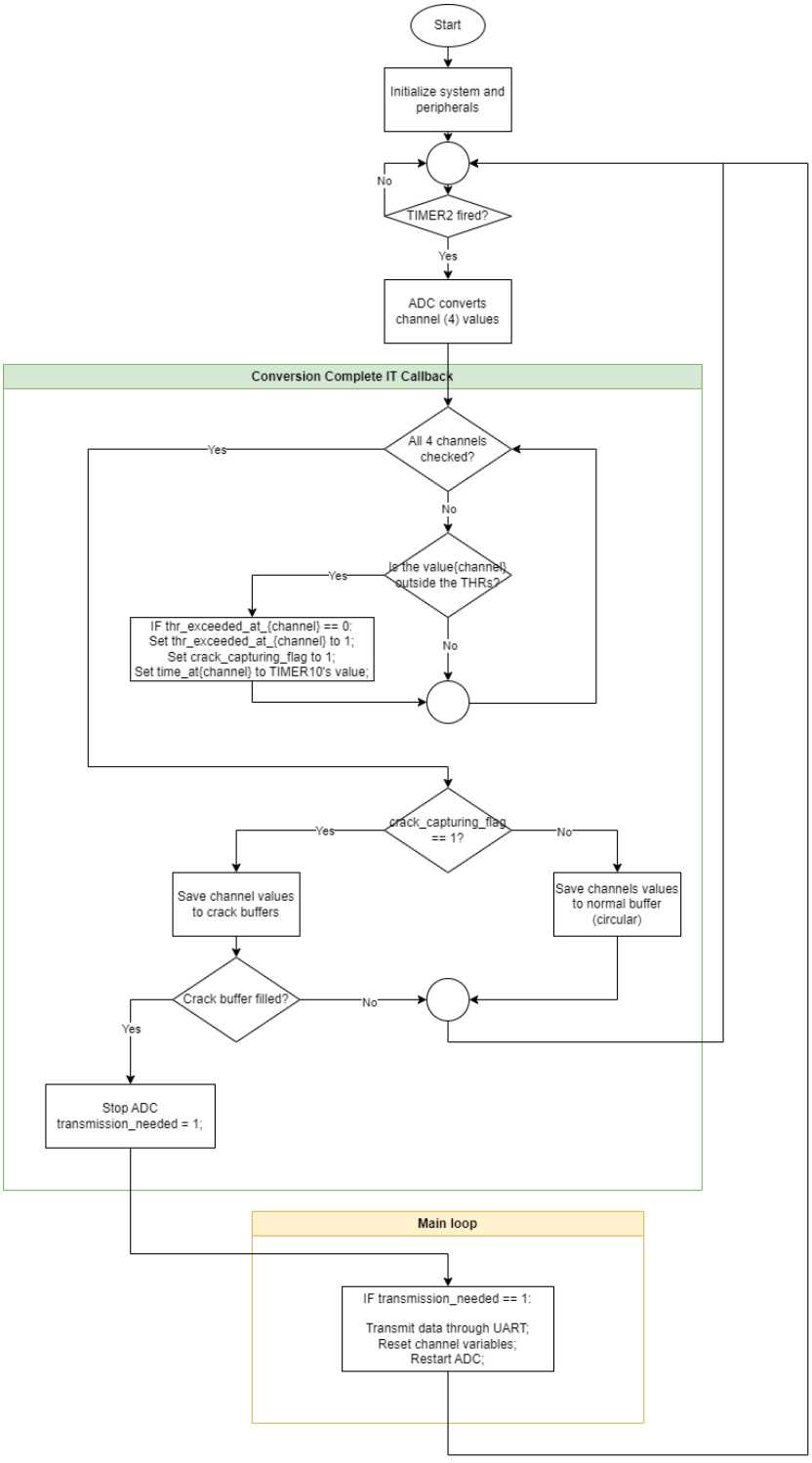


Figure 4.19: The final firmware's flowchart

3. A third alternative was to start saving each signal into their *'crack buffer'* when each surpasses their threshold independently.

Option two was quickly discarded as this solution would risk losing the crack completely or getting stuck if the threshold was not exceeded in one of the channels. Ultimately, the first option was selected for multiple reasons. First and foremost, it was the simplest to implement. Secondly, it was possible to see the arrival time by looking at either the sample index when the threshold was exceeded or the timer value saved for each channel when the particular channel exceeded the threshold. This solution provides the easiest way to compare the different channels. Option three would not offer good comparability and would have a more complex implementation.

Consequently, once one of the channels surpassed the threshold, all channels had to be saved into their respective crack buffers. As the channels' values arrived at the same time - once all four ADC channels finished converting and the interrupt fired - it was possible to use the same code to check if the value was out of the threshold. For this reason, a function was created that implemented this, visible in 4.8.

Listing 4.8: The check threshold function

```
1
2  static inline void check_threshold(uint16_t val, volatile uint8_t *
       thr_exceeded, uint16_t *time_at_crack, uint16_t *onset_idx) {
3      /*
4       * checks if the current values is outside of the threshold given
        it has not been exceeded before
5       * sets crack capturing flag to signal that crack happened
6       *
7       * args:     value to check against
8       *           whether the threshold has been exceeded before
9       *           time at crack to update if check succeeds
10      *           onset index if check succeeds
11      */
12     if (*thr_exceeded == 0 && (val > THR_HIGH || val < THR_LOW)) {
13         *thr_exceeded = 1;
14         *time_at_crack = __HAL_TIM_GET_COUNTER(&htim10);
15         crack_capturing_flag = 1;
16         *onset_idx = crack_it - NORMAL_ARRAY_LEN - 2;
17     }
18 }
```

For each channel, the function checks if their respective value has exceeded the upper or the lower threshold, given that it had not exceeded it previously. If this is the case, it sets the given channel's *thr_exceeded* flag to one, saves the current time, and the current index which will be useful to visualize the onset time, and sets the

global *crack_capturing_flag* to one, notifying that a crack was detected in one of the channels. As explained before, it starts saving the converted data in different buffers. It is vital to notice the *inline* decorator in the function definition. The *inline* decorator notifies the compiler to replace the function with the actual code rather than perform the standard function call mechanism, enhancing performance by diminishing the function call overhead. Therefore, readability and maintainability increase while maintaining performance.

Peripherals

Various changes had to be made to the components and how they were used to enable multi-channel sampling. The most significant transformation was made to how the ADC is operated. As the goal was to move to multiple channelss this had to be enabled in the ADC. To do this, one has to set the *ScanConvMode* to *ENABLE*, which directs the ADC to scan through the defined channels. To tell the firmware how many channels are active, the *NbrOfConversion* variable was set to *4*. Generally, the Nucleo F446-RE microcontroller uses one physical pin for multiple purposes, restricting the parallel use of specific components, including the ADC. This property means that out of all the available ADC channels, not all could be used contemporarily. For the *ADC1* that was used for the thesis, channels *1, 2, 4, 8* were selected to ensure a conflict-free operation. Moreover, each channel had to be configured identically to how it was done for the single channel configuration (see 4.1).

An essential and obvious consequence of having four channels instead of one is that not one but four values had to be read and handled circularly, meaning it was not possible with the previous code structure. To facilitate this, a DMA had to be used to copy the ADC values to an array upon conversion of each channel. This feature made handling the value and performing the same actions as before possible. Notably, the ADC's *EOCSelection* was set to *ADC_EOC_SEQ_CONV*, which means an interrupt is only raised once all four channels have been converted. Other than this, the *DMAContinuousRequests* also had to be enabled, and now the ADC had to be started with the DMA mode (*HAL_ADC_Start_DMA()*). The relevant DMA initialization code can be seen in 4.9.

Listing 4.9: DMA initialization code for the ADC

```
static void MX_DMA_Init(void)
{
  __HAL_RCC_DMA2_CLK_ENABLE();
  /* DMA interrupt init */
  /* DMA2_Stream0_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn);
}
```

Crucially, the ADC had already struggled to keep up with the pace dictated by the software trigger when only one channel was enabled. However, with only one channel it was capable to process data without major crashes. Thus, it was expected that it could not handle four channels commanded by a software trigger. To avoid these problems, the ADC's conversion start was changed from a software trigger to a hardware trigger. Most often, this hardware trigger is a clock source. For this project, *Timer2* was selected to fulfill this task. Consequently, as now a periodic hardware signal started the conversion, the *ContinuousConvMode* was set to *DISABLE*. Moreover, as the ADC had to use *Timer2* as a trigger, *ExternalTrigConv* was set to *ADC_EXTERNALTRIGCONV_T2_TRGO* and *ExternalTrigConvEdge* was set to *ADC_EXTERNALTRIGCONVEDGE_RISING* to start the conversion at the clock's rising edge. Using an external hardware trigger permitted to decrease the *SamplingTime* to *3* cycles. The completed ADC initialization code can be seen in 4.10.

Listing 4.10: ADC initialization for 4 channels, with DMA and HW trigger

```
1  static void MX_ADC1_Init(void)
2  {
3    ADC_ChannelConfTypeDef sConfig = {0};
4
5    /* Configure the global features of the ADC (Clock, Resolution,
       Data Alignment and number of conversions)
6    */
7    hadc1.Instance = ADC1;
8    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
9    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
10   hadc1.Init.ScanConvMode = ENABLE;
11   hadc1.Init.ContinuousConvMode = DISABLE;
12   hadc1.Init.DiscontinuousConvMode = DISABLE;
13   hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
14   hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T2_TRGO;
15   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
16   hadc1.Init.NbrOfConversion = 4;
17   hadc1.Init.DMAContinuousRequests = ENABLE;
18   hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
19   if (HAL_ADC_Init(&hadc1) != HAL_OK)
20   {
21     Error_Handler();
22   }
23
24   sConfig.Channel = ADC_CHANNEL_0;
25   sConfig.Rank = 1;
26   sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
27   if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
28   {
29     Error_Handler();
```

```
30    }
31
32    sConfig.Channel = ADC_CHANNEL_1;
33    sConfig.Rank = 2;
34    if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
35    {
36      Error_Handler();
37    }
38
39    sConfig.Channel = ADC_CHANNEL_4;
40    sConfig.Rank = 3;
41    if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
42    {
43      Error_Handler();
44    }
45
46    sConfig.Channel = ADC_CHANNEL_8;
47    sConfig.Rank = 4;
48    if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
49    {
50      Error_Handler();
51    }
52  }
```

The timer followed a relatively standard initialization. The frequency was chosen to be 1 MHz by setting the *Prescaler* to *1 - 1* and setting the *Period* to 84 - 1 to achieve the 84 MHz peripheral clock that powers the general-use timers. When the Period reaches the maximum value, an interrupt starts the ADC conversion. The only irregularity was that the *MasterOutputTrigger* had to be set to *TIM_TRGO_UPDATE* to serve as the ADC trigger. The timer initialization code can be seen in 4.11.

Listing 4.11: Timer 2 initialization that triggers the ADC

```
1  static void MX_TIM2_Init(void)
2  {
3    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
4    TIM_MasterConfigTypeDef sMasterConfig = {0};
5
6    htim2.Instance = TIM2;
7    htim2.Init.Prescaler = 0;
8    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
9    htim2.Init.Period = 84 - 1 ;
10   htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
11   htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
12   if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
13   {
14     Error_Handler();
15   }
```

```
16    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
17    if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) !=
       HAL_OK)
18    {
19      Error_Handler();
20    }
21    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
22    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
23    if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig)
       != HAL_OK)
24    {
25      Error_Handler();
26    }
27  }
```

The UART communication has also changed, now having a higher baud rate of *115200*. The data format was updated, the channel ID was also transmitted, alongside the onset index showing at which sample the crack exceeded the threshold, if it did at all. The updated data format can be seen in Figure 4.20. The communication was changed back to blocking operation as two consecutive cracks quickly happening after one another is believed to have a low probability. In any case, the length of the crack buffer can be increased arbitrarily to make sure to include any potential after effects. For the time of the transmission, the ADC conversion stops, and once all data is transmitted, the conversion restarts.

0	1	[2, N + 1]	[N + 2, N + 2 + C]	N+2+C+1	N+2+C+1+1
Channel ID	Index	Normal data (N)	Crack data (C)	Time at crack	Onset time idx

Figure 4.20: Multi channel UART data frame

Python

The Python client did not undergo radical changes compared to the previous version; instead, more quality-of-life improvements were performed in addition to the necessary components to support multi-channel processing and visualization.

The most noteworthy feature was adding interactive, live data visualization utilizing *matplotlib*'s *plt.ion()* function that displays the data in a new window as it arrives from the board. The channels now had predefined colors for each channel to evade confusion. Two horizontal lines are also added that represent the lower and higher thresholds when the data is plotted. Using the onset index coming through the UART two magenta colored dots are printed over the channels to show the users where the channels surpassed the threshold. In case they did not, the dot is not printed on the graph.

Other than this, it was also enabled to save the raw data coming from the UART sensor before it went through any preprocessing. This allows the script to use this data to simulate the microcontroller's behavior in case it is not present.

Furthermore, two new visualization functions were created that can plot the saved and processed data, either per crack (i.e. four channels on one plot) or per channel.

Finally, an additional function was created titled *find_onset_time*. This function can determine the channels' onset time more precisely than the microcontroller does. To do this, the function computes the average of the first N samples and from then on, computes the cumulative sum, adding one new sample to the previous sum at a time to compute a new average. The onset time is found when the currently analyzed sample is outside the previously defined average. This results in a more optimal onset time definition. The function returns the index of the onset time as well as the lower and higher thresholds that the user can define. The function's use can be seen in 4.21. In the plot, the magenta dot shows the onset time determined by the microcontroller, while the red dot shows the onset time calculated by the Python client. Although not perfect, the Python client notably improves the onset time placement.

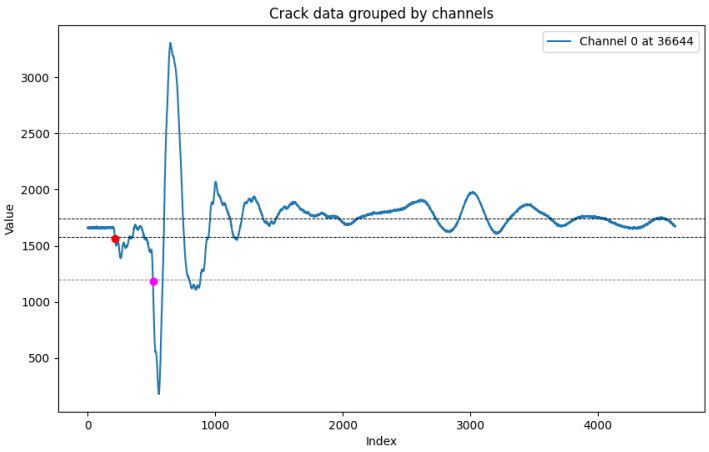


Figure 4.21: Onset time determination in Python

The updated Python client code can be seen in B.3.

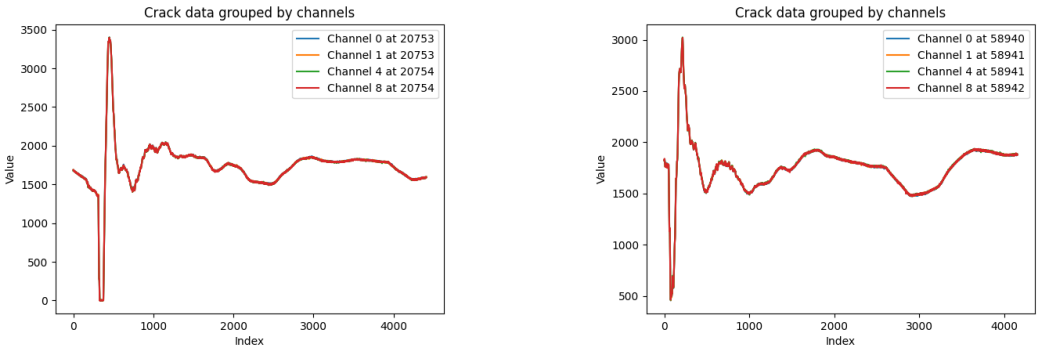
Results

Before the final real-life tests, preliminary trials were performed to verify that the solution worked as expected. The tests used the same acoustic emission sensor connected to the four different ADC input pins. Despite generating the same data

four times, this setup could verify if the whole processing chain worked as expected and visualize if the captured data arrived correctly at the test PC. Moreover, this also meant that it was not necessary to build four different instances of the same amplification circuit; it was enough to use one.

After creating the data and performing PLB tests on a flat surface where the PE sensor was lying, the returned results were reassuring. They showed that the solution could process and visualize the data in real-time, detect cracks when they occur, transmit them via the serial link, and display them on the Python client.

Figure 4.22 shows some of the obtained results. It is essential to say that it is normal for the data to be identical for all four channels as they were fed with the same data. On the contrary, the more similar the data, the better it is, which means the system does not suffer from big noise. It can be seen that there are only minor differences in the corresponding signals for the different channels. Another important detail is that the signals' arrival times are similar; they only differ usually in 2 clock ticks. This observation will be further elaborated in the Results Chapter 5.



(a) Crack A **(b)** Crack B

Figure 4.22: Results of 4 different ADC channels using the same sensor

These results confirmed the correctness of the developed embedded solution. This allowed the focus to shift to the final phase of the work, which involved assembling three other circuits and performing real-life PLB tests on concrete and marble blocks.

Chapter 5

Results

This chapter details the preparatory steps, test setup, and methodology leading up to the final tests on the embedded solution. It then presents the conducted tests, analyzes the obtained results, and interprets their significance. Additionally, the chapter includes an evaluation of the solution's maximum performance.

The simulated test with one PE sensor duplicated to the four ADC inputs gave satisfactory results, so it was time to build three additional amplifier circuits to utilize all four PE sensors. A significant difference exists between the three other circuits constructed at this point and the original circuit. While the original circuit employed a special BNC to wire/banana cable, only one such device was available. For this reason, the new circuits operated with classical BNC-crocodile cables. Although they fulfilled their role, their comparably longer length meant they had more significant parasitic capacitances and were more noise-prone. Otherwise, the three additional circuits and the sensors were identical. The circuits were built on a standard breadboard on three separate lanes to make them more modular.

After building the three circuits and ensuring each was correctly assembled and free of mechanical or component errors, it was time to perform PLB tests. Two different test configurations were created: the first test sequence was performed on a marble block, while the second test configuration was executed on concrete. The test setups can be seen in 5.1 and 5.2. The author wishes to highlight the numbering of the sensors, as it will serve as a source of important analysis. Sensor 1 corresponds to channel 0, sensor 2 equals channel 1, sensor 3 is transformed by channel 4 and finally, sensor 4 is converted by channel 8. Another important point is that sensor *two* and sensor *three* were not properly touching the surface of the stones, which meant that the data received from this sensor was suboptimal. This happened because the cable's weight lifted one end of the sensor. Unfortunately, it was impossible to attach the sensors properly to the surface without permanent glue. The paper-based tape used to remedy the negative effect on sensor two was also insufficient in both configurations. To show that the problem was only a matter

of the improper contact specific tests were carried out to show that the sensors were functional. Furthermore, older results also back up this fact. Originally, the PLB tests were performed using a mechanical pencil with a lead of 0.5 mm, but this has proven to produce too weak of a signal. Thus, the mechanical pencil was substituted with a screwdriver with a cone head to generate a stronger signal.

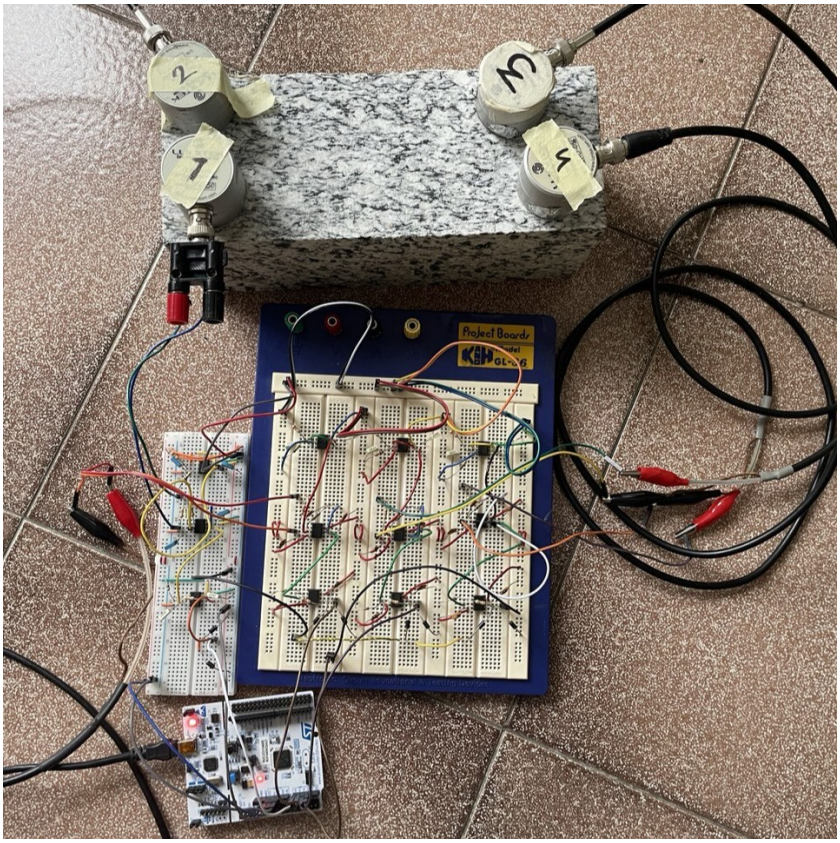


Figure 5.1: Test setup on marble block

At first, the PC used to process the serial data was plugged into the power supply. After the first tests were performed, data did not show the expected behavior when the transmitted signals were plotted. Such a result can be seen in Figure 5.3. After some investigation, it turned out that the AC voltage crept into the circuit, causing a significant noise that destroyed the AE signal. Hence, it is essential to note that the current setup only works with the PC operating from the battery. This behavior can happen because the capacitors responsible for eliminating the noise do not successfully eradicate the power supply's AC voltage. This error could be fixed by changing the decoupling diodes to ones with proper capacitance values. Another point to consider is changing the charge amplifier so that its HPF better attenuates the 50 Hz AC signal.

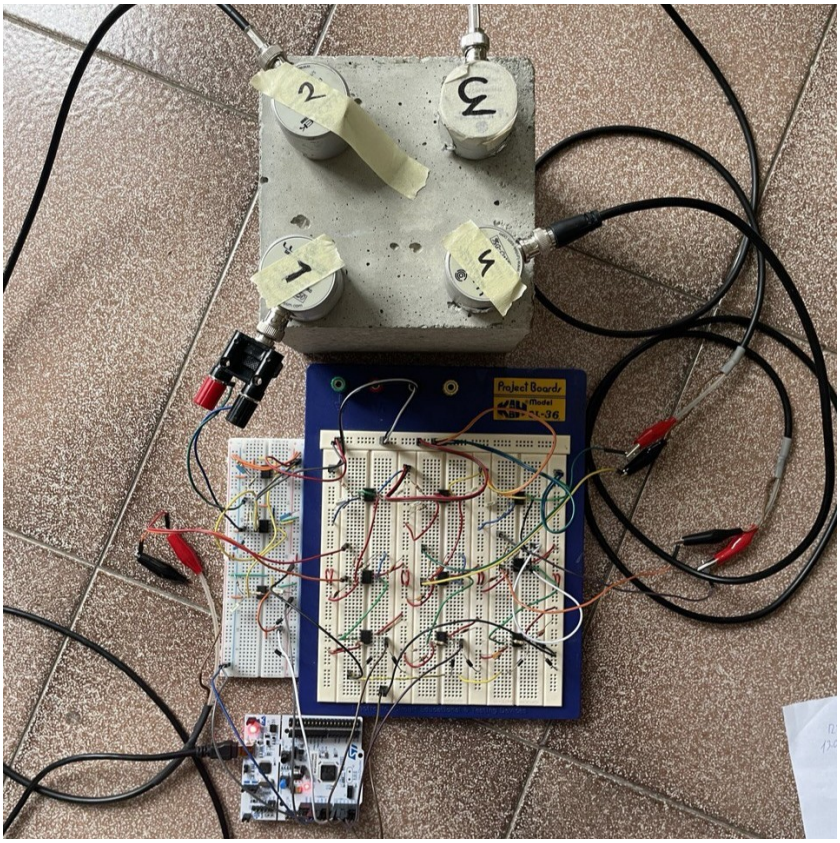


Figure 5.2: Test setup on concrete block

However, when the power supply was unplugged from the PC, the received signals followed the expected behavior. Four different tests were performed after each other, in both the marble and the concrete configurations:

1. A PLB test was performed close to sensor 1,
2. The next test was performed close to sensor 2,
3. The third test was performed between sensor 3 and sensor 4,
4. The last test was performed in the middle of the block.

5.1 PLB tests on marble

The first PLB test's result can be seen in Figure 5.4. This figure plots the results against each other for better compatibility. It can be seen that sensor 1 detects the impact the fastest. According to the threshold surpassing (magenta dots),

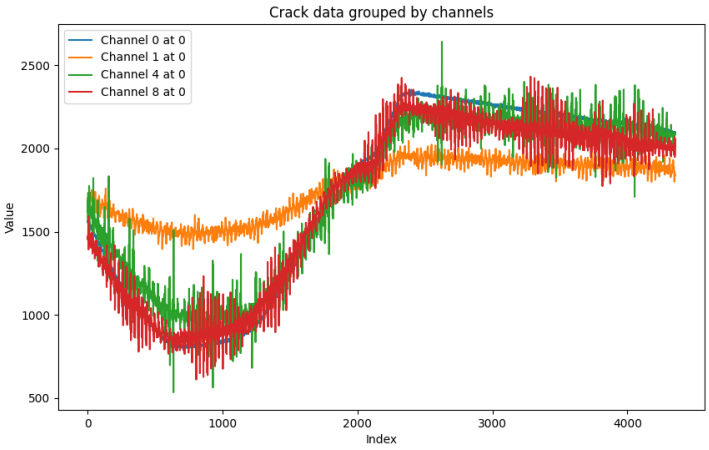


Figure 5.3: Received data when the power supply's noise affected the original AE signal

the second sensor to detect the crack is sensor 3 and channel 8. Despite sensor 2 (channel 1) exceeding the threshold much later, it has a significantly larger maximum peak, suggesting it was closer to the source of the impact. This notion is supported by the fact that the onset time detecting algorithm in Python (black dots) selects this signal as the second in the order. It can also be seen that channel 0 and channel 1 capture almost identical signals, barring some magnitude differences. Finally, channel 4 did not detect a crack at all, which is due to improper contact with the surface.

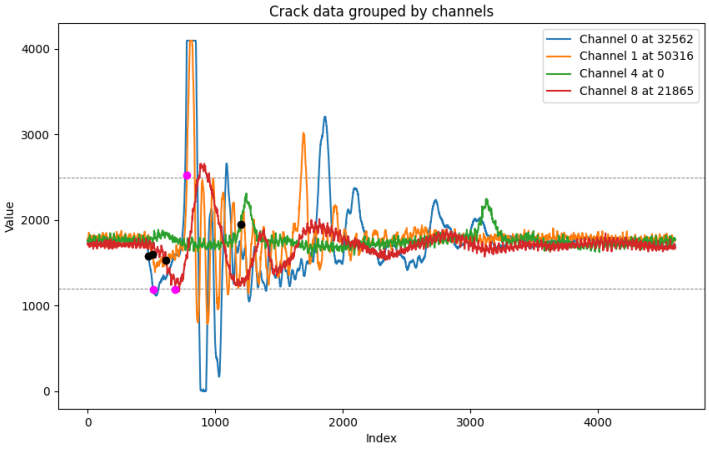


Figure 5.4: Crack on marble, close to sensor 1

Looking at the timer values for each channel, the received numbers might seem to be counterintuitive. However, it is important to highlight that the clock to command *Timer10* had a frequency ten times higher than the clock of the ADCs. Hence, channel 8's smaller timer value is because the timer has overflowed and started counting again, from 0 in the meantime.

The second PLB test was conducted by making an impact near sensor 2, visible in Figure 5.5. Despite the deliberate effort, neither sensor 2 nor sensor 3 — channels 1 and 4, respectively — detected the crack. A deeper analysis reveals that channel 4 captured a weaker signal, indicating it was somewhat in contact with the surface. This is confirmed by the placement of the onset detection algorithm of the Python script (black dot on the green graph). Channels 0 and 8, captured the signal accurately, with post-determined onset times close to each other and to channel 4.

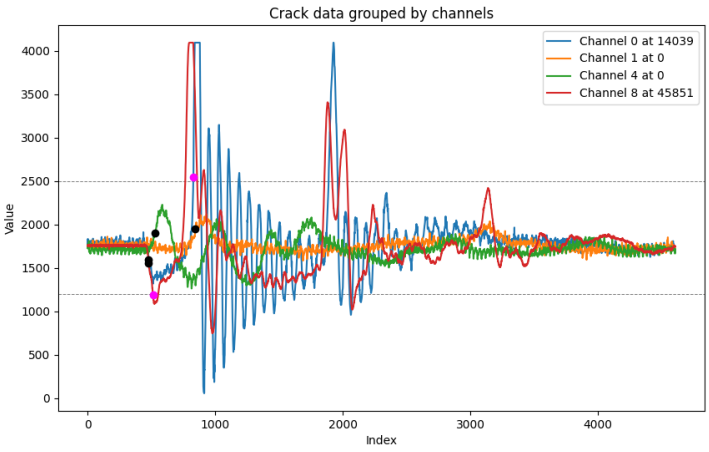


Figure 5.5: Crack on marble, close to sensor 2

The third test was performed in the vicinity of sensor 4 - see Figure 5.6. In this case, the sensor easily captured the signal, having the highest amplitudes out of the four channels. Even though according to the data channel 0 did not capture the crack, the reader can notice it was only a matter of a few millivolts as the sensor one's peak was only a little below the 2500 mark. Channel 8 detected the crack essentially at the same time as channel 4 but with an opposite phase, and with smaller amplitude. Finally, the crack arrived at channel 1 at last having the smallest voltage swings as well.

Finally, the last PLB test was executed by making a crack in the middle of the marble block that can be seen in Figure 5.7. In this case, channels 0, 4, and 8 registered the cracks essentially at the same time, with channel 4 having a slightly smaller peak than the other two channels. Channel 1 had a much weaker signal,

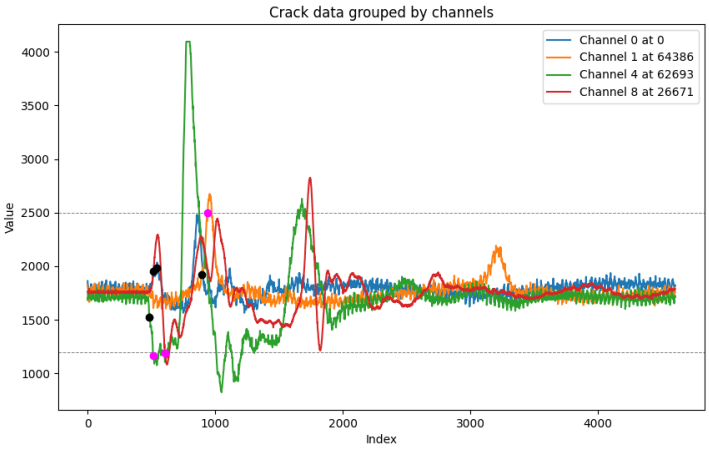


Figure 5.6: Crack on marble, close to sensor 3

but ultimately the threshold was crossed for this channel. It can be seen that the Python function also placed the onset times for the three normal channels at the same time. Even though channel 1's onset times (both the magenta and the black) were placed much later than for the other channel, it can be discerned that this signal also started fluctuating at the same time as the others; even if only a little.

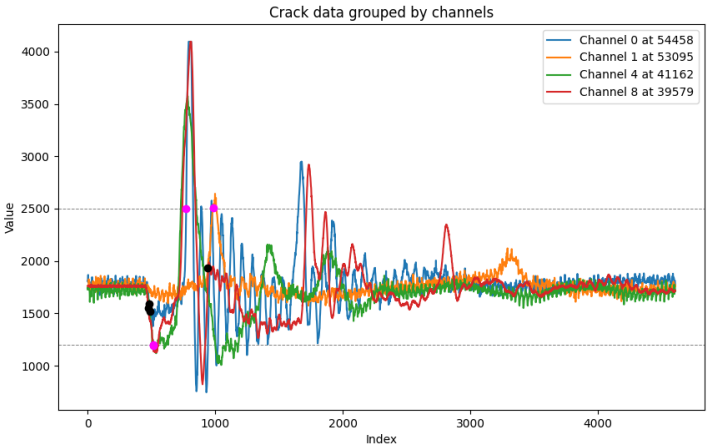
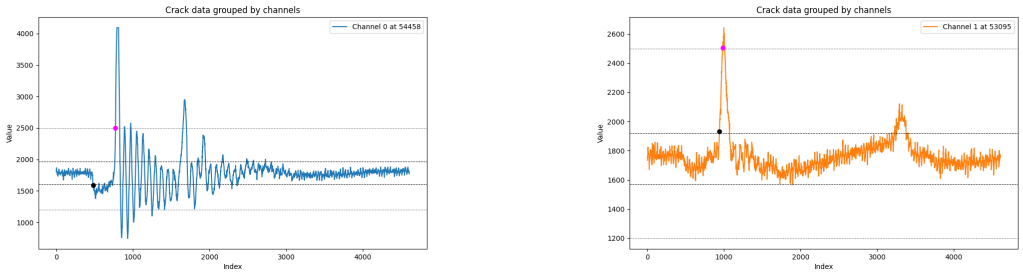


Figure 5.7: Crack on marble, in the middle of the block

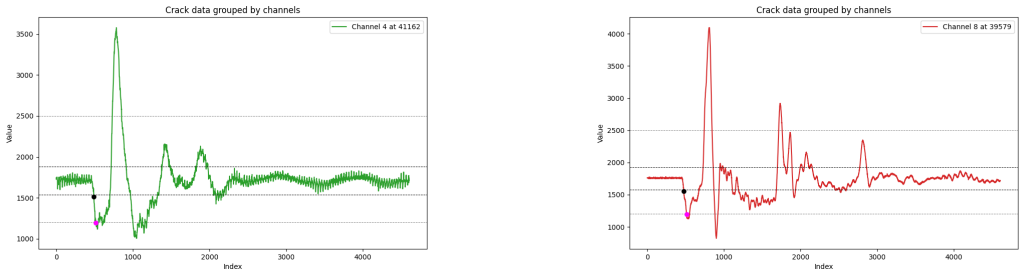
It is beneficial to inspect the channels separately for this test as the channels show fairly similar properties. Looking at the separate plots 5.8, 5.9, it can be stated that channels 4 and 8 exhibit a similar start of the crack, both of them

starting with a voltage dip, crossing the lower threshold line, which is followed by a massive voltage swing towards, the maximum voltage value. This signal portion is followed by a fairly standard attenuation. The microcontroller's onset index is similar for both channels (magenta), and the calculated onset time by the Python script (black). Overall, channel 0 also presents analogous properties; the most important differences are that this channel lacks the initial dip in the voltage and has much more frequent voltage swings than the other two channels. The latter leads to a later threshold surpassing, but the Python script places the onset time in the same area as for the other two signals. Finally, channel 1 did not behave as the other three channels. However, the separate plot helps us confirm that indeed there is a voltage dip at a comparable point in time, even though the high noise of this channel conceals it. The large positive peak is also present in this channel even though it is lesser than in the other channels.



(a) Channel 0 **(b)** Channel 1

Figure 5.8: Channels shown separately for the PLB test in the middle - 1



(a) Channel 4 **(b)** Channel 8

Figure 5.9: Channels shown separately for the PLB test in the middle - 2

Analyzing the arrival times, it can be seen that the time difference between the signals is quite significant. To have a proof of concept about the distance

measurement it is advantageous to compare the closest arrival times, that is channel 8's and channel 4's. *Timer10* was set to have a clock frequency of 84 MHz, yielding a 18.834μs time difference of onset times, that is derived in 5.1.

$$\Delta t = (41162 - 39579) \cdot \frac{1}{84000000Hz} = 1583 \cdot \frac{1}{84000000Hz} = 18.845\mu s \quad (5.1)$$

Assuming that the acoustic emission signal has an estimated travel speed of 3500 m/s in marble and in concrete [86], Equation 5.2 yields a difference of around *6.5 cm-s* between the two sensors, which is very precise when looking at the test setup and the placement of sensor 3 and 4 (see 5.1). This result underlines the correctness of the solution. However, it must be said that this computation did not use the proper formulas, so it can only be used as an estimate.

$$\Delta s = 3500m/s \cdot 18.845\mu s = 0.0659m = 6.59cm \quad (5.2)$$

Although the obtained results are more than satisfactory, it might be considered to slow down *Timer10*'s frequency. Even though the high 84 MHz clock gives the highest precision for cracks that are close to each other, a higher frequency might make the measurements more difficult for sensors that are further from each other, because of the overflows. To handle this either a variable had to be introduced to store the number of timer overflows or the timer's frequency had to be slowed down. As this decision is application-dependent, the thesis did not consider selecting a proper frequency, it focused more on the maximum achievable performance which is why the 84 MHz was selected.

5.2 PLB tests on concrete

Looking at the first result performed on a concrete block that had an impact close to sensor 1 5.10, it can be seen that the crack first arrives at sensor 1, then shortly after at sensor 4 - depicted by channels 0 and 8 on the plot, respectively. Both channels have a peak of the ADC's maximum, 4095 value, but the dip preceding this peak is greater in absolute value for channel 0. It can be seen that although the microcontroller detected the crack much later (magenta dot on the yellow plot) than it did for the previous two channels, the Python code places the onset time almost at the same time as for the previous two channels (black dots). This confirms that in reality the the crack arrived at sensor 2 much earlier than what is shown by the timers. Channel 4 however did not record a crack, due to the unreliable contact with the concrete's surface.

Even though the second PLB test created a crack close to sensor 2, it could not record the crack 5.11, due to the wrong surface contact. Unlike channel 1,

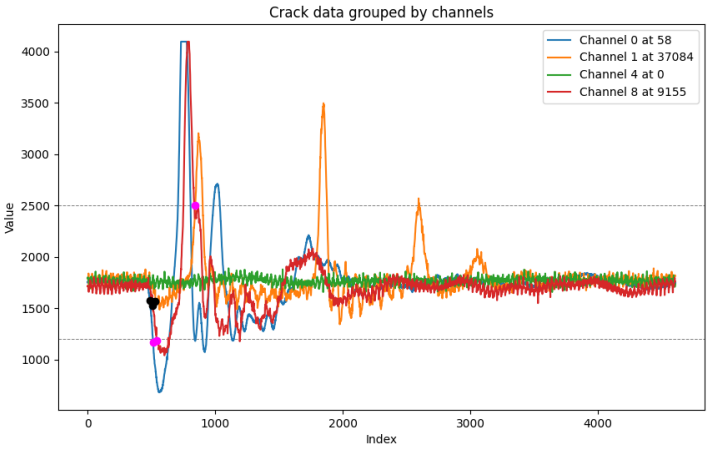


Figure 5.10: Crack on concrete, close to sensor 1

channel 0 and channel 8 recorded a crack with a big low dip, that already crossed the threshold. This cannot be stated for channel 0, even though this channel also had a voltage dip at the same time as channel 8, but it did not penetrate the threshold. Channel 4 did not cross the threshold, but it came close to it at a similar time as the other two channels that did detect a crack. The Python script's onset time detection places the onset time close to the other two channels (black dots). Channel 1 remained so stagnant that even the Python code did not manage to determine an onset time.

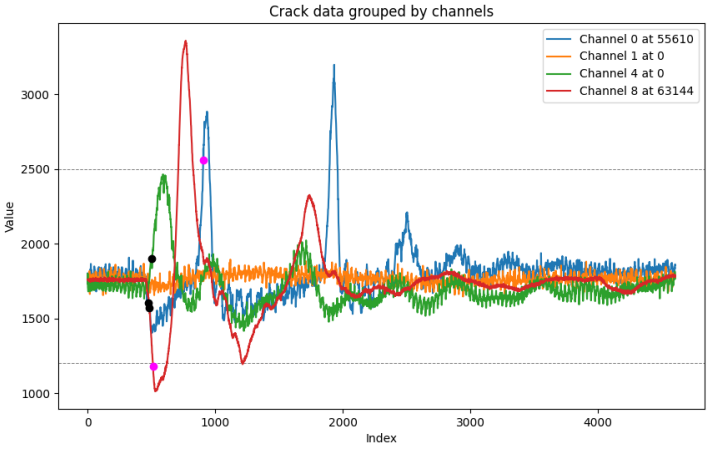


Figure 5.11: Crack on concrete, close to sensor 2

The third PLB test impacted sensors 3 and 4 5.12. Channels 4 and 8 managed to pick up these cracks almost at the same time but with opposite voltage swings. Indeed, there is only a 1000-tick difference between the two onset times, which translates to 12μs-s. Both channels register the maximum peak at 4095, with similar characteristics over the whole signal span. Instead, channel 0 only detects the crack at the time when the other two channels register the maximum peak. Unlike these channels, channel 1 did not register the crack, but there is a minimal dip in the middle. This is, however, so small that the Python code couldn't detect it either.

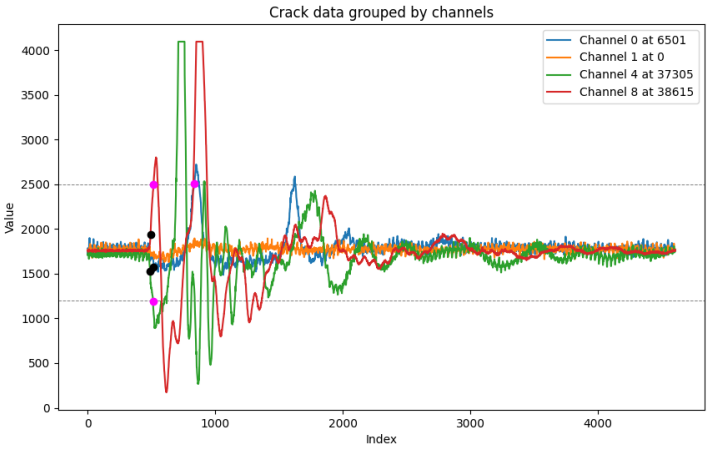


Figure 5.12: Crack on concrete, between sensor 3 and 4

Finally, the last PLB test hit the marble in the middle of the concrete block: 5.13. Again, channel 0 and channel 8 passed the threshold at the same time during the negative voltage swing, while channel 1 recorded it only during the positive voltage swing. However, the Python code's onset detection algorithm places channel 1's onset time at the same point as the other two channels. It can also be witnessed that all three channels register a similar crack evolution over time. Channel 4 did not record any cracks.

In conclusion, it can be stated that the PLB tests performed on the concrete block exhibit similar properties as the PLB tests on the marble block. What cannot be seen from these results, is that it took more runs to create appropriate results, because the sensors often failed to record the signals. This is probably due to the structural differences between the two materials. Despite this, the embedded solution worked well on this material as well.

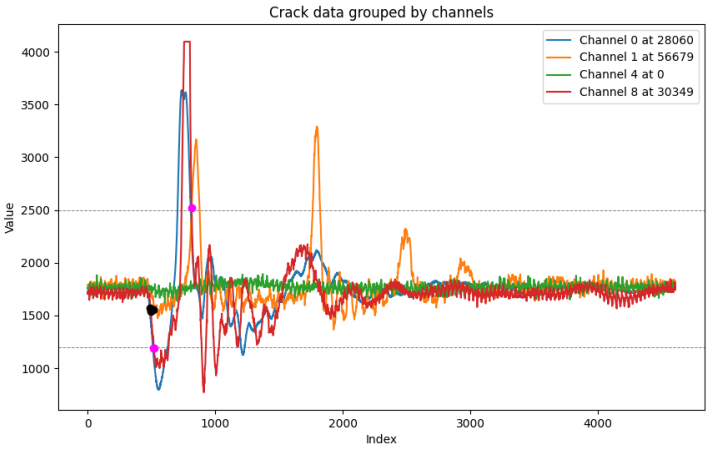


Figure 5.13: Crack on concrete, in the middle of the concrete block

5.3 Analysis of the maximum performance

At this point, it is helpful to study the maximum achievable sampling time, code execution, and precision with the current setup and determine the embedded solution's upper-performance limit.

Inspecting the maximum speed of the ADC, it can be seen from the board's datasheet [77] that every ADC is powered by the *APB2* clock, whose maximum clock frequency is 90 MHz. In our setting, *APB2* is set to 84 MHz, with an *ARR* register minimally being (2 - 1), resulting in a 42 MHz frequency. Hence, the theoretical maximum frequency for the ADC's clock is 42 MHz. However, after further inspection of the datasheet, the ADC characteristics section states that the maximum ADC clock frequency is 18 MHz, which is still not the final result, as the solution uses an external trigger for the ADC. The related table states that the maximum external ADC trigger that can be kept up with is *1.764 MHz*. This number is the maximum frequency that the embedded system can operate with. This value aligns with the requirements as a minimum of *100 kHz* sampling was asked. In the final configuration, the ADC is triggered with a *1 MHz* external trigger. The current ADC setup uses a 12-bit resolution that requires 15 clock cycles to convert and three clock cycles sampling time, totaling 18 clock cycles. For the four channels, it is a total of 72 clock cycles. Using the maximum achievable frequency of *1.764 MHz*, it would be a total of *41 us*, while using the selected *1 MHz* it results in *72 us*. Using two ADC-s instead of one could half the required conversion time to *36* or *20 us*, depending on the clock speed. Using three ADCs instead of two wouldn't bring further improvements as one ADC would need to convert two channels, remaining at the same speed.

Nonetheless, looking at other limiting components in the processing chain is worthwhile. Another component whose execution time can influence the results and the onset time measurement's precision is the execution of the *check_threshold* function. As *Timer10* was already initialized and used for high-speed measurements it was selected to measure and save the time before and after the function call to measure its execution time. This was performed for all function calls, which can be seen in 5.1. After making more than 50 measurements, the maximum execution time was around *40* ticks, which using the 84 MHz clock translates to *0.5 us*. This, compared to the 72-microsecond ADC conversion is negligible (less than 1 percent). This measurement also confirms the efficiency of inline functions.

Listing 5.1: The execution time measurements of the check_threshold function

```
1     volatile int t1 = __HAL_TIM_GET_COUNTER(&htim10);
2     check_threshold(raw_adc_values[0], &thr_exceeded_at0, &
    time_at_crack0, &onset_index0);
3     volatile int t2 = __HAL_TIM_GET_COUNTER(&htim10);
4     check_threshold(raw_adc_values[1], &thr_exceeded_at1, &
    time_at_crack1, &onset_index1);
5     volatile int t3 = __HAL_TIM_GET_COUNTER(&htim10);
6     check_threshold(raw_adc_values[2], &thr_exceeded_at4, &
    time_at_crack4, &onset_index4);
7     volatile int t4 = __HAL_TIM_GET_COUNTER(&htim10);
8     check_threshold(raw_adc_values[3], &thr_exceeded_at8, &
    time_at_crack8, &onset_index8);
9     volatile int t5 = __HAL_TIM_GET_COUNTER(&htim10);
```

The final component that was analyzed in the firmware was the maximum execution time of the ADC callback function. To measure its execution time, *Timer10* was used similarly; its value was saved at the beginning and the end of the callback. On average, the maximum execution time was around *360* ticks, which means *4.2 us* in total. This value is still one magnitude less than the ADC's conversion time.

Overall, it can be declared that the solution's performance satisfies the initial requirements of the 100 kHz sampling time as it would translate to a conversion every *10 us*. Despite the final configuration being a little slower (around 75 us) it has been demonstrated that the solution can reach a speed of up to 20 microseconds, which is on the required level. The results obtained also support that the system achieves satisfactory results.

Chapter 6

Conclusion

This thesis developed an embedded system capable of detecting acoustic emission signals in real-time using piezoelectric sensors. It introduced the necessary background knowledge to understand the field. In the first phase, in-depth research was performed on state-of-the-art technologies in structural health monitoring.

The second phase focused on developing an appropriate analog circuit to transform the piezoelectric sensor's signal to increase interpretability. A charge amplifier was chosen to transform the current to a more usable voltage level using an op-amp. The signal was shifted to the midway of the ADC's input range by another op-amp and amplified by a third op-amp stage. The developed circuit was employed with Schottky protection diodes to shield the microcontroller from harmful ESD strikes and overvoltages of different natures. Thereafter, the focus was switched to the embedded firmware development, where a program was created that could sample four ADC channels in parallel, detect acoustic emission signals, and transmit them over a serial connection for later processing. An ADC was configured to convert the four sensor values connected to its four channels, commanded by an external timer trigger, and moved to memory by a DMA instance. The communication was executed using a UART protocol, and a proprietary data frame was designed. The transmitted data was then displayed and saved by a Python client. The client could interactively process and display the received data on the fly.

The thesis concluded by building a total of four identical amplifier circuits to enable the platform to localize the crack in 3D. It was followed by testing the developed embedded solution on both a marble and a concrete block with PLB tests, demonstrating that the system can process and detect the generated signals in real-time. Results showed that the embedded solution can process the data with the required speed and precision to determine the onset time and localize the crack inside the stone block. The utilized timers provide a possibility to determine the position of the crack, which can be implemented in the future. An analysis of the performance and the maximum execution time was performed, yielding that the

solution complies with the initial requirements.

Even though the thesis provided a verified solution to a pressing problem in the structural health monitoring domain, there is room for improvement in various areas. Even though the developed analog circuit does not require a power supply other than the microcontroller's own, the system's power consumption could be further improved by utilizing the low-power modes provided by the ST ecosystem. The system could be put into a low-power state between processing samples to optimize energy consumption. As detailed in the thesis, the developed circuit often suffered from noise from the PC's power supply. Hence, another future improvement is to make the circuit more noise-resistant.

A possible problem that did not manifest during the PLB tests is that the sensors might not be sensitive enough to detect more distant cracks coming from tens or hundreds of meters, so this could also be improved in the future. A good advancement of the project is to implement a crack localization algorithm either in the Python client or in the board itself. An AI agent could also be trained on the data generated with the embedded solution, and this agent could be deployed in the embedded system to evaluate the agent's capability of detecting the onset time. Finally, another consideration is to redesign the sensor and the amplification circuit to have a wireless connection to the central node to lose the constraints posed by limited cable lengths.

Appendix A

Experimental signal amplifier circuits

A.1 Circuit 1

The first circuit that was designed included four stages. The first stage provided the necessary offset of 1.65 V to the ADC's input, and the sensor connected in parallel with a $1M\Omega$ resistance. The idea behind this resistor was to discharge the accumulated charge between the sensor's two terminals. The next stage of the circuit served as an overvoltage protection. This stage included 2 1N5819 [87] Schottky rectifier diodes. The diodes were connected in series between the 3.3 Volts supply and ground, while the output of the previous stage was connected between the two diodes. The third stage was a simple RC LP filter intended as an anti-aliasing filter. The resistance and the capacitance were chosen so that the LPF had a cutoff frequency of 20 kHz.

$$f_c = 20kHz = \frac{1}{2\pi RC} \overset{R=10k\Omega}{=} \frac{1}{2\pi 10^4 C} \tag{A.1}$$

$$C = \frac{1}{2\pi \cdot 2 \cdot 10^8} = 0.79nF \tag{A.2}$$

The closest available capacitance in the laboratory was 1 nF, resulting in a 16 kHz cutoff frequency.

Finally, the circuit concluded with a non-inverting amplifier that amplified the input signal 11 times. As op amp, a TLC271 [83] was selected, as the board could power it.

The whole circuit can be seen in Figure A.1. It can be easily seen that this circuit could not perform its intended purpose for many reasons. Most trivially, even though it amplified the sensor's voltage to the correct range of +/- 1.5 V, it

also amplified the offset to a staggering 15V in theory, which would, in practice, translate to the circuit's supply voltage (5 V). For this reason, this design was rapidly discarded.

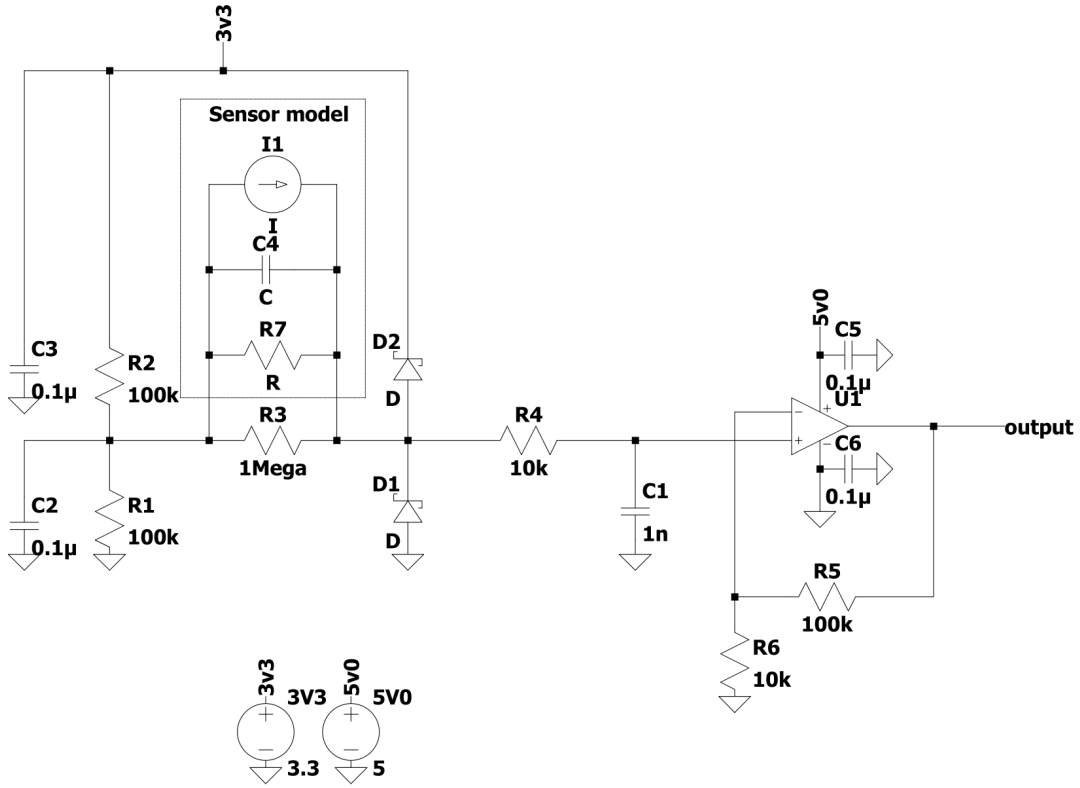


Figure A.1: Experimental circuit design 1

A.2 Circuit 2

The second experimental circuit introduced a different approach to inserting the sensor into the design. In this layout, the sensor was connected to the ground and directly to the midpoint of the protection Schottky diodes detailed in the previous design. The next stage of the circuit was the most complex part of the design, which included a non-inverting op-amp with a positive reference voltage. The design was based on a TI design tutorial [88].

The first step in the design was to choose the desired gain (G) for the circuit. Based on the sensor's voltage output and the ADC's input range, the gain can be computed by the following equation, where O denotes output, and I denotes input.

$$\frac{V_{Omax} - V_{Omin}}{V_{Imax} - V_{Imin}} = \frac{3.3V - 0V}{0.15V - -0.15V} = 11 \tag{A.3}$$

The second step was to select the values of the *R1* and *R4* resistors. These values can be selected freely; however, the values within the feedback are recommended to be less than $100k\Omega$. Following this, the value of *R1* was chosen as $1k\Omega$ and the value of *R4* was chosen as $1M\Omega$. The latter was purposefully selected to be so large as to limit the current creeping in and out through this branch.

Subsequently, the values of *R2* and *R3* were calculated using a system of equations. Skipping the detailed derivation of the solution, the exact values of the resistors are: $R2 = 91\Omega$ and $R3 = 90k\Omega$.

Nevertheless, these resistor values are very particular; they were unavailable in the laboratory, so the closest resistors used were 100 Ohm and 100 kOhm, slightly changing the offset and gain values obtained:

$$G = \frac{10^6}{10^6 + 10^5} \cdot \frac{10^3 + 100}{100} = 10 \tag{A.4}$$

$$Offset = V_{ref} \cdot \frac{R_3}{R_3 + R_4} \cdot \frac{R_1 + R_2}{R_2} = 1.65V \cdot \frac{10^5}{10^5 + 10^6} \cdot \frac{1000 + 100}{100} = 1.65V \tag{A.5}$$

These values are close to the desired theoretical numbers, so the resistors were kept.

The final two stages of the circuit were taken from the previous design. The third stage is a low-pass filter with a unit-gain op amplifier to stabilize the signal. The final portion is the clamping circuitry of two 1N5819 Schottky diodes.

The complete circuit can be seen in Figure A.2.

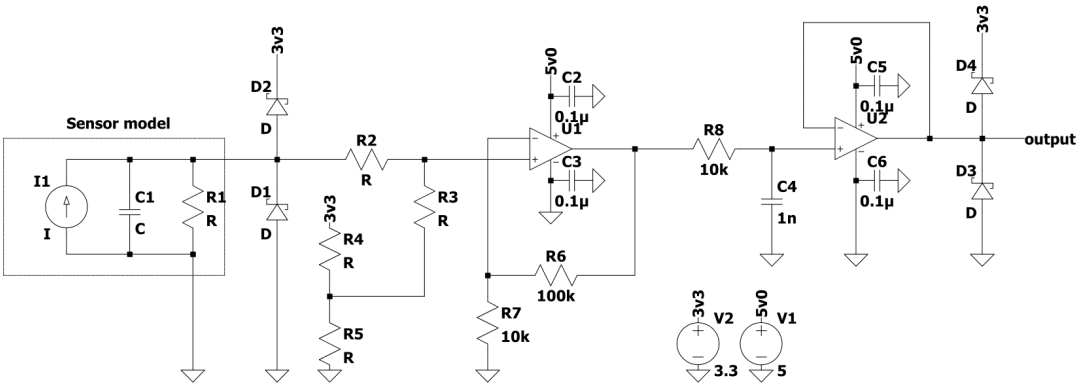


Figure A.2: Experimental circuit number 2

Firstly, the behavior of the low-pass filter was verified. The filter was tested at two different frequencies, at 3 kHz and 50 kHz. As expected, the LPF greatly attenuated the signal at a larger frequency. The results can be seen in Figure A.3, A.4, and A.5.

(a) 5 kHz input signal **(b)** 5 kHz LPF response

Figure A.3: LPF behavior at 5 kHz

(a) 5 kHz input signal **(b)** 5 kHz LPF response

Figure A.4: LPF behavior at 50 kHz

Afterward, the whole chain was tested with a signal generator and a standard power supply. The circuit behaved according to the expectations when tested with a signal generator similar to the sensor's signal. The behavior at 3 kHz can be seen in Figure A.6, while in Figure A.7, the characteristics at 33 kHz are presented.

There are, nonetheless, essential differences between the sensor's behavior and the signal generator's. While the signal generator directly creates a voltage between the two terminals it is connected, the sensor's output is current, meaning it has to pass through some resistance to create a perceivable voltage on the circuit's output. Moreover, due to the sensor's capacitive nature, the sensor behaves as a cut when the input is static, meaning it will act as if the sensor's input was left floating. In that case, the offset voltage will be sent to the output in an amplified fashion. Since the offset is around 1.5V, the circuit's output will be the supply

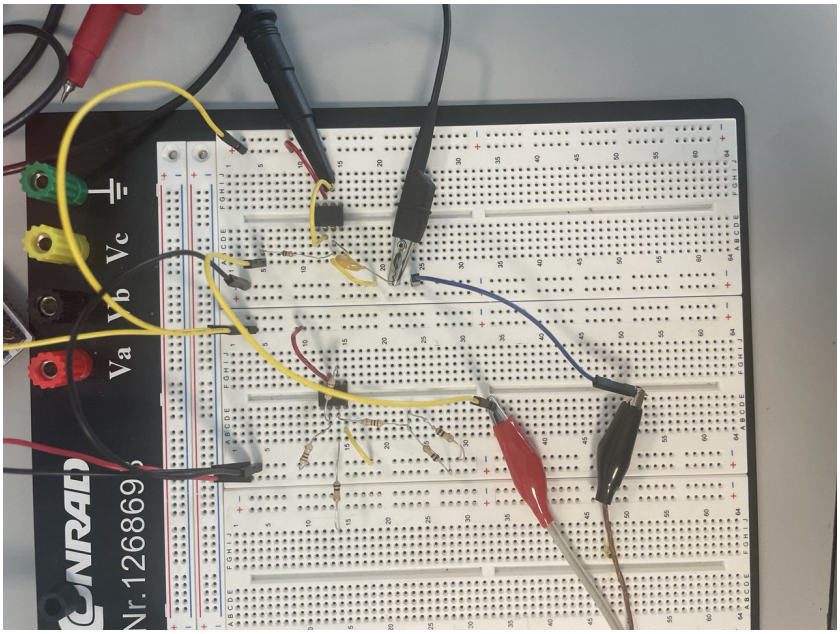
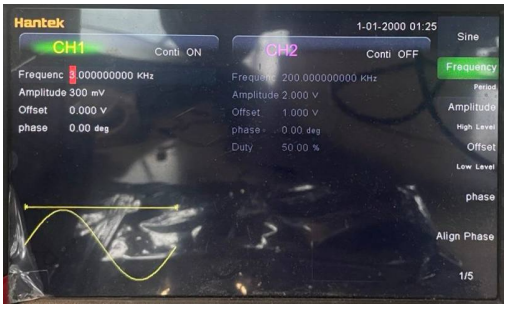


Figure A.5: Assembled circuit with only the LPF



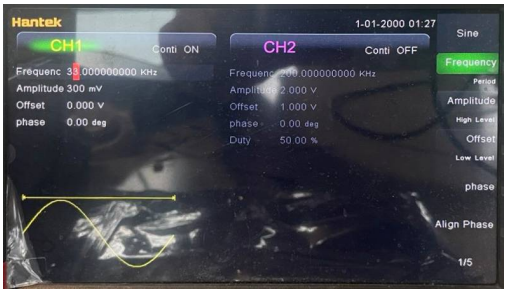
(a) 3 kHz input signal

(b) Circuit response at 3 kHz

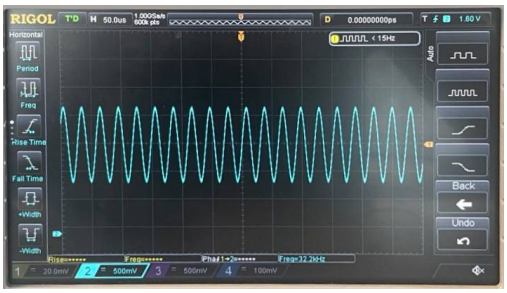
Figure A.6: Circuit behavior at 3 kHz

voltage $V_{dd} = 3.6V$. This demeanor is captured in A.8. Channel 1 (yellow) shows the sensor's signal, while channel 2 (blue) captures the circuit's output. Channel 1's vertical resolution is 50mV per grid, while channel 2's unit is 2 V per grid. It can be seen that the output is constantly at the supply voltage except when the sensor's negative voltage swing counters the constant offset voltage to decrease the overall voltage.

It is essential to mention that the fact that the sensor was outputting current, not voltage, was not yet understood at this development stage. Subsequently, the following design did not try to solve this issue but the problem of the amplified



(a) 33 kHz input signal



(b) Circuit response at 33 kHz

Figure A.7: Circuit behavior at 33 kHz

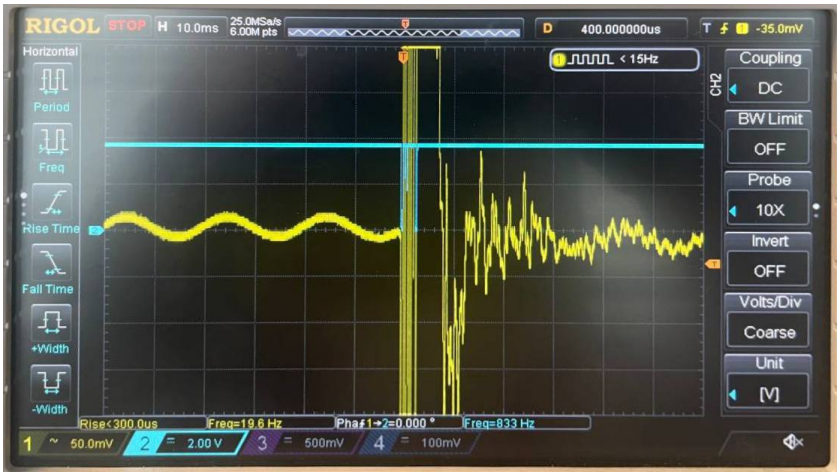


Figure A.8: The circuit's response to the sensor's signal

offset voltage for constant inputs.

A.3 Circuit 3

Another more precise design was created to solve the incorrect output signal perceived in the previous design. This layout included a precision component, an INA, discussed more in detail in 2. Particularly a TI INAx126 [89]. An important characteristic of this circuit is the change to a dual-supply topology to power the INA introduced. The sensor was connected to the INA's input terminals. This active component features a configurable amplification that can be set by placing the appropriate resistor between the dedicated terminals. To achieve a $\tilde{1}0$-fold amplification a $15k\Omega$ resistor was chosen. Decoupling diodes were placed on the supply lines to eliminate every possible noise entering the component, powering

the INA.

The other input branch of the circuit included a voltage reference with a negative feedback unit gain inverting amplifier. As for now, there was no need to place single supply components, so a different, dual-supply op-amp was used, the TL082 from Texas Instruments [82]. The offset voltage was set to 1.65V and consisted of 2 resistors.

The final addition to the design was the introduction of the summing amplifier, also discussed in 2. Further amplification was unnecessary as the sensor's signal was already amplified to the ADC's correct range. Consequently, every resistor inside the summing amplifier was set to be 22 kOhms to obtain a unit gain.

Finally, the same LPF and clamping diodes were added to the circuit as in the previous design. However, this one was also changed to a TL082 like the other op-amp. The complete circuit's design can be seen in Figure A.9.

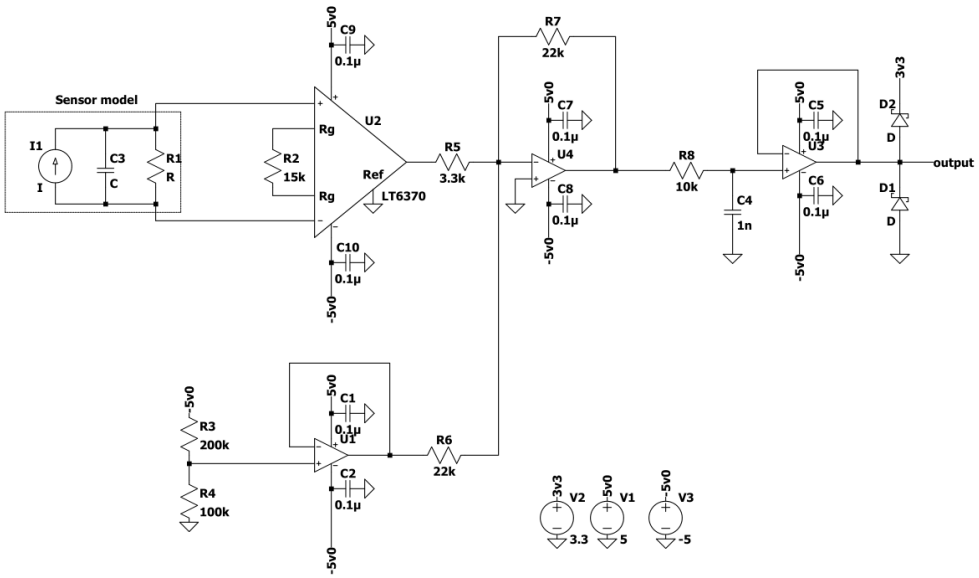


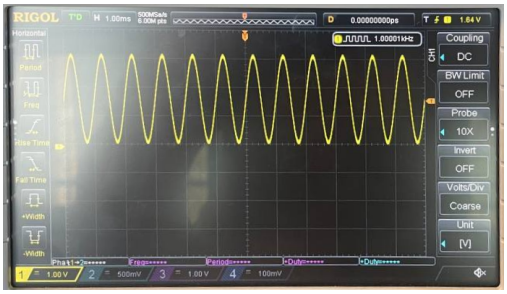
Figure A.9: Experimental circuit design 3

First, the INA was tested with the sensor only to determine if the amplification worked correctly. Results showed that the INA amplified the sensor's signal to the correct range A.10.

Leveraging this positive outcome, the chain excluding the LPF and the protection diodes was tested first against a signal generator-generated signal. When tested, the circuitry showed promising results, correctly amplifying and shifting the input signal to the desired range. Results can be seen in A.11, while the assembled circuit can be seen in A.12.

Figure A.10: the INA's output when connected the sensor is connected to its inputs

(a) 1 kHz input signal



(b) Circuit response at 1 kHz

Figure A.11: INA circuit response to 1 kHz sine wave

Consequently, the circuit was tried with the sensor as input, excluding the LPF and the protection diodes. Unfortunately, in this case, the behavior of the sensor torpedoed the circuit's success in correctly amplifying and shifting the signal. Results show a rectangular, almost peak-to-peak signal between the supply voltages A.13.

These results indicated a fundamental problem in how these circuits were designed and how the sensor's behavior was interpreted. This latest letdown demonstrated a need for further research and understanding of such sensors. Additional research was deemed fruitful, and the following design resulted in a working solution presented in chapter 4.

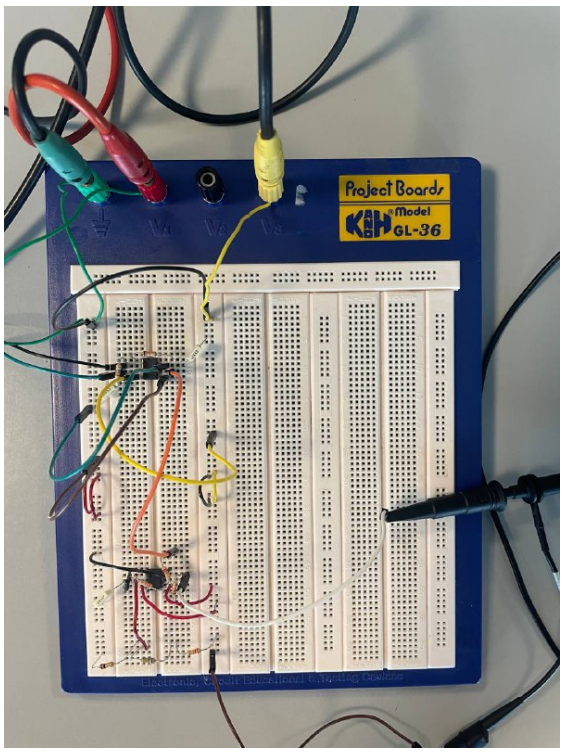


Figure A.12: Assembled circuit with INA and probe connected to its output

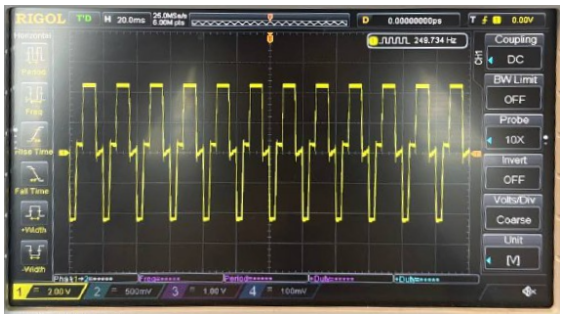


Figure A.13: Experimental circuit 3's (including INA) response to the signal as input

Appendix B

Python scripts

B.1 Python script for single ADC channel and polling

Listing B.1: First Python script to save data

```
import serial
import time
import matplotlib.pyplot as plt

CRACK_ARRAY_LEN = 1024
UART_PORT = 'COM3'
UART_BAUDRATE = 57600
TERMINATOR = b'\xFF\xFF'


def read_from_UART(port, baud_rate, length, delay=0.01):

    with serial.Serial(port, baud_rate) as ser:

        time.sleep(delay)
        data = ser.read(length*2)  # 2 bytes for uint16_t
        if len(data) != length*2:
            print('Error: Data length mismatch')

        values = []
        for i in range(0, len(data), 2):
            value = int.from_bytes(data[i:i+2], byteorder='little')
            values.append(value)

        return values


```

```
28 if __name__ == '__main__':
29
30     normal_array_len = read_from_UART(UART_PORT, UART_BAUDRATE, 1)[0]
31     print('Normal array length:', normal_array_len)
32
33     crack_prior = read_from_UART(UART_PORT, UART_BAUDRATE,
       normal_array_len)
34     crack = read_from_UART(UART_PORT, UART_BAUDRATE, CRACK_ARRAY_LEN)
35
36     whole_crack = crack_prior + crack
37     plt.plot(whole_crack)
38     print('Data length:', len(whole_crack))
```

B.2 Python script for single ADC channel with interrupts

Listing B.2: Improved Python script with threading and data visualization

```
1  import serial
2  import time
3  import threading
4  import struct
5  import matplotlib.pyplot as plt
6
7  # iterator + normal data + crack data + time at crack
8  NORMAL_LEN = 512
9  CRACK_LEN = 4096
10 ARRAY_LEN = 1 + NORMAL_LEN + CRACK_LEN + 1
11 UART_PORT = 'COM3'
12 UART_BAUDRATE = 57600
13
14
15 def input_thread(stop_event):
16     while not stop_event.is_set():
17         if input().lower() == 'q':
18             stop_event.set()
19             break
20
21
22 def read_fixed_length_data(ser):
23     expected_bytes = ARRAY_LEN * 2  # 2 bytes per uint16_t
24     data_bytes = ser.read(expected_bytes)
25
26     if len(data_bytes) != expected_bytes:
27         raise ValueError(f"Expected {expected_bytes} bytes, got {len(
       data_bytes)}")
```

```
28
29      return struct.unpack(f'<{ARRAY_LEN}H', data_bytes)
30
31
32  def process_data(visualize=False):
33      stop_event = threading.Event()
34      input_thread_handle = threading.Thread(target=input_thread,
35                                             args=(stop_event,))
36      input_thread_handle.start()
37
38      with serial.Serial(UART_PORT,
39                         UART_BAUDRATE,
40                         parity=serial.PARITY_NONE,
41                         stopbits=serial.STOPBITS_ONE,
42                         timeout=10) as ser:
43
44          print(f"Connected to {ser.name}")
45          print("Press 'q' to quit and Enter to quit the program")
46
47          while not stop_event.is_set():
48              try:
49                  print("Waiting for data...")
50                  data = read_fixed_length_data(ser)
51
52                  # index of the last element in the normal data
53                  last_index = data[0]
54
55                  normal_data = data[1:1+NORMAL_LEN]
56                  # rotate the crack data to the beginning
57                  normal_data = normal_data[last_index:] + normal_data
    [:last_index]
58                  crack_data = data[1+NORMAL_LEN:-1]
59                  time_at_crack = data[-1]
60
61                  whole_data = (time_at_crack,) + normal_data +
    crack_data
62
63                  date = time.strftime("%Y-%m-%d", time.localtime())
64                  with open(f"results/crack_data{date}.txt", "a") as f:
65                      f.write(" ".join(map(str, whole_data)) + "\n")
66
67                  print(f"Total data length: {len(whole_data)}")
68                  if visualize:
69                      plt.plot(normal_data + crack_data)
70                      plt.xlabel("Index")
71                      plt.ylabel("Value")
72                      plt.title("Crack data at time: " + str(
    time_at_crack))
73                      plt.show()
```

```
74
75              except ValueError as e:
76                  print(f"Error reading data: {e}")
77              except serial.SerialException as e:
78                  print(f"Serial port error: {e}")
79                  break
80              except struct.error as e:
81                  print(f"Data unpacking error: {e}")
82                  continue
83              except Exception as e:
84                  print(f"Unexpected error: {e}")
85                  continue
86
87      stop_event.set()
88      input_thread_handle.join()
89      print("Program terminated.")
90
91
92  def visualize_data(mfile):
93      with open(mfile, "r") as f:
94          data = f.readlines()
95
96      start_times = []
97      cracks = []
98      for line in data:
99          line = line.strip()
100         line = line.split(" ")
101         start_times.append(line[0])
102         cracks.append(line[1:])
103
104     cracks = [[int(x) for x in crack] for crack in cracks]
105
106     plt.figure(figsize=(10, 6))
107     for crack in cracks:
108         plt.plot(crack)
109         plt.xlabel("Index")
110         plt.ylabel("Value")
111         plt.title("Crack data")
112
113
114 if __name__ == '__main__':
115     process_data(visualize=True)
116
117     today = time.strftime("%Y-%m-%d", time.localtime())
118     datafile = f"results/crack_data{today}.txt"
119     visualize_data(datafile)
```

B.3 Python script for multiple ADC channels

Listing B.3: Final Python script with threading and interactive data visualization

```
1
2  import serial
3  import time
4  import threading
5  import struct
6  import matplotlib.pyplot as plt
7  from collections import defaultdict
8
9  NORMAL_LEN = 512
10 CRACK_LEN = 4096
11 # channel + index + normal_data + crack data + time + onset_index
12 ARRAY_LEN = 1 + 1 + NORMAL_LEN + CRACK_LEN + 1 + 1
13 UART_PORT = 'COM3'
14 UART_BAUDRATE = 115200
15 PLOT_WINDOW_SIZE = 5000  # Number of points to display per channel
16 LOWER_THRESHOLD = 1200
17 UPPER_THRESHOLD = 2500
18
19 color_mapping = {'0': '#1f77b4', '1': '#ff7f0e', '4': '#2ca02c', '8':
       '#d62728'}
20
21 # Initialize data storage
22 channel_data = defaultdict(lambda: {
23     'time_at_crack': [],
24     'values': [],
25     'onset_index': []
26 })
27
28 # Define the heights for horizontal lines
29 horizontal_lines = [LOWER_THRESHOLD, UPPER_THRESHOLD]
30
31
32 def input_thread(stop_event):
33     while not stop_event.is_set():
34         if input().lower() == 'q':
35             stop_event.set()
36             break
37
38
39 def read_fixed_length_data(ser):
40     """Reads a fixed length of data from the serial port
41
42     Args:
43         ser (): serial.Serial object
44
```

```
45      Raises:
46          ValueError: If the number of bytes read is not equal to the
    expected number of bytes
47
48      Returns:
49          Tuple: Tuple of uint16_t values, that are read from the
    serial port
50      """
51      expected_bytes = ARRAY_LEN * 2  # 2 bytes per uint16_t
52      data_bytes = ser.read(expected_bytes)
53
54      if len(data_bytes) != expected_bytes:
55          raise ValueError(f"Expected {expected_bytes} bytes, got {len(
    data_bytes)}")
56
57      return struct.unpack(f'<{ARRAY_LEN}H', data_bytes)
58
59
60  def process_data(visualize: bool = True, use_existing_data: bool =
    False, save_rx_data: bool = True) -> None:
61      """
62      dynamic data processing function,
63      fist opens the serial connection, then reads the data from the
    serial port,
64      optionally saves the raw data to a file for later simulation use,
65      and finally visualizes the data in real time and saves the
    processed data to a file
66
67      Args:
68          visualize (bool, optional): whether to visualize live.
    Defaults to True.
69          use_existing_data (bool, optional): whether to use existing
    data
70              WARNING: COM connection still needed. Defaults to False.
71          save_rx_data (bool, optional): whether to save received raw
    data
72              (used for debug and communication simulation when board
    is not available). Defaults to True.
73
74      Raises:
75          ValueError: If the data length exceeds the plot window size
76          SerialException: If there is an error with the serial port
77          struct.error: If there is an error unpacking the data
78          Exception: For any other unexpected errors
79      """
80      stop_event = threading.Event()
81      input_thread_handle = threading.Thread(target=input_thread, args
    =(stop_event,), daemon=True)
82      input_thread_handle.start()
```

```
83
84      with serial.Serial(UART_PORT,
85                         UART_BAUDRATE,
86                         parity=serial.PARITY_NONE,
87                         stopbits=serial.STOPBITS_ONE,
88                         timeout=10) as ser:
89          print(f"Connected to {ser.name}")
90          print("Press 'q' and Enter to quit the program")
91          if visualize:
92              plt.ion()  # Turn on interactive mode
93              fig, ax = plt.subplots(figsize=(12, 8))
94              # dicts to prevent multiple channels on the plot
95              lines = {}
96              scatter_plots = {}
97              # color coding of channels
98              colors = {0: 'red', 1: 'blue', 4: 'orange', 8: 'green'}
99              ax.set_xlabel("Index")
100             ax.set_ylabel("Value")
101             ax.set_title("Crack Data for All Channels")
102
103             # Plot horizontal threshold lines
104             for height in horizontal_lines:
105                 ax.axhline(y=height, color='gray', linestyle='--',
    linewidth=0.7)
106
107         date = time.strftime("%Y-%m-%d_%H-%M", time.localtime())
108         while not stop_event.is_set():
109             try:
110                 if use_existing_data:
111                     with open("raw_data/crack_data2025-03-04_19-24.
    txt", "r") as f:
112                         data = f.readlines()
113                         data = data[1].strip().split(" ")
114
115                     # Convert data to integers
116                     data = list(map(int, data))
117                 else:
118                     print("Waiting for data...")
119                     # attempt to read data
120                     data = read_fixed_length_data(ser)
121
122                     # join tuple into a string and write to file
123                     res = ' '.join(str(val) for val in data)
124                     with open(f"raw_data/crack_data{date}.txt", "a")
    as f:
125                         f.write(res + '\n')
126
127                 # the first element is channel
128                 channel = data[0]
```

```
129                 # second element is the latest value in the normal
    array
130                 last_index = data[1] - 2
131
132                 # reordering the data because circular buffer
    probably filled up
133                 # if not, then 0s at the beginning are better than bw
     normal and crack data
134                 normal_data = data[2:2+NORMAL_LEN]
135                 # rotate the crack data to the beginning
136                 normal_data = normal_data[last_index:] + normal_data
    [:last_index]
137                 # crack data is from the last index of normal data to
     the end - 2
138                 crack_data = data[2+NORMAL_LEN:-2]
139                 # time at crack is the last element
140                 time_at_crack = data[-1]
141                 # onset index is the last element, but we need to add
     the length of normal data
142                 onset_index = data[-2] + len(normal_data)
143
144                 # Combine normal and crack data
145                 combined_data = normal_data + crack_data
146
147                 # Store data
148                 channel_data[channel]['time_at_crack'] = [
    time_at_crack]
149                 channel_data[channel]['values'] = combined_data
150                 channel_data[channel]['onset_index'] = [onset_index]
151
152                 # Limit data to PLOT_WINDOW_SIZE
153                 if len(channel_data[channel]['values']) >
    PLOT_WINDOW_SIZE:
154                     raise ValueError("Data length exceeds plot window
     size")
155
156                 # Save to file
157                 if save_rx_data:
158                     with open(f"results/crack_data{date}.txt", "a")
    as f:
159                         whole_data = (channel, time_at_crack,
    onset_index) + tuple(combined_data)
160                         f.write(" ".join(map(str, whole_data)) + "\n"
    )
161
162                 print(f"Channel {channel}: Total data length: {len(
    channel_data[channel]['values'])}")
163
164                 if visualize:
```

```
                    if channel not in lines:
                        # Initialize line for new channel
                        lines[channel], = ax.plot(
                            channel_data[channel]['values'],
                            label=f"Channel {channel} at {time_at_crack}",
                            color=colors.get(channel, 'black')
                        )
                    else:
                        # Update existing line
                        lines[channel].set_ydata(channel_data[channel]['values'])
                        lines[channel].set_xdata(range(len(channel_data[channel]['values'])))
                        lines[channel].set_label(f"Channel {channel} at {time_at_crack}")

                        # Remove previous scatter plot if it exists
                        if channel in scatter_plots:
                            scatter_plots[channel].remove()

                    # Plot dot at onset_index, calculated by the board
                    scatter_plots[channel] = ax.scatter(onset_index,
                                                        channel_data[channel]['values'][onset_index],
                                                        color='#FF00FF',
                                                        zorder=5)

                    ax.relim()
                    ax.autoscale_view()
                    ax.legend(loc='upper right')
                    plt.draw()
                    plt.pause(0.01)

            except ValueError as e:
                print(f"Error reading data: {e}")
            except serial.SerialException as e:
                print(f"Serial port error: {e}")
                break
            except struct.error as e:
                print(f"Data unpacking error: {e}")
                continue
            except Exception as e:
                print(f"Unexpected error: {e}")
                continue

    stop_event.set()
```

```
207     input_thread_handle.join()
208 
209     if visualize:
210         plt.ioff()
211         plt.show()
212     print("Program terminated.")
213 
214 
215 def visualize_per_channel(mfile: str) -> None:
216     """visualization from processed file per channel (4 lines)
217 
218     also adds better calculated onset index to the plot, with upper
    and lower limits
219     Args:
220         mfile (str): string of file containing the processed data
221     """
222     with open(mfile, "r") as f:
223         data = f.readlines()
224 
225     samples = []
226     valid_channels = {'0', '1', '4', '8'}
227     for line in data:
228         line = line.strip()
229         line = line.split(" ")
230         if line[0] not in valid_channels:
231             continue
232         sample = {
233             'channel': line[0],
234             'start_time': line[1],
235             'onset_index': int(line[2]),
236             'crack': [int(x) for x in line[3:]],
237         }
238         samples.append(sample)
239 
240     for i, sample in enumerate(samples):
241         fig, ax = plt.subplots(figsize=(10, 6))
242         # Plot horizontal lines
243         for height in horizontal_lines:
244             ax.axhline(y=height, color='gray', linestyle='--',
    linewidth=0.7)
245 
246         ax.plot(sample['crack'],
247                 label=f"Channel {sample['channel']} at {sample['
    start_time']}",
248                 color=color_mapping[sample['channel']])
249 
250         # calculate and plot upper and lower limits, onset time
251         _, upper, lower, outlier_index = find_onset_time(samples[i]['
    crack'], 50, 10)
```

```
252             if outlier_index != -1:
253                 ax.axhline(y=upper, color='black', linestyle='—',
    linewidth=0.7)
254                 ax.axhline(y=lower, color='black', linestyle='—',
    linewidth=0.7)
255                 ax.scatter(outlier_index, samples[i]['crack'][
    outlier_index], color='black', zorder=5)
256 
257             # add onset index of the crack, calculated by the board, if
    it is outside the threshold
258             # (previously the onset index reset was not added to the c
    code,
259             # so it is not sure if it contains the latest onset index)
260             onset_value = sample['crack'][sample['onset_index']]
261             if onset_value > UPPER_THRESHOLD or onset_value <
    LOWER_THRESHOLD:
262                 ax.scatter(sample['onset_index'], onset_value, color='#
    FF00FF', zorder=5)
263 
264             ax.set_xlabel("Index")
265             ax.set_ylabel("Value")
266             ax.set_title("Crack data grouped by channels")
267             ax.legend()
268             plt.show()
269             plt.savefig(f"images/concrete_GOOD_run3/crack{(i // 4) + 1}
    _channel{sample['channel']}.png")
270             plt.close(fig)
271 
272 
273 def visualize_per_crack(mfile: str) -> None:
274     """visualizes data per crack (1 plot = 1 channel)
275 
276     also adds better calculated onset index to the plot
277     Args:
278         mfile (str): file containing the data
279     """
280     with open(mfile, "r") as f:
281         data = f.readlines()
282 
283     samples = []
284     valid_channels = {'0', '1', '4', '8'}
285     for line in data:
286         line = line.strip()
287         line = line.split(" ")
288         if line[0] not in valid_channels:
289             continue
290         sample = {
291             'channel': line[0],
292             'start_time': line[1],
```

```
293             'onset_index': int(line[2]),
294             'crack': [int(x) for x in line[3:]],
295         }
296         samples.append(sample)
297 
298     grouped_samples = []
299     current_group = []
300     current_channels = set()
301 
302     for sample in samples:
303         channel = sample['channel']
304         if channel not in current_channels and len(current_channels)
    < 4:
305             current_group.append(sample)
306             current_channels.add(channel)
307         else:
308             grouped_samples.append(current_group)
309             current_group = [sample]
310             current_channels = {channel}
311 
312     if current_group:
313         grouped_samples.append(current_group)
314 
315     for group in grouped_samples:
316         fig, ax = plt.subplots(figsize=(10, 6))
317         # Plot horizontal lines
318         for height in horizontal_lines:
319             ax.axhline(y=height, color='gray', linestyle='--',
    linewidth=0.7)
320 
321         for sample in group:
322             ax.plot(sample['crack'],
323                     label=f"Channel {sample['channel']} at {sample['
    start_time']}",
324                     color=color_mapping[sample['channel']])
325 
326             # add onset index of the crack, calculated by the board,
    if it is outside the threshold
327             # (previously the onset index reset was not added to the
    c code,
328             # so it is not sure if it contains the latest onset index
    )
329             onset_value = sample['crack'][sample['onset_index']]
330             if onset_value > UPPER_THRESHOLD or onset_value <
    LOWER_THRESHOLD:
331                 ax.scatter(sample['onset_index'], onset_value, color=
    '#FF00FF', zorder=5)
332 
```

```
333             _, _, _, onset_idx = find_onset_time(sample['crack'], 50,
    10)
334             if onset_idx != -1:
335                 ax.scatter(onset_idx, sample['crack'][onset_idx],
    color='black', zorder=5)
336 
337         ax.set_xlabel("Index")
338         ax.set_ylabel("Value")
339         ax.set_title("Crack data grouped by channels")
340         ax.legend()
341         plt.show()
342         plt.close(fig)
343 
344 
345 def find_onset_time(samples: list[str], initial_N: int, threshold:
    int) -> tuple:
346     """Finds the onset time of the crack
347 
348     Args:
349         samples (list[str]): data
350         initial_N (int): number of initial samples to average
351         threshold (int): threshold for outlier detection in pct %
352 
353     Raises:
354         ValueError: if initial_N is larger than the number of samples
     available
355 
356     Returns:
357         tuple: average, upper limit, lower limit, index of the
    outlier
358     """
359     if initial_N > len(samples):
360         raise ValueError("initial_N is larger than the number of
    samples available")
361 
362     cumulative_sum = sum(samples[:initial_N])
363     for index in range(initial_N, len(samples)):
364         average = cumulative_sum / index
365         upper_limit = average * (1 + threshold / 100)
366         lower_limit = average * (1 - threshold / 100)
367 
368         if samples[index] > upper_limit or samples[index] <
    lower_limit:
369             return average, upper_limit, lower_limit, index
370 
371         cumulative_sum += samples[index]
372 
373     return average, upper_limit, lower_limit, -1  # Return -1 if no
    outlier is found
```

```
374
375
376 if __name__ == '__main__':
377     process_data(visualize=True, use_existing_data=False,
    save_rx_data=True)
378
379     testfile = "results/concrete/crack_data2025-03-09_13-16_GOOD.txt"
380     visualize_per_crack(testfile)
381     visualize_per_channel(testfile)
```

Bibliography

[1] Samira Gholizadeh, Z Leman, BTHT Baharudin, et al. «A review of the application of acoustic emission technique in engineering». In: *Struct. Eng. Mech* 54.6 (2015), pp. 1075–1095 (cit. on p. 3).

[2] Daniel Balageas, Claus-Peter Fritzen, and Alfredo Güemes. *Structural Health Monitoring*. Google Books, 2006 (cit. on p. 3).

[3] Xiaoqing Huang, Pei Wang, Song Zhang, Xiongtao Zhao, and Yupeng Zhang. «Structural health monitoring and material safety with multispectral technique: A review». In: *Journal of Safety Science and Resilience* 3.1 (2022), pp. 48–60. ISSN: 2666-4496. DOI: 10.1016/j.jnlssr.2021.09.004. URL: https://www.sciencedirect.com/science/article/pii/S2666449621000499 (cit. on pp. 3, 4).

[4] D. Inaudi. «Structural health monitoring of bridges: General issues and applications». In: *Structural Health Monitoring of Civil Infrastructure Systems*. Ed. by Vistasp M. Karbhari and Farhad Ansari. Woodhead Publishing Series in Civil and Structural Engineering. Woodhead Publishing, 2009, pp. 339–370. ISBN: 978-1-84569-392-3. DOI: 10.1533/9781845696825.2.339. URL: https://www.sciencedirect.com/science/article/pii/B9781845693923500116 (cit. on p. 3).

[5] Charles R. Farrar and Keith Worden. «An introduction to structural health monitoring». In: *Phil. Trans. R. Soc. A.* (2007). ISSN: 1365-3032. DOI: 10.1098/rsta.2006.1928. URL: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2006.1928 (cit. on p. 3).

[6] Yong Xia, Yozo Fujino, Masato Abe, and Jun Murakoshi. «Short-term and long-term health monitoring experience of a short highway bridge: Case study». In: *Bridge Structures* 1.1 (2005), pp. 43–53. DOI: 10.1080/15732480412331294696. URL: https://doi.org/10.1080/15732480412331294696 (cit. on p. 3).

[7] Diogo Montalvão, N. Maia, and A. Ribeiro. «A Review of Vibration-based Structural Health Monitoring with Special Emphasis on Composite Materials». In: *The Shock and Vibration Digest* 38 (July 2006), p. 295. DOI: 10.1177/0583102406065898 (cit. on p. 4).

[8] A.C. Raghavan and Carlos Cesnik. «Review of Guided-Wave Structural Health Monitoring». In: *The Shock and Vibration Digest* 39 (Mar. 2007), pp. 91–114. DOI: 10.1177/0583102406075428 (cit. on p. 4).

[9] Keith Worden, Charles R. Farrar, Graeme Manson, and Gyuhae Park. «The fundamental axioms of structural health monitoring». In: *Proceedings of the Royal Society* (2007). ISSN: 1471-2946. DOI: 10.1098/rspa.2007.1834. URL: https://royalsocietypublishing.org/doi/10.1098/rspa.2007.1834 (cit. on p. 4).

[10] Md. Sazedul Islam, Md. Shahruzzaman, M. Nuruzzaman Khan, Md. Minhajul Islam, Sumaya Farhana Kabir, Abul K. Mallik, Mohammed Mizanur Rahman, and Papia Haque. «Composite materials: Concept, recent advancements, and applications». In: *Renewable Polymers and Polymer-Metal Oxide Composites*. Ed. by Sajjad Haider and Adnan Haider. Metal Oxides. Elsevier, 2022, pp. 1–43. ISBN: 978-0-323-85155-8. DOI: 10.1016/B978-0-323-85155-8.00011-X. URL: https://www.sciencedirect.com/science/article/pii/B978032385155800011X (cit. on p. 5).

[11] F.M. Shuaeib, K.Y. Benyounis, and M.S.J. Hashmi. «Material Behavior and Performance in Environments of Extreme Pressure and Temperatures». In: *Reference Module in Materials Science and Materials Engineering*. Elsevier, 2017. ISBN: 978-0-12-803581-8. DOI: 10.1016/B978-0-12-803581-8.04170-9. URL: https://www.sciencedirect.com/science/article/pii/B9780128035818041709 (cit. on p. 5).

[12] J. Schijve. «Fatigue of Structures and Materials in the 20th Century and the State of the Art». In: *Materials Science* 39.3 (May 2003), pp. 307–333. DOI: 10.1023/b:masc.0000010738.91907.a9. URL: https://www.gruppofrattura.it/ocs/index.php/esis/ECF14/paper/viewFile/7980/5243 (cit. on p. 5).

[13] Chetan J. Chitte. «Study on Causes and Prevention of Cracks in Building». In: *International Journal for Research in Applied Science and Engineering Technology* 6.3 (Mar. 2018), pp. 453–461. DOI: 10.22214/ijraset.2018.3073. URL: https://www.researchgate.net/profile/Chetan-Chitte/publication/325783838_Study_on_Causes_and_Prevention_of_Cracks_in_Building/links/6006b70e92851c13fe1f76b9/Study-on-Causes-and-Prevention-of-Cracks-in-Building.pdf (cit. on p. 5).

[14] Grishma Thagunna. «Building cracks - Causes and remedies». In: *International Journal of Advanced Structures and Geotechnical Engineering* 4.1 (2015), pp. 2319–5347 (cit. on p. 5).

[15] Hyo-Gyoung Kwak, Soo-Jun Ha, and Jin-Keun Kim. «Non-structural cracking in RC walls: Part I. Finite element formulation». In: *Cement and Concrete Research* 36.4 (2006), pp. 749–760. ISSN: 0008-8846. DOI: 10.1016/j.cemconres.2005.12.001. URL: https://www.sciencedirect.com/science/article/pii/S0008884605003091 (cit. on p. 5).

[16] TF Drouillard. «Acoustic Emission—A Bibliography for 1970–1972». In: *Monitoring Structural Integrity by Acoustic Emission* 571 (1975), p. 241 (cit. on p. 5).

[17] M.J.S. Lowe. «ULTRASONICS». In: *Encyclopedia of Vibration*. Ed. by S. Braun. Oxford: Elsevier, 2001, pp. 1437–1441. ISBN: 978-0-12-227085-7. DOI: 10.1006/rwvb.2001.0143. URL: https://www.sciencedirect.com/science/article/pii/B0122270851001430 (cit. on p. 5).

[18] John P. McCrory, Safaa Kh. Al-Jumaili, Davide Crivelli, Matthew R. Pearson, Mark J. Eaton, Carol A. Featherston, Mario Guagliano, Karen M. Holford, and Rhys Pullin. «Damage classification in carbon fibre composites using acoustic emission: A comparison of three techniques». In: *Composites Part B: Engineering* 68 (2015), pp. 424–430. ISSN: 1359-8368. DOI: 10.1016/j.compositesb.2014.08.046. URL: https://www.sciencedirect.com/science/article/pii/S1359836814003849 (cit. on p. 6).

[19] Jonathan J Scholey, Paul D Wilcox, Michael R Wisnom, Mike I Friswell, Martyn Pavier, and Mohammad R Aliha. «A GENERIC TECHNIQUE FOR ACOUSTIC EMISSION SOURCE LOCATION.» In: *Journal of Acoustic Emission* 27 (2009) (cit. on p. 6).

[20] Tawhidul Islam, Nagafuchi Sunichi, and Mehedi Hasan. «Structural Damage Localization by Linear Technique of Acoustic Emission». In: *Open Journal of Fluid Dynamics* (Jan. 2014), pp. 425–432. DOI: https://doi.org/10.4236/ojfd.2014.45032. URL: https://www.scirp.org/journal/paperinformation?paperid=52849&utm_source=chatgpt.com (cit. on p. 6).

[21] Jonathan Melchiorre, Amedeo Manuello Bertetto, Marco Martino Rosso, and Giuseppe Carlo Marano. «Acoustic Emission and Artificial Intelligence Procedure for Crack Source Localization». In: *Sensors* 23.2 (Jan. 2023), pp. 693–693. DOI: https://doi.org/10.3390/s23020693. URL: https://www.mdpi.com/1424-8220/23/2/693 (cit. on pp. 6, 7, 19, 21).

[22] Matthew R Jones, Tim J Rogers, Keith Worden, and Elizabeth J Cross. *A Bayesian methodology for localising acoustic emission sources in complex structures*. 2020. URL: https://arxiv.org/abs/2012.11058?utm_source=chatgpt.com (cit. on p. 7).

[23] The Britannica. *Piezoelectricity | Piezoelectricity, Acoustic Wave, Ultrasound*. Oct. 2024. URL: https://www.britannica.com/science/piezoelectricity (cit. on p. 7).

[24] Malin Edvardsson. 2018. URL: https://www.biolinscientific.com/blog/what-is-piezoelectricity (cit. on p. 8).

[25] Jie Jiang, Saloni Pendse, Lifu Zhang, and Jian Shi. «Strain related new sciences and devices in low-dimensional binary oxides». In: *Nano Energy* 104 (2022), p. 107917. ISSN: 2211-2855. DOI: 10.1016/j.nanoen.2022.107917. URL: https://www.sciencedirect.com/science/article/pii/S2211285522009958 (cit. on p. 8).

[26] Wayne Storr. *Inverting Operational Amplifier - The Inverting Op-amp*. Aug. 2013. URL: https://www.electronics-tutorials.ws/opamp/opamp_2.html (cit. on p. 11).

[27] Wayne Storr. *Non-inverting Operational Amplifier - The Non-inverting Op-amp*. Aug. 2013. URL: https://www.electronics-tutorials.ws/opamp/opamp_3.html (cit. on p. 11).

[28] Wayne Storr. *Summing Amplifier is an Op-amp Voltage Adder*. Aug. 2013. URL: https://www.electronics-tutorials.ws/opamp/opamp_4.html (cit. on p. 12).

[29] Wayne Storr. *Differential Amplifier - The Voltage Subtractor*. Aug. 2013. URL: https://www.electronics-tutorials.ws/opamp/opamp_5.html (cit. on p. 13).

[30] Contributors. *Instrumentation Amplifier - An Electronic Amplifier*. Sept. 2003. URL: https://en.wikipedia.org/wiki/Instrumentation_amplifier (cit. on p. 13).

[31] M. Hill, P.J. Mekdara, B.A. Trimmer, and R. D. White. «Structural Vibration for Robotic Communication and Sensing on One-Dimensional Structures». In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 2160–2165. DOI: 10.1109/IROS.2015.7353666 (cit. on p. 14).

[32] Wayne Storr. *Active Low Pass Filter - Op-amp Low Pass Filter*. Aug. 2013. URL: https://www.electronics-tutorials.ws/filter/filter_5.html (cit. on p. 15).

[33] Alan Turing. «Computing Machinery and Intelligence». In: *Mind* 59.236 (Oct. 1950), pp. 433–460. DOI: https://doi.org/10.1093/mind/lix.236.433. URL: https://courses.cs.umbc.edu/471/papers/turing.pdf (cit. on p. 15).

[34] Frank Rosenblatt. «The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain». In: *Psychological Review* (1958). URL: https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf (cit. on p. 15).

[35] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: https://doi.org/10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (cit. on p. 15).

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Communications of the ACM* 60.6 (May 2012), pp. 84–90 (cit. on pp. 15, 17).

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017. URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 15, 17).

[38] David Silver et al. «Mastering the game of Go with deep neural networks and tree search». In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: https://doi.org/10.1038/nature16961. URL: https://www.nature.com/articles/nature16961 (cit. on p. 15).

[39] Karl Pearson. «On lines and planes of closest fit to systems of points in space». In: *University College London* (1901). URL: https://pca.narod.ru/pearson1901.pdf (cit. on p. 16).

[40] Corinna Cortes and Vladimir Vapnik. «Support-vector networks». In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. DOI: https://doi.org/10.1007/BF00994018 (cit. on p. 16).

[41] Leo Breiman. «Random Forest». In: *Machine Learning* 45.1 (Jan. 2001), pp. 5–32. DOI: https://doi.org/10.1023/a:1010933404324. URL: https://link.springer.com/article/10.1023/A:1010933404324 (cit. on p. 16).

[42] Detlof von Winterfeldt and Ward Edwards. *Decision trees*. Cambridge University Press, 1986, pp. 63–89. ISBN: 0-521-27304-8 (cit. on p. 16).

[43] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. «Rethinking the Inception Architecture for Computer Vision». In: *Cvfoundation.org* (2016), pp. 2818–2826. URL: https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Szegedy_Rethinking_the_Inception_CVPR_2016_paper.html (cit. on p. 17).

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. URL: https://arxiv.org/abs/1512.03385 (cit. on p. 17).

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. URL: https://arxiv.org/abs/1505.04597 (cit. on p. 17).

[46] Alberto Carpinteri, Giuseppe Lacidogna, and Gianni Niccolini. «Critical Behaviour in Concrete Structures and Damage Localization by Acoustic Emission». In: *Key Engineering Materials* 312 (June 2006), pp. 305–310. DOI: 10.4028/www.scientific.net/kem.312.305 (cit. on p. 18).

[47] «The equivalence of generalized least squares and maximum likelihood estimates in the exponential family». In: *Journal of the American Statistical Association* (1974). DOI: https://doi.org/10.1080//01621459.1976.10481508. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1976.10481508 (cit. on p. 18).

[48] Alberto Carpinteri, Giuseppe Lacidogna, and Amedeo Manuello. «Damage Mechanisms Interpreted by Acoustic Emission Signal Analysis». In: *Key Engineering Materials* 347 (Sept. 2007), pp. 577–582. DOI: 10.4028/www.scientific.net/kem.347.577 (cit. on p. 18).

[49] Masayasu Ohtsu. «Moment Tensor Analysis». In: *Springer eBooks* (July 2008), pp. 175–200. DOI: 10.1007/978-3-540-69972-9_8. URL: https://link.springer.com/chapter/10.1007/978-3-540-69972-9_8 (cit. on p. 18).

[50] F. Bai, D. Gagar, P. Foote, and Y. Zhao. «Comparison of Alternatives to Amplitude Thresholding for Onset Detection of Acoustic Emission Signals». In: *Mechanical Systems and Signal Processing* 84 (Sept. 2016), pp. 717–730. DOI: 10.1016/j.ymssp.2016.09.004. URL: https://www.sciencedirect.com/science/article/pii/S0888327016303430 (cit. on pp. 18, 19).

[51] Jochen H. Kurz, Christian U. Grosse, and Hans-Wolf Reinhardt. «Strategies for Reliable Automatic Onset Time Picking of Acoustic Emissions and Ultrasound Signals in Concrete». In: *Ultrasonics* 43 (Dec. 2004), pp. 538–546. DOI: 10.1016/j.ultras.2004.12.005. URL: https://www.sciencedirect.com/science/article/abs/pii/S0041624X04003166 (cit. on pp. 18, 19).

[52] M. Eaton, Rhys Pullin, and Karen Holford. «Towards Improved Damage Location Using Acoustic Emission». In: *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 226 (Sept. 2012), pp. 2141–2153. DOI: 10.1177/0954406212449582 (cit. on p. 19).

[53] Andrea Rocchi, Eleonora Santecchia, Fabrizio Ciciulla, Paolo Mengucci, and Gianni Barucca. «Characterization and Optimization of Level Measurement by an Ultrasonic Sensor System». In: *IEEE Sensors Journal* 19.8 (2019), pp. 3077–3084. DOI: 10.1109/JSEN.2018.2890568 (cit. on p. 19).

[54] Manfred Baer and Urs Kradolfer. «An Automatic Phase Picker for Local and Teleseismic Events». In: *Bulletin of the Seismological Society of America* 77 (1987), pp. 1437–1445. URL: https://api.semanticscholar.org/CorpusID:130279060 (cit. on p. 19).

[55] John Sadowsky. «Investigation of Signal Characteristics Using the Continuous Wavelet Transform». In: *JOHNS HOPKINS APL TECHNICAL DIGEST* 258.3 (1996). URL: https://secwww.jhuapl.edu/techdigest/Content/techdigest/pdf/V17-N03/17-03-Sadowsky.pdf (cit. on p. 19).

[56] Hirotugu Akaike. «Markovian representation of stochastic processes and its application to the analysis of autoregressive moving average processes». In: *Annals of the Institute of Statistical Mathematics* 26.1 (Dec. 1974), pp. 363–387. DOI: https://doi.org/10.1007/bf02479833. URL: https://www.ism.ac.jp/editsec/aism/pdf/026_3_0363.pdf (cit. on p. 19).

[57] Zilong Zhou, Ruishan Cheng, Yichao Rui, Jing Zhou, and Haiquan Wang. «An Improved Automatic Picking Method for Arrival Time of Acoustic Emission Signals». In: *IEEE Access* 7 (2019), pp. 75568–75576. DOI: 10.1109/ACCESS.2019.2921650 (cit. on p. 19).

[58] A. Carpinteri, J Xu, G. Lacidogna, and A. Manuello. «Reliable onset time determination and source location of acoustic emissions in concrete structures». In: *Cement and Concrete Composites* 34.4 (Dec. 2011), pp. 529–537. DOI: https://doi.org/10.1016/j.cemconcomp.2011.11.013. URL: https://www.sciencedirect.com/science/article/abs/pii/S0958946511002149 (cit. on p. 19).

[59] D V Hinkley. «Inference about the Change-Point from Cumulative Sum Tests». In: *Biometrika* 58.3 (Dec. 1971), pp. 509–509. DOI: https://doi.org/10.2307/2334386. URL: https://www.jstor.org/stable/2334386 (cit. on p. 19).

[60] Steven M Ziola and Michael R Gorman. «Source location in thin plates using cross-correlation». In: *The Journal of the Acoustical Society of America* 90.5 (Nov. 1991), pp. 2551–2556. DOI: https://doi.org/10.1121/1.402348. URL: https://pubs.aip.org/asa/jasa/article-abstract/90/5/2551/807982/Source-location-in-thin-plates-using-cross?redirectedFrom=fulltext (cit. on p. 19).

[61] F Ciampa and M Meo. «Acoustic emission source localization and velocity determination of the fundamental mode A0 using wavelet analysis and a Newton-based optimization technique». In: *Smart Materials and Structures* 19.4 (Mar. 2010), p. 045027. DOI: https://doi.org/10.1088/0964-1726/19/4/045027 (cit. on p. 19).

[62] «A fractal-based algorithm for detecting first arrivals». In: *ResearchGate* (1996). DOI: https://doi.org/10.1190//1.1444030. URL: https://www.researchgate.net/publication/215754932_A_fractal-based_algorithm_for_detecting_first_arrivals (cit. on p. 20).

[63] Vahid Emamian, Mostafa Kaveh, Ahmed H Tewfik, Zhiqiang Shi, Laurence J Jacobs, and Jacek Jarzynski. «Robust Clustering of Acoustic Emission Signals Using Neural Networks and Signal Subspace Projections». In: *EURASIP Journal on Advances in Signal Processing* 2003.3 (Mar. 2003). DOI: https://doi.org/10.1155/s1110865703210027. URL: https://asp-eurasipjournals.springeropen.com/articles/10.1155/S1110865703210027 (cit. on p. 20).

[64] H So and Hing Cheung. *Time Delay Estimation: Applications and Algorithms*. URL: https://sigport.org/sites/default/files/Time_Delay_Estimation.pdf (cit. on p. 20).

[65] Omkar; Sundaram; Raghavendra; Mani. «Acoustic emission signal classification using fuzzy C-means clustering». In: *ResearchGate* (2025). DOI: https://doi.org/10.1109//ICONIP.2002.1198989. URL: https://www.researchgate.net/publication/4014441_Acoustic_emission_signal_classification_using_fuzzy_C-means_clustering (cit. on p. 20).

[66] Mostafa Kaveh Nuri F Ince C.-S. Kao. «A machine learning approach for locating acoustic emission». In: *ResearchGate* (2010). DOI: https://doi.org/10.1155//2010//895486. URL: https://www.researchgate.net/publication/50282709_A_Machine_Learning_Approach_for_Locating_Acoustic_Emission (cit. on p. 21).

[67] Mengxi Zhang, Mingchao Li, Jinrui Zhang, Le Liu, and Heng Li. «Onset detection of ultrasonic signals for the testing of concrete foundation piles by coupled continuous wavelet transform and machine learning algorithms». In: *Advanced Engineering Informatics* 43 (Jan. 2020), pp. 101034–101034. DOI: https://doi.org/10.1016/j.aei.2020.101034. URL: https://www.sciencedirect.com/science/article/abs/pii/S1474034620300033 (cit. on p. 21).

[68] Ping-Hung Chen; Jian-Jiun Ding; Jin-Yu Huang; Tzu-Yun Tseng. «Accurate onset detection algorithm using feature-layer-based deep learning architecture». In: *ResearchGate* (2020). DOI: https://doi.org/10.1109//ISCAS45731.2020.9181255. URL: https://www.researchgate.net/publication/349293261_Accurate_Onset_Detection_Algorithm_using_Feature-Layer-Based_Deep_Learning_Architecture (cit. on p. 21).

[69] Federica Zonzini, Denis Bogomolov, Tanush Dhamija, Nicola Testoni, Luca De Marchi, and Alessandro Marzani. «Deep Learning Approaches for Robust Time of Arrival Estimation in Acoustic Emission Monitoring». In: *Sensors* 22.3 (Jan. 2022), p. 1091. DOI: https://doi.org/10.3390/s22031091. URL: https://www.mdpi.com/1424-8220/22/3/1091 (cit. on p. 21).

[70] Omar M. Saad and Yangkang Chen. «CapsPhase: Capsule Neural Network for Seismic Phase Classification and Picking». In: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2022), pp. 1–11. DOI: https://doi.org/10.1109/tgrs.2021.3089929. URL: https://ieeexplore.ieee.org/abstract/document/9467532 (cit. on p. 21).

[71] Annamaria Mesaros, Toni Heittola, Tuomas Virtanen, and Mark D. Plumbley. «Sound Event Detection: A tutorial». In: *IEEE Signal Processing Magazine* 38.5 (Sept. 2021), pp. 67–83. DOI: https://doi.org/10.1109/msp.2021.3090678. URL: https://arxiv.org/abs/2107.05463 (cit. on p. 21).

[72] Jonathan Melchiorre, Leo D'Amato, Federico Agostini, and Antonino Maria Rizzo. «Acoustic emission onset time detection for structural monitoring with U-Net neural network architecture». In: *Developments in the Built Environment* 18 (Apr. 2024), pp. 100449–100449. DOI: https://doi.org/10.1016/j.dibe.2024.100449. URL: https://www.sciencedirect.com/science/article/pii/S2666165924001303 (cit. on p. 21).

[73] Chenglong Yu, Jianchao Du, Meng Li, Yunsong Li, and Weibin Li. «An improved U-Net model for concrete crack detection». In: *Machine Learning with Applications* 10 (Nov. 2022), pp. 100436–100436. DOI: https://doi.org/10.1016/j.mlwa.2022.100436. URL: https://www.sciencedirect.com/science/article/pii/S2666827022001116 (cit. on p. 21).

[74] Jing Zheng, Jiren Lu, Suping Peng, and Tianqi Jiang. «An automatic microseismic or acoustic emission arrival identification scheme with deep recurrent neural networks». In: *Geophysical Journal International* 212.2 (Nov. 2017), pp. 1389–1397. DOI: https://doi.org/10.1093/gji/ggx487. URL: https://academic.oup.com/gji/article/212/2/1389/4604781 (cit. on p. 22).

[75] Tuan-Khai Nguyen, Zahoor Ahmad, and Jong-Myon Kim. «A Scheme with Acoustic Emission Hit Removal for the Remaining Useful Life Prediction of Concrete Structures». In: *Sensors* 21.22 (Nov. 2021), pp. 7761–7761. DOI: https://doi.org/10.3390/s21227761. URL: https://www.mdpi.com/1424-8220/21/22/7761 (cit. on p. 22).

[76] Pengcheng Jiao, King-James I. Egbe, Yiwei Xie, Ali Matin Nazar, and Amir H. Alavi. «Piezoelectric Sensing Techniques in Structural Health Monitoring: A State-of-the-Art Review». In: *Sensors* 20.13 (2020). ISSN: 1424-8220. DOI: 10.3390/s20133730. URL: https://www.mdpi.com/1424-8220/20/13/3730 (cit. on p. 23).

[77] STMicroelectronics. 2024. URL: https://www.st.com/en/evaluation-tools/nucleo-f446re.html (cit. on pp. 26, 36, 67).

[78] James Karki. *Signal Conditioning Piezoelectric Sensors*. Tech. rep. SLOA033A. Texas Instruments, Sept. 2000 (cit. on p. 27).

[79] Texas Instruments. *Analog Engineer's Circuit: Charge Amplifier Circuit*. URL: https://www.ti.com/lit/an/sboa287/sboa287.pdf?ts=1718012208393&ref_url=https%253A%252F%252Fwww.google.com%252F (cit. on p. 27).

[80] Robert Keim. *Understanding and Implementing Charge Amplifiers for Piezoelectric Sensor Systems*. URL: https://www.allaboutcircuits.com/technical-articles/understanding-and-implementing-charge-amplifiers-for-piezoelectric-sensor-s/ (cit. on p. 27).

[81] Robert Keim. *How to Design Charge Amplifiers for Piezoelectric Sensors*. URL: https://www.allaboutcircuits.com/technical-articles/how-to-design-charge-amplifiers-piezoelectric-sensors/ (cit. on p. 27).

[82] Texas Instruments. *TL08xx FET-Input Operational Amplifiers*. URL: https://www.ti.com/lit/ds/symlink/tl082.pdf (cit. on pp. 29, 77).

[83] Texas Instruments. *TLC271, TLC271A, TLC271B LinCMOS Programmable Low-Power Operational Amplifiers*. URL: https://www.ti.com/lit/ds/symlink/tlc271.pdf (cit. on pp. 34, 71).

[84] ST Microelectronics. *STM32 CUBE IDE*. URL: https://www.st.com/en/development-tools/stm32cubeide.html (cit. on p. 37).

[85] Balint Bujtor et al. *STM32 Nucleo Continuous ADC mode only converts once*. URL: https://community.st.com/t5/stm32-mcus-products/stm32-nucleo-continuous-adc-mode-only-converts-once/td-p/721761l (cit. on p. 44).

[86] Dongxue Li et al. «Acoustic Emission Wave Velocity Attenuation of Concrete and Its Application in Crack Localization». In: *Traffic Infrastructure Sustainability in Autonomous Driving and Smart Pavement Environments* (2020). URL: %E2%80%8Chttps://www.mdpi.com/2071-1050/12/18/7405 (cit. on p. 64).

[87] Mouser. *STMicroelectronics 1N5819*. 2025. URL: https://eu.mouser.com/datasheet/2/389/1n5817-1848842.pdf (cit. on p. 71).

[88] Texas Instruments. *Non-Inverting Op Amp with Non-Inverting Positive Reference Voltage Circuit*. 2024. URL: https://www.ti.com/lit/an/sboa263a/sboa263a.pdf (cit. on p. 72).

[89] Texas Instruments. *INAx126 MicroPower Instrumentation Amplifiers*. URL: https://www.ti.com/lit/ds/symlink/ina126.pdf (cit. on p. 76).