UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Programme of MSc Studies

MSc in Science and Technology of Electrical and Computer Engineering

On the Design Space Exploration of Vibration Analysis Systems

Master's Thesis

Vasileios Dimitriou

Supervisor: Karakonstantis Georgios

September 2025

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Programme of MSc Studies

MSc in Science and Technology of Electrical and Computer Engineering

On the Design Space Exploration of Vibration Analysis Systems

Master's Thesis

Vasileios Dimitriou

Supervisor: Karakonstantis Georgios

September 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Πρόγραμμα Μεταπτυχιακών Σπουδών

Επιστήμη και Τεχνολογία Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διερεύνηση Σχεδίασης Συστημάτων Ανάλυσης Δονήσεων Μηχανών

Μεταπτυχιακή Διπλωματική Εργασία

Βασίλειος Δημητρίου

Επιβλέπων/πουσα: Καρακωνσταντής Γεώργιος

Σεπτέμβριος 2025

Approved by the Examination Committee:

Supervisor **Karakonstantis Georgios**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Stamoulis Georgios**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Nestoras Evmorfopoulos**
Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Vasileios Dimitriou

Master's Thesis

On the Design Space Exploration of Vibration Analysis Systems

Vasileios Dimitriou

Abstract

This thesis addresses the design of a predictive maintenance apparatus, contributing to the data acquisition and analysis. The scientific area of the thesis encompasses mechanical fault diagnosis, vibration analysis and Internet of Thing (IoT) technologies.

The main challenge is the development of a low-cost and best value for money set-up that has the capacity to analyze the acceleration data created by some machine. Its ultimate objective is to predict misalignments, bearing failures and other defects, providing the maintenance personnel readings, metrics and historical data that can by analyzed and conclude what action needs to be taken.

The methodology is centered on a model-based approach, utilizing a triple-axis analog accelerometer (ADXL335) for data acquisition. Vibration signals are captured in the time-domain and processed using the Fast Fourier Transform (FFT) algorithm to generate frequency spectra. The frequency spectra illustrate a healthy state of the machine and a theshold line at 1g indicating that there is some concerning defect.

The main steps involved the design and integration of a hardware setup featuring an Arduino microcontroller for sensor data collection and a Raspberry Pi microcomputer for data processing and analysis. Communication between these components was established via a UART protocol. The developed system successfully demonstrated the ability to collect high-fidelity vibration data, perform spectral analysis, and illustrate frequency data.

The main results confirm the viability of using low-cost, off-the-shelf components to create an effective predictive maintenance tool. The conclusions highlight the critical role of FFT-based spectral analysis in mechanical diagnostics and the potential of IoT technologies to reinforce the preventive maintenance strategies for a wider range of industrial applications.

Keywords: Predictive Maintenance, Accelerometer, Vibration Analysis, Fast Fourier Transform (FFT), Internet of Things (IoT)

Μεταπτυχιακή Διπλωματική Εργασία

Διερεύνηση Σχεδίασης Συστημάτων Ανάλυσης Δονήσεων Μηχανών

Βασίλειος Δημητρίου

Abstract

Η παρούσα διατριβή πραγματεύεται τον σχεδιασμό και την ανάπτυξη μιας συσκευής προγνωστικής συντήρησης, με στόχο την απόκτηση και ανάλυση δεδομένων.

Η βασική πρόκληση που επιχειρεί να αντιμετωπίσει η εργασία είναι η δημιουργία ενός οικονομικού αλλά ταυτόχρονα αξιόπιστου συστήματος, το οποίο θα μπορεί να επεξεργάζεται δεδομένα επιτάχυνσης που προέρχονται από βιομηχανικές μηχανές. Ο τελικός στόχος είναι η πρόβλεψη πιθανών αστοχιών, όπως κακή ευθυγράμμιση, φθορά ρουλεμάν ή άλλα μηχανικά ελαττώματα, παρέχοντας στο προσωπικό συντήρησης χρήσιμες μετρήσεις, δείκτες και ιστορικά δεδομένα.

Η μεθοδολογία βασίστηκε σε μια προσέγγιση μοντελοποίησης, με τη χρήση ενός τριαξονικού αναλογικού επιταχυνσιομέτρου (ADXL335) για την απόκτηση δεδομένων. Τα σήματα δόνησης καταγράφηκαν στο πεδίο του χρόνου και αναλύθηκαν με τον Γρήγορο Μετασχηματισμό Fourier (FFT), ώστε να εξαχθούν φάσματα συχνοτήτων. Τα φάσματα αυτά χρησιμοποιήθηκαν για τη σύγκριση της «υγιούς» λειτουργικής κατάστασης μιας μηχανής με πιθανές καταστάσεις βλάβης, με την ύπαρξη ορίου στο 1g να σηματοδοτεί ενδεχόμενο πρόβλημα.

Η υλοποίηση περιλάμβανε τον σχεδιασμό και την ολοκλήρωση ενός hardware setup με μικροελεγκτή Arduino για τη συλλογή δεδομένων από τον αισθητήρα και Raspberry Pi για την ανάλυση και επεξεργασία τους.

Τα αποτελέσματα της μελέτης δείχνουν ότι είναι εφικτή η χρήση εξαρτημάτων χαμηλού κόστους και ευρείας διαθεσιμότητας για την ανάπτυξη ενός λειτουργικού εργαλείου προγνωστικής συντήρησης. Τα συμπεράσματα αναδεικνύουν τον καθοριστικό ρόλο της φασματικής ανάλυσης με FFT στη διάγνωση μηχανικών βλαβών, ενώ ταυτόχρονα υπογραμμίζουν τις δυνατότητες που προσφέρουν οι τεχνολογίες IoT στη βελτίωση στρατηγικών προληπτικής και προγνωστικής συντήρησης σε πλήθος βιομηχανικών εφαρμογών.

Λέξεις-κλειδιά:

Predictive Maintenance, Accelerometer, Vibration Analysis, Fast Fourier Transform (FFT), Internet of Things (IoT)

Contents

List of Figures

Abbreviations

MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
FFT	Fast Fourier Transform
IoT	Internet of Things
ML	Machine Learning
RFID	Radio Frequency Identification
IP	Internet Protocol

Chapter 1

Introduction

Most of the industrial plants nowadays face severe harness related to unplanned production downtimes. For this reason most of the industrial manager crave the development and usage of smart devices that are able to predict these machinery breakdowns. This fact would also require their taking action so as to manage some upcoming machine failure.

It is worth mentioning at this point that the equipment failure can be handled by taking some action in the majority of the cases. This ensues that the maintenance teams is always responsible and can most of the time alleviate potential production issues. For this purpose, engineers need data readings, such that they can analyze and conclude what the actual situation is.

The data reading aim at the collection of data related mostly on vibration. The most common cause of the machine failures is this of bearings failure or missalignment. In most of the cases, the missalignment case stresses the whole mechanism (shaft, coupling and bearings) and this could lead to the complete destruction of all the machine components.

Another case that needs to be studied is this of the bearing failure. Bearings have a specific operational number of hours of life. Especially the bearings that require periodical greasing are susceptible to failure because of lack of lubrication. The bearing failure can be detected by measuring using some sensing devices. This could predict the failure of a bearing even in the preliminary stages of the fault.

There is plethora of different devices that may contribute to this concept. This wide variety of sensors and measuring devices yields the opportunity of receiving and analyzing different machines making the most of distinct approaches.

In this chapter the readers will get some intuition into the object of the diploma thesis,

along with its contribution and structure.

Last but not least, the main concept in this technical report is this of the Predictive Maintenance. Wu et al articulate that advancements of Internet of Things(IoT) technology has made predictive maintenance prominent nowadays [11]. The authors explain that it is a condition-based maintenance practice, which in turn requires proactive maintenance.

In the aforementioned article, it is clearly highlighted that the authors focus on the computational resources and the data collection. Wu et al pursue the development of empirical machine learning models so as to predict performance drops and potential issues [11]. Machine learning is the most common practice so as to create a model-based prediction, but is not the only deployed implementation.

The most common reference feature when it comes to failure prediction is the machine body vibration. Chen et al focus on the vibration-based analysis of some machinery, utilizing mainly machine learning (ML) algorithms [3]. But there is also the approach of the FFT (Fast Fourier Transform) for the feature extraction of the Vibration Signals. This requires the usage of the FFT algorithm (see Chapter 2).

1.1 Main Objectives of the Diploma Thesis

Elaboration on a system that predicts the failure of mechanical components based on historical data. This thesis focuses on resolving the issue of unplanned shutdowns due to corrective maintenance of machines that have bearings.

The problem that is being investigated is the creation of a set-up that collects and sends some reading to a database. The main focus of the project will be the usage of low-cost components and free-of-charge services.

Moreover, the operational efficiency of the whole set-up is to be investigated. The ultimate objective is the development of a product that is best value for money, aiming at the development of a low-cost device that can efficiently gather reading from machinery, and this must be combined with the the development of an algorithm that is able to analyze the readings and predict any potential upcoming issues.

The main challenges here lie into the fact that the sensor has some limitations and requirement of case-specific data. This derives from the fact that every machine has its own vibration pattern, and this could be attributed to an abundance of factors (e.g. the foundation,

the bearings clearance, the route of the issue itself etc), for this reason the most reliable - ideal practice would be the collection of readings and trend tracking, under the condition that the machine data starts being collected from the first run.

In addition to the challenges that have already been mentioned, there is always of the relevant data collection in the case of the development of machine learning models. Datasets that resemble the actual machine conditions are required in this case scenario.

The sensing component of the set-up must be attached on the studied machinery. The reason behind this is the efficiency and the accuracy of the readings. The sensor must be fixed on the machinery frame so as to get vibration readings of it. In addition to this, there is the need of a microcomputer and a microcontroller that could gather and analyze the reading.

The most significant feature of the product of this diploma thesis is the fact that the devices within a safe network convey the data to some online database, which in turn are presented online and remote. To put accurately, the reading can be analyzed and investigated remotely, at any time and place, on personal devices irrelevant to the industrial equipment.

1.1.1 The Concept of Predictive Maintenance

Getting a good grasp on the upcoming required maintenance activities and being aware of what is of utmost importance, is vital for the operation of a plant. Periodical status checks of different machines is crucial for the maintenance engineers, since this could yield a clear understanding of what priority is at that instance of time. Therefore the prediction of an upcoming machine failure would constitute an invaluable tool.

The concept of preventive maintenance lies into the fact that the maintenance personnel needs to follow some periodical activities related to lubrication and tightness checks among others. In addition to this straight forward practice, they need to keep track of some data readings so as to figure out what the status of their machines are especially if they notice some abnormal status.

So within the concept of the preventive maintenance there is a subfield named predictive maintenance that focuses mostly on the readings and data. For this purpose the best practice is taking periodical readings and keep track of the health status of the machine. Nodem et al claim that preventive maintenance is usually planned and aimed at increasing the reliability of a deteriorating system or decreasing the operating costs of repairable systems [12].

1.1.2 Contribution

In the context of this diploma thesis:

1. a large number of try and error algorithms took place in practice
2. developed a set-up of algorithms so as to convey the final result considering the datasheet information
3. assessed the performance of the algorithms and stresses the system with data
4. the algorithms ran in microcomputers and microcontrollers
5. the algorithms illustrated the after-analysis data
6. thresholds and good status reference values were included so as to have it easier for the reader to come to some conclusion

1.2 Structure of the Thesis

This thesis is structured to systematically explore the development and implementation of a predictive maintenance system using vibration analysis. The organization of the chapters is designed to guide the reader from foundational concepts to the final implementation and validation of the proposed system.

Chapter 2 provides the essential technical background for this research. It introduces the core concept of predictive maintenance through vibration analysis and details the critical bearing fault frequencies—BPFO, BPFI, BSF, and FTF—that are fundamental to diagnosing mechanical failures. The chapter thoroughly explains the signal processing methodology, with a focus on the Fast Fourier Transform (FFT) algorithm, which is pivotal for converting time-domain vibration data into interpretable frequency spectra. Finally, it outlines the Internet of Things (IoT) architecture and communication protocols, such as UART, that form the basis of the real-time data acquisition and monitoring system developed in this work.

Chapter 3 discusses the system architecture and hardware design. It details the selection of the sensing unit, specifically the ADXL335 accelerometer [13], and the communication protocols used for data acquisition. The chapter also covers the integration of the Arduino microcontroller and Raspberry Pi microcomputer, explaining their roles in the data collection and transmission pipeline [14, 7].

In Chapter 4, we develop the core signal processing methodology. The focus is on the implementation of the Fast Fourier Transform (FFT) algorithm [15, 16] for converting time-domain vibration signals into the frequency domain.

Chapter 5 presents the experimental results and performance evaluation. It describes the data collection and processes the performance of the microcontroller and microcomputer under some memory stress. The results of the fault diagnosis are analyzed to assess the system's accuracy. In addition to this, readings are illustrated.

Finally, Chapter 6 concludes the thesis by summarizing the key findings and contributions. It discusses the implications of the results. In addition to this it discusses some new concepts that could be implemented so as to improve the performance of the project.

Chapter 2

Background

2.1 Introduction

2.1.1 The Scope of Predictive Maintenance

There is a tremendous abundance of applications where the vibration analysis is a pre-requisite. Readings and recordings are of vital importance in the context of the predictive maintenance implementation. Most of the industries nowadays strive to make use of accurate preventive maintenance plans, which requires a well-rounded continuous monitoring system, that keeps data of the whole equipment. The ultimate goal of the preventive maintenance plan is the elimination of the unwanted shutdowns due to a machine failure and the increase of plant reliability [12].

Most of the machines need bearings, since bearings are mechanical components designed to support rotating shafts and minimize friction. The bearings failure may have a wide variety of causes, lubrication shortage, seal failure, high temperatures, misalignment are some of the causes among others. Therefore vibration analysis focuses on predicting all theses possible causes of bearings failure. Some of the preventive actions against the bearings failure are described bellow [17].

Proper lubrication plays a vital role in bearing longevity. Rather than choosing oils based solely on moderate operating temperatures, engineers must consider factors like bearing speed, size, and load. Maintaining an adequate oil film is critical, and minimum viscosities are established based on bearing type. This oil film ensure that no metal-to-metal contact takes place for a predetermined load and speed. If these thresholds aren't met, metal-to-metal

contact occurs, significantly reducing reliability [1].

Alignment is another critical cause of bearings failure. A misalignment of some equipment -typically consisted of a motor and the main equipment- leads to bearings failure. For this purpose vibration and temperature monitoring are significant. For this reason, alignment practice upon machine installation is mandatory. According to Bloch the proper initial shaft alignment is necessary in any case and it is proven that the most advantageous technique to do so is the laser optic one [1].

A detailed alignment procedure is also presented by Bloch. On Figure 2.1 is depicted a typical aligning tool set-up. The laser headers must be installed on machined surfaces, which entails that it will be settled on the motor shaft and the shaft of the rotated mechanism. This laser mechanism has an emitter system, that produces a laser beam, and the laser beams are received by some receivers. Usually a three reference point alignment is what the optic laser machines offer, and this procedure takes place whle the motor and the rotated machine are coupled with each other.

The laser mechanism set-up (laser emitters and receivers) are connected to some alignment computer, which is powered by some special software. This yields some reading and some alignment correction recommendations. The alignment criteria depend on the mechanical design of the equipment and the rotating speed as well [1].

However there could be some more causes of machinery failure, specifically a loose machinery base or tightening bolt is another that could lead to a components failure. Hence, the way to alleviate this issue is to align and tighten the machinery back to the frame.

Vibration diagnosis contributes to operational reliability and cost efficiency in engineering applications immensely. The implementation of vibration-analysis-based predictive maintenance helps in the direction of troubleshooting the cause of some issue. In addition, this vital information could alert the working personnel for some upcoming failure, which leads to proper planning and equipment damage prevention.

2.1.2 The Lack of Preventive Maintenance

It is worth mentioning that there is a large number of possible faults that may lead to vibration, when the preventive maintenance is not implemented properly. The most common defect that is encountered on machinery is this of the bearing failure (see Figure 2.2 some bearings components and on Figure 2.3 a bearing’s component defect).

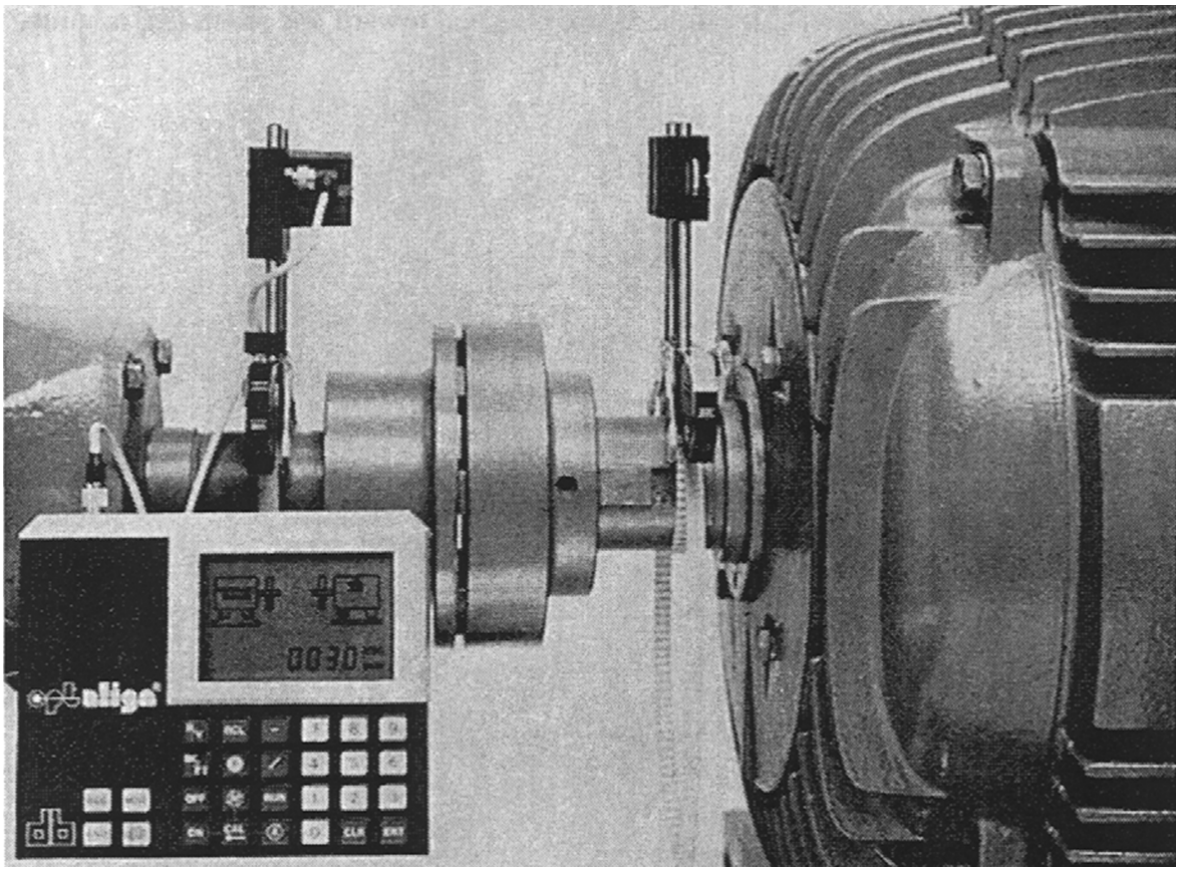

Figure 2.1: Aligning laser tool. Source: [1].

A bearing's defective component will certainly lead to greater problems. Specifically, this could lead to a severe shaft wearing or even thread damage (see 2.4). In some other cases this could lead to a bearing housing damage.

Chen et al included in their research some more photos related to faulty bearings and gearsets [3].

2.1.3 The Reliability

To begin with, the ultimate objective of the preventive maintenance concept is to improve the dependability. In essence, it is a concept that encompasses different subconcepts like Reliability, Availability, Maintenability and others.

The reliability interprets the successful uninterrupted operation of an adequately-performing device or an engineering system. This entails that the operating system must retain not only a low Failure rate, but the most important trait of a device in operation must be the absence of failures so as to assert that a system is ”reliable system”.

According to Martz et al, reliability analysis is a statistical and engineering practice so

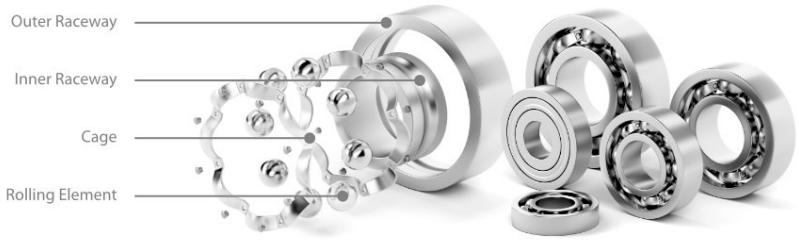

Figure 2.2: Different bearings components (source: [2]).

Figure 2.3: A bearing's defective outer raceway.

as to extract metrics of reliability of a working machine. It can be utilized to assess possible break down modes and their causes. This study on the health status of some machine is used in order to have a numerical-quantified estimate of the machinery reliability [18].

Hence, it is worth it utilizing the aforementioned metrics (MTTF and MTBF), when contemplating reliability, since these constitute the actual metrics of reliability.

The initial aspect of the theoretical background to be elaborated upon is Mean Time to Failure (MTTF) is a fundamental reliability metric used to estimate the average operational lifetime of non-repairable components and systems. In the context of this very article, the main focus of the study is mechanical components (e.g bearings, gears) as well as a system-machine in operation.

Some of the most important functions that are being used in the context of the metrics interpretation are the following:

- $f(x)$ is the probability density function of the time to failure.
- $R(t)$ interprets the probability that the device is still operational at time t [18].

Figure 2.4: Possible defects of shafts.

Figure 2.5: Faulty bearings and gears (source: [3]).

- $F(t)$ is the cumulative distribution function of the time to failure, interpreting the probability that the system has failed by time t.

Interpretation of the Mean Time To Failure

MTTF (Mean Time To Failure) can be modeled as the integral of the reliability function $R(t)$ over time, from zero to infinity [18]:

$$\text{MTTF} = \int_0^\infty R(t)\, dt$$

Where:

$$R(t) \ = Pr(T > t) = \int_t^\infty f(x)\, dx = 1 - F(t)$$

The MTTF interprets the expected time until failure for a large number of identical devices under similar operating conditions.

MTTF is used to predict the average lifespan of a device, providing critical information for maintenance planning, warranty determination, and lifecycle management. A higher MTTF indicates a more reliable device with a longer expected operational life. This metric

is crucial for designing reliable products and for making informed decisions about product development and quality assurance.

Interpretation of the Mean Time Between Failures

The MTBF (Mean Time Between Failures) is another critical metric used in reliability engineering, particularly for repairable systems. It represents the average time between failures for a system or component. For repairable systems, MTBF is calculated as the sum of the operational time divided by the number of failures [18]:

$$\text{MTBF} = \frac{\text{Total operational time}}{\text{Number of failures}}$$

In this case, the reliability function is expressed as follows:

$$R(t) = e^{-\lambda t}$$

Mathematically, MTBF is the inverse of the failure rate λ for systems with a constant failure rate [18]:

$$\text{MTBF} = \int_0^\infty t f(t)\, dt = \int_0^\infty R(t)\, dt = \frac{1}{\lambda}$$

MTBF (Mean Time Before Failure) is an indicative for planning significant maintenance activities tacking the minimum risk possible. The greater this number is the better the reliability of a system is [18]. This ensues that a system go through a lower number of failures.

Interpretation of the Failure Rate

The FT (Failure Rate), often designated as λ, is defined as the frequency with which an engineered system or component fails. The units of the FT is the number of failures per unit time.

The Failure rate is expressed as follows:

$$\lambda(t) = \frac{f(t)}{R(t)} = \frac{f(t)}{1 - F(t)}$$

There is three failure rate cases that may be encountered:

- A constant λ is indicative of a normal-operating system, since the amount of failures over time does not fluctuate.
- An increasing $\lambda(t)$ indicates that the system is getting stressed and becoming more susceptible to failures.
- A decreasing $\lambda(t)$ is indicative of a system that becomes more reliable in terms of time. This happens after the starting phase of its operational life.

Constant Failure Rate

Here comes the interpretation of the Constant Failure Rate, the failure rate λ is constant over time:

$$\lambda = \text{constant}$$

In this case, the reliability function $R(t)$ and the probability density function $f(t)$ are given by:

$$R(t) = e^{-\lambda t}$$

$$f(t) = \lambda e^{-\lambda t}$$

Interpretation of the Availability

Availability (A) is typically expressed as a percentage and is calculated using the following mathematical expression [19]:

$$A = \frac{MTBF}{MTBF + MTTR} \times 100\%$$

Where:

- MTBF (Mean Time Between Failures): The average time interval between two consecutive expected failures of a system component or mechanism.
- MTTF (Mean Time To Repair): The average time required to repair a shut system or mechanism and set it back to operation.

2.1.4 Vibration Analysis Instruments

There is a great number of vibration monitoring products. Some of them are listed hereunder:

- amplitude velocity and acceleration (range of motion) vibrometers
- frequency (rate of oscillation repetition) vibrometers
- accelerometers

The accelerometers and vibrometers, both can be remote - cordless sensor units or wired sensor units connected to some computer. The latter device is a smart device powered by some software so as to store and analyze data. In many cases these devices may also offer some online-cloud options, storing data on some database yielding accessibility for multiple users.

The device depicted in the image exemplifies a standard configuration of a vibration meter and an accelerometer (see Figure 2.6).

The accelerometer measures vibration events in mg (or in some cases in mm/s^2 and interprets these events into frequency domain data. While the vibrometer stores reading of vibration velocity (mm/s) of vibration acceleration (mm/s^2).

These devices have the following common traits, they require the machinery size and rotation as an input, so as to proceed with the results extraction. In addition to this, these can execute this calculation on all three directions simultaneously. And to top it all off, they are expensive.

A deeper understanding of the available in the market sensor is given by Lilly et al [20]. They divide vibration equipment into three levels of detail:

- hand-held sensor
- single-channel FFT
- multi-channel FFT

Vibration measurement tools vary significantly in their technical capabilities. Basic hand-held devices offer convenient portability for field use but have several limitations. These compact units are confined to single-location measurements and typically operate within a

Figure 2.6: Typical vibrometer set-up.

10-1000 Hz frequency window. Using velocity transducers, they provide only fundamental vibration amplitude data without phase information or sophisticated analytical features.

More advanced single-channel FFT instruments deliver superior performance for spectral examination. These systems cover a broader 0.5-20 kHz frequency span with enhanced resolution, supporting up to 1600 spectral lines. While they enable more thorough vibration spectrum evaluation, their single-input design prevents multi-point correlation studies or phase-based diagnostics.

For comprehensive vibration analysis, multi-channel FFT platforms are necessary. These high-performance systems accommodate multiple measurement points simultaneously across an extensive DC-100 kHz bandwidth. With ultra-fine resolution reaching 6400 spectral lines, they capture precise phase relationships vital for resonance detection and operational deflection pattern visualization. Such systems incorporate specialized functions including rotational

speed tracking, coherence verification, and system response characterization - all crucial for detailed mechanical fault investigation and structural dynamic studies.

The primary technical distinctions between these solutions involve: their measurement approach (single versus multiple points), frequency analysis quality (basic versus high-definition), phase measurement availability, and specialized analytical features. Choosing the appropriate tool involves matching the instrument's capabilities to the diagnostic requirements - from preliminary condition checks with hand-held units to in-depth failure analysis using multi-channel systems.

There is also some obsolete set-ups, for instance the station used by Betta et al for their experiment [4]. This station encompasses an acceleration sensor (8710A50M1 by Kistler™) with 5g measuring range, a signal conditioner with adjustable gain (5118B1 by Kistler™) for data translation, a DPC/C40 by Loughborough Sound Images™ for FFT-Analysis and a host PC (see Figure 2.7).

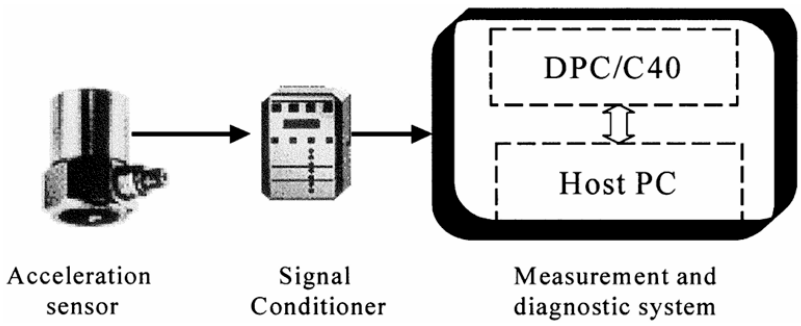

Figure 2.7: Block diagram of the designed station for vibration analysis (source: [4]).

There is some more accessible alternatives. Specifically, there is some small sized low-cost acceleration sensor on breakouts that can be used on related projects (see Figure 2.8).

2.1.5 Typical Applications of Vibration Analysis

Some typical applications that use vibration analysis implementations are turbine motors and shafts, motors connected pumps and motors connected to roll equipment among others.

Li and McKee explain in their research that most of the vibration analysis applications are related to electrical and mechanical equipment that utilize roller bearings [17].

Similarly, Lilly and Marscher have studied the vibration and noise derived from pumps and motors separately [20]. In addition, the concept of translational and torsional vibration is

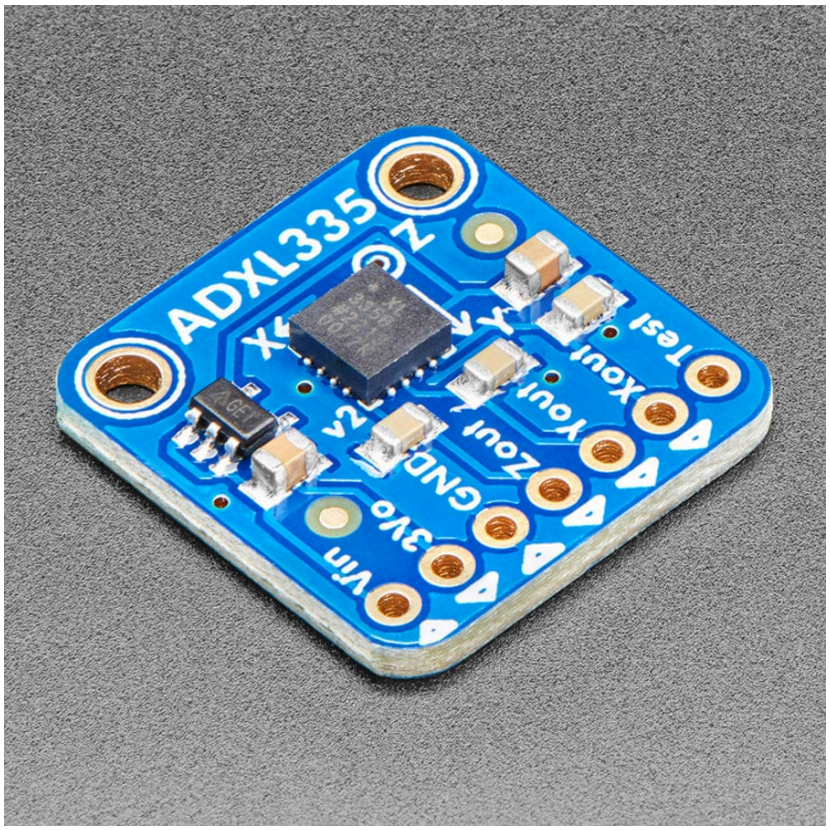

Figure 2.8: ADXL 335 accelerometer.

introduced. In addition to this, the concept of vibration analysis on piping is studied as well.

The translational vibration analysis is made for the engine-pump-support structure, investigating the lateral vibration. For this study the usage of high-quality sensors is a prerequisite. There is also the concept of the torsional vibration analysis, which is also subtle and requires special study on the oscillating torque. This application is difficult to be studied.

Moreover, Lilly and Marscher investigate also the possibility of using some actuators that may compensate the noise of the motor-pump compound, making the most of the electronically generated sound waves. This technique aims at the cancellation of noise pollution [20].

The study by Jiao et al. demonstrates the efficacy of time-domain feature extraction combined with Independent Component Analysis (ICA) for gearbox fault diagnosis. Their approach maximizes mutual information to isolate fault-related vibrations from noisy multi-channel measurements, achieving high accuracy in classifying faults such as teeth surface damage and loose foundations [21].

There is also the chance of implementing advanced signal processing techniques, such as envelope analysis and cyclostationary methods, to enhance fault detection in complex

systems like helicopters, where high noise and low signal-to-noise ratios are prevalent [22].

In the context of unmanned aerial vehicles (UAVs), Mohd Ghazali and Rahiman employ vibration sensors and artificial intelligence (AI) techniques, including fuzzy logic and neural networks, to detect faults in drone arms. Their framework addresses the challenges of real-time decision-making and integrates IoT for remote monitoring, showcasing the adaptability of vibration analysis to emerging technologies [23].

Meanwhile, Girondin et al. focus on bearing faults in helicopters, proposing redundancy-based harmonic analysis to overcome limitations imposed by low sampling frequencies and environmental noise. Their geometric mean criterion proves robust in identifying fault frequencies despite parasitic vibrations [22].

Theoretical advancements in vibration analysis also emphasize the role of preprocessing steps, such as whitening and Minimum Entropy Deconvolution (MED), to accentuate impulsive features indicative of faults [22]. These methods, combined with machine learning classifiers like Support Vector Machines (SVM) and neural networks, enable automated fault diagnosis with minimal human intervention, as demonstrated in gearbox and bearing applications [21, 23].

Vibration analysis constitutes significant asset of the modern industry, from traditional rotating machinery to cutting-edge UAVs. Its evolution—driven by innovations in signal processing, AI, and IoT—continues to evolve while data bases are still developing.

2.2 Fundamentals of Bearing Fault Frequencies

Li and McKee analyze the failure modes of bearing and yield a brief explanation of what each failure mode entails. It is highlighted by their research that wide-band impulses are generated when rollers pass over a defect. As far as the diagnostics algorithm is concerned, in this research there is presented statistical parameters that help in finding bearings localized defects. In addition to this, they stress that the frequency-related algorithms like the Hilbert transform are the most prominent in bearing diagnostics.

The most significant challenge that Betta et al had to deal with is that of the faulty cases detection. For this reason, fault and unfault models have been developed, and series of comparisons between the actual model generated by apparatus measurements and these two situational models are being executed. This development constitutes a real challenge since this

is the main criterion that identifies the state of the motor [4].

2.2.1 Unfault and Fault Models

Betta et al make use of a model-based approach, which entails that the main concept is comparison-oriented. Specifically, there is the unfault model, which contains a high-amplitude peak at the shaft rotating frequency F_ω. In addition to that, it can be noticed in Figure 2.9 that some other patterns distinct themselves from the previously-mentioned unfault model. Regarding the unbalance model, it is obvious that the amplitude at the first tone is much greater. In the case of the misalignment model, it can be noticed that the tone at $2F_\omega$ has a much greater amplitude while the amplitude at the tone of the shaft speed F_ω remains at a high level. In the case of looseness, there are some more tones prevailing on the spectrum, among the tones related to the shaft speed. To top it all off, the most important vibration spectrum is that of the bearing failure (Refer to Figure 2.10). Every bearing is characterized by four critical frequencies, given by its manufacturer, and these frequencies are associated with the four bearing components (inner and outer ring, balls, and cage) [4], [24]:

- BPFO (Ball Pass Frequency Outer) - outer race failure
- BPFI (Ball Pass Frequency Inner) - inner race failure
- BSF (Ball Spin Frequency) - rolling element failure
- FTF (Fundamental Train Frequency) - cage failure

The procedure to calculate critical frequencies involves two steps: first computing the bearing frequency factors, then deriving the vibration frequencies. The following analysis covers the case where the inner race rotates while the outer race remains stationary [17], [4].

$$F_{\text{OR}} = \frac{Z}{2}\left(1 - \frac{d}{D}\cos\alpha\right) \qquad \text{(Outer Race Fault)} \qquad (2.1)$$

$$F_{\text{IR}} = \frac{Z}{2}\left(1 + \frac{d}{D}\cos\alpha\right) \qquad \text{(Inner Race Fault)} \qquad (2.2)$$

$$F_{\text{FTF}} = \frac{1}{2}\left(1 - \frac{d}{D}\cos\alpha\right) \qquad \text{(Cage Frequency, Inner Race Rotating)} \qquad (2.3)$$

$$F_{\text{BS}} = \frac{D}{d}\left[1 - \left(\frac{d}{D}\right)^2 \cos^2\alpha\right] \qquad \text{(Ball Spin Frequency)} \qquad (2.4)$$

Then, the aforementioned factors when multiplied with shaft speed (f) give specific critical bearing vibration frequencies:

$$\text{BPFO} = f \times F_{\text{OR}} \tag{2.5}$$

$$\text{BPFI} = f \times F_{\text{IR}} \tag{2.6}$$

$$\text{BSF} = f \times F_{\text{BS}} \tag{2.7}$$

$$\text{FTF} = f \times F_{\text{FTF}} \tag{2.8}$$

where:

f	: Shaft rotational speed (Hz)
BPFI	: Ball pass frequency, inner race
BPFO	: Ball pass frequency, outer race
FTF	: Fundamental train frequency
BSF	: Ball spin frequency
Z	: Number of rolling elements
D	: Pitch circle diameter of the bearing (mm)
d	: Rolling element (ball) diameter (mm)
α	: Contact angle ($^\circ$)

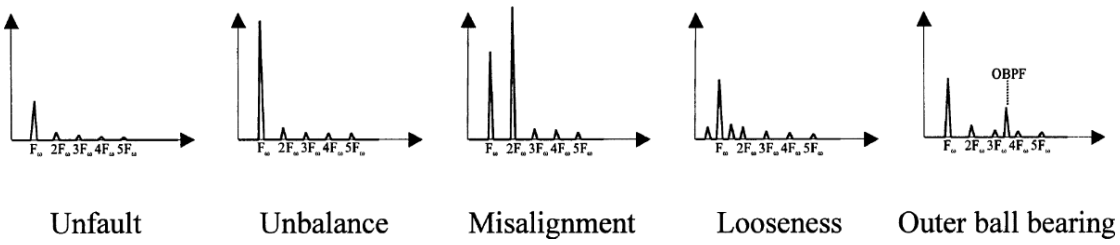

Figure 2.9: Fault and unfault spectrum graphs (source: [4]).

It is also worth noting that the bearing frequency factors can be obtained through:

- Manufacturer-provided values
- Direct calculation using the above equations (2.1)–(2.4)
- Reverse calculation from measured vibration frequencies (2.5)–(2.8)

To summarize the preceding analysis, Figure 2.10 provides a clear visualization of the fault diagnosis procedure. When analysis reveals elevated frequency tones, a systematic approach must be followed to identify the source of these dominant frequencies. This process essentially involves mapping characteristic vibration patterns to specific machine faults on a frequency spectrum plot.

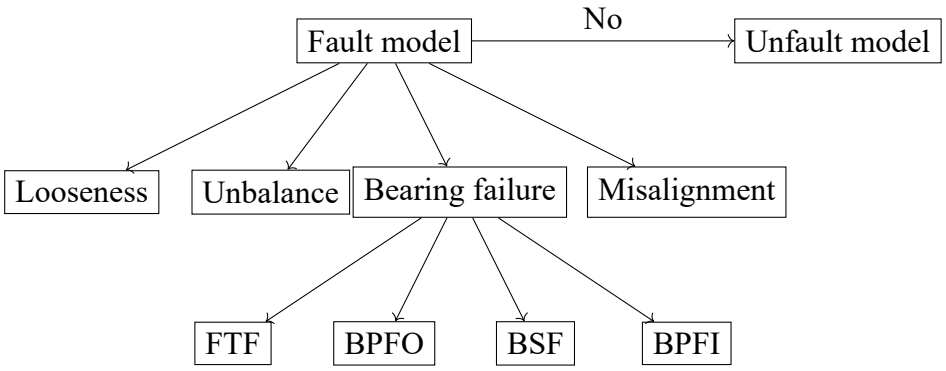

Figure 2.10: Fault tree analysis diagram showing the relationship between machine conditions and specific fault frequencies.

It must be articulated that a fault-emulation method can be used in such cases to simulate the faulty models. This entails that additional tones can be stressed, and crucial spectra modifications may take place for research and comparison purposes.

2.3 Algorithms and Technologies

2.3.1 Algorithm Introduction

This section aims at yielding a brief introduction into the concept of bearing failure predictive maintenance plan. The set-up utilizes an accelerometer which collects data that are going to be used for model assessment, by the means of Fast-Fourier Analysis.

In most of the actual applications a machine learning implementation is a common practice. A figure that depicts this modeling has been presented by Dharmarathne et al [5].

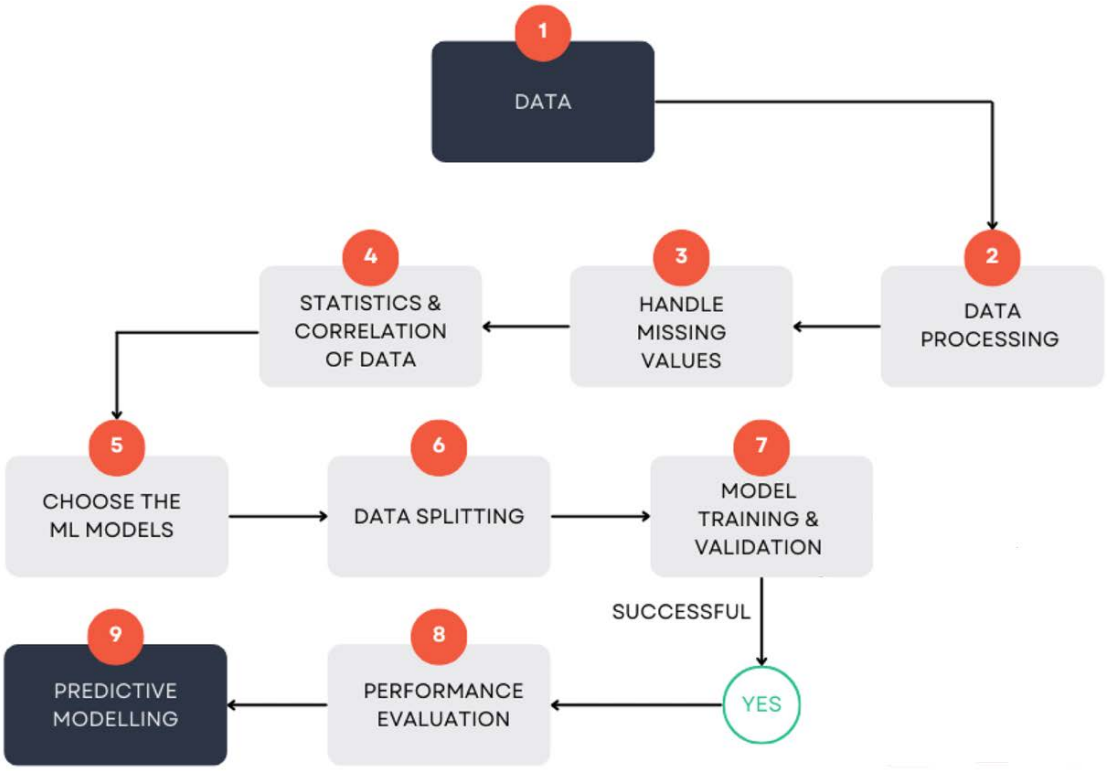

Figure 2.11: Data collection and assessment algorithm for bearing failure prediction. Source: [5].

2.3.2 Signal Processing

Fourier Analysis

In this section a deeper elabotation on Fourier Analysis and Transform is going to take place.

Serafeimidou explains in detail the theoritical background of the Fourier Analysis. For an arbitrary finite range [-L,L] there is a family of functions, which meet the Dirichlet condition, and these functions can be expressed as a trigonometric sequence [25]:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos(n\pi\omega x) + b_n \sin(n\pi\omega x) \right), \tag{2.9}$$

where $\omega = \frac{\pi}{L}$.

In addition its constants can be expressed as follows [25]:

$$a_n = \frac{1}{L} \int_{-L}^{L} f(u) \cos(n\omega u)\, du \tag{2.10}$$

$$b_n = \frac{1}{L}\int_{-L}^{L} f(u)\sin(n\omega u)\,du \tag{2.11}$$

The equation (2.9) needs to be expressed as a valid form within the whole $x \in \mathbb{R}$. For this reason the trigonometric integral (which is called Fourier integral) can be expressed as follows [25]:

$$F(x) = \int_0^{\infty} \left[A(\omega)\cos(\omega x) + B(\omega)\sin(\omega x)\right]\,d\omega \tag{2.12}$$

where:

$$A(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(u)\cos(\omega u)\,du \tag{2.13}$$

$$B(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(u)\sin(\omega u)\,du \tag{2.14}$$

Assuming that there exist real functions $f(x)$ defined in $\mathbb{R}$ meeting the following points:

- $f(x)$ and its derivative $f'(x)$ are continuous in the range $[-L, L] \subset \mathbb{R}$.
- $f(x)$ is integrable in $\mathbb{R}$:

$$\int_{-\infty}^{\infty} |f(x)|\,dx < \infty.$$

Hence, given that the aforementioned conditions are met, $A(\omega)$ (see Eq. (2.13)) and $B(\omega)$ (see Eq. (2.14)) tend to attain a real value for ω, which implies that $A(\omega)$ and $B(\omega)$ are defined for every $\omega \in \mathbb{R}$. In addition to this, the integral of the equation (2.12) tends to attain a real value for x, which ensues that F(x) is defined for every $x \in \mathbb{R}$ [25].

$F(x)$ is identical to $f(x)$ whenever $f(x)$ is a continuous function. There are some special cases when $f(x)$ is not a continuous function, and to deal with these cases, we define [25]:

$$F(x) = \frac{1}{2}\left[\lim_{x\to x^+} f(x) + \lim_{x\to x^-} f(x)\right] \tag{2.15}$$

This ensures that $F(x)$ is properly handled even in cases of discontinuity.

All things considered the Fourier Transform $F(\omega) = \mathcal{F}\{F(x)\}$ can be mathematically expressed as follows [25]:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}\,dt \tag{2.16}$$

where the frequency can derive from the following form: $\omega = \frac{2\pi jk}{N}$.

In practice, the Discrete Fourier Transform (DFT) boasts an extremely wide variety of applications, facilitating the Fourier Transform when applied to sampled signals.

Hence, the mathematical formulation for determining the amplitude of each frequency, given a waveform $x_0, x_1, \ldots, x_{n-1}$ consisting of real values, is structured as follows:

$$X(j) = \sum_{k=0}^{n-1} x(k) e^{-\frac{i2\pi jk}{n/2}}, \quad \text{for } j = 0, 1, \ldots, n-1. \tag{2.17}$$

Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT), expessed by the equation (2.17) and its inverse. One of the most prominent FFT algorithms, the Cooley-Tukey algorithm [15], employs a divide-and-conquer approach to reduce the computational complexity from $\mathcal{O}(n^2)$ (for the naive DFT) to $\mathcal{O}(2n \log n)$, enabling significant speed improvement for large amount of data.

This reduces complexity from $\mathcal{O}(n^2)$ (naive DFT entails direct implementation of the equation (2.17)) to less than $\mathcal{O}(2n \log n)$ (a Fast Fourier Transform implementation), enabling major time savings for complex and large-scale transforms. It is also worth articulating that the complexity can be interpreted as the number of operations executed [15], [16].

Both algorithms are implemented in major numerical packages including MATLAB and NumPy's routines [26]. What is highlighted by Singeleton is that the Cooley-Tukey approach is considerably faster and decently performing especially when working along with some additional practices so as to execute the Fast Fourier Transform [16].

The computation time according to the aforementioned article decreases drastically by making use of the Cooley-Tukey algorithm. In addition, this very algorithm performs and adapts better to different applications compared to Good's, Danielson's and Goertzel's approaches.

Cooley and Tukey proved that the sample length can be composite size where it is preferable to make use of sample points sized as power of 2, especially when it comes to personal computers with binary arithmetic system. In addition to this the authors have proven that the power of 3 is formally the most efficient approach, however in the context of the fast Fourier Analysis the so-called Radix-2 implementation is the prominent one [15].

$$N = 2^m \tag{2.18}$$

It is hereby worth it elaboration on the Fast Fourier Transform algorithm [15]. The indices of the equation (2.17) can be expressed as follows:

$$\begin{aligned} j &= j_{m-1} \cdot 2^{m-1} + j_{m-2} \cdot 2^{m-2} + \cdots + j_1 \cdot 2 + j_0, \\ k &= k_{m-1} \cdot 2^{m-1} + \cdots + k_1 \cdot 2 + k_0, \end{aligned} \tag{2.19}$$

where j_v and k_v are the contents of the respective bit positions within the binary representation, which ensues that they are equal to 0 or 1.

The aforementioned set-up (see eq. (2.17)) leads to the following equation (2.21), encompassing twiddle factors expressed as [15]:

$$W = e^{-2\pi i/N} \tag{2.20}$$

And the Fast Fourier Transform [15]:

$$X(j_{m-1}, \cdots, j_0) = \sum_{k_0} \sum_{k_1} \cdots \sum_{m-1} x(k_{m-1}, \cdots, k_0) \cdot W^{j \cdot k_{m-1} \cdot 2^{m-1} + \cdots + j \cdot k_0} \tag{2.21}$$

where as per the first step:

$$W^{j \cdot k_{m-1} \cdot 2^{m-1}} = W^{j_0 \cdot k_{m-1} \cdot 2^{m-1}} \tag{2.22}$$

Furthermore, in the context of the first calculation, the first array can be expressed as follows [15]:

$$x_1(j_0, k_{m-2}, \cdots, k_0) = \sum_{\mathbf{k_{m-1}}} x(\mathbf{k_{m-1}, \cdots, k_0}) \cdot W^{\mathbf{j_0 \cdot k_{m-1} \cdot 2^{m-1}}} \tag{2.23}$$

And hereunder can be contemplated that proceeding to the next innermost-successive sum [15]:

$$W^{j \cdot k_{m-1} \cdot 2^{m-1}} = W^{(j_{l-1} \cdot 2^{l-1} + \cdots + j_0) \cdot k_{m-l} \cdot 2^{m-l}} \tag{2.24}$$

followed by the calculation of the arrays [15]:

$$x_l(j_0, \cdots, j_{l-1}, k_{m-l-1}, \cdots, k_0) = \sum_{\mathbf{k_{m-l}}} x_{l-1}(j_0, \cdots, j_{l-2} \cdots, k_{m-l}, \cdots, k_0) \cdot W^{(j_{l-1} \cdot 2^{l-1} + \cdots + j_0) \cdot k_{m-l} \cdot 2^{m-l}} \tag{2.25}$$

for l = 1, 2, ... , m

And this could constitute a recursive procedure obtaining the successive arrays. And last but not least, this finally could lead to the calculation of X spectrum. It can be contemplated that from the Equation (2.25), there is at all times two consecutive values, the even and the odd value that can be stored at a specific location corresponding to an index of [15]:

$$(j_0 2^{m-1} + \cdots + j_{l-1} 2^{m-l} + k_{m-l-1} 2^{m-l-1} + \cdots + k_0) \tag{2.26}$$

It is also worth it recalling that m is defined by $N = 2^m$. Since bit positions $2^(m - l)$, meaning the j and k coefficients, can only get the values 0 and 1, which ensues that there is multiple-storage locations that can be calculated simultanously.

The array calculated gives the desired Fourier sums [15]:

$$X(j_{m-1}, \cdots, j_0) = A_m((j_0, \cdots, j_{m-1}) \tag{2.27}$$

At this point there is a need to simplify the Fast Fourier Transform algorithm. It is mendatory to recall the (2.21) so as to depict the algorithm.

The calculations workload can be reduced, by decomposing the original input sequence $X[k]$, of length N, into two subsequences [15]:

- A sequence consisting of elements at even-numbered indices: $X_{\text{even}}[k]$
- A sequence consisting of elements at odd-numbered indices: $X_{\text{odd}}[k]$

By processing these smaller sequences independently, the algorithm substantially reduces the computational workload. The FFT is applied recursively to each half, and this decomposition continues until each subproblem has length 1, at which point the computation becomes trivial (the amplitude value corresponding to a specific frequency). Furthermore, the results from the smaller FFTs are recombined using twiddle factors (see Equation (2.20)).

An significant property of this algorithm is that that, in the case of $j > \frac{n}{2}$, the values are repeated due to the sinusoidal property of $\exp$, which is known as the symmetry identity. This entails that for the values $j = 0, 1, \ldots, n/2 - 1$:

$$X_{\text{even}}(j + n/2) = X_{\text{even}}(j) \quad \text{and} \quad X_{\text{odd}}(j + n/2) = X_{\text{odd}}(j) \tag{2.28}$$

The evens and odds decomposition:

$$X_{even}(j) = \sum_{k=0}^{n/2-1} x(2k) e^{-\frac{i2\pi jk}{n/2}}, \quad \text{for } j = 0, 1, \ldots, n/2 - 1. \tag{2.29}$$

$$X_{odd}(j) = \sum_{k=0}^{n/2-1} x(2k+1)e^{-\frac{i2\pi jk}{n/2}}, \quad \text{for } j = 0, 1, \ldots, n/2-1. \tag{2.30}$$

So from the Equation (2.21):

$$X(j) = X_{even}(j) + e^{-\frac{i2\pi j}{n}} X_{odd}(j) \tag{2.31}$$

And this procedure is scalable since each and every X_{even} and X_{odd} can be deplited into a subsequent set of X_{eve} and X_{odd} to the point that the length 1 is reached.

Last but not least, the symmetry identity as described by the Equation (2.28), leads to the following set of equations (see also (2.31)):

$$X(j+n/2) = X_{even}(j) + e^{-\frac{i2\pi(j+n/2)}{n}} X_{odd}(j) \tag{2.32}$$

For the values $j = 0, 1, \ldots, n/2-1$.

The objective of this algorithm is to get to the point that the length is of the sums is 1, meaning that the sums of the Equations (2.29) and (2.30) are applicable for k=0. The only remaining exponential terms $e^{-\frac{i2\pi j}{n}}$ in the Equation (2.31) - as previously described by Equation (2.25).

The aforementioned procedure can be further repeated, by breaking the branches problem into sub-branches. The radix-2 DIT algorithm to be formulated as follows on Figure 2.12, depicting the concept of narrowing down the problem.

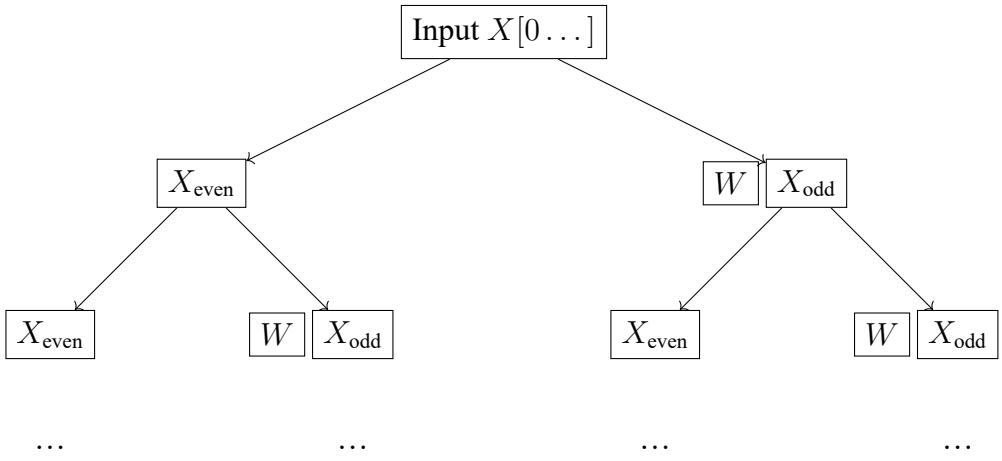

Figure 2.12: Recursive decomposition using even-odd splitting.

All in all, this section yields an overview of utmost importance highlighting the most invaluable Fourier Transform algorithms expressed by equation (2.17) and equation (2.21). As

discussed in this section, the Cooley-Tukey Fast Fourier Transform (FFT) algorithm significantly accelerates the computation of the Discrete Fourier Transform (DFT) by recursively breaking the problem into smaller, more manageable subproblems. This divide-and-conquer approach is fundamental to the algorithm's performance advantage.

2.4 Internet of Things

2.4.1 Introduction to Internet of Things & Key Technologies

The ultimate objective of this project is to create a set-up that is well-rounded and has a real-time monitoring and assessment competency. Two of the most prominent technologies that are being incorporated by Internet of Things (IoT) is the cloud computing and the wireless communication. It is also worth mentioning that, Internet of things encompasses interconnection of physical devices and applicaitons making the most of sensors, networks and data analysis.

2.4.2 Cloud-Oriented Perspective

Gubbi et al describe the IoT (Internet of Things) as a network of sensing and actuating devices that share information and interact with each other, since there is a combination of different standards (e.g cloud computing and data analytics). This approach discusses only the purpose of IoT [27].

This approach is mostly ”Cloud”-oriented. The authors focus mostly on the concept of ubiquitous information and evolving communication networks, powered by Smart Connectivity. Moreover, IoT (Internet of Things) is fueled by technologies such as Bluetooth, radio frequency identification (RFID) and Wi-Fi[27].

Nevertheless, Hardware-sensor breakouts are critically important, as they serve as the primary means of data collection in IoT systems. Middleware and computing tools for data analytics are equally invaluable, as they enable the processing of raw sensor data into actionable insights. Finally, visualization tools and dashboards play a key role in end-user interaction with the IoT environment, while also providing stakeholders with an intuitive and accessible representation of the data [27].

2.4.3 Internet- vs. Things-Oriented Perspectives

As regards the approach of Atzori et al, there is given a different definition by considering two different approaches, the ”Internet”-oriented approach and the ”Things”-oriented approach. The ”Things”-oriented approach targets mainly at Big Data applications, since according to Atzori et al the time, the place and multipe recipients are the most important factors of this approach. In essnece the ”Things” that contribute to this perspective are the Radio-Frequency IDentification (RFID), Near Field Communications (NFC) and Wireless Sensor and Actuator Networks (WSAN) among others.

The ”Internet”-oriented vision given by Atzori et al, it is highlighted that the IP (Internet Protocol) constitutes the core technology of Internet of Things (IoT). The Internet Protocol (IP) is lightweight, scalable, and capable of connecting smart objects worldwide—even on low-power devices. The authors also introduce Internet Ø, a similar approach that further simplifies IP to enable ”IP over anything,” ensuring seamless IoT integration. The ultimate goal of both strategies is to make all objects addressable, reachable, and interoperable through IP, moving from a basic ”Internet of Devices” to a fully connected ”Internet of Things” [6].

2.4.4 Semantic-Oriented Approach

In addition to the aforementioned perspectives there is an emerging need for representation and storage of the invaluable information, which leads to the ”Semantic”-oriented approach (see 2.13). This final approach creates a well-rounded Internet of Things (IoT) concept, yielding more details on it.

The whole concept of the vast majority of different projects requires a semantic approach so as to set some thresholds and yield some feedback or take some action over something. In addition to this the semantic vision according to Atzori et al expresses the need for a representation of the collected data [6].

Atzori defines Internet of Things (IoT) as a broad network (worldwide network) of unique entities that interact with each other via standard communication protocols.

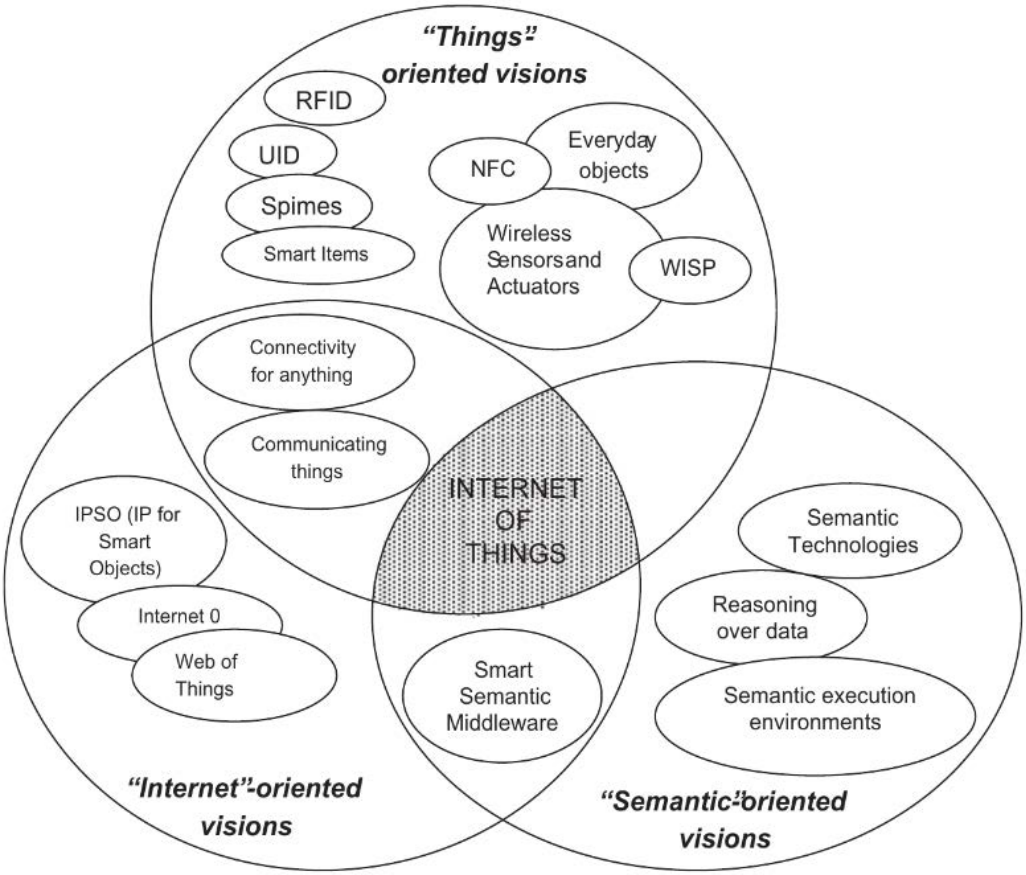

Figure 2.13: The Internet of Things paradigm as a result of the convergence of different visions. Source: [6].

2.5 Project Implementation: Sensing & Communication

2.5.1 The Sensor Specifications

As it has been highlighted by the Internet of Things definitions, the sensors are among the most important devices of this technology. The sensing unit in this project utilizes an ADXL335 accelerometer to collect physical acceleration data, which connects to the micro-controller through an analog input interface.

The accelerometer ADXL335 is a single component sensing X, Y and Z, having a wide measuring range in all three axes. Typical applications of this sensor is structure deflection projects. This sensor is the ideal choice when it comes to vibration analysis applications, since it retains a measuring range of $\pm 3g$. This range adequate with respect to vibration analysis because in most of the rotating equipment applications an acceleration greater than 1,5g indicates a defective component. This entails that the $\pm 3g$ is sufficient in the context of this

project. In addition to this, it offers higher resolution, up to 300mV/g for vibration detection, which means it has relatively high sensitivity to small vibrations. As regards its frequency bandwidth, it ranges 0.5-1600Hz. As regards its noise density, it equals to $150\,\mu g/\sqrt{Hz}$, which is ideal for low-amplitude vibration waveforms[13].

The Adafruit ADXL335 is a high performance sensor breakout, utilizing capacitors so as to decouple the noise of the power supply and filters om analog outputs for antialiasing and noise reduction.

As for the sensor's theory of operation, the ADXL335 is fabricated using a polysilicon surface-micromachining process on a silicon wafer. The sensing element is suspended above the wafer by polysilicon springs, which provide a restoring force against acceleration. When acceleration occurs, the suspended mass is displaced, and this movement is detected through a differential capacitive structure composed of fixed electrodes and electrodes attached to the movable mass. The fixed electrodes are driven by two square-wave signals that are 180° out of phase. Any displacement of the mass causes an imbalance in the differential capacitance, generating an output signal whose amplitude is directly proportional to the applied acceleration. Using phase-sensitive demodulation, the sensor extracts both the magnitude and direction of this acceleration [13].

This combination of mechanical design and signal processing provides robust acceleration measurement without requiring temperature compensation circuits -since this breakout retains a large tolerance into operation temperatures- or complex calibration procedures [13].

2.5.2 The Communication Protocol

The system architecture of the ADXL335 breakout supports analog communication protocol option, however there is a number of different communication protocols frequently utilized by the Adafruit breakouts:

- **Analog Input**: Currently implemented for direct sensor interfacing (ADXL335)
- **Digital Input**: Available for threshold-based detection applications or PWM
- **SPI (Serial Peripheral Interface)**: Suitable for high-speed (up to 10 Mbps) communication with multiple peripherals
- **I^{2}C (Inter-Integrated Circuit)**: Enables multi-device communication using only two wires (SDA/SCL)

- **UART (Universal Asynchronous Receiver-Transmitter)**: Implemented at a predetermined baud rate for Arduino-to-Raspberry Pi communication

The Universal Asynchronous Receiver-Transmitter Protocol

The Arduino board is connected to the Raspberry Pi microcomputer via its USB port. The USB port of the Arduino is in essence a UART device, that has two options, not working simultaneously. The first one out of the two is the Rx - Tx pins option and the second one is the USB-cable option. In the case of the Arduino-Raspberry Pi interconnection, the desired set-up requires a USB-cable.

The Universal Asynchronous Receiver-Transmitter (UART) is a peripheral device for asynchronous serial communication. The baud rate and the data format is configurable, based on the project needs. After hands-on try and error, it has been found out that a baud rate of 115200 performs adequately offering a higher speed on data transferring, which is vital for the sake of this very project. The data transferring speed is of vital importance, since the higher the amount of data sent to the database is, the larger the frequency domain data range can be.

The raw UART (transmitted via USB) data is in a binary format, and upon receipt by the Raspberry Pi the data batch needs to be converted into a string format. For this reason it is mandatory to use UTF-8 (8-bit Unicode Transformation Format) to handle ASCII characters at this point.

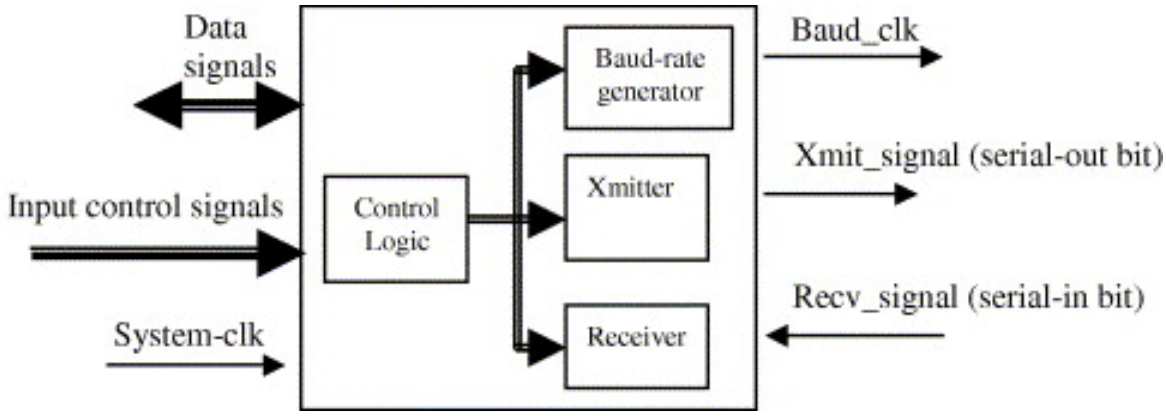

Figure 2.14: The UART block diagram. Source: [7].

A standard UART includes the following key modules (see Figure 2.14):

- **Transmitter** (parallel-to-serial conversion)

- **Receiver** (serial-to-parallel conversion)
- **Baud-rate generator**
- **Control logic**

The baud-rate generator derives a *baud clock* from the system clock. This clock typically operates at 16 times the desired baud rate:

$$\text{baud\_clock} = 16 \times \text{baud\_rate}$$

This oversampling approach ensures precise timing for bit detection and improves communication reliability.

UART transmits data in structured frames composed of the following elements:

- **Start bit**: Always logic 0
- **Data bits**: Typically 6 to 8 bits (configurable)
- **Optional parity bit**: For basic error detection
- **Stop bits**: Logic 1, usually 1 or 2 bits

The receiver "synchronizes" its internal clock to that of the transmitter's at the beginning of every data packet received. This allows the receiver to sample the data bit at the bit-cell center. A data packet is composed of 1 start-bit, which is always a logic 0, followed by a programmable number of data bits (typically between 6 and 8). The receiver detects the start bit by sensing a falling edge (logic 1 to 0 transition) and then samples each subsequent bit at the center of its bit period, using the internally generated baud clock. The stop bit must always be logic 1.

UART's simplicity and robustness make it a widely used solution for embedded systems and serial communication with peripheral devices [7].

Analog Protocol

The Arduino ATmega controllers used for the Arduino contain an onboard 6 channel analog-to-digital (A/D) converter. The converter has 10 bit resolution, returning integers from 0 to 1023. While the main function of the analog pins for most Arduino users is to read analog

sensors, the analog pins also have all the functionality of general purpose input/output (GPIO) pins (the same as digital pins 0 - 13) [14].

In this context, it is suggested that the Arduino analogRead function work along with some source code that executes calibration function before starts gathering the actual data. This would increase the reliability of the readings with a small toll on the amount of data spent on measuring. The reason why mapping is required is that the Arduino can only return integers from 0 to 1023. Arduino analog pins compare the input voltage on each pin to a reference voltage, which in turn is the operating voltage 5V. So this practice interprets the measurement input from the sensor as a value within the range [0-1023].

$$\text{analogRead()} = \left(\frac{V_{\text{reading}}}{V_{\text{reference}}} \right) \times 1023 \tag{2.33}$$

Another practice that can also increase the reliability of the readings is the usage of some statistical means that would decrease the chances of chances of reading false data. Some exquisite practices are the avarage value, the median value and the avarage value after getting rid of the minimum and maximum values.

The aforementioned practices are not mandatory for this context. In addition to this they reduce severely the amount of data within the reading, which leads to smaller amount of data on frequency domain, after carrying out the Fast Fourier Transform. For this reason, raw data that can go through some mapping-calibration is the ideal practice.

Chapter 3

System Engineering and Design

3.1 Introduction

As it has already been articulated, the objective of this diploma thesis is to study, design, and develop a complete measuring system that integrates a sensor, a microcomputer, and a microcontroller into a coherent and functional unit. The intention behind this effort is not only to demonstrate the feasibility of such a system, but also to provide a framework that can be applied to real-world industrial monitoring scenarios.

The system is specifically conceived for low-cost, real-time vibration monitoring of rotating machinery, a task of significant importance in predictive maintenance and condition monitoring. By capturing the vibrational behavior of machines during operation, valuable insights can be obtained about their mechanical health, potential faults, and overall performance. This contributes directly to reducing downtime, optimizing maintenance schedules, and extending the operational lifespan of equipment.

The chosen architecture relies on a modular set-up that combines hardware and software components of major importance. At its core lies a sensor element capable of detecting vibrations within a predetermined, application-relevant frequency range. The signals acquired are then transferred to a microcontroller, which is tasked with initial data handling, conditioning, and communication. Subsequently, the processed data is relayed to a microcomputer that ensures reliable storage in a structured database, while also providing the computational capacity required for further analysis and processing. In this manner, the system covers the complete cycle of measurement: from data acquisition, to secure storage, and finally to intelligent interpretation.

The following sections of this thesis provide a more detailed overview of the system requirements, addressing both functional requirements (such as measurement accuracy, sampling rates, and communication protocols) and non-functional requirements (such as cost efficiency, scalability, and reliability). Furthermore, a thorough elaboration on the system set-up is presented (see Figure 3.1), offering a comprehensive description of the interaction between hardware modules, data flow, and the software infrastructure that binds them together.

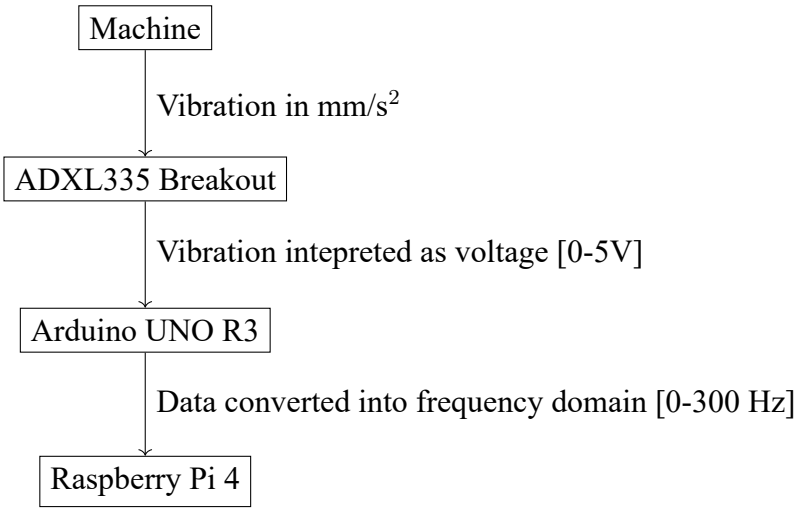

Figure 3.1: The concept layout.

3.2 Hardware Architecture

3.2.1 ADXL335 Accelerometer

The ADXL335 is a small, low-power, 3-axis accelerometer capable of measuring acceleration with a minimum full-scale range of $\pm 3\,g$. It is designed for motion- and tilt-sensing applications, making it suitable for mobile devices, gaming systems, disk drive protection, image stabilization, and sports and health devices. The device operates on a single supply voltage ranging from $1.8\,V$ to $3.6\,V$ and consumes only $350\,\mu A$ of current, making it ideal for power-sensitive applications [13].

The Adafruit breaout of this very sensor has a power supply at 5V which is cut off according to the needs of the sensor. The VCC takes up to 5V in and regulates it to 3.3V with an output pin. The analog outputs are ratiometric: that means that 0g measurement output is always at half of the 3.3V output (1.65V), -3g is at 0v and 3g is at 3.3V with full scaling in between.

The aforementioned Adafruit ADXL335 accelerometer boasts a wide variety of amendments in comparison with the original stand-alone sensor. Most importantly, this breakout is an ideal choice when it comes to interfacing with 5V microcontrollers like Arduino. It is compatible with microcontrollers because there is pending connections on the breakout. Alternatively, a PCB must be designed so as to solder the small scale and hard to manage pins of the sensor itself.

Another significant feature of the breakouts is that manufacturing research has been conducted and this has lead to the optimal product development. This entails that these breakouts encompass decoupling capacitors, filters and power supply regulation that assures the pinout signals to be as accurate as possible.

It can measure the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion, shock, or vibration.

A figure depicting the ADXL335 accelerometer is depicted on Figure 2.8 of Chapter 2. The ADXL335 is housed in a 16-lead LFCSP (Low Profile Chip Scale Package). Table 3.1 describes its key pin functions.

Table 3.1: ADXL335 Pin Functions [13]

Pin Number	**Pin Name**	**Description**
2	ST	Self-Test pin. When set to V_S, it generates an electrostatic force to test functionality.
3,5,6,7	COM	Common ground pins.
12	X_{OUT}	X-axis analog output.
10	Y_{OUT}	Y-axis analog output.
8	Z_{OUT}	Z-axis analog output.
14,15	V_S	Supply voltage pins ($1.8\,V$ to $3.6\,V$).
-	Exposed Pad	Not internally connected but should be soldered for mechanical integrity.

Table 3.2 summarizes the key specifications of the ADXL335 accelerometer [13].

All in all, the ADXL335 is widely used in applications requiring precise motion detection and tilt sensing. Its excellent temperature stability and low power consumption make it suitable for portable and battery-operated devices. The device's performance is further enhanced by its low noise density and adjustable bandwidth, allowing for customization based

Table 3.2: ADXL335 Key Specifications [13]

Parameter	**Value**
Measurement Range	$\pm 3\,g$ (minimum), $\pm 3.6\,g$ (maximum)
Sensitivity	$300\,mV/g$ (typical) at $V_S = 3\,V$ (ratiometric)
Zero g Bias	$1.5\,V$ (typical) for X and Y axes, $1.5\,V$ (typical) for Z axis
Noise Performance (X,Y axes)	$150\,\mu g/\sqrt{Hz}$ (typical)
Noise Performance (Z axis)	$300\,\mu g/\sqrt{Hz}$ (typical)
Bandwidth (X,Y)	Adjustable up to $1600\,Hz$ via external capacitors
Bandwidth (Z)	Adjustable up to $550\,Hz$ via external capacitors
Operating Temperature	$-40°C$ to $+85°C$
Shock Survival	$10,000\,g$ (unpowered or powered)

on specific application requirements [13].

For further details, refer to the ADXL335 datasheet [13].

3.2.2 Arduino Uno

The Arduino UNO represents one of the most accessible and widely adopted microcontroller platforms in the maker community. Built around the reliable ATmega328P architecture, this development board strikes an optimal balance between capability and user-friendliness, serving equally well as an educational tool for novices and a rapid prototyping solution for experienced engineers. Its plug-and-play design eliminates complex setup procedures - users can immediately begin development by connecting via standard USB or utilizing various power input methods including the convenient barrel jack for 9V battery operation.

The specifications of Arduino are listed bellow [28]:

- **ATmega328P Microcontroller Unit** – Serving as the computational core, this component delivers:
  - Efficient 8-bit AVR RISC processing at 16 MHz clock speed
  - Substantial 32 kilobyte Flash memory for program storage
  - 2 kilobytes of volatile SRAM for dynamic memory operations
  - 1 kilobyte EEPROM for persistent data storage across power cycles

- **System Reliability Features:**
  - Automatic Power-On Reset (POR) circuitry for stable initialization
  - Brown-Out Detection (BOD) protection against voltage fluctuations
- **Integrated Hardware Peripherals:**
  - Dual 8-bit and single 16-bit hardware timers with advanced control capabilities
  - Full-duplex USART interface with sophisticated clock generation
  - Flexible SPI and I2C serial communication controllers
  - Precision analog comparator with configurable reference
  - System watchdog for fail-safe operation
  - Six-channel PWM generation for analog signal emulation
  - Pin-change interrupt system for efficient event handling
  - Seamless bridge between USB host and USART peripheral
  - Automatic protocol conversion (USB packets $\leftrightarrow$ serial frames)
  - Built-in drivers for major operating systems
- **USB Interface Controller (ATmega16U2):**
  - Dedicated 8-bit processor managing USB protocol
  - 16 kilobyte reprogrammable Flash memory
  - 512 bytes each of EEPROM and SRAM
  - On-chip debugging through debugWIRE interface
- **Input/Output Configuration:**
  - Fourteen programmable digital I/O lines (with six supporting PWM)
  - Six-channel 10-bit analog-to-digital conversion
  - Standard USB Type-B connector for programming and power
  - In-circuit serial programming (ICSP) header
  - Dedicated hardware reset circuit

- **Power Management System:**
  - Wide operating voltage range (2.7-5.5V DC)
  - Multiple power input options including regulated 5V USB power

A concise overview of the available pins is given in Table 3.3. For completeness, the full pinout information is provided in the appendix in Tables .1 and .2, which list all analog, digital, and power connections in detail. These complement the schematic representation shown in Figure 3.2.

Table 3.3: Summary of Arduino Pin Functions

Category	**Typical Pins**	**Description**
Power Supply	+3.3V, +5V, GND, VIN	Provides operating voltages and ground
Analog Inputs	A0–A5	6 analog channels, some also used for I^2C
Digital I/O	D0–D13	General-purpose digital pins, PWM on some
Communication	SDA, SCL, MOSI, MISO, SCK, SS	Interfaces for I^2C and SPI
Special	Reset, IOREF, AREF	Reset, voltage reference, and analog reference

An exquisite trait of Arduino boards, is that these devices boast a built-in set up of pull-up and pull-down resistors, which is extremely helpful when it comes to implementing projects making use of some sensors.

Moreover, the Arduino UNO cleverly bridges modern USB communication with traditional serial (UART) communication through a two-chip design. The main ATmega328P microcontroller handles the actual programming and serial communication using simple UART protocol, while a secondary ATmega16U2 chip acts as a translator, converting between the computer's USB signals and the microcontroller's UART signals. This translation happens automatically in both directions - when you upload code, the USB chip converts your computer's USB commands into UART signals the main chip understands, and when the Arduino sends serial data back, the USB chip converts it back to USB for your computer. The system creates a virtual COM port on your computer, making the Arduino appear as a simple

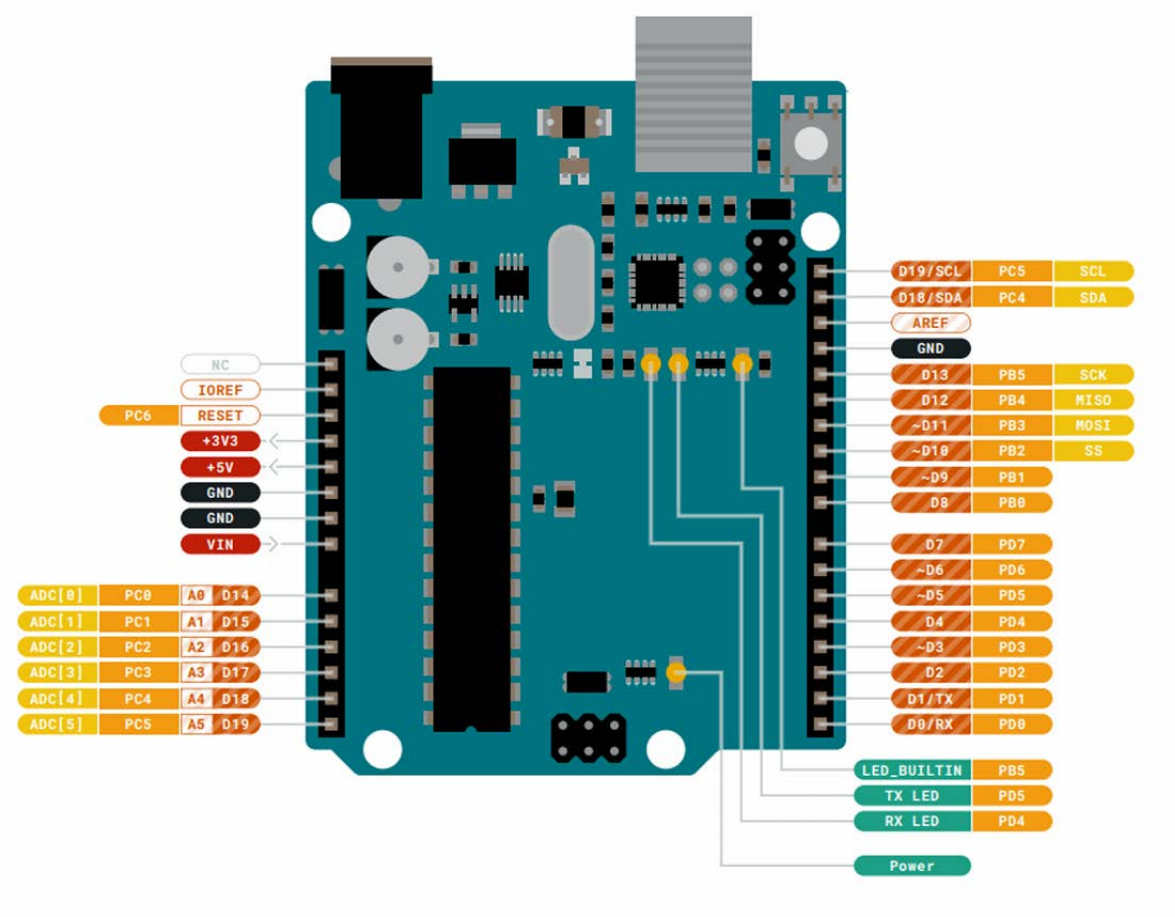

Figure 3.2: Arduino schematic pinout (refer to Tables .1 and .2 for details).

serial device even though it's actually communicating via USB. This elegant solution provides plug-and-play USB connectivity while maintaining compatibility with traditional serial communication, all without requiring any additional components or complex setup from the user. The dual-chip approach also keeps the main processor free from USB protocol handling, allowing it to focus on running your programs efficiently.

The Arduino UNO's serial communication typically operates at standard baud rates ranging from 300 to 115200 bits per second, with 9600 baud being the most commonly used default speed for basic communication. The USB-UART bridge supports higher speeds up to 2 Mbps in theory, but in practice, most applications reliably use 115200 baud or lower. The actual stable maximum baud rate depends on several factors, including the clock accuracy (the UNO's 16MHz crystal provides $\pm 0.3\%$ stability), the quality of USB connections, and the timing requirements of the specific application. For most beginner projects and serial monitoring, 9600 or 115200 baud provide the best balance of speed and reliability, while advanced applications can push higher speeds if properly implemented. The baud rate must match on

both the Arduino and the receiving device for successful communication.

These is also a tremendous amount of source code that is adaptable to different projects. All this sketches can be easily downloaded, stored and modified (or just even tweak some parameters) so as to fit it in each project.

All in all, the UNO is the most popular and well-documented board in the Arduino ecosystem, making it perfect for learning electronics and coding. Its processor ATmega328P chip, in most of the cases is not welded and this allows easy replacement procedure. In addition Arduino UNO is robust, flexible, and beginner-friendly platform for a wide variety of projects. Most importantly it supports easy and robust communication between Raspberry Pi and Arduino.

For further details, refer to the Arduino UNO R3 datasheet [28].

3.2.3 Raspberry Pi 4

The Raspberry Pi is a series of credit-card sized single-board computers developed by the Raspberry Pi Foundation. Designed originally for education, these low-cost computers have evolved into versatile platforms for computing and digital making [8].

There is a plethora of current Raspberry Pi boards (e.g. Raspberry Pi 4 and Raspberry Pi 5). They are all extremely powerful and can cover the needs of highly demanding projects, in terms of memory and computational power [8]. The Raspberry Pi's complete technical specifications are systematically organized in Tables 3.4, 3.5, and 3.6, yielding detailed hardware and software characteristics.

Table 3.4: Key Specifications of Raspberry Pi

Component	**Specification**
Processor	Broadcom BCM2712 (Pi 5) or BCM2711 (Pi 4) ARM Cortex
Memory	4GB LPDDR4X RAM
Storage	MicroSD card (64GB recommended)
Power Supply	5V USB-C (3A recommended)
Connectivity	Dual-band 802.11ac Wi-Fi, Gigabit Ethernet
Video Output	Dual micro HDMI (up to 4K resolution)
GPIO	40-pin header

Table 3.5: Raspberry Pi OS Features

Feature	**Description**
OS Type	Debian-based Linux distribution
Desktop	Complete desktop environment
Development	Python tools pre-installed
Hardware Access	GPIO control libraries

Table 3.6: Supported Communication Protocols

Protocol	**Description**
Digital I/O	General purpose input/output pins
SPI (Serial Peripheral Interface)	High-speed serial communication
I^2C (Inter-Integrated Circuit)	Two-wire serial bus
UART	Asynchronous serial communication

A clear representation of the Raspberry Pi 4 hardware is depicted on the Figure 3.3. Raspberry Pi boasts a wide variety of input and output pins, resembling the capacity of Arduino. It is also programmable, having a source code back-up for multiple hardware options (sensors). The most frequently used coding language for this context is PYTHON.

Table 3.7 details the 40-pin GPIO header configuration, which maintains backward compatibility with previous Raspberry Pi models. The header provides multiple voltage rails (3.3V and 5V) along with dedicated ground pins. The GPIO header serves as the primary interface for hardware projects, supporting everything from simple LED controls to complex sensor networks and custom communication protocols.

The Raspberry Pi 4 Model B offers comprehensive connectivity options as detailed in Table 3.8. The board features dual display outputs through micro-HDMI ports capable of driving 4K resolution simultaneously. For peripheral connectivity, it provides a balanced mix of USB 3.0 and 2.0 ports, with the USB-C power input supporting higher current requirements. The onboard wireless interfaces deliver dual-band Wi-Fi and Bluetooth 5.0 for modern wireless applications.

For further details, refer to the Raspberry Pi 4 Model B datasheet [8].

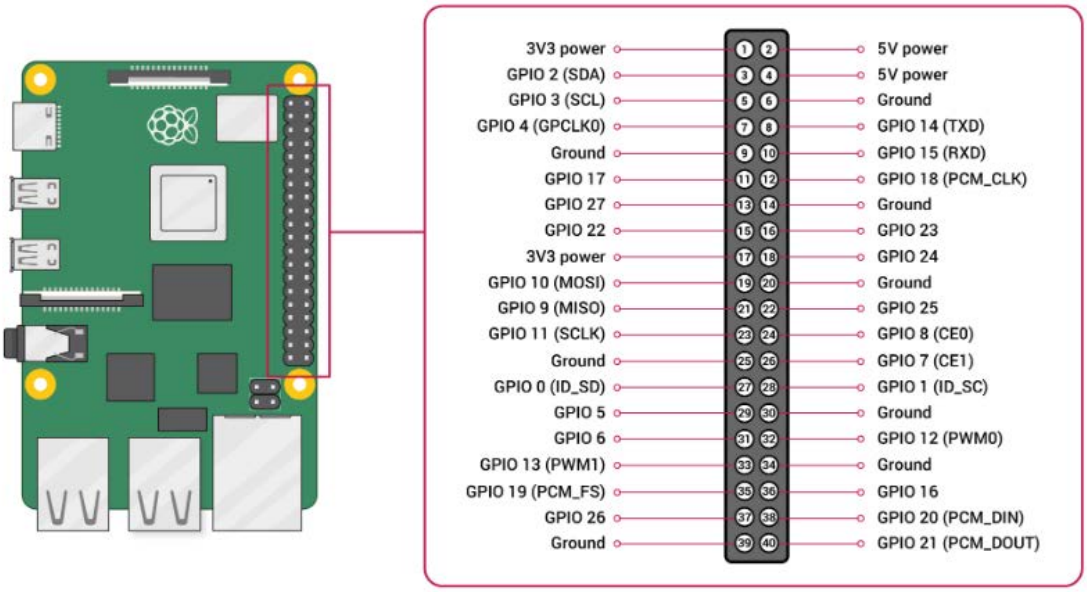

Figure 3.3: 40-PIN GPIO of Raspberry Pi 4 (source [8]).

Table 3.7: GPIO Pinout Details (An extended version of this table is on Appendix - extended version of the following table is the Table .3 (see Chapter 6).

Pin #	Function	Alt Function	Voltage	Notes
1	3.3V	-	3.3V	50mA max
2	5V	-	5V	-
3	GPIO2	SDA1	3.3V	I²C
4	5V	-	5V	-
5	GPIO3	SCL1	3.3V	I²C
6	GND	-	-	-
7	GPIO4	-	3.3V	-
...	...	...	...	...
37	GPIO26	-	3.3V	-
38	GPIO20	MOSI1	3.3V	SPI
39	GND	-	-	-
40	GPIO21	MISO1	3.3V	SPI

Table 3.8: Raspberry Pi 4 Model B Ports and Connectors

Port Name	Type	Quantity	Specifications
Primary External Ports			
USB 3.0	Type-A	2	5Gbps, BC1.2 charging support
USB 2.0	Type-A	2	480Mbps
Ethernet	RJ45	1	Gigabit (Realtek RTL8111G)
HDMI	micro-HDMI	2	HDMI 2.0 (up to 4K@60Hz)
Power	USB-C	1	5V/3A (minimum)
Audio	3.5mm jack	1	Stereo output + composite video
GPIO Header			
GPIO	40-pin	1	2×20 pin header, 0.1” spacing
Other Connectors			
CSI	15-pin FPC	1	Camera Serial Interface
DSI	15-pin FPC	1	Display Serial Interface
MicroSD	Slot	1	UHS-I compatible
PoE	4-pin header	1	802.3af support (requires HAT)
Wireless Interfaces			
Wi-Fi	Antenna	1	2.4GHz/5GHz 802.11ac
Bluetooth	Antenna	1	BT 5.0, BLE

3.2.4 The whole set-up

Following the presentation of the different ports of both the Raspberry Pi 4 and the Arduino UNO R3, it is important to also describe the interconnection between these two devices and the way in which data flows through the system. The accelerometer module (Adafruit ADXL335 breakout) generates analog voltage signals corresponding to the acceleration along the three axes (X, Y, and Z). These signals are connected to the analog input pins of the Arduino UNO, which then uses its built-in Analog-to-Digital Converter (ADC) to digitize the incoming data.

After conversion, the Arduino transmits the measurements via serial communication to the Raspberry Pi 4. On the Raspberry Pi side, a Python script is used to set up the serial connection, send requests for new measurements, and receive the acceleration values.

Specifically, the Raspberry Pi sends a command string to the Arduino, which responds with a comma-separated line containing the three acceleration components. The incoming data stream is subsequently retrieved as a text line, converted into a human-readable string, and then parsed into numerical values corresponding to the three acceleration axes. . Finally the data is split into three floating-point numbers representing the X, Y, and Z axes.

The baud rate of the serial communication, which must be pre-set identically on both the Arduino and the Raspberry Pi, plays a crucial role in ensuring reliable data transfer. The baud rate defines the speed at which bits are transmitted per second, and therefore determines how much information can be sent within a given time window. If the chosen baud rate is too low compared to the frequency and volume of data produced by the Arduino, the serial buffer may overflow, leading to delays or data loss. On the other hand, selecting a sufficiently high baud rate allows the system to transmit each line of acceleration data quickly enough to keep up with the sensor output, ensuring that all measurements are captured in real time. For example, at 9600 bps only around 960 bytes per second can be transmitted, which may not be adequate for high-frequency sampling, whereas at 115200 bps more than 11,000 bytes per second can be handled, providing ample capacity for continuous vibration monitoring. In this case, both devices can work well with a baud rate at 115200 bps.

In summary, the flow of information can be described as follows: the ADXL335 breakout produces analog signals sending them to the Arduino UNO samples and digitizes them. Afterwards the data is transmitted through the serial interface to the Raspberry Pi which in turn requests, receives, and parses the acceleration values for further storage and analysis. This interaction ensures a continuous and reliable stream of vibration data, which is schematically represented in Figure 3.4.

3.2.5 Network Infrastructure Specifications

The network infrastructure was configured to ensure reliable and secure data transmission from the sensor system to the cloud database. The router configuration provided optimal performance for IoT data transmission with excellent signal strength and enterprise-grade security protocols, supporting the structured src layout organization of the project.

The data transmission pathway followed a secure flow: *HW Setup → Local Router (192.168.2.1) → Internet → Supabase API (HTTPS)*. All communications used HTTPS with TLS 1.3 encryption and API key authentication, consistent with the security practices implemented

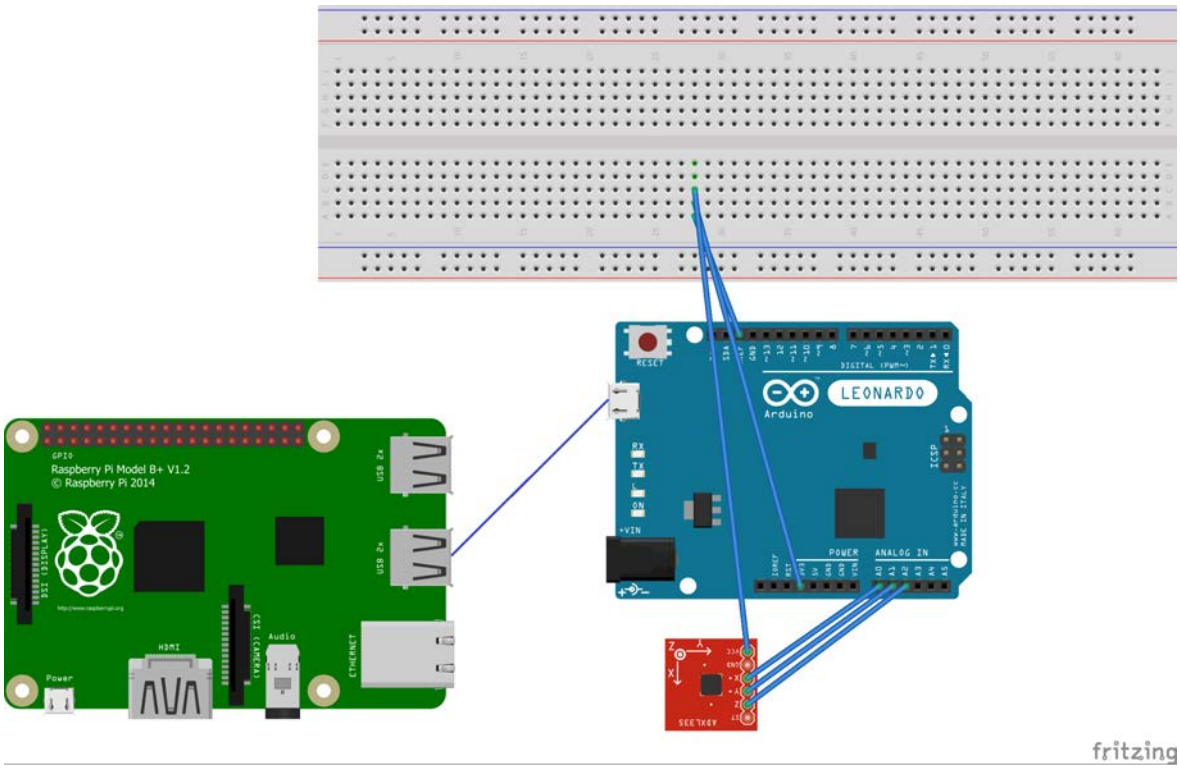

Figure 3.4: Hardware architecture with signal flow.

through environment variables and Poetry dependency management. The system transmitted data in JSON format via HTTPS POST requests with a maximum payload size of 1 MB at 30-second intervals, incorporating timestamps, sensor IDs, and checksums for data integrity verification, following the structured payload schema defined in the API implementation.

3.3 System Requirements

3.3.1 Functional Requirements

In addition to the aforementioned design traits of the system as a whole, there is also some design requirements related to the measuring limits of the sensor and communication among different components.

To begin with, the most commonly rotating speed of machinery goes up to 1500 [rpm] corresponds to 25 [Hz]. This relationship is defined by the following equation:

$$f\,[\mathrm{Hz}] = \frac{r\,[\mathrm{RPM}]}{60} \tag{3.1}$$

The aforementioned relationship entails that the rotational speed of a machine can be

interpreted into frequency, which in turn is of great help when related to Fourier Transform applications.

All in all, the functional requirements are highlighted hereunder:

- **Frequency Range**: 0–300 Hz (covers common machine faults like imbalance, misalignment).
- **Real-Time Processing**: On-the-fly FFT computation for immediate fault detection.
- **Scalability**: Modular design for multi-sensor deployments.

3.3.2 Non-Functional Requirements

- **Cost**: Minimize BOM cost (justifies Arduino + RPi4 over industrial PLCs).
- **Power**: Optimize for continuous operation (e.g., no active cooling).
- **Accuracy**: ADXL335's ±3g range suffices for industrial vibrations (cite datasheet).

3.3.3 Network Requirements

The network requirements for the IoT sensor system were defined to ensure secure, reliable, and efficient transmission of data to the Supabase cloud database. The design considered the characteristics of the sensor payload, network infrastructure, and API constraints to optimize performance and maintain data integrity. Each data batch consisted of approximately 12,847 measurements in JSON format, with payload sizes ranging from 50 KB to 200 KB per transmission.

The system used a local router at IP 192.168.2.1 with MAC address 94:e3:ee:f0:20:bc, connected via a 2.4 GHz Wi-Fi band on channel 11. The Wi-Fi network utilized WPA2-Personal (AES/CCMP) encryption to secure local traffic.

With a 10 Mbps internet connection (1.25 MB/s), the network had sufficient bandwidth to handle sensor data uploads reliably. The Wi-Fi link speed (16.25 MB/s) ensured that local transmissions were fast and would not create a bottleneck for the payloads.

To ensure reliability and avoid Supabase API rate limits, each POST request was kept under 1 MB. Batching of multiple measurements per request optimized network utilization while preserving data accuracy. High signal strength and low packet loss guaranteed consistent transmission. Each payload included timestamps, sensor IDs, and optional checksums

for integrity verification. Retry mechanisms and local buffering were implemented to recover from temporary network outages, ensuring zero data loss.

All data transmissions used HTTPS with TLS 1.3 encryption, and API key authentication was enforced. Local traffic was protected by WPA2. Optional signing or checksum verification of payloads further enhanced security against tampering.

The network infrastructure and transmission logic were designed to accommodate additional sensors, higher sampling rates, or larger payloads without compromising performance, reliability, or security. Overall, the combination of strong Wi-Fi signal, robust security protocols, and careful payload management ensured that the network could reliably support continuous, secure, and accurate transmission of IoT data to the Supabase cloud database.

Chapter 4

Implementation

4.1 Implementation Introduction

The implementation of the aforementioned system concept requires a study of the fundamental limitations of its individual components.

A primary potential bottleneck is the hardware serial transmit buffer of the Arduino Uno. For this project, the maximum capacity of the UART interconnection between the Arduino Uno and the Raspberry Pi 4 is a critical design parameter (the connection setup is detailed in Chapter 3). The Arduino's operation is defined by a batch-processing strategy: it must first read and store data from the ADXL335 accelerometer in its internal memory. Only after a predetermined amount of data is stored is the entire dataset transmitted to the Raspberry Pi 4. Figures 3.1 and 3.4 provide a clear visualization of this dataflow.

Subsequently, the Raspberry Pi 4 receives, aggregates, and processes this dataset. Its computational role is to execute a Fast Fourier Transform (FFT) and calculate the signal magnitude for analysis, translating the real-world time-domain data into the frequency domain (see Chapter 2). And the processed results are sent to a database for storage and further use.

Following the frequency-domain analysis, the Raspberry Pi 4 executes the final stage of the data pipeline by transmitting the processed results to a cloud-based PostgreSQL database hosted on Supabase. This is achieved by leveraging Supabase's auto-generated RESTful API, which provides a secure and programmatic interface for data operations. The Pi utilizes HTTPS requests, specifically authenticated POST calls, to insert structured JSON payloads containing the computed FFT magnitudes and frequency bins directly into the designated tables within the Supabase schema. This serverless architecture eliminates the need for man-

aging database infrastructure, while the built-in row-level security (RLS) ensures that data access is strictly governed by predefined policies. By interfacing with the Supabase API, the system ensures that analyzed sensor data is persistently stored in a scalable and readily queryable data store, facilitating further visualization, historical analysis, or integration with other applications within the Supabase ecosystem.

A flowchart that illustrates a brief dataflow is depicted on Figure 4.1.

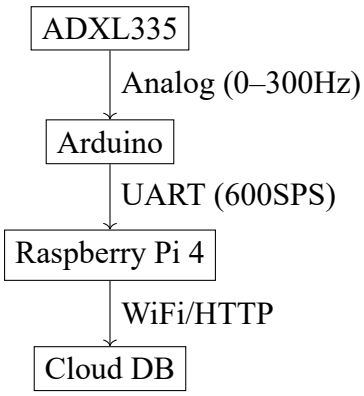

Figure 4.1: End-to-end data flow with protocols

More technical details on the aforementioned topics are briefly analyzed within the present chapter.

4.2 Software Architecture and Data Flow

4.2.1 Dataflow Block Diagram

Figure 4.2 illustrates a layout of the algorithms in the following sections.

Arduino UNO collects data and sends data in data batches to the serial port, where data in bit format are stored in the memory of Raspberry Pi and are first converted to g-units and then transformed in the frequency domain. Following this step, data is stored to the Supabase and then presented on a webpage upon request.

The terminal recipient of the collected readings is the React application of Vercel that illustrates the collected data. There is a threshold value at 1[g] magnitude of the frequency spectrum that works as a decisive factor for identifying the failure of some mechanical components.

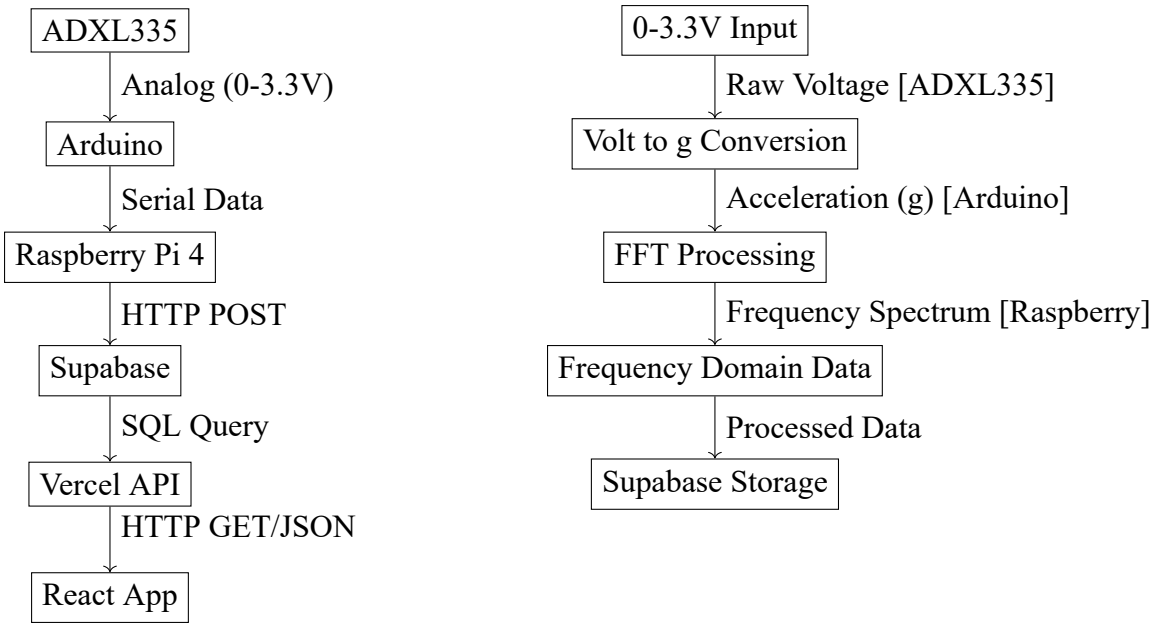

Figure 4.2: Condition monitoring system dataflow.

4.2.2 The Arduino UNO Algorithm

The main focus of the Arduino UNO sketch is to implement a batched data processing strategy that enables sequential transmission of sensor data to the Raspberry Pi 4. This design methodology specifically addresses the memory constraints of the Arduino UNO by minimizing the instantaneous SRAM utilization through carefully sized data batching.

The Arduino UNO receives analog signals from the sensor breakout board through its integrated analog-to-digital converter (ADC), which features 10-bit resolution across six input channels. Each designated analog input pin (A0, A1, A2) continuously monitors voltage levels in the range of 0,V to 5,V, corresponding to the physical phenomena being measured by the sensors, which entails that the reading can be interpreted as g's or mm/s^2. The ADC performs successive approximation conversion at a sampling rate of approximately 10,000 samples per second, transforming continuous analog voltages into discrete digital values ranging from 0 (representing 0V) to 1023 (representing 5V). Each converted value is stored as a 16-bit signed integer (2 bytes) in memory, maintaining compatibility with the Arduino's architecture while preserving measurement precision.

The data acquisition process follows a structured double-buffering approach where readings are temporarily stored in a two-dimensional array before processing and transmission. This architecture ensures temporal consistency between related sensor measurements while optimizing memory utilization through fixed-size allocation.

Algorithm 1 Snippet of Data Acquisition & Transmission (see the whole code on Appendix Section 2.1)

Require: $numChannels = 3$, $analogPins = [A0, A1, A2]$, $bufferSize = 10$

Ensure: Serial transmission of analog data

Initialize $analogBuffer[numChannels][bufferSize]$, $tempVal \leftarrow 0$
while true **do**
  for $j = 0$ to $bufferSize - 1$ **do**
    for $i = 0$ to $numChannels - 1$ **do**
      $analogBuffer[i][j] \leftarrow$ analogRead$(analogPins[i])$
    end for
  end for
  if $tempVal = 100$ **then**
    $tempVal \leftarrow 0$ {Calibration}
  else
    for $j = 0$ to $bufferSize - 1$ **do**
      for $i = 0$ to $numChannels - 1$ **do**
        Serial.print($analogBuffer[i][j]$)
        if $i < numChannels - 1$ **then**
          Serial.print(" , ")
        end if
      end for
      Serial.println()
    end for
    $tempVal \leftarrow tempVal + 1$
  end if
end while

A snippet of the Arduino UNO algorithm implementation follows hereunder (Algorithm 1):

The memory requirement in this case can be estimated , combining all received data as analog signal by the sensor breakout:

$$\text{Total Memory} = (\text{numChannels} \times \text{bufferSize} \times \text{sizeof(int)}) + \text{Overhead} = 3 \times 10 \times 2 = 60$$

The core data storage therefore occupies exactly 60 bytes for sensor readings. This buffer size was determined through extensive empirical testing and validation using methodical trial-and-error optimization. Increasing the buffer size beyond this optimal value consistently resulted in system instability and memory allocation failures, while smaller buffers reduced data throughput efficiency.

Although 60 bytes appears minimal compared to the Arduino UNO's 2048 bytes of SRAM and 32 KB of flash memory (as extensively discussed in Chapter 3), several critical memory overhead factors must be considered. The hardware serial communication infrastructure requires 128 bytes of dedicated SRAM space, comprising dual 64-byte circular buffers for transmit (TX) and receive (RX) operations. Additional memory is consumed by program stack operations, interrupt service routines, and the Arduino core libraries, which typically utilize 200-300 bytes during normal operation.

Furthermore, the script incorporates several control variables and system parameters that contribute to cumulative memory consumption. These include loop counters, temporary variables, and the program instruction pointer, which collectively add approximately 20-30 bytes of overhead. This memory fragmentation, combined with the inherent memory management characteristics of the AVR architecture, creates a persistent risk of stack-heap collisions and memory exhaustion scenarios. The implementation therefore employs conservative memory practices, including static allocation and avoidance of dynamic memory operations, to mitigate these risks and ensure system reliability during extended operation periods.

The chosen architecture represents an optimal balance between data throughput requirements and memory constraints, ensuring reliable operation while maximizing the available resources for future expansions or additional functionality.

The FFT is applied to the acceleration data collected from the Arduino to analyze its frequency content. The data from all three axes (X, Y, Z) and the combined amplitude are transformed from the time domain to the frequency domain. The actual sampling rate F_s is calculated from the timestamps of the collected data, and the FFT is performed using a configurable number of points L. The result is a single-sided amplitude spectrum for each signal. A larger F_s increases the maximum measurable frequency, while a larger L improves frequency resolution. The detailed steps of this process are summarized in Algorithm 2.

Algorithm 2 FFT Processing of Arduino Sensor Data

Require: Time-domain sensor data for X, Y, Z axes and combined amplitude, measurement timestamps, FFT length L

Ensure: Single-sided amplitude spectrum $P1$ and frequency vector f for each signal

Compute differences between successive timestamps
Calculate sampling rate: $F_s \leftarrow 1/\text{average(time differences)}$
Choose number of FFT points L
for each signal in [Amplitude, X, Y, Z] **do**
    Compute FFT of the signal with length L
    Compute magnitude and normalize by L
    Keep first half of the FFT (single-sided spectrum)
    Multiply all values except first and last by 2 to conserve energy
    Generate frequency vector: $f \leftarrow F_s/L\times$ bin indices
end for

4.2.3 The Raspberry Pi 4 Algorithm

In contrast with Arduino UNO, the Raspberry Pi 4 would require an external Analog to Digital converter so as to receive data from ADXL335. For this reason Arduino UNO is mandatory for the needs of this project.

While the Arduino UNO sketches seem to be quite straight forward and the only difficulty that had to be overcome was its memory limitations, the Raspberry Pi 4 algorithms are far more sophisticated.

An efficient way to organise the files of a project is the ”src layout” [29]. A well-structured project layout, such as the src-based approach with clearly separated folders for source code, tests, configs, and scripts, is highly effective because it enforces organization and scalability as a project grows. By isolating source code under src/, you avoid common import conflicts and ensure your code behaves the same way in both development and production.

Configuration files and environment variables are separated from the core logic, which improves maintainability and allows developers to adapt the project across environments (local, staging, production) without modifying source code. This separation of concerns makes collaboration easier, onboarding smoother, and scaling more manageable as features and complexity increase.

Within this setup, Poetry and .env files play crucial roles. Poetry simplifies dependency management by consolidating everything into a single pyproject.toml, handling versioning, virtual environments, and packaging in a reproducible way. This ensures consistency across machines and deployments, preventing "works on my machine" problems. On the other hand, .env files keep sensitive information (like API keys, database credentials, and tokens) out of the codebase, promoting security and flexibility. Paired with a .env.example, they also document required environment variables without leaking secrets. Together, Poetry and .env form the backbone of a professional, maintainable, and secure Python project setup.

An overview of the core processing workflow implemented in the project is presented in Algorithm 3, highlighting the principal computational processes and methodological practices employed. This snippet captures the essential data acquisition, processing, and transmission pipeline that forms the foundation of the vibration monitoring system.

For a comprehensive examination of the complete project architecture, readers are directed to the Appendix section. Specifically, Section 2.2 provides detailed algorithmic descriptions of all constituent scripts, including their functional specifications, inter-module communication protocols, and implementation details. The appendix offers complete visibility into the system's modular design, encompassing data acquisition, signal processing, API communication, and utility functions that collectively enable the real-time vibration monitoring capabilities.

The srclayout project structure looks like this:

```
vibration-monitoring-project/
├── .venv/                  # Python virtual environment
├── config/
│   └── config.yaml         # Application configuration
├── sourceCode/
│   ├── main.py             # Main application
│   ├── api.py              # API communication
│   ├── data_processing.py # Signal processing
│   ├── serial_communication.py # Serial communication
│   └── utils.py            # Utility functions
├── testing/
│   └── test_methods.py     # Test cases
```

```
├── poetry.lock            # Dependency lock
└── .env                   # Environment variables
```

Algorithm 3 Raspberry Pi Data Processing Pipeline

Require: Serial port $port$, baud rate $baud\_rate$

Ensure: Processed vibration analysis results

1: $analogReference(EXTERNAL)$
2: Initialize serial connection: $ser \leftarrow setup\_serial\_connection(port, baud\_rate)$
3: $history \leftarrow [\,]$, $history_X \leftarrow [\,]$, $history_Y \leftarrow [\,]$, $history_Z \leftarrow [\,]$
4: $start\_time \leftarrow current\_time()$, $time\_window \leftarrow config['time\_window']$
5: **while** $current\_time() - start\_time < time\_window$ **do**
6: $timestamp \leftarrow current\_time()$
7: $x, y, z \leftarrow read\_serial\_data(ser)$
8: $amplitude \leftarrow calculate\_amplitude(x, y, z)$
9: Convert the 10-bit ADC value to voltage, then get acceleration in g.
10: Append values to $history$, $history_X$, $history_Y$, $history_Z$
11: **end while**
12: $Fs \leftarrow calculate\_sampling\_rate(timestamps)$
13: $L \leftarrow Desiredlength$ {FFT points}
14: **for** each $dataset$ in $[history, history_X, history_Y, history_Z]$ **do**
15: $spectrum, frequencies \leftarrow compute\_fft(dataset, Fs, L)$
16: $result \leftarrow detect\_unbalancing(frequencies, rpm = 1800, spectrum, threshold = 0.1)$
17: Send results to API: $api.send\_measurement(...)$
18: **end for**
19: **return** Processing complete

The serial communication module, as previously highlighted in the system architecture, establishes a connection with the Arduino microcontroller at a specified baud rate, initializing the serial port with a timeout parameter to prevent blocking operations.

Specifically, the data received by Arduino UNO are getting analyzed. The sensors yields the 3-axis readings (X, Y and Z), when the whole dataset, for a predetermined time-window

is received by Raspberry Pi, the amplitude of the 3-axis readings is calculated and afterwards frequency domain data are extracted.

Upon initialization, the Raspberry Pi waits until the Arduino Uno shares the sensor readings. The received data string, containing three comma-separated acceleration values from the sensor's X, Y, and Z axes, is decoded from UTF-8 format, stripped of extraneous characters, and parsed into floating-point numerical values representing the triaxial acceleration measurements in g-forces, which are subsequently returned for further processing in the vibration monitoring pipeline, consistent with the data flow described in earlier sections.

When the ADXL335 is powered with a 3.3 V supply, each axis outputs an analog voltage centered at about $1.65\,\mathrm{V}$ when experiencing $0\,g$ (no acceleration). The sensitivity in this case is approximately $330\,\mathrm{mV}/g$, meaning the output shifts by $0.33\,\mathrm{V}$ for every $\pm 1\,g$ of acceleration along that axis. To read this with a 10-bit ADC (0–1023 counts at 5 V reference), you first convert the raw ADC value to voltage as

$$V = \text{raw} \times \frac{3.3}{1023.0}.$$

Then, subtract the zero-g bias ($1.65\,\mathrm{V}$) (which corresponds to the half of the 3.3 V - the supply of the ADXL335 sensor) and divide by the sensitivity (0.330) to convert to g-units:

$$a_g = \frac{V - 1.65}{0.330}.$$

This procedure should be repeated for the X, Y, and Z axes, after which you may optionally compute the vector magnitude to determine total acceleration:

$$|a| = \sqrt{a_x^2 + a_y^2 + a_z^2}.$$

It can be noticed here that the breakout improves the data acquisition procedure and conveys better and more accurate value. there is a benefit of utilizing the 3.3V, because we can make the most of the whole 0-3.3V range of the analog read pins in this set-up. In case the 10bit values have a reference voltage at 5 V , which entails that the Arduino will never receive 10bit values greater than 676. This leads to a sensitivity decrease.

After this the Fast-Fourier-Transform is taking place. When performing an FFT on accelerometer data, the constant component caused by sensor orientation appears only at 0 Hz (the DC component). By ignoring this 0 Hz peak, the FFT reflects solely the dynamic vibrations of the system, meaning the amplitudes at non-zero frequencies correspond to the actual

mechanical motion rather than the static effect of gravity. As a result, even if the time-domain signal has a significant constant offset due to tilt, the vibration magnitudes extracted from the frequency domain remain small and accurately represent only the true oscillatory behavior of the component.

Moreover, the API communication for transmitting vibration data to the Supabase backend is implemented through a structured workflow that begins with authentication and configuration validation. Upon class instantiation, the system employs the requests.post() method to transmit JSON-formatted vibration data to the Supabase REST API endpoint, ensuring secure communication through properly configured HTTP headers containing authentication tokens and content type specifications.

The requests.post() command functions as the core mechanism for data transmission, operating as an HTTP client that serializes the vibration data payload into JSON format and dispatches it via POST request to the designated Supabase endpoint. This method automatically handles connection management, timeout protocols, and response processing while incorporating the pre-configured headers containing the essential API key for authentication. The implementation includes robust error handling that captures potential exceptions such as connection failures, timeout errors, or invalid response formats, while also validating the HTTP status codes returned by the server to ensure successful data ingestion. The method returns a response object containing the server's status code and optional response data, which the system then processes through comprehensive logging mechanisms for both debugging and operational monitoring purposes.

For each data transmission request, the system first resolves the sensor identifier by querying the Supabase database using the sensor name, establishing a relational link between the measurement data and the specific sensor entity. The payload construction encapsulates critical metadata including the sensor ID, precise time intervals, measurement coordinate (X, Y, Z, or magnitude), and the actual FFT-processed values array. The HTTP POST request is executed with proper error handling and comprehensive logging, featuring response validation that checks for empty or malformed JSON responses while maintaining robust exception handling for network failures or authentication errors. The implementation includes diagnostic logging at multiple stages, providing visibility into the payload structure, response status codes, and JSON content for debugging and monitoring purposes, ensuring reliable data transmission to the cloud-based storage infrastructure.

In the frequency-domain power spectrum, a threshold of $1\,g$ is set as a warning line. Any spectral component exceeding this level may indicate abnormal vibrations or potential mechanical issues.

Moreover it is worth it mentioning that the development and testing environment spanned multiple hardware platforms, utilizing Poetry for consistent dependency management across all systems:

- **Raspberry Pi devices**: Model 4B (4GB RAM) running Raspberry Pi OS
- **Testing PCs**: Fedora 40 and Windows 11 systems

Poetry ensured identical dependency versions across all platforms through the `pyproject.toml` specification. Key packages maintained consistency included:

- **pyserial** v3.5: Uniform serial communication across platforms
- **numpy** v1.24.3: Consistent numerical computation
- **requests** v2.28.2: Standardized HTTP client functionality
- **python-dotenv** v1.0.0: Uniform environment variable handling

The cross-platform approach provided several advantages:

- Development flexibility across Windows and Linux environments
- Consistent behavior between development and deployment on Raspberry Pi
- Simplified testing across multiple operating systems
- Streamlined collaboration through version-locked dependencies

Testing covered all platform combinations to ensure compatibility and consistent performance across different devices. The environment successfully supported the complete development lifecycle from initial coding to final deployment on production hardware.

4.3 Database Set-up

As regards the database set-up, the Supabase database consists of the following tables:

- `customers`
- `sensors`
- `measurements`

Each customer can own several sensors that measure different points, i.e., different pieces of equipment. Each sensor has a name based on the machine and its function. All measurement data is stored in the main table `measurements`.

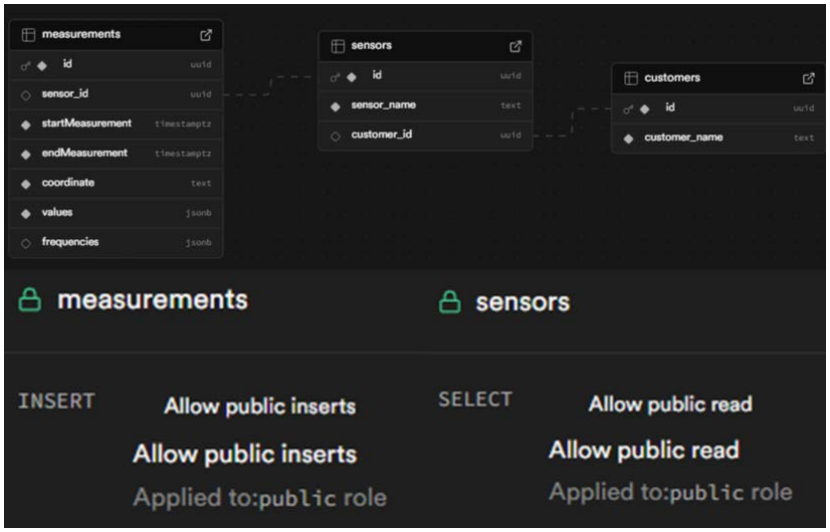

Figure 4.3: Entity-relationship schema of the Supabase database.

4.3.1 Vercel Implementation and Supabase Integration

After sending all data to Supabase, you can deploy this Next.Js-friendly application that visualizes your sensor measurements. The project structure is optimized for Next.js with TypeScript and includes all necessary configuration files. When you deploy this application to Vercel, you'll receive a unique URL that you can share with specific email addresses.

It is worth mentioning at this point that there is discrete data rows for each of the 3-axis. This entails that data stored independent from each other (see Section 4.3). For this reason, Vercel identifies the set of (X,Y,Z and their Magnitude) based on the instance of time (there is the startMeasurement and endMeasurement columns).

The aforementioned separate data are posted on different figures so as to distinct from each other and focus on the axis that deals with the more intense vibration.

Once the Next.js application is deployed on Vercel, the web page becomes accessible from any remote device using the assigned Vercel domain (e.g., `https://yourproject.vercel.app`). When a user opens this page, the frontend code hosted on Vercel executes in the browser and communicates directly with the Supabase API using the project URL and the public `anon` key. Supabase then applies the defined Row Level Security (RLS) policies to determine which data can be returned. As a result, users can view the permitted Supabase results on the web page from any location, making the system fully accessible online.

Vercel is an online platform for hosting web applications and can be used to create dynamic pages that fetch and display real-time data. In your project, Vercel hosts a Next.js dashboard that retrieves sensor measurements from a Supabase API. Using classic client-side rendering (CSR), your React component calls fetch('/api/measurements') on the client, awaits the data, and then uses it to create plots. The charts are rendered only after the data arrives, allowing users to see the most recent measurements dynamically without requiring a page rebuild.

The dashboard is designed to compare current sensor readings with a stored healthy baseline for each machinery coordinate (X, Y, Z, Magnitude). The earliest measurement for each coordinate is considered the healthy state, while the latest measurement is treated as the current state. The data is processed in the client by grouping measurements and combining the healthy and current values for each frequency point, producing arrays that are then used for plotting. This structure allows for a clear visual comparison of baseline versus current vibration data for each axis.

Each plot displays two lines on the same chart: one representing the healthy baseline and the other representing the current measurements. This enables users to immediately identify deviations or anomalies in the machinery's performance. The system can be enhanced with deviation highlighting, real-time updates via Supabase Realtime, or server-side preprocessing for large datasets. Overall, this approach creates a fully dynamic, interactive monitoring dashboard hosted on Vercel that provides immediate visual feedback on machinery health.

The access control can be manipulated by configuring Vercel's authentication settings to restrict the deployment URL to only specific email addresses, ensuring your sensor data remains private while being easily accessible to authorized viewers.

These packages enabled secure communication with the Supabase backend while maintaining optimal performance in the Vercel environment. The application structure was organized as follows:

```
my-nextjs-project/
├── .next/
├── node_modules/
├── public/
├── src/
│ └── app/
│ | └──api/
│ | | └── route.ts
│ | ├── fonts/
│ | ├── lib/
│ | | └── supabase.ts
│ | ├── favicon.ico
│ | ├── globals.css
│ | ├── layout.tsx
│ | └── page.tsx
├── .env.local
├── .eslintrc.json
├── tsconfig.json
├── package.json
├── package-lock.json
├── next.config.ts
├── next-env.d.ts
├── tailwind.config.ts
├── postcss.config.js
```

The most important algorithm that is being called is the page.tsx component, which serves as the main data visualization dashboard that fetches sensor measurements from the Supabase database through the API endpoint, processes the time-series data, and renders interactive

charts displaying X, Y, Z axis readings along with magnitude values using the Recharts library for comprehensive sensor data analysis and monitoring.

This algorithm implements the core functionality of different end points. It fetches, processes and renders charts. More details on this can be looked upon the algorithms of the Subsection 2.3, while a snippet yields some intuition into the aforementioned practice (see Algorithm 4).

Algorithm 4 Sensor Data Visualization Algorithm

Import React, Recharts components
Define Measurement type structure

Function Home():
    Initialize state for measurements and error
    Fetch data from API endpoint on component mount
    Handle errors and loading states
    Combine X, Y, Z coordinate data of reference and current state
    Render interactive charts using Recharts of X , Y , Z and their magnitude separately
End Function

API Function GET():
    Query Supabase database for measurements
    Return data as JSON response
    Handle database errors appropriately
End Function

The Vercel deployment platform was utilized to host a Next.js application that provides real-time visualization of sensor measurements. Next.js provides the /api directory, allowing you to write server-side code (like route.ts) alongside your React components. The implementation required several Supabase integration packages to ensure secure and efficient data handling:

- `@supabase/supabase-js`: The official JavaScript client for Supabase services (database, authentication, storage)

- `@supabase/ssr`: Server-side rendering support for Next.js applications (replacing the deprecated `@supabase/auth-helpers-nextjs`)

The React component serves as the primary user interface for accessing and interpreting the collected sensor measurements, forming the core of the data visualization pipeline. The page.tsx file defines both the UI structure and client-side logic, demonstrating the use of React for the frontend implementation.

This software runs in real-time keeping track of all measurements of the actual mechanism. It stores the measurements of the first run as a healthy-reference point and every other measurement will be compared to that healthy-state reference dataset. These historical data can be critical so as to locate the critical frequencies and any potential peaks magnitude increase.

In summary, the presentation layer—implemented as a React application—consumes data via a dedicated RESTful API endpoint deployed on Vercel. This endpoint is granted direct database access by handling authentication and attains data-fetching utilizing the Supabase backend. This architecture successfully facilitates the visualization of real-time sensor data on a dynamic webpage. The complete system data flow, from physical vibration to cloud-based visualization, is illustrated in Figure 4.4.

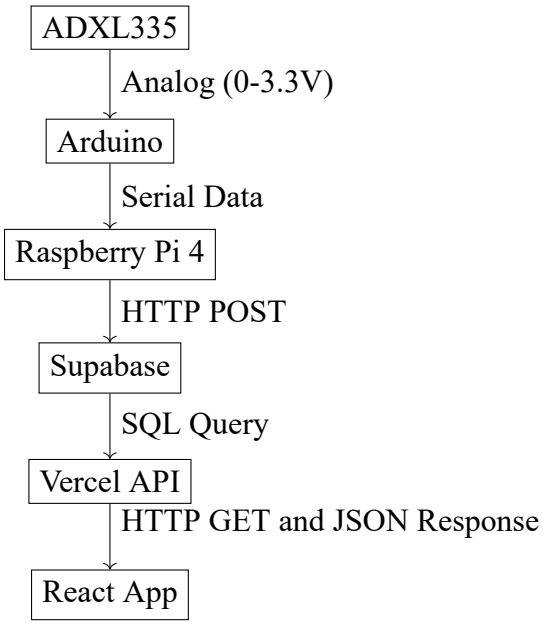

Figure 4.4: Complete system architecture and data flow from physical sensor to web visualization dashboard

Chapter 5

Results

5.1 Introduction

First and foremost, comprehensive testing protocols were implemented to validate system functionality and reliability across all components. Performance metrics were tracked to identify potential delays, data loss, or software malfunctions.

As previously discussed in Chapter 4, concerns regarding data receiving and transmission integrity were addressed. The Arduino UNO's hardware architecture imposes specific constraints on data handling capabilities, with its 64-byte serial buffers limiting maximum amount of data input to approximately 600 samples (considering the set-up and the implementation of this very project, which ensues that this is the estimated amount of data based on the dataset format) per second when transmitting three-axis acceleration data. This necessitated careful design of the communication protocol to prevent buffer overflows while maintaining real-time monitoring capabilities.

In addition, further investigation is carried out considering the Raspberry Pi data handling capacity, since the latter one has to receive, store, process and post data, which activity is far more demanding.

Last but not least, The data acquired by the React application are posted with an orange line as a machinery health status indication. If the magnitude peak at any frequency exceed this limit it is considered as an maintenance alarm. This concept has been implemented first in the testing phase in the python environment and later on it is used for the deployment and data is depicted using Vercel.

Finally, for testing and deployment, the Raspberry Pi 4 maintained a continuous connec-

tion to a host PC within the same local network. This configuration utilized the Raspbian lite operating system—a lightweight, Debian-based Linux distribution optimized for embedded systems—running on the Raspberry Pi. Communication between the host PC and Raspberry Pi was established through SSH (Secure Shell protocol), which provided secure remote access and full terminal control over the Raspberry Pi. This setup enabled real-time remote monitoring, debugging, and management of the data acquisition processes without requiring direct physical access to the embedded system.

5.2 Precommissioning System Verification and Validation

5.2.1 ADXL335 to Arduino Validation

In Figure 5.1, data are transmitter from ADXL335 to Arduino in 10 bit format (0 = 1023). The figure illustrates what is printed to the serial port, meaning what is expected to be sent to Raspberry Pi 4 via the USB port.

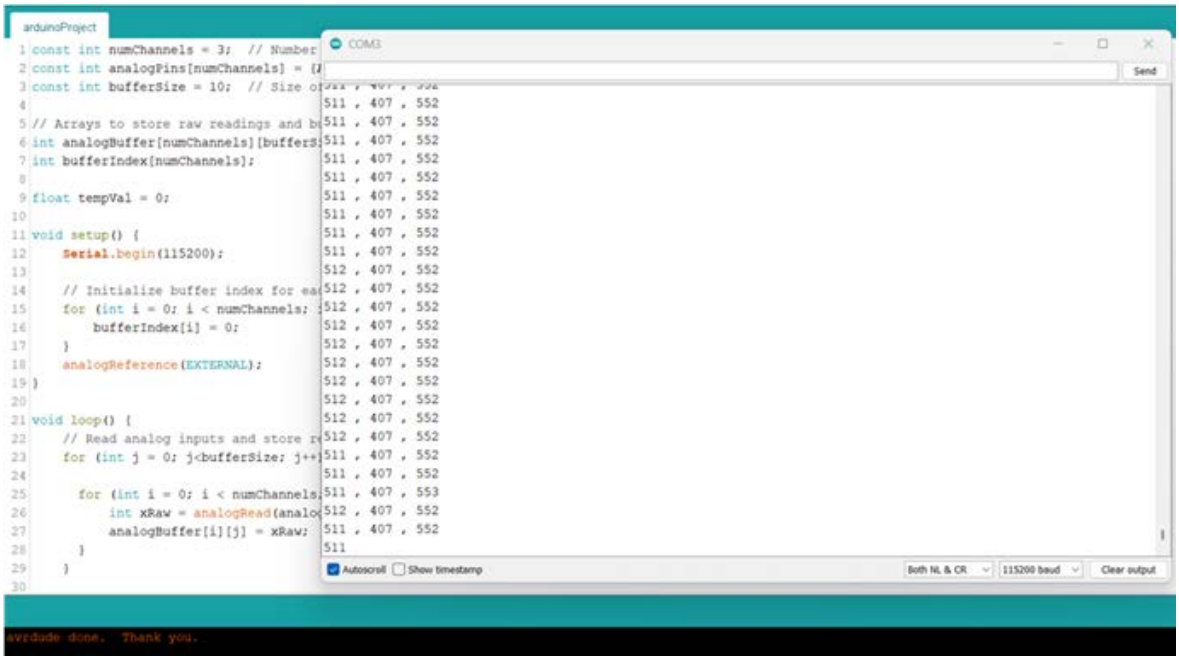

Figure 5.1: Transmitting system operational validation ADXL335 to Arduino.

The amount of data cannot be measured at this point using this sketch, but the data are formatted the way it has been explained in 4.

The Arduino reads three analog channels and sends the data to the Raspberry Pi via serial communication at 115,200 baud. Each reading is converted to a number (0–1023) and sepa-

rated by a newline. Here, the 115,200 baud rate seems to work well along with the ADXL335 sensor breakout. This conclusion is verified by the fact that the data have a normal looking format, without any missing data resembling the last row.

5.2.2 Arduino to Raspberry Pi Data Transmission Performance Metrics

The system was tested across all stages of data transmission, with careful consideration of the Arduino's operational limits. Sensor readings were first collected by the Arduino, which processed the raw signals from each axis and calculated the amplitude. The Arduino successfully managed the incoming sensor data without loss or timing issues, operating well within its 64-byte buffer constraints at the configured 115200 baud rate.

The Arduino's competency in receiving and transmitting data can be evaluated using Algorithm 7: "Testing Results" in the Appendix (see Section 2.2). This algorithm allows measuring the effective sampling performance of the Arduino under typical operating conditions.

The sampling rate of the Arduino reflects its ability to acquire data from sensors. As discussed in Chapter 4, memory and processing limitations make data handling complex. In executed experiments, the sampling rate F_s was calculated from the timestamps of acquired data by taking the average interval between successive samples and inverting it. This produced an average sampling rate of approximately 700 Hz, reflecting the Arduino's hardware capabilities (350 [Hz] is the expected amount of data since the spectra are mirrored for values greater than 350 [Hz]).

When performing a Fast Fourier Transform (FFT) on the acquired data, the frequency resolution depends on both the sampling rate F_s and the number of FFT points L:

$$\Delta f = \frac{F_s}{L}$$

The Arduino reads three analog channels and sends the data to the Raspberry Pi via serial communication at 115,200 baud. Each reading is converted to a number (0–1023) and sent as a line of text with comma separators and a newline.

With a buffer size of:

$$N_b = 10 \text{ [samples/buffer]},$$

Since there is approximately 18 bytes per sample line in total due to the data format, it is

estimated that a full buffer sends roughly:

$$B_{\text{send}} \approx 180 \text{ [bytes/sending]}.$$

At a baud rate of:

$$R_b = 115{,}200 \text{ [baud]} = 115{,}200 \text{ [bits/s]},$$

the serial connection can transmit about:

$$R_{\text{bytes}} = \frac{R_b}{10} = \frac{115{,}200}{10} = 11{,}520 \text{ [bytes/s]},$$

since each byte requires 10 bits for UART transmission.

This allows approximately:

$$f_{\text{buffers}} = \frac{R_{\text{bytes}}}{B_{\text{send}}} = \frac{11{,}520}{180} \approx 64 \text{ [buffer sends/s]}.$$

A larger sampling rate F_s increases the maximum measurable frequency, while a larger L improves the frequency resolution. The limitation that we encountered here is the upper limit of:

$$f_{\max} = \frac{F_s}{2} = \frac{700}{2} = 350 \text{ [Hz]}$$

in the frequency domain results, while the sampling rate is

$$F_s = 700 \text{ [Hz]}.$$

For instance, a measurement window of 30 seconds:

$$T = 30 \text{ [s]},$$

Leads to the fact that the Raspberry Pi is expected to receive about:

$$N_{\text{estimated samples / second}} = F_s \cdot T = 640 \cdot 30 = 19{,}200 \text{ [samples]}.$$

It can be concluded here that no matter the amount of data received, the implementation described in Section 4.2.2 ensures proper handling.

Subsequently, the Arduino transmitted these batches to the Raspberry Pi via serial communication using an optimized practice. In this synchronous communication pattern, the Arduino sends exactly one line of three comma-separated acceleration values at a time, effectively preventing buffer overflows while ensuring data integrity.

30-second measuring window

The measured data of $20{,}684$ samples closely matches the expected estimate, confirming that the system reliably handles the anticipated data rate without any loss, as shown in Figure 5.2.

For the 30-second measurement period, the actual sampling rate was calculated as:

$$F_{s,30} = \frac{N_{30}}{30} = \frac{20{,}684}{30} \approx 689.5 \text{ [samples/s]}.$$

The Raspberry Pi efficiently received, buffered, and organized the incoming data, demonstrating that it could manage the expected data rate of approximately 640 [samples/s] without issues.

It is also worth noting that even when the volume of transmitted data increases, the Arduino handles data transmission efficiently by sending readings in batches of 10 values per serial line. In this case, the transmission success rate was 100%.

60-second measuring window

The measured data of $41{,}434$ samples closely matches the expected estimate, confirming that the system reliably handles the anticipated data rate without any loss, as shown in Figure 5.3. This is almost double of the one calculated over the 30 seconds readings window.

For the 60-second measurement period, the actual sampling rate was:

$$F_{s,60} = \frac{N_{60}}{60} = \frac{41{,}434}{60} \approx 690.6 \text{ [samples/s]}.$$

During this longer measurement period, the Arduino transmitted buffered batches to the Raspberry Pi using an optimized technique. The Arduino sends exactly one line containing three comma-separated acceleration values at a time, effectively preventing buffer overflows while maintaining data integrity.

The Raspberry Pi successfully received, buffered, and organized all incoming data, demonstrating that the system could handle the expected data rate of approximately 640 [samples/s] over an extended period without any loss.

Even when the total transmitted data is doubled compared to the 30-second test, the Arduino continues to transmit efficiently in batches of 10 values. In this test as well, the transmission success rate was 100%.

5.2.3 Raspberry Pi to Supabase and System Performance Metrics

Although verification of the data transmitted from the Arduino to the Raspberry Pi and ultimately posted to a Vercel dashboard was the most crucial part of this project, there is always the possibility of issues in other sections, such as Vercel's data reception. For this reason, the amount of data received and posted by Vercel was also checked, and it was validated that the entire list of 500 data points was properly read.

The following tables relate to the performance of the Raspberry Pi, which can handle even larger amounts of data:

FFT Points	Runtime (s)
250	36
500	39
750	41
1000	43
1500	43
2500	44

Table 5.1: Runtime Raspberry Pi performance for different FFT output lengths

Time Window (s)	Rows of Raw Data	Runtime (s)
30	20,684	40
60	41,433	68
120	82,933	131

Table 5.2: Runtime Raspberry Pi performance for different time windows

The whole system was further checked for performance using cronjob commands, performing recursive data reading over an entire day. No temperature or similar failures of the electronics, microcomputer, or microcontroller were detected, confirming that the system operated reliably.

All in all, the Raspberry Pi transmitted the data to the Supabase API. By comparing the readings collected on the Raspberry Pi with the data successfully posted to Supabase, it was confirmed that the entire pipeline—from sensor to cloud—worked reliably. In this case scenario, there is a 100% successful data transmission in all instances.

30-second measuring window

Each component of the system—from the sensor to the Arduino, from the Arduino to the Raspberry Pi, and from the Raspberry Pi to Supabase—successfully managed the data load while maintaining data integrity, timing requirements, and reliability within the Arduino's operational constraints of 9,000–10,000 bytes per second.

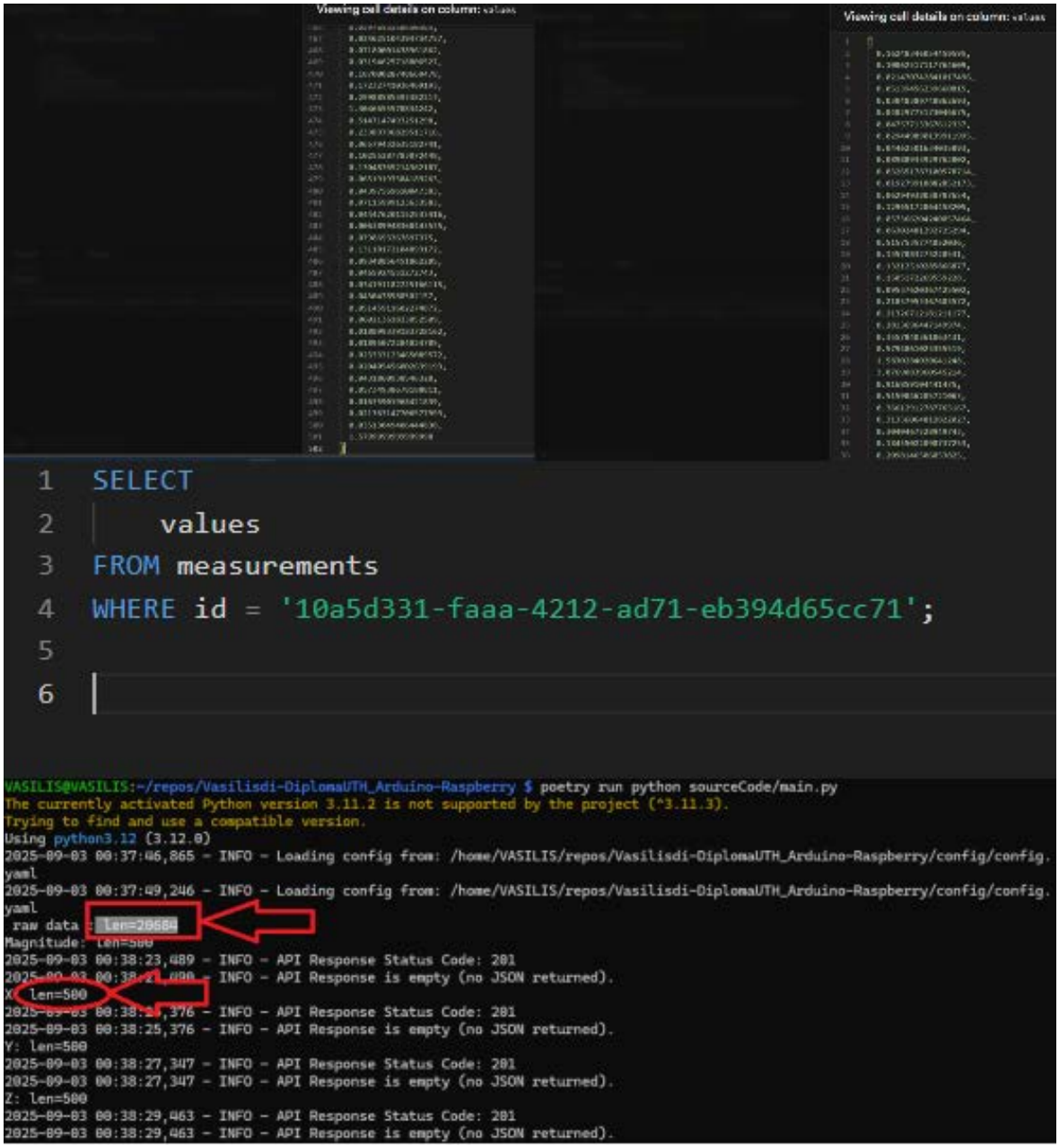

Figure 5.2: System operational validation: data transmission from Arduino to Raspberry Pi for a 30-seconds measurement.

This is illustrated in Figure 5.2, where the raw data from the Arduino consists of 20,684 samples, the data received by the Raspberry Pi is also 20,684 samples, and the processed data

of length 500 has been successfully stored in Supabase. This can be easily verified using the ‘values‘ column.

60-second measuring window

Moreover, a trial on greater measuring window took place. Similarly, for the 60-second measurement period, each component of the system—from sensor to Arduino, Arduino to Raspberry Pi, and Raspberry Pi to Supabase—handled the increased data load while maintaining data integrity, timing requirements, and reliability within the Arduino’s operational constraints of 9,000–10,000 bytes per second.

This is illustrated in Figure 5.3, where the raw data from the Arduino consists of 41,434 samples, the data received by the Raspberry Pi is also 41,434 samples, and the processed data of length 2500 has been successfully stored in Supabase. Verification can be done making the comparison to the ‘values‘ column.

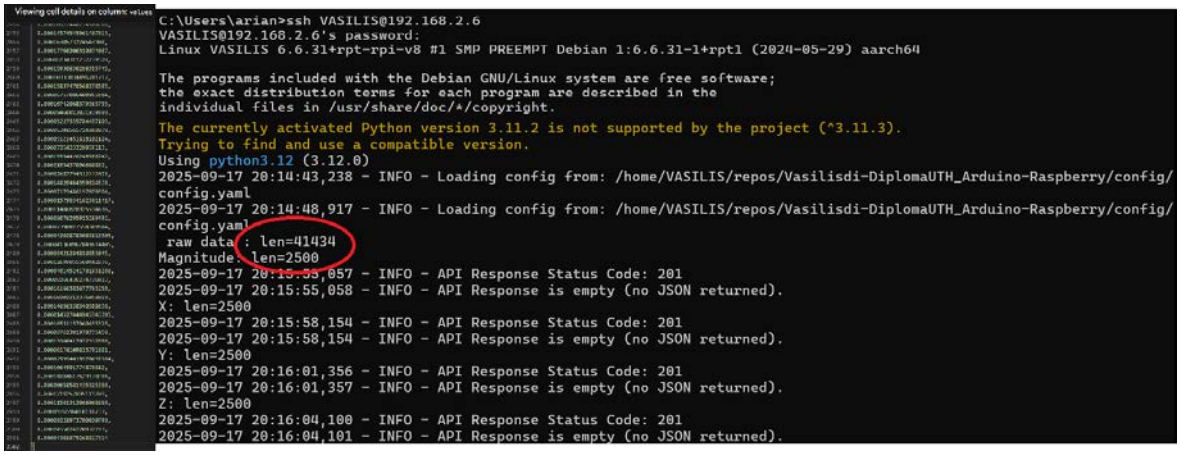

Figure 5.3: System operational validation: data transmission from Arduino to Raspberry Pi for the 60-seconds measurement.

5.3 Summarizing Testing System Performance

In the Appendix Section 2.4 there is many cases of mechanical machine faults. The depicted examples extracted from the Datasets [10] and [9], depict the frequency domain bits related to these faults as well as the raw data wave, where all these frequency domains where calculated from.

All the waveforms are readings of acceleration sensors, which entails that the result are in [g]. This facilitates immensely the process since the algorithms used for this project utilize this this unit as well.

The threshold that is being used as a metrics of dealing with a faulty case, is the 1[g]. So in case the magnitude of the some bit exceeds this threshold this entails that there is a faulty case.

In Figure .5 the 1 [g] threshold is depicted, indicating the critical upper limit. In Section 4.2.1, the concept of 1 [g] as an alarm has been put. In Section 2.2, the faults have been analyzed, yielding some illustration of the faulty machinery spectra. For this reason visual checks take place, considering the aforementioned features so as to decide frequency spectra could reveal.

First, the rotational speed of the machine constitutes the most crucial frequency. In Figure .5 it can be noticed that a motor working at 3000 [rpm], which equals to 50[Hz] (see Equation 3.1), has bit peak at the rotational speed of the machine. According to Section 2.2 this was the expected illustration.

Secondly, the peak exceed some te threshold of the 1 [g], which leads to some maintenance alarm.

Third trait of a mechanical fault is this of the bearing components crucial frequencies. In Section 2.2, it was discussed that in the case of bearing component faults, the resulting vibration spectrum often exhibits frequency peaks at integer multiples of the rotational speed.

Table 5.3: Summary of critical system failures

Fault Condition	Speed	Test Cases	Correct	Incorrect Prediction
Unbalance	50 Hz	11	0	No Alarm
Misalignment	50 Hz	14	0	Correct Alarm
Ball Fault	24 Hz	3	0	No Alarm
Ball Fault	50 Hz	4	0	No Alarm
Inner Race Fault	24 Hz	4	0	No Alarm
Outer Race Fault	24 Hz	3	0	No Alarm

A summary of all the previously concept test is summarized in Table 5.4:

The characteristic bearing fault frequencies can be expressed in the general form:

$$f_{\text{fault}} = n \times f \times F_{\text{bearing}} \tag{5.1}$$

Table 5.4: Summary of system test results

Task / Condition	Test Cases	Success Rate
Misalignment Prediction	14	100%
Other Bearing Failure	25	0%
Data Transmission (Raspberry → Supabase)	3	100%
Data Transmission (Arduino → Raspberry Pi)	3	100%

where n is an integer and F_{bearing} is a specific bearing factor.

Using the aforementioned knowhow, it can be estimated what the faulty condition might be. What can be noticed in the table 5.3 is that there is a prediction pattern. It is easier to predict the misalignment rather than any other mechanical fault. The threshold concept seems to be ineffective in the case of small-size mechanical components, since the only case that it predicted successfully is this of the misalignment.

5.4 Summarizing System Deployment

The deployment phase concerns the 30- and 60-second window readings. The two different concepts retain the following traits:

- 30 seconds of readings
  - 20684 data reading rows
  - Sampling rate: 689.5
  - Frequency domain data length: 500
- 60 seconds of readings
  - 41434 data reading rows
  - Sampling rate: 690.6
  - Frequency domain data length: 2500

Real machinery operation data visualization is taking place in the second phase, following the testing phase. This concept focuses on developing a condition monitoring, cloud-based, application. This concept works by retaining a reference healthy-status frequency curve, so as

to compare future results to this initial reference point. Moreover, further vibration analysis can be executed utilizing all historical data for all 3-axis.

The sensing set-up has been deployed on an actual fan rotating at 35 [Hz]. This fan is loose. The small-scale motored fan is mounted loosely (not screwed on some foundation) and this leads to the peaks, as analyzed bellow.

Prior to the system verification described in the previous section, testing was conducted using a simplified algorithm derived from the main implementation. These preliminary implementations can be found in Appendix 2.2 (Algorithm 1 and Algorithm 7).

The purpose of this algorithm was to validate initial data visualization, demonstrating that both raw time-domain signals and frequency-domain data could be successfully processed and simulated within the Python interface. This approach proved efficient during testing, as it enabled rapid verification of expected patterns. In particular, the frequency spectra produced by the FFT could be readily compared with patterns described in the relevant literature, serving as an early confirmation of correct signal behavior.

Network performance is exquisite with a 100 percent transmission success rate across all API calls, zero timeout incidents, and no security breaches detected. This performance validates the robust error handling and logging mechanisms implemented in the HTTPS communication methodology.

For the Vercel-based interface, visualization tools and interactive charts were employed to depict the transmitted sensor data, supporting the analysis of amplitudes, frequency trends, and potential anomalies over time. Upon accessing the deployed webpage, the dashboard dynamically retrieves sensor data from Supabase, compares the current measurements against stored healthy baseline data, and plots both datasets on the same charts. This comparative visualization enables users to identify deviations at a glance, making the dashboard an effective, real-time monitoring tool deployed efficiently on Vercel.

The following figures illustrate the deployed Vercel-based dashboard used for vibration data monitoring. The interface presents a series of interactive plots that allow comparison between the stored healthy baseline state of the machinery and the latest current measurements retrieved from Supabase. Each chart provides a clear visual distinction between the two states, enabling the detection of deviations across the X, Y, and Z axes as well as the overall vibration magnitude. By scrolling through the interface, the user can review each coordinate in detail, observing the amplitude-frequency relationships that characterize both

normal and potentially anomalous machine conditions.

It must also be noted that on each plot two lines are depicted: the blue line corresponds to the healthy state reference, while the green line illustrates the current state of the machinery. Moreover, it is worth noting that in the case of the values in the frequency domain exceeding the value 1[g], then an orange horizontal line will show up, indicating the dangerous zone. This ensues that in the casing of having peaks exceeding this values some mechanical components needs to be further investigated or even replaced.

The difference between the illustrations of the Figures 5.4 and 5.5, is that the second plot 5.5 retains more detailed view of the situation since it has a processed data frequency length of 2500. Comparing the results of these two it can be concluded that the results resemble each other.

As already has been articulated, in the case of 5.4 and 5.5, the data visualized here are about a loose base fan. This entails that we expect to receive finding similar to this of Figure 2.9. Indeed, the arbitrary peak bits about the Fs peak of the rotational speed, indicate that there os the case of a loose support.

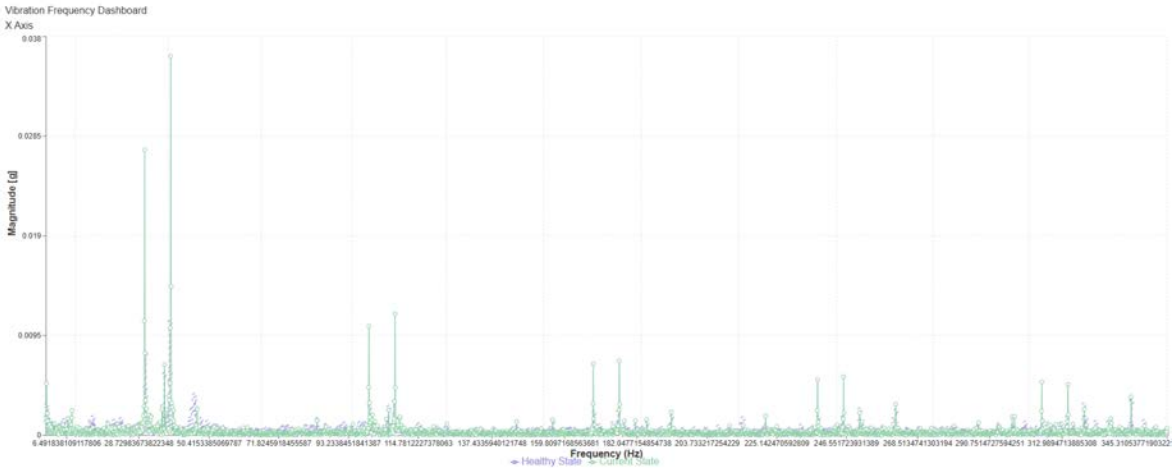

Figure 5.4: 30 seconds deployment, 500 frequency points and 20684 rows.

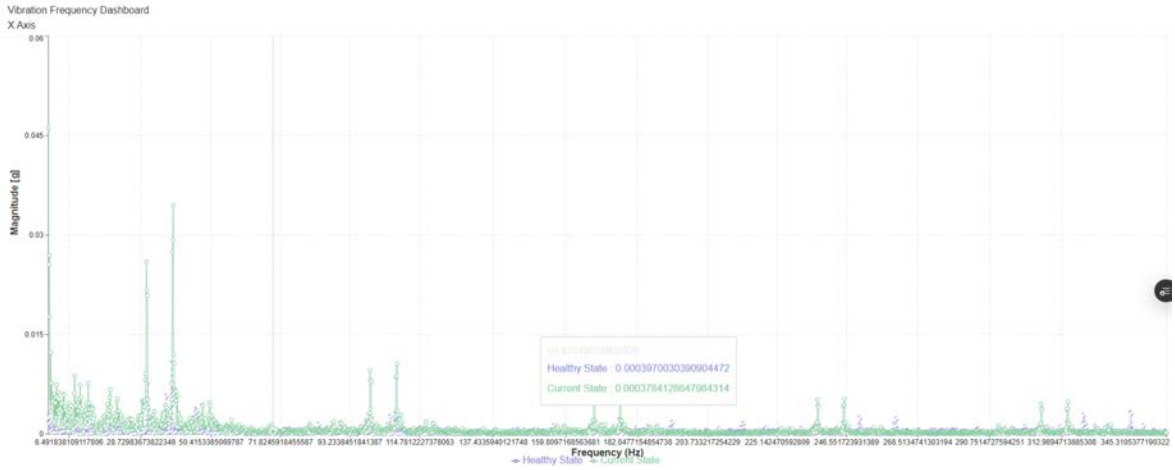

Figure 5.5: 60 seconds deployment, 2500 frequency points and 41434 rows.

Chapter 6

Conclusions

This research has successfully designed, implemented, and validated a complete end-to-end system for machine condition monitoring with a specific focus on bearing defect detection through vibration analysis. The implemented solution effectively integrates hardware components including Arduino UNO, ADXL335 sensor, and Raspberry Pi 4 with software protocols and cloud services such as Supabase and Vercel, creating a reliable system for data acquisition, processing, transmission, and visualization.

The fault identification methodology is based on comparative frequency analysis, where vibration spectra from operational machinery are compared against pre-established healthy baseline spectra. The system processes raw time-domain data through Fast Fourier Transform to generate frequency spectra, which are then visually presented on an interactive Vercel dashboard. The simultaneous display of both healthy baseline and current operational data across X, Y, Z axes and magnitude enables immediate visual identification of potential anomalies.

Data acquisition initiates at the sensor level with the ADXL335 accelerometer mounted on the target machinery and connected to the Arduino UNO microcontroller. The Arduino receives the triaxial acceleration signals and calculates the resultant magnitude. This Arduino algorithm ensures the Arduino accumulates data in batches of 10 samples per axis and transmits comma-separated data lines containing X, Y, Z, and magnitude values only upon receiving explicit requests from the Raspberry Pi, maintaining a consistent sampling rate of approximately 700 Hz without data loss.

Data processing takes place on the Raspberry Pi, where time-domain data undergoes Fast Fourier Transformation to produce frequency-domain spectra essential for bearing fault , missalignment and loose mounting diagnosis. The processed frequency data is subsequently

structured and transmitted to Supabase, which serves as the central data repository for both historical baseline information and current measurements. The final analytical stage occurs on the Vercel dashboard, where frequency spectra are retrieved and presented through interactive visualizations. Although the current implementation requires human interpretation of spectral deviations, the infrastructure supports complete real-time data synchronization and availability.

6.1 Summary and Conclusions

This research has successfully achieved its primary objective of developing and validating a proof-of-concept condition monitoring system utilizing affordable, commercially available components. The implemented solution demonstrates that effective condition monitoring can be accomplished without relying on expensive proprietary systems, thereby significantly reducing the barriers to implementation for predictive maintenance applications.

The system design effectively addresses critical embedded constraints, particularly the Arduino's limited buffer capacity, through an innovative communication protocol that ensures data integrity despite hardware limitations.

Validation testing confirmed the system's operational reliability, with the Raspberry Pi consistently receiving and processing exactly 20,684 samples per channel during 30-second measurement windows (the same deployment implementation took place for a 60-seconds time-window and the results seem to be similar comparing these two cases). The deployment of a dynamic, real-time dashboard on Vercel provides an effective interface for operational monitoring and comparative analysis. The user may tweak the time-span, aiming at receiving greater amount of samples.

In addition, the accelerometer ADXL335 works at 3.3V power supply, which according to its datasheet increaeses its performance. The 5V supply would yield negative results. This lays into the fact that the power supply of the sensor breakout is at 5V (powered by Arduino UNO), and the analog values read by arduino (can only go up to 3.3V), which is the return value of the sensor breakout. This entails that Arduino can only read 10bit value up to 676.

Several limitations and challenges were identified during system development and validation. The analytical upper frequency boundary is constrained by the Arduino's effective sampling rate. The current system implementation necessitates human intervention for visual

spectrum comparison and fault interpretation, introducing subjectivity and limiting scalability. Additionally, the comparative analysis methodology requires the availability of a verified healthy baseline spectrum for each specific machine under monitoring. This threshold is in essence the first run readings or the healthy state readings.

It is also worth stressing that despite the limitations of the Arduino UNO R3 and Raspberry Pi 4, mostly potential bugs due to data accumulation and system malfunctions have been overcome. This ensues that the way the whole project set-up was designed lead to great results and no issues during the operation simulation were noticed.

Last but not least, programming-wise, this project would require much greater accuracy of fault predictions. This could be achieved by making use of machine learning implementations.

In conclusion, this thesis establishes that cost-effective IoT-based solutions can provide viable and efficient vibration-based fault detection capabilities. The research contributes a validated architectural blueprint for constructing such systems, addressing practical challenges related to data handling, serial communication, and full-stack deployment. The resulting implementation provides a solid foundation for developing automated, real-time condition monitoring solutions for industrial applications.

6.2 Future Research

The research presented in this thesis provides a comprehensive foundation for vibration data acquisition and visualization, yet the current dependence on visual comparison by human operators for identifying faults at characteristic frequencies represents a significant limitation. This manual interpretation approach introduces subjectivity, requires substantial time investment, and proves impractical for monitoring multiple machines simultaneously. Consequently, the most critical and logical progression for future research involves transitioning from a visualization platform to an automated diagnostic system.

The highest priority for subsequent investigation involves implementing automated fault detection and diagnosis through machine learning methodologies. This direction would require developing advanced feature extraction techniques to automatically identify key characteristics within frequency spectrum data, including amplitudes at specific characteristic fault frequencies—such as Ball Pass Frequency Outer race (BPFO), Ball Pass Frequency In-

ner race (BPFI), Ball Spin Frequency (BSF), and Fundamental Train Frequency (FTF) (see Chapter 2.

In addition some metrics related to the preventive maintenance practices and even in smaller scale - meaning machine bearing prediction - may be implemented utilizing the reliability theory, analyzed in chapter 2. This would constitute a KPI for the equipment and the maintenance personnel themselves. A Mean Time To Repair and Mean Time Between Failures archive would constitute great metrics for future reference and improvement.

Finally, system expansion through multi-sensor data fusion would further improve diagnostic accuracy. Incorporating additional sensors for temperature monitoring and acoustic emission detection would enable the collection of multi-modal data streams. Sophisticated artificial intelligence models could then fuse these diverse data sources to provide more comprehensive and reliable machine health assessments while reducing false positive rates.

Bibliography

[1] Heinz P. Bloch. Maintenance cost reduction. In Heinz P. Bloch, editor, *Improving Machinery Reliability*, volume 1 of *Lecture Notes in Computer Science*, pages 434–484. Gulf Professional Publishing, 1998.

[2] Bearing fault detection using vibration analysis. `https://ncd.io/blog/bearing-fault-detection-vibration-analysis/`. Accessed: 02-06-2025.

[3] Zhiqiang Chen, Xudong Chen, Chuan Li, René Vinicio Sanchez, and Huafeng Qin. Vibration-based gearbox fault diagnosis using deep neural networks. *Journal of Vibroengineering*, 19:2475–2496, 2017.

[4] Giovanni Betta, Consolatina Liguori, Alfredo Paolillo, and Antonio Pietrosanto. A dsp-based fft-analyzer for the fault diagnosis of rotating machine based on vibration analysis. *IEEE Transactions on Instrumentation and Measurement*, 51:1316–1321, 12 2002.

[5] Gangani Dharmarathne, A. M.S.R. Abekoon, Madhusha Bogahawaththa, Janaka Alawatugoda, and D. P.P. Meddage. A review of machine learning and internet-of-things on the water quality assessment: Methods, applications and future trends. *Results in Engineering*, 26, 6 2025.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54:2787–2805, 10 2010.

[7] Liakot Ali, Roslina Sidek, Ishak Aris, Alauddin Mohd. Ali, and Bambang Sunaryo Suparjo. Design of a micro-uart for soc application. *Computers and Electrical Engineering*, 30:257–268, 6 2004.

[8] About raspberry pi 4. `https://www.raspberrypi.com/`. Ημερομηνία πρόσβασης: 07-08-2025.

[9] Hoang Si Hong and Thuan Nguyen. Hust bearing: a practical dataset for ball bearing fault diagnosis, 2023.

[10] Haris Ihsannur, Bagus Tris Atmaja, Suyanto, and Dhany Arifianto. Vbl-va001: Lab-scale vibration analysis dataset, August 2022.

[11] Si WU, Pu YANG, Dingqian LI, Guanghao CHEN, and Zhe WANG. Towards predictive maintenance: A performance evaluation framework for cooling towers in hvac systems. *Building and Environment*, 284, 10 2025.

[12] F. I. Dehayem Nodem, J. P. Kenné, and A. Gharbi. Simultaneous control of production, repair/replacement and preventive maintenance of deteriorating manufacturing systems. *International Journal of Production Economics*, 134:271–282, 11 2011.

[13] Adxl335 accelerometer breakout. `https://www.analog.com/en/products/adxl335.html`, 2023. Accessed: October 17, 2025.

[14] Analog input. `https://docs.arduino.cc/learn/microcontrollers/analog-input/`. Ημερομηνία πρόσβασης: 06-07-2025.

[15] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[16] Richard C. Singleton. On computing the fast fourier transform. *Communications of the ACM*, 10(10):647–654, Oct. 1967.

[17] C.J. Li and K. McKee. *BEARING DIAGNOSTICS*, pages 143–152. Elsevier, 1 2001.

[18] Michael S. Hamada, Alyson G. Wilson, C. Shane Reese, and Harry F. Martz. Bayesian reliability. *Springer New York, NY*, 1:143–159, 7 2001.

[19] Patrick Ruane, Patrick Walsh, and John Cosgrove. Validation of a digital simulation model for maintenance in a high-volume automated manufacturing facility. In *5th IFAC Workshop on Advanced Maintenance Engineering, Services and Technologies AMEST 2022*, volume 55, pages 127–132. Elsevier B.V., 7 2022.

[20] Jerry G. Lilly, William D. Marscher, Roger J. Cronin, Ray A. Gall, Richard O. Garbus, Willard O. Keightley, J. Davis Miller, Aloysius M. Mocek, William E. Nelson, Randall R. Parks, Jerry P. Pollack, James W. Schettler, and Theodore B. Whiton. *Vibration and Noise*, pages 22.1–22.65. Butterworth-Heinemann, 3 edition, 1 2008.

[21] Weidong Jiao, Zhijing Huang, and Yonghua Jiang. Fault diagnosis of gearboxes using feature extraction in time domain. In *2017 9th International Conference on Modelling, Identification and Control (ICMIC)*, pages 573–577, 2017.

[22] Victor Girondin, Herve Morel, Jean Philippe Cassar, and Komi Midzodzi Pekpe. Vibration-based fault detection of sharp bearing faults in helicopters. *IFAC Proceedings Volumes*, 45:180–185, 1 2012.

[23] Mohamad Hazwan Mohd Ghazali and Wan Rahiman. Vibration-based fault detection in drone using artificial intelligence. *IEEE Sensors Journal*, 22:8439–8448, 5 2022.

[24] Chao Liu, Guofa Li, Tianzhe Wang, and Yixiong Wang. Sparse decomposition-based adaptive kurtogram analysis for bearing compound fault diagnosis. *Measurement: Journal of the International Measurement Confederation*, 256, 12 2025.

[25] Μαρία Κωνσταντινίδου-Σεραφειμίδου. *Διαφορικές Εξισώσεις*. Σοφία, Θεσσαλονίκη, 3 edition, 2009.

[26] Discrete fourier transform. `https://numpy.org/doc/2.2/reference/routines.fft.html`. Ημερομηνία πρόσβασης: 02-06-2025.

[27] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29:1645–1660, 2013.

[28] About arduino. `https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf/`. Ημερομηνία πρόσβασης: 07-08-2025.

[29] About organising projects in python. `https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout//`. Ημερομηνία πρόσβασης: 07-08-2025.

APPENDICES

Appendix

Tables and Algorithms

1 Tables

Table .1: Analog Pins (JANALOG)

Pin	Function	Type	Description
1	NC	NC	Not connected
2	IOREF	IOREF	Reference for digital logic V (connected to 5V)
3	Reset	Reset	Reset
4	+3V3	Power	+3.3V Power Rail
5	+5V	Power	+5V Power Rail
6	GND	Power	Ground
7	GND	Power	Ground
8	VIN	Power	Voltage Input
9	A0	Analog/GPIO	Analog input 0 / GPIO
10	A1	Analog/GPIO	Analog input 1 / GPIO
11	A2	Analog/GPIO	Analog input 2 / GPIO
12	A3	Analog/GPIO	Analog input 3 / GPIO
13	A4/SDA	Analog input/I2C	Analog input 4 / I2C Data line
14	A5/SCL	Analog input/I2C	Analog input 5 / I2C Clock line

Table .2: Digital Pins (JDIGITAL)

Pin	**Function**	**Type**	**Description**
1	D0	Digital/GPIO	Digital pin 0 / GPIO
2	D1	Digital/GPIO	Digital pin 1 / GPIO
3	D2	Digital/GPIO	Digital pin 2 / GPIO
4	D3	Digital/GPIO	Digital pin 3 / GPIO
5	D4	Digital/GPIO	Digital pin 4 / GPIO
6	D5	Digital/GPIO	Digital pin 5 / GPIO
7	D6	Digital/GPIO	Digital pin 6 / GPIO
8	D7	Digital/GPIO	Digital pin 7 / GPIO
9	D8	Digital/GPIO	Digital pin 8 / GPIO
10	D9	Digital/GPIO	Digital pin 9 / GPIO
11	SS	Digital	SPI Chip Select
12	MOSI	Digital	SPI Main Out Secondary In
13	MISO	Digital	SPI Main In Secondary Out
14	SCK	Digital	SPI Serial Clock Output
15	GND	Power	Ground
16	AREF	Digital	Analog Reference Voltage
17	A4/SD4	Digital	Analog input 4 / I2C Data line (duplicated)
18	A5/SD5	Digital	Analog input 5 / I2C Clock line (duplicated)

Table .3: Complete GPIO Pinout for Raspberry Pi 4 Model B (Part 1/2)

Pin #	Function	Alt Function	Voltage	Notes
1	3.3V	–	3.3V	50mA max
2	5V	–	5V	Power supply
3	GPIO2	SDA1	3.3V	I²C Data
4	5V	–	5V	Power supply
5	GPIO3	SCL1	3.3V	I²C Clock
6	GND	–	–	Ground
7	GPIO4	GPCLK0	3.3V	General purpose
8	GPIO14	TXD0	3.3V	UART Transmit
9	GND	–	–	Ground
10	GPIO15	RXD0	3.3V	UART Receive
11	GPIO17	–	3.3V	General purpose
12	GPIO18	PCM_CLK	3.3V	PWM0
13	GPIO27	–	3.3V	General purpose
14	GND	–	–	Ground
15	GPIO22	–	3.3V	General purpose

Table .4: Complete GPIO Pinout for Raspberry Pi 4 Model B (Part 2/2) (Continued from Table .3)

Pin #	**Function**	**Alt Function**	**Voltage**	**Notes**
16	GPIO23	–	3.3V	General purpose
17	3.3V	–	3.3V	50mA max
18	GPIO24	–	3.3V	General purpose
19	GPIO10	MOSI0	3.3V	SPI0
20	GND	–	–	Ground
21	GPIO9	MISO0	3.3V	SPI0
22	GPIO25	-	3.3V	General purpose
23	GPIO11	SCLK0	3.3V	SPI0
24	GPIO8	CE0	3.3V	SPI0 Chip Enable 0
25	GND	-	-	-
26	GPIO7	CE1	3.3V	SPI0 Chip Enable 1
27	ID_SD	-	-	EEPROM ID
28	ID_SC	-	-	EEPROM Clock
29	GPIO5	-	3.3V	General purpose
30	GND	-	-	-
31	GPIO6	-	3.3V	General purpose
32	GPIO12	PWM0	3.3V	PWM channel 0
33	GPIO13	PWM1	3.3V	PWM channel 1
34	GND	-	-	-
35	GPIO19	PCM_FS	3.3V	PWM/MISO1
36	GPIO16	-	3.3V	General purpose
37	GPIO26	-	3.3V	General purpose
38	GPIO20	MOSI1	3.3V	SPI1
39	GND	-	-	-
40	GPIO21	MISO1	3.3V	SPI1

Table .5: Router and Network Configuration Specifications

#	**Property**	**Value**	**Description**
1	Router IP (Gateway)	192.168.2.1	Local private IP address serving as network gateway
2	Operating Band	2.4 GHz	Longer range frequency, suitable for IoT applications
3	Channel	11	Optimized channel selection to minimize interference
4	Security Protocol	WPA2-Personal (AES/CCMP)	Enterprise-grade encryption for data protection
5	Signal Strength (RSSI)	93% / -42 dBm	Excellent connection quality with minimal packet loss
6	Link Speed	130 Mbps	More than adequate bandwidth for sensor data transmission
7	Internet Bandwidth	10 Mbps	Sufficient for external data transmission requirements

2 Algorithms

2.1 Arduino Data Acquisition Algorithm

Algorithm 1: Data Acquisition and Transmission System

Initialization:
numChannels $= 3$ {Number of analog channels}
analogPins $= [A0, A1, A2]$ {Analog input pins}
bufferSize $= 10$ {Buffer size per channel}
analogBuffer $=$ array$[3][10]$ {Data storage array}
bufferIndex $=$ array$[3]$ {Buffer indices array}
tempVal $= 0$ {Control variable}

Initialize serial communication at 115200 baud
for $i = 0$ to numChannels $- 1$ **do**
    bufferIndex$[i] \leftarrow 0$ {Initialize buffer index}
end for
$analogReference(EXTERNAL)$

while true **do**
    Phase 1: Data Acquisition and Storage
    for $j = 0$ to bufferSize $- 1$ **do**
        for $i = 0$ to numChannels $- 1$ **do**
            $xRaw =$ analogRead(analogPins$[i]$)
            analogBuffer$[i][j] = xRaw$
        end for
    end for

    Phase 2: Processing and Transmission
    if tempVal $= 100$ **then**
        Execute calibration procedures
        tempVal $= 0$
    else

```
29:       for j = 0 to bufferSize − 1 do
30:          for i = 0 to numChannels − 1 do
31:             Serial.print(analogBuffer[i][j])
32:             if i < numChannels − 1 then
33:                Serial.print(, )
34:             end if
35:          end for
36:          Serial.println() {End of data line}
37:       end for
38:       tempVal = tempVal + 1
39:    end if
40: end while
```

2.2 Raspberry Pi Algorithms

Algorithm 1: main.py Implementation

```
from datetime import datetime, timedelta, timezone
from api import VibrationMonitoringAPI
from data_processing import calculate_amplitude, compute_fft
from serial_communication import setup_serial_connection, read_serial_data
from utils import get_api_key, load_config
import numpy as np

# Load configuration
config = load_config()
# Load API key from environment variable
api_key = get_api_key()
if not api_key then
    raise ValueError("API key not found. Please set the API_KEY environment variable.")
end if
vibration_api = VibrationMonitoringAPI()
baud_rate = config['serial']['baud_rate']
port = config['serial']['port_linux']
# port = config['serial']['port_windows']
ser = setup_serial_connection(port, baud_rate)
history = []
history_X = []
history_Y = []
history_Z = []
time_window = config['time_window']
timestamps = []
start_time = datetime.now(timezone.utc)
time_delta = timedelta(seconds = time_window)

try:
while datetime.now(timezone.utc) − start_time < time_delta do
```

timestamp = datetime.now(timezone.utc)
timestamps.append(timestamp.timestamp())
col_1, col_2, col_3 = read_serial_data(ser)
amplitude = calculate_amplitude(col_1, col_2, col_3)
history.append(((amplitude * (5/1023)) − 2.5)/0.3)
history_X.append(((col_1 * (5/1023)) − 2.5)/0.3)
history_Y.append(((col_2 * (5/1023)) − 2.5)/0.3)
history_Z.append(((col_3 * (5/1023)) − 2.5)/0.3)
end while
# Calculate the actual sampling rate
time_diffs = np.diff(timestamps)
Fs = 1/np.mean(time_diffs)
L = 1000 # number of FFT points
for $i = 0$ **to** 3 **do**
values, f = compute_fft(data, Fs, L)
titles = [
”Single-Sided Amplitude Spectrum of sqrt(X(t)^2 + Y(t)^2 + Z(t)^2)”,
”Single-Sided Amplitude Spectrum of X(t)”,
”Single-Sided Amplitude Spectrum of Y(t)”,
”Single-Sided Amplitude Spectrum of Z(t)”
]
ylabels = [”|P1(f)|”, ”|P1X(f)|”, ”|P1Y(f)|”, ”|P1Z(f)|”]
coordinate = [”Magnitude”, ”X”, ”Y”, ”Z”]
api = VibrationMonitoringAPI()
status_code = api.send_measurement(”Test”, coordinate[i], values.tolist(), start_time, timestamp)
end for
except KeyboardInterrupt:
ser.close()

Algorithm 2: api.py Implementation

```
import requests
import json
import logging
from utils import load_config, get_api_key, get_supabase_url_posting, get_supabase_url_selecting


from datetime import datetime, timezone
from zoneinfo import ZoneInfo
config = load_config()

class VibrationMonitoringAPI:
    def __init__(self):
        # Load API key and Supabase URL using the helper functions
        self.api_key = get_api_key()
        self.purl = get_supabase_url_posting()
        self.surl = get_supabase_url_selecting()
        # Ensure both the API key and Supabase URL are available
        if not self.api_key:
            logging.error("API Key is missing.")
            raise ValueError("API Key is required for the API.")
        if not self.purl:
            logging.error("Supabase URL is missing.")
            raise ValueError("Supabase URL is required for the API.")
        if not self.surl:
            logging.error("Supabase URL is missing.")
            raise ValueError("Supabase URL is required for the API.")
        # Initialize headers using the loaded API key
        self.headers = {
            'accept': 'application/json',
            'apikey': self.api_key,
            'Content-Type': 'application/json'
        }
```

```
31:         # Define the Athens time zone using ZoneInfo
32:         self.athens_tz = ZoneInfo('Europe/Athens')
33:     def get_time(self, start_measurement, end_measurement):
34:         # Convert the start and end times to Athens time zone
35:         start_measurement_athens = start_measurement.astimezone(self.athens_tz)
36:         end_measurement_athens = end_measurement.astimezone(self.athens_tz)
37:         # Format the timestamps in the required format with Athens time zone
38:         start_measurement_str = start_measurement_athens.strftime('%Y-%m-%dT%H:
    %M:%S.%f')[:-3] +' Z'
39:         end_measurement_str = end_measurement_athens.strftime('%Y-%m-%dT%H:
    %M:%S.%f')[:-3] +' Z'
40:         return start_measurement_str, end_measurement_str # end_measurement is now is not specified
41:
42:     def send_measurement(self, sensor_name, coordinate, values, start_measurement =
    datetime.now(timezone.utc), end_measurement = datetime.now(timezone.utc)) :
43:         start_measurement_str, end_measurement_str = self.get_time(start_measurement, end_measure
44:         # Lookup sensor_id from sensor_name
45:         sensor_id = self.get_sensor_id(sensor_name)
46:         if not sensor_id:
47:             logging.error(f"Sensor with name '{sensor_name}' not found.")
48:             return None
49:         payload = {
50:             "sensor_id": sensor_id,
51:             "startMeasurement": start_measurement_str,
52:             "endMeasurement": end_measurement_str,
53:             "coordinate": coordinate,
54:             "values": values
55:         }
56:         response = requests.post(self.purl, headers = self.headers, data = json.dumps(payload))
57:         # Log the response status and data
58:         logging.info(f"API Response Status Code: {response.status_code}")
59:         #Check if response has content before parsing JSON
```

```
60:         if response.text.strip(): # Ensure response is not empty
61:             try:
62:                 logging.info(f"API Response JSON: " + str(response.json()))
63:             except requests.exceptions.JSONDecodeError:
64:                 logging.warning("API Response is not JSON (empty or invalid format).")
65:         else:
66:             logging.info("API Response is empty (no JSON returned).")
67:         return response.status_code
68:
69:     # New method to get sensor_id from sensor_name
70:     def get_sensor_id(self, sensor_name):
71:         query_url = f"{"self.surl"}?sensor_name=eq.{sensor_name}"
72:         response = requests.get(query_url, headers = self.headers)
73:         if response.status_code == 200 :
74:             data = response.json()
75:             if data:
76:                 return data[0]['id']
77:         else:
78:             logging.error(f"Failed to fetch sensor_id for sensor_name '{sensor_name}': {response.status

79:         return None
```

Algorithm 3: data_processing.py Implementation

```
import numpy as np
import matplotlib.pyplot as plt
import math
import scipy.signal as signal

def calculate_amplitude(col_1, col_2, col_3):
    return math.sqrt(col_1**2 + col_2**2 + col_3**2)

def compute_fft(data, Fs, L):
    fft_result = np.fft.fft(data, L)
    P2 = np.abs(fft_result[1 :]/L)
    P1 = P2[: L//2]
    P1[1 : -1] = 2 * P1[1 : -1]
    return P1, Fs/L * np.arange(1, L//2 + 1)

def compute_envelope(data):
    analytic_signal = signal.hilbert(data)
    envelope = np.abs(analytic_signal)
    return envelope

def plot_fft(P1, f, title, ylabel):
    plt.figure()
    plt.plot(f, P1, linewidth = 3)
    plt.title(title)
    plt.xlabel(f(Hz))
    plt.ylabel(ylabel)
    plt.show()

def unbalancing(frequency, rpm, spectrum, threshold):
    #index = f.index(rpm) to be used in main.py
    index = np.where((frequency > 0.999 * rpm)&(frequency < 1.001 * rpm))
```

```
    #print(index)
    frq = frequency[index]
    amplitude = spectrum[index]
    if amplitude > threshold :
        status = 'Shaft needs to be balanced'
    else:
        status = 'Balancing is ok'
    return f"RPM: {frq}, Amplitude: {amplitude}, Status: {status}"
```

Algorithm 4: serial_communication.py Implementation

```
import serial
import time

def setup_serial_connection(port, baud_rate):
    ser = serial.Serial(port, baud_rate, timeout = 1.0)
    time.sleep(1)
    ser.reset_input_buffer()
    return ser

def read_serial_data(ser):
    ser.write('rasp\n'.encode())
    while ser.in_waiting <= 0 :
        time.sleep(0.001)
    response = ser.readline().decode('utf - 8').rstrip()
    col_1, col_2, col_3 = map(float, response.split(','))
    return col_1, col_2, col_3
```

Algorithm 5: utils.py Implementation

```
import os
import yaml
import logging
from pathlib import Path
from dotenv import load_dotenv

# Configure logging with timestamps
logging.basicConfig(
    level = logging.INFO,
    format = %(asctime)s - %(levelname)s - %(message)s,
)

def load_env_variables():
    ”Load environment variables from .env file.”
    load_dotenv()

def load_config(config_path = None):
    ”Load YAML configuration file.”
    if config_path is None:
        config_path = Path(__file__).resolve().parent.parent/'config'/'config.yaml'
    else:
        config_path = Path(config_path).resolve()
    logging.info(f”Loading config from: {config_path}”)
    # Check if file exists
    if not config_path.is_file():
        raise FileNotFoundError(f”Config file not found at: {config_path}”)
    try:
        with config_path.open('r') as file :
            # More readable pathlib usage
            config = yaml.safe_load(file)
    except yaml.YAMLError as e:
```

```
32:         raise ValueError(f"Error parsing YAML file at {config_path}: {e}")
33:     except Exception as e:
34:         raise RuntimeError(f"Unexpected error loading config: {e}")
35:     return config
36:
37: def get_api_key():
38:     "Retrieve API key from environment variables."
39:     load_env_variables()() # Ensure environment variables are loaded
40:     api_key = os.getenv('API_KEY')
41:     if not api_key:
42:         logging.warning("API_KEY is not set in the environment.")
43:         return None # Return None instead of raising an error
44:     return api_key
45:
46: def get_supabase_url_posting():
47:     "Retrieve Supabase URL from environment variables."
48:     load_env_variables()() # Ensure environment variables are loaded
49:     purl = os.getenv('SUPABASE_URL_FOR_POSTING')
50:     if not purl:
51:         logging.warning("SUPABASE_URL is not set in the environment.")
52:         return None # Return None if SUPABASE_URL is not set
53:     return purl
54:
55: def get_supabase_url_selecting():
56:     "Retrieve Supabase URL from environment variables."
57:     load_env_variables()() # Ensure environment variables are loaded
58:     surl = os.getenv('SUPABASE_URL_FOR_SELECTING')
59:     if not surl:
60:         logging.warning("SUPABASE_URL is not set in the environment.")
61:         return None # Return None if SUPABASE_URL is not set
62:     return surl
```

Algorithm 6: config.yaml Implementation

```
serial:
    # This applies in the case of Linux OS
    port_linux: ”/dev/ttyACM0”
    # This applies in the case of Windows OS
    port_windows: ”COM3”
    baud_rate: 115200
time_window: 30
# user defined
rpm: 300
bpfi: 3.4
bpfo: 2
ftf: 10
bsf: 5
threshold: 5
```

Algorithm 7: Testing Results

```
1: import time
2: import numpy as np
3: from sourceCode.utils import loadconfig
4: from sourceCode.dataprocessing import computeenvelope, computefft, calculateamplitude, plotfft, unl
5: from sourceCode.serialcommunication import setupserialconnection, readserialdata
6:
7: config = loadconfig()
8:
9: baudrate = config['serial']['baudrate']
10: port = config['serial']['portwindows']
11: rpm = config['rpm']
12: threshold = config['threshold']
13: ser = setupserialconnection(port, baudrate)
14:
15: time.sleep(1)
16: ser.resetinputbuffer
17:
18: history = []
19: historyX = []
20: historyY = []
21: historyZ = []
22: timewindow = config['timewindow']
23: timestamps = []
24: starttime = time.time()
25:
26: try:
27:     while time.time() - starttime < timewindow:
28:         timestamp = time.time()
29:         timestamps.append(timestamp)
30:         col1, col2, col3 = readserialdata(ser)
```

```
31:             amplitude = calculateamplitude(col1, col2, col3)
32:             history.append(amplitude)
33:             historyX.append(col1)
34:             historyY.append(col2)
35:             historyZ.append(col3)
36:
37:         timediffs = np.diff(timestamps)
38:         Fs = 1 / np.mean(timediffs)
39:
40:         L = 1000
41:         ylabels = ["|P1(f)|", "|P1X(f)|", "|P1Y(f)|", "|P1Z(f)|"]
42:
43:         for i, data in enumerate([history, historyX, historyY, historyZ]):
44:             P1, f = computefft(data, Fs, L)
45:             titles = [
46:                 "Single-Sided Amplitude Spectrum of resultant acceleration",
47:                 "Single-Sided Amplitude Spectrum of X-axis",
48:                 "Single-Sided Amplitude Spectrum of Y-axis",
49:                 "Single-Sided Amplitude Spectrum of Z-axis"
50:             ]
51:             print(unbalancing(f, rpm, P1, threshold))
52:             plotfft(P1, f, titles[i], ylabels[i])
53:
54:         for i, data in enumerate([history, historyX, historyY, historyZ]):
55:             Envelope = computeenvelope(data)
56:
57:     except KeyboardInterrupt:
58:         ser.close()
```

2.3 Vercel Webpage Implementation

Vibration Frequency Data Visualization Dashboard Implementation - React Frontend

```
'use client';
import { useEffect, useState } from 'react'
import { LineChart, Line, XAxis, YAxis, CartesianGrid, Tooltip,
Legend, ResponsiveContainer } from 'recharts'
type Measurement = {
  id: string;
  startMeasurement: string;
  endMeasurement: string;
  coordinate: string;
  values: number[];
  frequencies: number[];
export default function Home() {
  const [measurements, setMeasurements] = useState<Measurement[]>([]

  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    const fetchData = async () => {
      try
        const res = await fetch('/api/measurements');
        if (!res.ok) throw new Error('Error fetching data');

        const data: Measurement[] = await res.json();
        setMeasurements(data);
      } catch (err) {
        setError((err as Error).message);
    fetchData();
  if (error) return <div>Error: {error}</div>;
  function getHealthyAndCurrent(measurements: Measurement[])
{
```

```
27:     const grouped: Record<string, { healthy?: Measurement;
   current?: Measurement }> = {};
28:     for (const m of measurements) {
29:       const coord = m.coordinate;
30:       if (!grouped[coord]) grouped[coord] = { healthy: m, current:
   m };
31:       else {
32:         if (new Date(m.startMeasurement).getTime() < new Date(groupe
   grouped[coord].healthy = m;
33:         if (new Date(m.endMeasurement).getTime() > new Date(grouped[
   grouped[coord].current = m;
34:       }
35:     }
36:     return grouped;
37:   }
38:
39:   const grouped = getHealthyAndCurrent(measurements);
40:
41:   const combineData = (healthy?: Measurement, current?: Measurement)
   => {
42:     if (!healthy and !current) return [];
43:     const length = healthy ? healthy.values.length : current!.values

44:     const combined: { frequency: number; healthyValue?: number;
   currentValue?: number }[] = [];
45:     for (let i = 0; i < length; i++) {
46:       combined.push({
47:         frequency: healthy?.frequencies[i] ?? current!.frequencies[i

48:         healthyValue: healthy?.values[i],
49:         currentValue: current?.values[i]
50:       });
```

```
51:       }
52:       return combined;
53:     };
54:
55:     const xData = combineData(grouped.X?.healthy, grouped.X?.current);

56:     const yData = combineData(grouped.Y?.healthy, grouped.Y?.current);

57:     const zData = combineData(grouped.Z?.healthy, grouped.Z?.current);

58:     const magnitudeData = combineData(grouped.Magnitude?.healthy,
    grouped.Magnitude?.current);
59:
60:     return (
61:       <div>
62:         <h1>Vibration Frequency Dashboard</h1>
63:         {measurements.length === 0 ? <p>Loading measurements...</p>
    : <>}
64:           /* X Axis */
65:           <h2>X Axis</h2>
66:           <ResponsiveContainer width=”100per cent” height=700><LineCha
    data=xData><CartesianGrid strokeDasharray=”3 3” /><XAxis dataKey=”fr
    label={{ value: ’Frequency (Hz)’, position: ’insideBottom’,
    offset: -5 }} /><YAxis label={{ value: ’Amplitude’, angle: -90,
    position: ’insideLeft’ }} /><Tooltip /><Legend /><Line type=”monoton
    dataKey=”healthyValue” stroke=”8884d8” name=”Healthy State”
    strokeWidth=3/><Line type=”monotone” dataKey=”currentValue”
    stroke=”82ca9d” name=”Current State” /></LineChart></ResponsiveConta

67:         /* Y, Z, Magnitude axes follow the same pattern as X
    Axis */
68:       </> }
```

```
69:         </div>);
```

Supabase Data Retrieval API Endpoint - Vercel API endpoint

```
1: import { supabaseAdmin } from '../../lib/supabase';
2: export async function GET() {
3:   try {
4:     console.log("Fetching data from Supabase...");
5:     const { data, error } = await supabaseAdmin
6:       .from('measurements')
7:       .select('*');
8:     if (error) {
9:       console.error("Error fetching measurements:", error);

10:      return new Response("Error fetching measurements: " +
   error.message, { status: 500 });
11:    }
12:    console.log("Fetched data:", data);
13:    return new Response(JSON.stringify(data)
14:      headers: { "Content-Type": "application/json" },
15:    });
16:  } catch (err) {
17:    console.error("Unexpected error:", err);
18:    // Cast err to Error type to access message
19:    const errorMessage = err instanceof Error ? err.message
   : "Unknown error";
20:    return new Response("Unexpected error: " + errorMessage,
   { status: 500 });
21:  }
```

2.4 Testing Figures

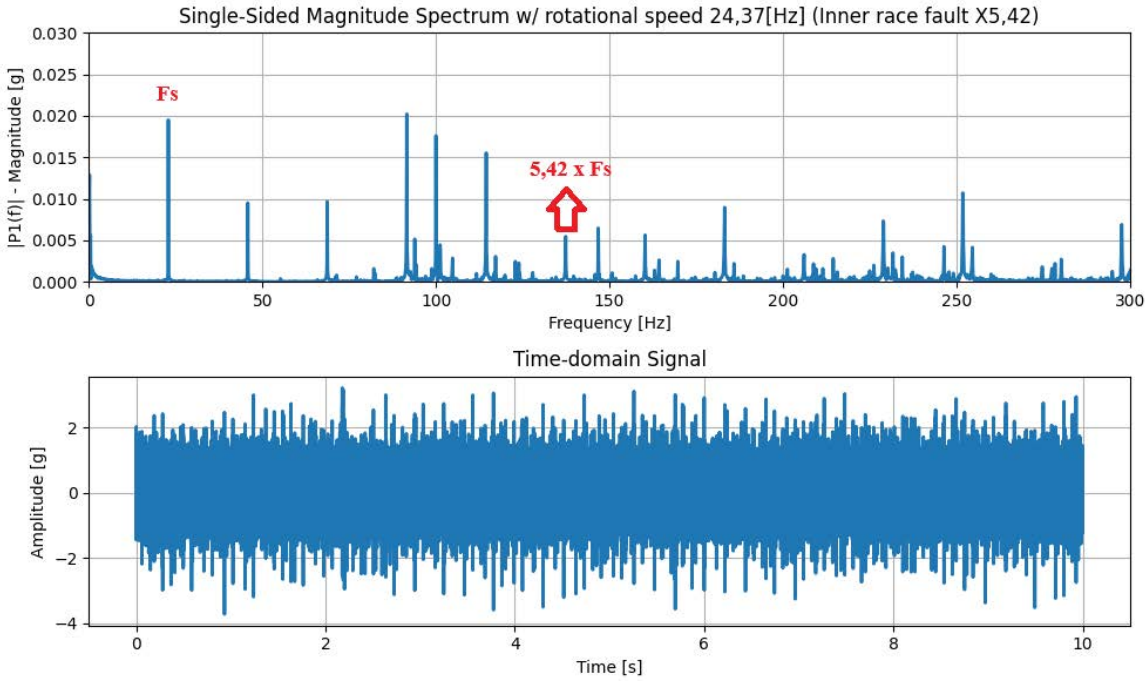

Figure .1: First bearing inner racing fault for rotational speed 24,37[Hz] [9].

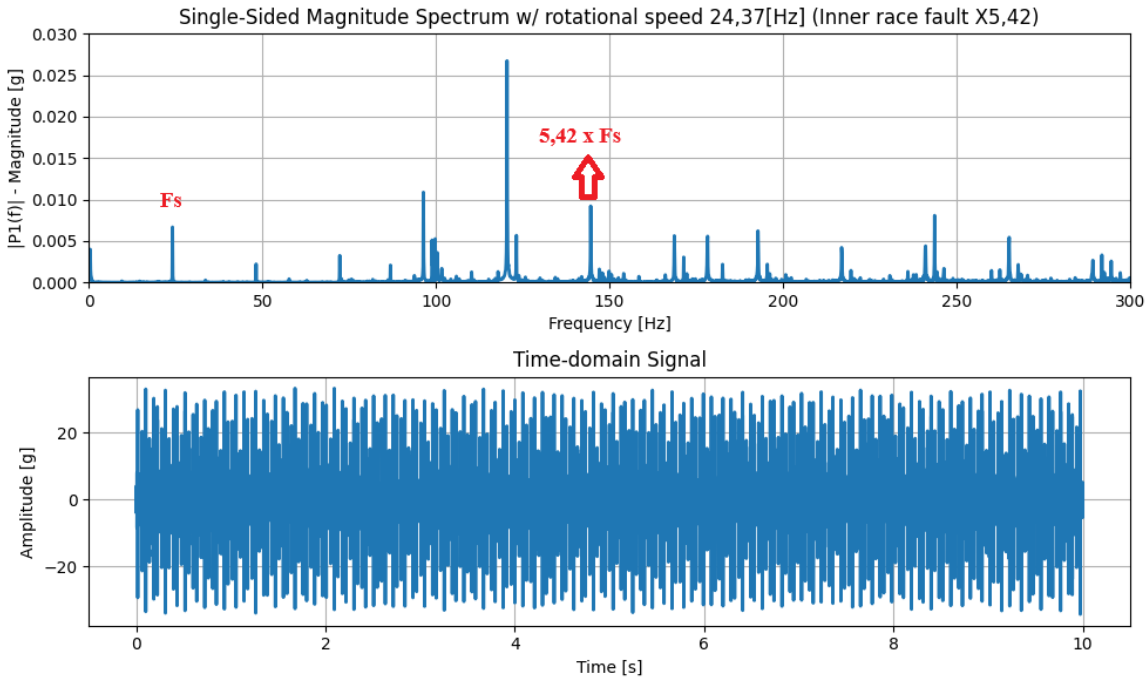

Figure .2: Second bearing inner racing fault for rotational speed 24,37[Hz] [9].

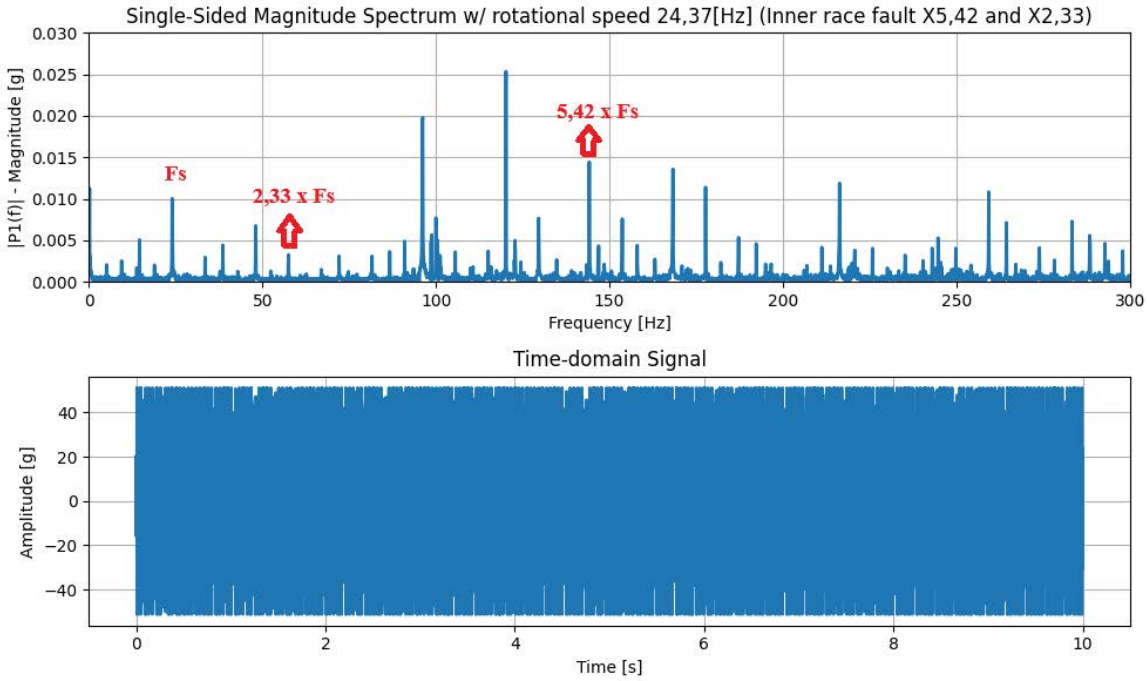

Figure .3: Third bearing inner racing fault for rotational speed 24,37[Hz] [9].

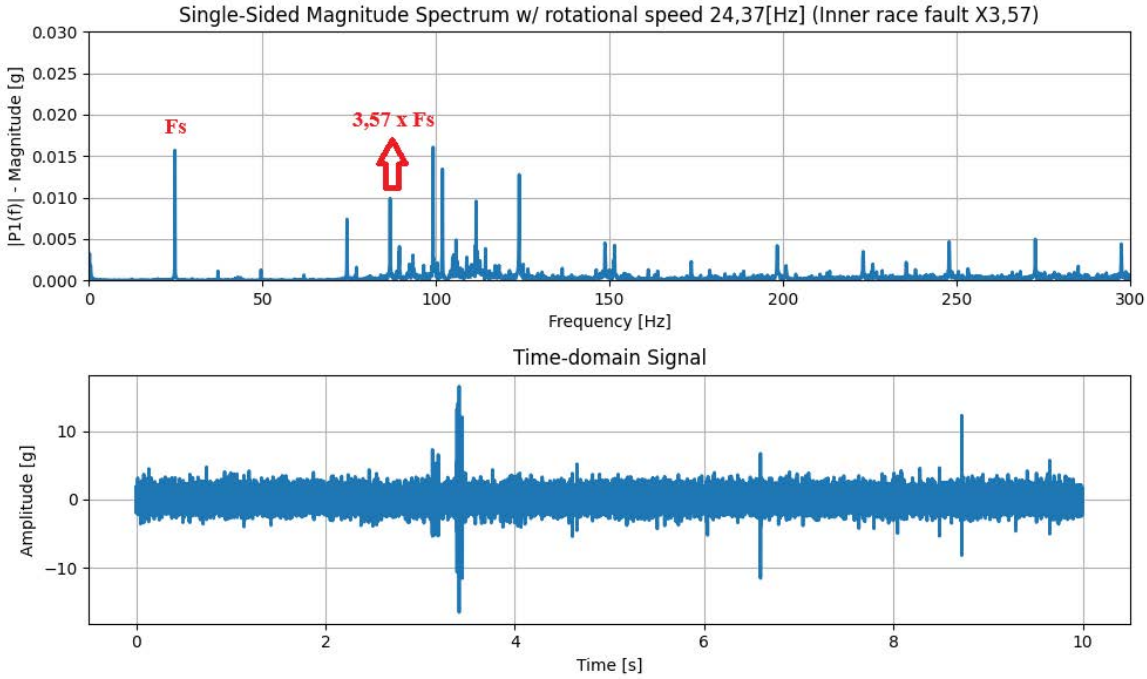

Figure .4: Bearing outer racing fault for rotational speed 24,37[Hz] [9].

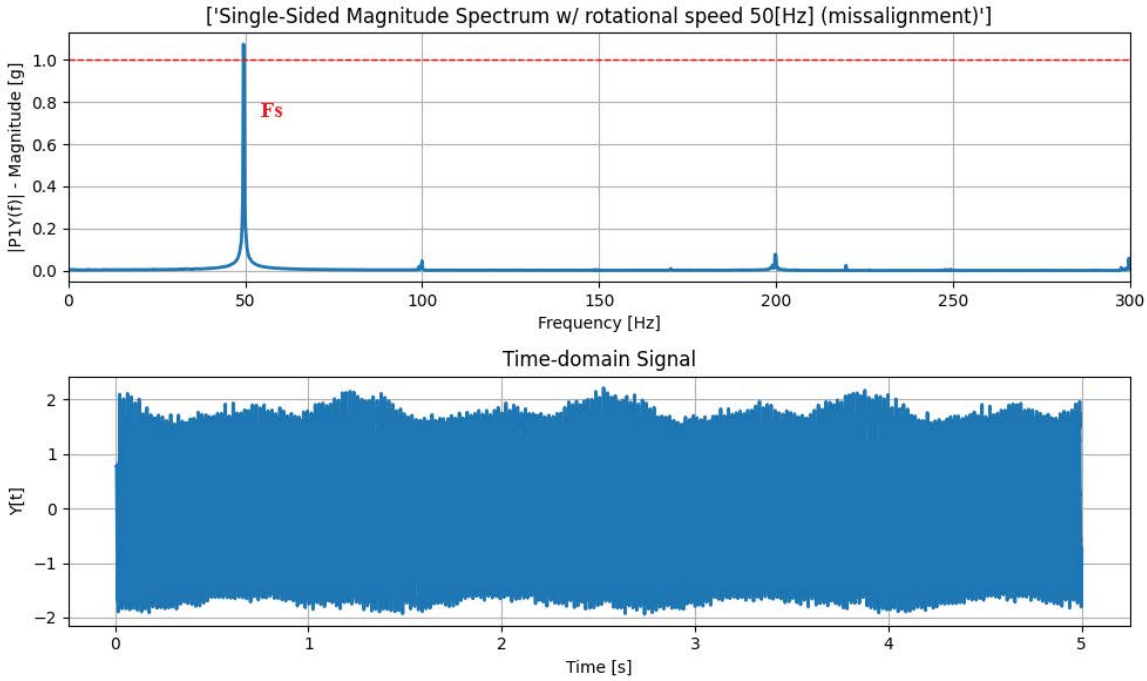

Figure .5: Motor misalignment fault at rotational speed 50[Hz] [10].

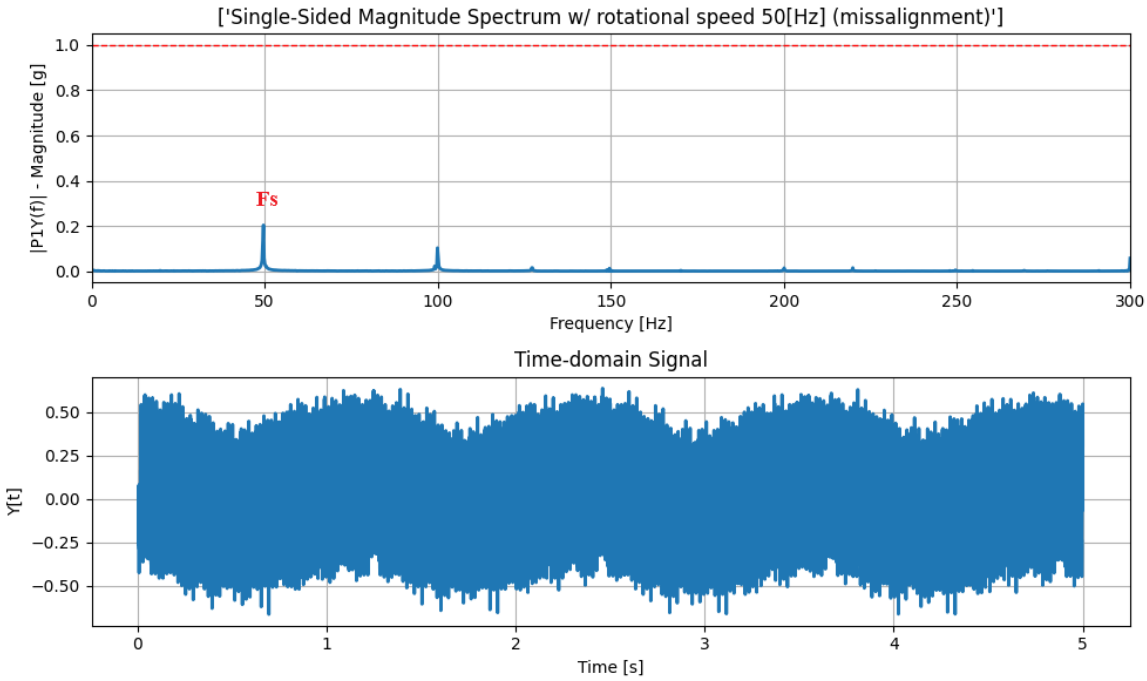

Figure .6: Motor unbalance fault at rotational speed 50[Hz] [10].