Department of Electronic and Telecommunication Engineering
University of Moratuwa

EN2160- Electronic Design Realization

Design Documentation

Vibration Damping System for Machine Tools

Group K (29)

210204R Hansindu K.A.A.

210341H Liyanaarachchi L.A.S.

Contents

1 Introduction

This design documentation presents a comprehensive overview of the development process, technical specifications, and detailed project progression for the Vibration Damping System for Machine Tools. The project aims to address the critical issue of mechanical vibrations in industrial machinery.

The report encompasses various stages of the project, starting from the conceptual design to component selection, system architecture, and detailed data acquisition and analysis. It also includes an extensive daily log of activities, reflecting the systematic approach and rigorous testing undertaken throughout the project. This documentation serves as a valuable resource for understanding the design decisions, challenges encountered, and solutions implemented, providing insights for future improvements and applications.

2 Background Research

Mechanical vibrations in industrial machinery are a critical concern as they can lead to performance degradation, increased wear and tear, and ultimately, machine failure. Understanding the causes and mechanisms of these vibrations is essential for effective maintenance and design improvement. This section provides an overview of different types of vibrations, methods for measuring them, and existing damping systems. This foundational knowledge is crucial for developing a robust vibration damping system for machine tools.

2.1 Types of Vibrations

Industrial machines, by their very nature, are subject to various types of vibrations due to the dynamic forces they encounter during operation. Understanding these vibrations is crucial for maintaining machinery performance, ensuring safety, and prolonging equipment life. Following are the types of vibrations commonly found in industrial machines:

2.1.1 Free Vibration

Free vibration occurs when a machine or a component is displaced from its equilibrium position and allowed to vibrate without continuous external force. This type of vibration can be further categorized into:

- **Damped Free Vibration:** The amplitude of vibration decreases over time due to the presence of damping elements like friction or material properties.
- **Undamped Free Vibration:** In an ideal scenario with no damping, the vibration continues indefinitely at a constant amplitude.

2.1.2 Forced Vibration

Forced vibration happens when an external force continuously influences the machine. The frequency of the vibration matches the frequency of the external force. Examples include vibrations caused by unbalanced rotating components, misalignment, or periodic impacts.

2.1.3 Self-Excited Vibration

Self-excited vibrations are sustained by the energy provided by the motion of the machine itself, rather than an external periodic force. These vibrations can grow in amplitude until limited by non-linear effects or energy dissipation mechanisms. Common examples are:

- **Chatter in machine tools:** Often occurs during cutting processes.
- **Flutter in aeroelastic systems:** Found in aircraft structures and turbine blades.

2.1.4 Transient Vibration

Transient vibrations are short-duration vibrations that occur in response to a sudden force or impact. These are often seen during events like startup, shutdown, or sudden loading/unloading of a machine.

2.1.5 Steady-State Vibration

Steady-state vibrations occur when a machine operates under constant conditions, leading to continuous and predictable vibration patterns. These are often observed in machinery operating at a constant speed or load.

2.1.6 Random Vibration

Random vibration lacks a predictable pattern and is caused by unpredictable forces. It is often modeled using statistical methods. Common sources include road surface roughness for vehicles, wind loads on structures, and turbulence in fluid systems.

2.1.7 Resonant Vibration

Resonant vibration occurs when the frequency of external forces matches the natural frequency of the system, leading to large amplitude vibrations. This phenomenon can cause severe damage if not controlled. Resonance is a critical concern in the design of industrial machines.

2.1.8 Torsional Vibration

Torsional vibrations involve angular oscillations around the rotational axis of a shaft or component. These are particularly important in power transmission systems and can lead to failures if not properly managed.

2.1.9 Axial Vibration

Axial vibrations occur along the axis of rotation or movement. They are often caused by imbalances or misalignments and can affect the performance and lifespan of bearings and shafts.

2.1.10 Lateral Vibration

Lateral vibrations occur perpendicular to the axis of rotation. These are commonly caused by rotor imbalances, misalignment, and bearing defects.

Understanding these types of vibrations and their causes is essential for designing robust industrial machinery and implementing effective maintenance strategies. Proper vibration analysis can lead to improved machine performance, reduced downtime, and enhanced safety in industrial operations.

2.2 Measuring Vibrations

Accurate measurement of vibrations is essential for analyzing and mitigating their effects on industrial machinery. Various methods and tools are employed to capture vibrational data, each suited for different types of vibrations and applications. Below are detailed descriptions of the primary methods for measuring vibrations.

2.2.1 Accelerometers

Definition: Accelerometers are sensors that measure the acceleration forces acting on an object. They are commonly used to capture vibrational data in three dimensions (X, Y, and Z axes).

Types:

- **Piezoelectric Accelerometers:** Generate an electric charge proportional to the applied acceleration.
- **Capacitive Accelerometers:** Measure changes in capacitance due to the movement of a microstructure within the sensor.
- **MEMS Accelerometers:** Utilize micro-electromechanical systems to detect acceleration changes.

Applications:

- Monitoring machine vibrations to detect imbalance, misalignment, and other mechanical issues.
- Used in handheld vibration meters for field measurements and in fixed installations for continuous monitoring.

Advantages:

- High sensitivity and accuracy.
- Capable of measuring a wide range of frequencies.
- Available in compact sizes, suitable for various applications.

2.2.2 Displacement Sensors

Definition: Displacement sensors measure the relative motion between two points. They are useful for capturing low-frequency vibrations and large displacements.

Types:

- **Linear Variable Differential Transformers (LVDTs):** Measure displacement by detecting changes in inductance.

- **Eddy Current Probes:** Measure displacement based on changes in inductance caused by the proximity of a conductive target.

Applications:

- Monitoring the movement of large machine components.
- Used in applications where precise measurement of displacement is critical, such as in turbine shafts and large motors.

Advantages:

- High accuracy for low-frequency vibrations.
- Suitable for measuring large displacements.

2.2.3 Velocity Sensors

Definition: Velocity sensors measure the speed of vibrating parts. They are often used in conjunction with accelerometers to provide a comprehensive analysis of vibrational behavior.

Types:

- **Electromagnetic Velocity Sensors:** Use the relative motion between a coil and a magnet to generate a voltage proportional to the velocity.

Applications:

- Monitoring the vibrational velocity of machine components.
- Used in predictive maintenance to assess the condition of bearings and other critical parts.

Advantages:

- Direct measurement of vibrational velocity.
- Effective for mid-range frequency measurements.

2.2.4 Laser Doppler Vibrometers

Definition: Laser Doppler vibrometers use laser beams to measure the velocity and displacement of vibrating surfaces without physical contact.

Mechanism: A laser beam is directed at the vibrating surface, and the frequency shift of the reflected light (Doppler effect) is measured to determine the velocity of the surface.

Applications:

- Non-contact measurement of vibrations in delicate or hard-to-reach components.
- Used in research and development for precise vibrational analysis.

Advantages:

- High precision and accuracy.
- Non-contact measurement, suitable for sensitive or inaccessible components.

2.2.5 Summary

Selecting the appropriate method for measuring vibrations depends on the specific application and the type of vibrations being analyzed. Accelerometers, displacement sensors, velocity sensors, laser Doppler vibrometers, and FFT analyzers each offer unique advantages and are suited for different measurement needs. By employing these tools, it is possible to gain a comprehensive understanding of vibrational behavior, enabling effective maintenance and design improvements in industrial machinery.

2.3 Damping Systems Present

To effectively manage and reduce vibrations in industrial machinery, various damping systems are employed. These systems are categorized based on their operating principles and mechanisms. Below are detailed descriptions of the primary types of damping systems used in industrial applications.

2.3.1 Passive Damping Systems

Passive damping systems rely on materials and structural design to absorb and dissipate vibrational energy. These systems are simple, reliable, and do not require external power sources.

Types of Passive Damping Systems

a) Viscoelastic Materials

- **Mechanism:** Viscoelastic materials exhibit both viscous and elastic properties. When subjected to vibration, these materials deform and dissipate energy through internal friction.
- **Applications:** Used in vibration dampers, pads, and mounts in various industrial equipment.
- **Advantages:** Simple to implement, low cost, and effective for a wide range of frequencies.

b) Damping Pads and Mounts

- **Mechanism:** Pads and mounts made from rubber or other damping materials are placed between vibrating components to reduce transmitted vibrations.
- **Applications:** Commonly used in engines, compressors, and other machinery to isolate vibrations.
- **Advantages:** Easy to install, provide both vibration isolation and shock absorption.

c) Tuned Mass Dampers (TMDs)

- **Mechanism:** TMDs add a secondary mass that is tuned to the natural frequency of the vibrating system. The secondary mass oscillates out of phase with the primary mass, reducing resonance amplitudes.
- **Applications:** Used in buildings, bridges, and tall structures to reduce wind and seismic vibrations.
- **Advantages:** Highly effective at reducing resonance and improving structural stability.

d) Mechanical Damping Systems

- **Mechanism:** The spring component stores mechanical energy when subjected to vibrations. It compresses and expands, thereby reducing the amplitude of the vibrations. The damper component dissipates the stored energy, often through viscous friction or material deformation, converting it into heat and thereby damping the vibrations.
- **Applications:** Widely used in automotive suspensions to improve ride comfort and vehicle stability, applied in industrial machinery to minimize operational vibrations and enhance precision, and employed in civil engineering for structural vibration control in buildings and bridges.

- **Advantages:** Simple and robust design, does not require external power sources, making it reliable and easy to maintain, and effective over a broad range of frequencies, providing versatile vibration control.

2.3.2 Active Damping Systems

Active damping systems use sensors, actuators, and control algorithms to dynamically counteract vibrations. These systems require external power and are capable of adapting to changing conditions.

Types of Active Damping Systems

a) Piezoelectric Actuators

- **Mechanism:** Piezoelectric materials generate an electric charge when subjected to mechanical stress. These actuators convert electrical signals into mechanical motion to counteract vibrations.
- **Applications:** Used in precision machining, aerospace, and automotive industries.
- **Advantages:** High precision, fast response, and effective over a wide range of frequencies.

b) Electromagnetic Actuators

- **Mechanism:** These actuators use electromagnetic forces to counteract vibrations. An electromagnet creates a magnetic field that interacts with a moving component to generate a damping force.
- **Applications:** Used in active suspension systems, vibration control in machinery, and structural damping.
- **Advantages:** Adjustable damping characteristics, rapid response, and effective in controlling low-frequency vibrations.

c) Hydraulic Dampers

- **Mechanism:** Hydraulic dampers use hydraulic fluid to absorb and dissipate vibrational energy. The fluid is forced through a restricted passage, converting kinetic energy into heat.
- **Applications:** Common in heavy-duty industrial applications, such as construction equipment and large machinery.
- **Advantages:** High energy dissipation capacity, adjustable damping levels, and robust design.

2.3.3 Semi-Active Damping Systems

Semi-active damping systems combine features of passive and active systems, allowing for real-time adjustment of damping characteristics based on feedback from sensors.

Types of Semi-Active Damping Systems

a) Magnetorheological (MR) Dampers

- **Mechanism:** MR dampers use a magnetic field to change the viscosity of a magnetorheological fluid. By varying the magnetic field, the damping characteristics can be adjusted in real time.
- **Applications:** Used in automotive suspensions, seismic damping in buildings, and vibration control in machinery.
- **Advantages:** Rapid adjustment of damping characteristics, high reliability, and low power consumption.

b) Variable Stiffness Dampers

- **Mechanism:** These dampers adjust the stiffness of damping elements to optimize vibration suppression across different operating conditions. This is achieved through mechanical or hydraulic means.
- **Applications:** Used in adaptive structures, precision manufacturing, and aerospace.
- **Advantages:** Versatile, effective over a wide range of frequencies, and capable of adapting to varying load conditions.

2.3.4 Conclusion

Understanding and implementing the appropriate damping system is crucial for reducing vibrations and enhancing the performance and longevity of industrial machinery. Passive damping systems offer simplicity and reliability, while active damping systems provide precision and adaptability. Semi-active damping systems offer a balance between the two, providing real-time adjustment capabilities. By leveraging these damping systems, industries can significantly improve operational efficiency, reduce maintenance costs, and ensure the safety and stability of their machinery.

3 System Architecture

The system architecture of the Vibration Damping System for Machine Tools is designed to ensure effective vibration monitoring and mitigation. The architecture comprises two primary components: the Data Acquisition System and Processing, and the Spring and Damper System mitigation of vibrations. Below is a detailed overview of each component and how they work together to achieve the desired outcomes.

3.1 Data Acquisition System and Processing

The Data Acquisition System (DAQ) and Processing unit is a critical component of the Vibration Damping System for Machine Tools. It is designed to capture, process, and analyze vibrational data in real-time, enabling precise monitoring and effective mitigation of vibrations. This section provides a detailed description of the components involved, their functions, and the workflow.

3.1.1 Components and Workflow

a) Accelerometers (MPU6050)

Description:

The MPU6050 is a high-performance accelerometer and gyroscope sensor capable of measuring acceleration in three dimensions (X, Y, and Z axes). It is widely used in various applications for its reliability and precision.

Features:

- Measures accelerations along three axes.
- Integrated 16-bit ADCs for digital output.
- Inbuilt digital motion processing (DMP) engine for advanced calculations.

Placement:

The accelerometer is securely held in place by the Accelerometer Holding Part, designed to ensure stability and precise measurement. This design minimizes external interference and ensures accurate data capture.

b) Microcontroller Unit (MCU) - ATmega328P

Description:

The ATmega328P microcontroller collects data from the MPU6050 accelerometer, processes it, and temporarily stores it. It serves as the central processing unit for the DAQ system.

Function:

- Receives digital output from the accelerometer.
- Performs initial filtering and calculations to ensure data accuracy.
- Temporarily stores processed data for transmission.

c) Data Transmission Interface

Description:

The data transmission interface, featuring the CH340C USB-to-serial converter, facilitates communication between the MCU and a computer.

Function:

- Transfers processed data from the MCU to a computer, enabling further analysis.
- Ensures reliable data transfer.

d) Computer and MATLAB

Description:

The computer, equipped with MATLAB, receives vibrational data from the MCU. MATLAB's powerful computational tools are used to perform detailed analysis, including frequency analysis.

Features:

- Real-time data monitoring and analysis.
- Advanced computational capabilities for detailed vibrational analysis.

Function:

- Analyzes vibrational data to diagnose issues and make data-driven decisions for vibration mitigation.

3.1.2 Workflow Summary

The Data Acquisition System and Processing unit follows a systematic workflow to ensure precise capture, processing, and analysis of vibrational data.

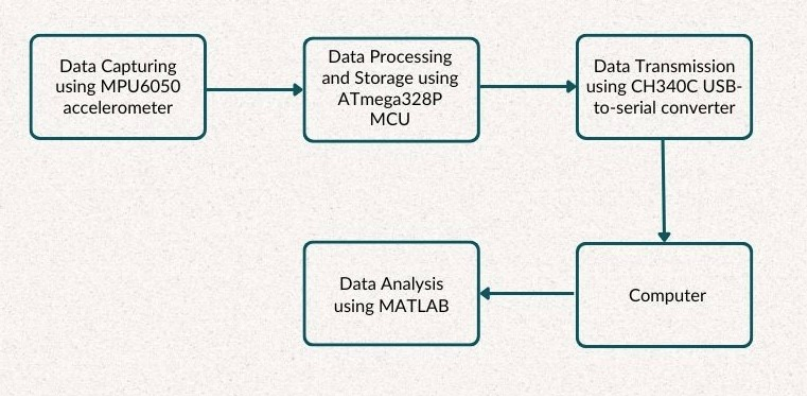


Figure 1: Workflow

Step 1: Data Capture The MPU6050 accelerometer, securely held in place by the Accelerometer Holding Part, captures vibrational data along the X, Y, and Z axes. The placement of the accelerometer ensures precise measurement of vibrations affecting the machine tool.

Step 2: Data Processing The ATmega328P MCU receives the digital output from the MPU6050 via the I2C communication interface. The MCU processes the raw vibrational data, performing initial filtering and calculations to ensure data accuracy and relevance.

Step 3: Data Storage The processed data is temporarily stored in the MCU's memory. This storage allows for real-time data handling and ensures that no data is lost during transmission.

Step 4: Data Transmission The CH340C USB-to-serial converter facilitates the transfer of processed data from the MCU to a computer. This interface ensures reliable and efficient data transmission, allowing for real-time monitoring and analysis.

Step 5: Data Analysis On the computer, the vibrational data is analyzed using MATLAB. MATLAB's powerful computational tools allow for detailed analysis, including frequency analysis, pattern recognition, and identification of vibration sources. The analysis results are used to diagnose issues and make data-driven decisions for vibration mitigation.

3.1.3 Advantages

1. **High Precision:** The use of the MPU6050 accelerometer, combined with the secure and stable Accelerometer Holding Part, ensures high-precision measurements of vibrational data.

2. **Robust Data Processing:** The ATmega328P MCU provides robust data processing capabilities, ensuring that the data captured is accurate and relevant for analysis.

3. **Reliable Data Transmission:** The CH340C USB-to-serial converter ensures reliable and efficient data transmission from the MCU to the computer, facilitating real-time monitoring.

4. **Comprehensive Data Analysis:** MATLAB's powerful analysis tools provide detailed insights into the vibrational behavior of the machine tool, enabling effective diagnosis and mitigation of vibration issues.

3.1.4 Conclusion

The Data Acquisition System and Processing unit is a vital part of the Vibration Damping System for Machine Tools. By capturing, processing, and analyzing vibrational data in real-time, this system ensures precise monitoring and effective mitigation of vibrations, enhancing the precision and longevity of machine tools. The combination of high-precision sensors, robust processing capabilities, and advanced analysis tools makes this system a reliable and efficient solution for vibration management.

3.2 Spring and Damper System

The Spring and Damper System is implemented to minimize vibrations and enhance the precision and longevity of machine tools. This passive damping system is designed to absorb and dissipate vibrational energy, reducing the impact of vibrations on the machine tool's performance. This section provides a detailed overview of its components, workflow, types of vibrations reducible, and its conclusion.

3.2.1 Components and Workflow

a) Springs

Function: Store mechanical energy when subjected to vibrations. They compress and expand, thereby reducing the amplitude of the vibrations.

Material: Made from the material that matches the calculated spring constant.

b) Dampers

Function: Dissipate the energy stored by the springs. This is usually done through viscous friction or material deformation, converting the vibrational energy into heat.

Type: Include a viscous fluid according to the calculated viscous coefficient.

c) Integration with Machine Tool

Placement: The spring and damper system is integrated into the machine tool's structure, strategically positioned to address the most significant sources of vibration.

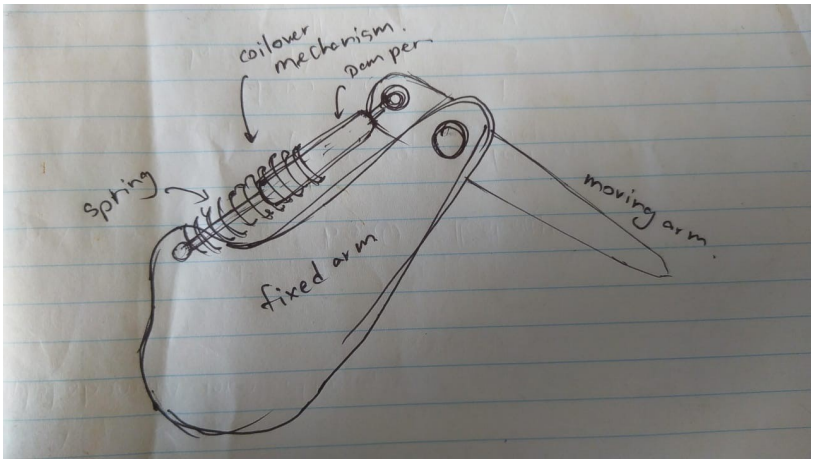


Figure 2: Proposed Structure

3.2.2 Workflow Summary

- Vibrations are absorbed by the springs, which compress and store the energy.
- The stored energy is dissipated by the dampers, reducing the amplitude of the vibrations.
- The overall effect is a significant reduction in vibrations, enhancing the precision and longevity of the machine tool.

3.2.3 Types of Vibrations Reducible Using Spring-Damper Systems

Spring-damper systems (also known as vibration dampers or isolators) are effective in reducing the following types of vibrations:

1. **Forced Vibration**
    - Rotational Imbalance: Absorb and reduce centrifugal forces caused by imbalances in rotating parts.
    - Misalignment: Mitigate vibrations caused by misalignment by absorbing periodic forces generated.
2. **Resonant Vibration**
    - Shift natural frequency away from operating frequency to avoid resonance. Absorb excessive energy and reduce the amplitude of vibrations.
3. **Transient Vibration**
    - Absorb shock loads during sudden events like startups or shutdowns, reducing transient vibrations.
4. **Steady-State Vibration**
    - Mitigate continuous vibrations from constant operating conditions by absorbing and dissipating vibrational energy.
5. **Axial and Lateral Vibration**
    - Reduce axial and lateral vibrations by isolating machinery from vibration sources and absorbing vibrational energy.

3.2.4 Conclusion

The Spring and Damper System is an essential part of the Vibration Damping System for Machine Tools. It provides a robust, reliable, and effective solution for minimizing vibrations, enhancing precision, and extending the lifespan of machine tools.

3.3 Conclusion

The system architecture of the Vibration Damping System for Machine Tools is designed to provide a comprehensive solution for vibration monitoring and mitigation. The Data Acquisition System and Processing component captures and analyzes vibrational data, while the Spring and Damper System effectively reduces vibrations. Together, these components ensure that the machine tools operate with enhanced precision and longevity, reducing maintenance costs and improving overall performance.

4 Electronic Design

The Electronic Design of the Vibration Damping System for Machine Tools encompasses the selection of components, schematic design, and PCB design. This section provides a detailed overview of each aspect, ensuring that the electronic system is robust, efficient, and reliable.

4.1 Component Selection and Justifications

4.1.1 MPU6050 Accelerometer

Justification: The MPU6050 is chosen for its high precision and reliability in measuring acceleration and gyroscopic data. Its integrated 16-bit ADCs and Digital Motion Processor (DMP) allow for accurate and real-time data capture, making it ideal for vibration monitoring.

Features:

- Measures acceleration in three dimensions (X, Y, and Z axes).
- Low power consumption.
- Integrated 16-bit analog-to-digital converters (ADCs).

4.1.2 ATMEGA 2560 Microcontroller

Justification: The ATMEGA 2560 is selected for its adequate memory, processing speed, and overall reliability, making it an ideal central processing unit for handling the data processing needs of the Vibration Damping System.

Key Features:

- Memory: 256 KB of Flash memory, 8 KB of SRAM, and 4 KB of EEPROM, providing sufficient space for handling the initial stages of data processing.
- Processing Speed: Operates at a clock speed of 16 MHz, meeting the demands of real-time data collection and processing.
- Reliability: Known for its stable performance and extensive use in various applications, ensuring dependable operation.
- Versatility: Supports a wide range of peripherals and interfaces, adaptable to various components used in the project.
- Community Support: Extensive documentation and a large community of users provide valuable resources and support.
- Cost-Effective: Offers a good balance between performance and cost.
- Proven Track Record: Used in numerous successful projects, indicating its reliability and robustness.

4.1.3 CH340C USB-to-Serial Converter

Justification: The CH340C is chosen for its reliable communication capabilities and compatibility with various operating systems, ensuring smooth data transfer between the MCU and a computer.

Key Features:

- Reliable Communication: Provides a reliable interface for serial communication over USB.
- Compatibility: Compatible with a wide range of operating systems.
- Cost-Effective: Economical solution for USB to serial conversion.

4.1.4 16 MHz Crystal Oscillator

Justification: A 16 MHz crystal oscillator is used to provide the clock signal for the ATMEGA 2560, ensuring precise timing and stable operation.

Key Features:

- Precision Timing: Provides a stable and precise clock signal necessary for accurate MCU operation.
- Frequency Stability: Ensures consistent performance by maintaining a stable clock frequency.
- Compatibility: Matches the operational requirements of the ATMEGA 2560.

4.1.5 100nF Decoupling Capacitors

Justification: Used to stabilize the power supply to the micro controller and other components, ensuring a stable and noise-free operation.

Key Features:

- Noise Reduction: Filters out noise and prevents voltage spikes.
- Improved Performance: Enhances overall performance and reliability by maintaining a clean power signal.
- Protection: Protects sensitive components from transient voltage fluctuations.

4.2 Schematics Design

The schematic design for the overall system was developed using the Altium Designer EDA platform. To ensure an organized and efficient design process, we adopted a hierarchical design methodology, emphasizing modularity and abstraction where we identified four main sub-parts:

1. Sensor Input Circuit
2. Micro controller Circuit (ATmega328P)
3. USB Port Circuit (CH340C)
4. Power Supply Circuit (LM7805)

This approach allowed for clear separation of different functional blocks, making the design more manageable and easier to troubleshoot. We planned to use Surface Mount Devices. During the design process, we referred to component selection platforms such as Mouser and LCSC. This enabled seamless integration of component footprints directly into our design, ensuring that all selected components were compatible with the SMD process.

4.2.1 Key Steps in Schematic Design

Sensor Input Circuit

The sensor input circuit is designed to interface with accelerometers, capturing vibration data.

Micro controller Circuit (ATMEGA 2560)

Included a 16 MHz crystal oscillator for clocking, decoupling capacitors for noise reduction, and necessary I/O configurations.

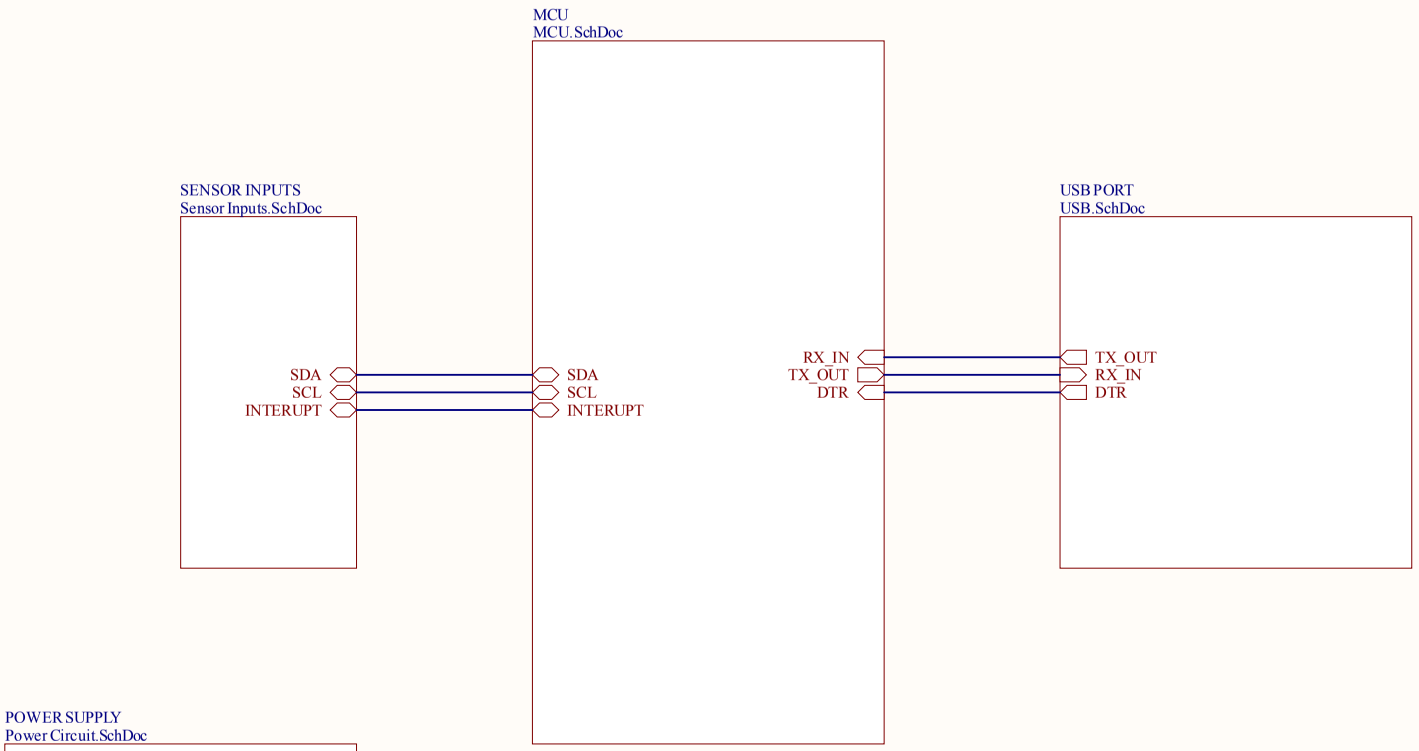
USB Communication Circuit

Designed with the CH340C USB-to-serial converter to facilitate reliable data transfer to a computer.

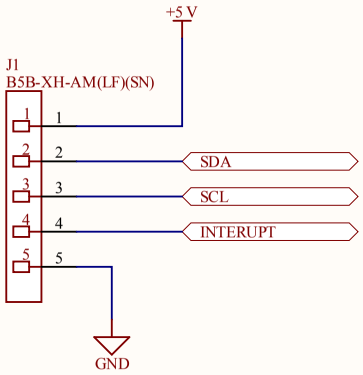
Power Supply Circuit

Designed to provide a stable 5V supply using the LM7805 voltage regulator and necessary filtering capacitors.

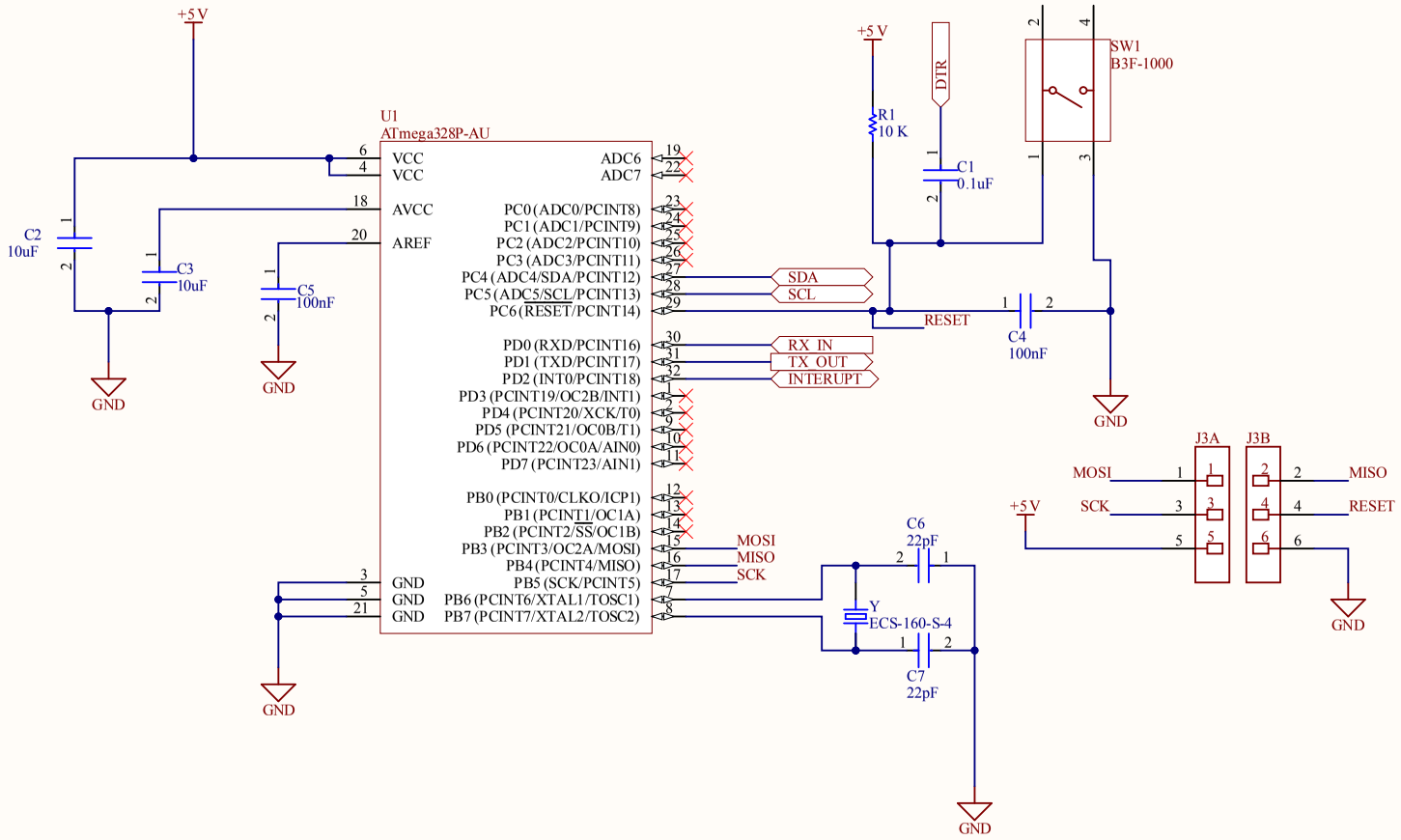
4.2.2 Schematic Diagrams



Title	Main		
Size: A4	Number: 1		Revision:
Date: 7/08/2024		Sheet 1 of 5	
File: D:\New folder (2)\..\Main.SchDoc		Drawn By: Achira Hansindu	

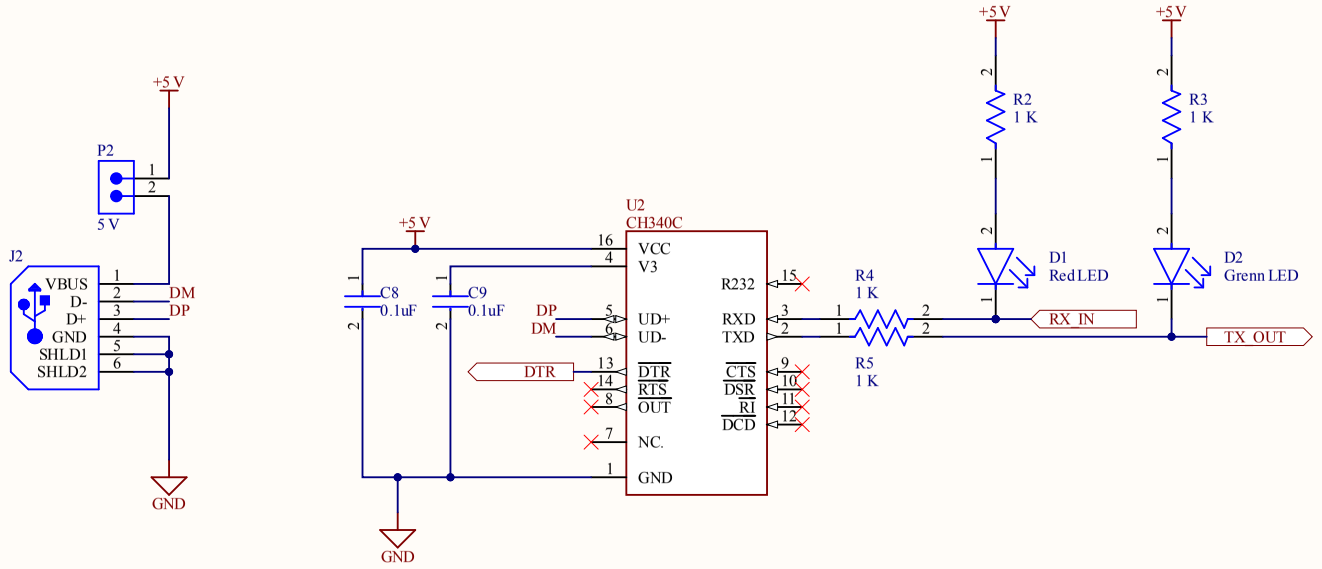


Title	Sensor Inputs		
Size: A4	Number: 2		Revision:
Date: 7/08/2024		Sheet 2 of 5	
File: D:\New folder (2)\..\Sensor Inputs.SchDoc		Drawn By: Achira Hansindu	

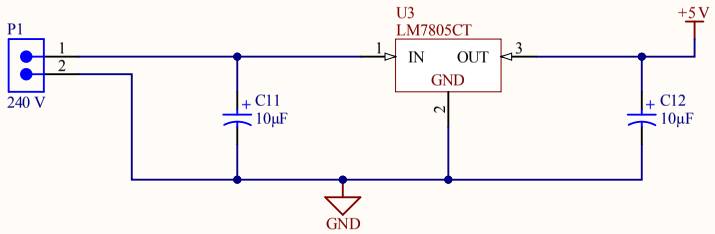


Decoupling capacitors(C2, C3, C5) should be as much as close to the IC

Title	Microcontroller		
Size: A4	Number: 3		Revision:
Date: 7/08/2024		Sheet 3 of 5	
File: D:\New folder (2)\..\MCU.SchDoc		Drawn By: Achira Hansindu	



Title	USB Port	
Size: A4	Number: 4	Revision:
Date: 7/08/2024		Sheet 4 of 5
File: D:\New folder (2)\..\USB.SchDoc		Drawn By: Achira Hansindu



Title	Power Supply		
Size: A4	Number: 5		Revision:
Date: 7/08/2024		Sheet 5 of 5	
File: D:\New folder (2)\..\Power Circuit.SchDoc		Drawn By: Achira Hansindu	

4.3 PCB Design

4.3.1 PCB Layout

The PCB layout integrates all schematic designs onto a single board, ensuring optimal component placement and efficient signal routing. Our design process prioritizes robustness, reliability, and ease of integration.

- Trace width: A trace width of 0.3 mm is chosen for signal traces, while a width of 1 mm is used for power components, ensuring adequate power delivery and signal integrity.
- PCB Size: 58.3 mm x 42.9 mm
- Component Placement: Components are strategically positioned to minimize trace lengths and optimize signal paths. Critical components such as the micro controller and ICs are centrally located to enhance stability and performance. Decoupling capacitors are placed close to the ICs to filter out noise and provide a stable power supply.
- Use of vias: Vias of sufficient diameter is used to transition traces between the top and bottom layers of the PCB.

The following pictures show different aspects of the PCB design.

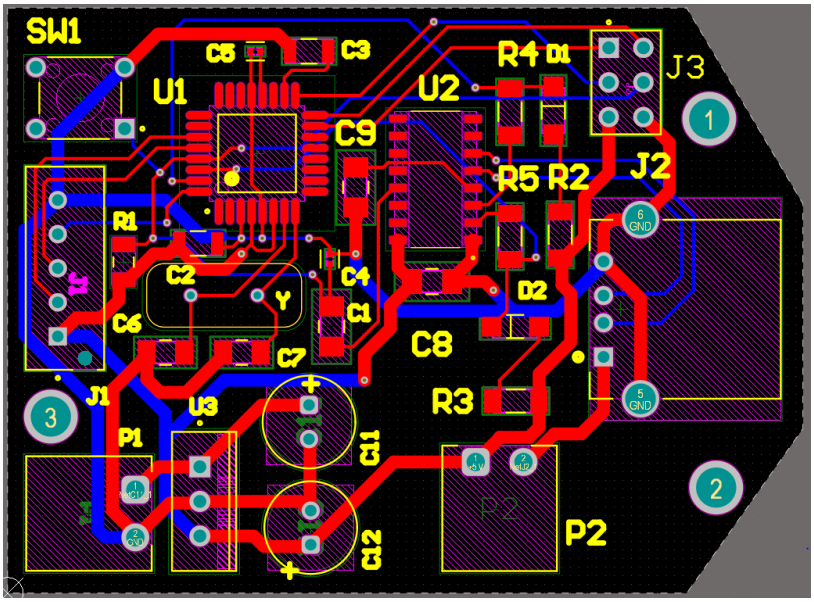


Figure 3: 2D View

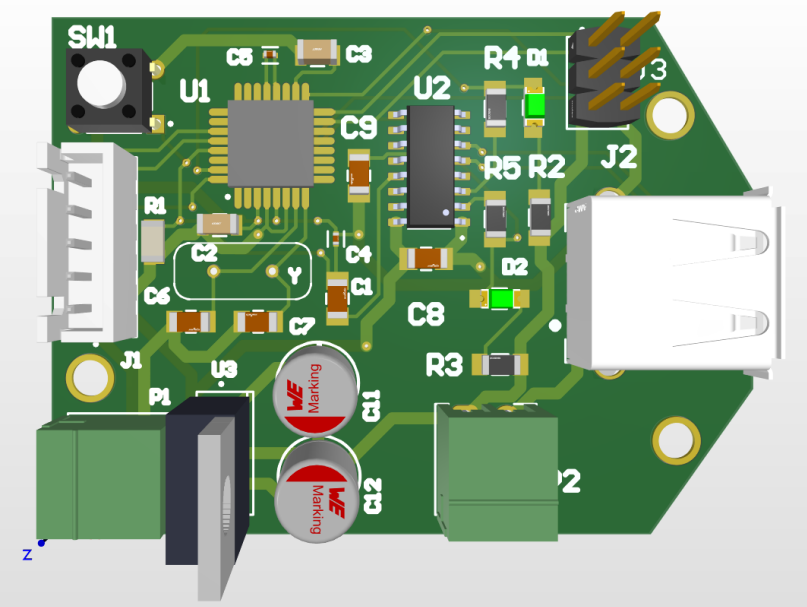


Figure 4: 3D View

4.3.2 Bare PCB

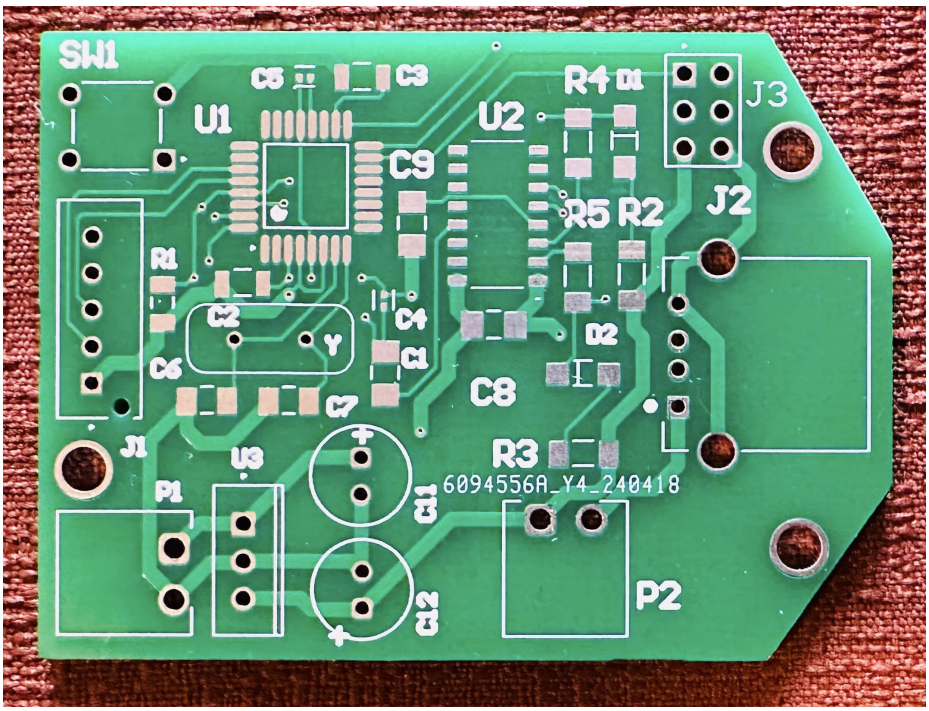


Figure 5: Top View - Bare PCB

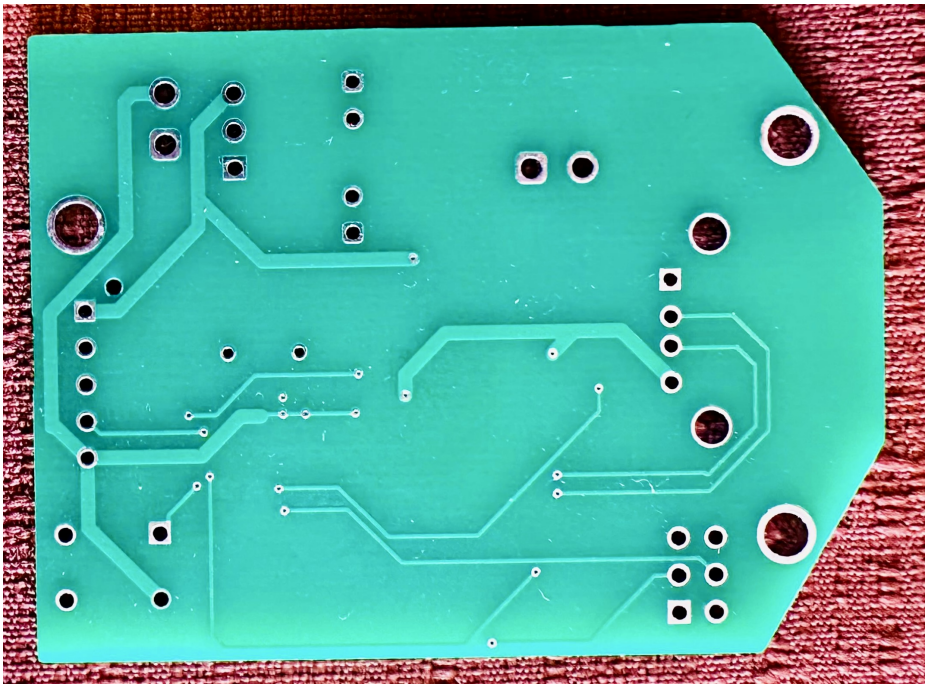


Figure 6: Bottom View - Bare PCB

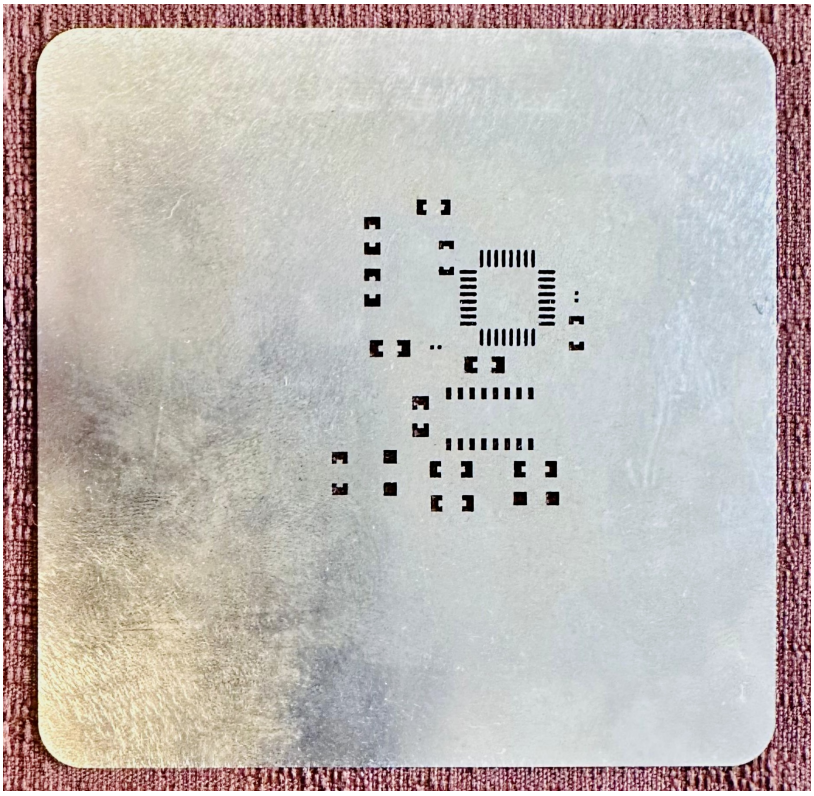


Figure 7: Stencil

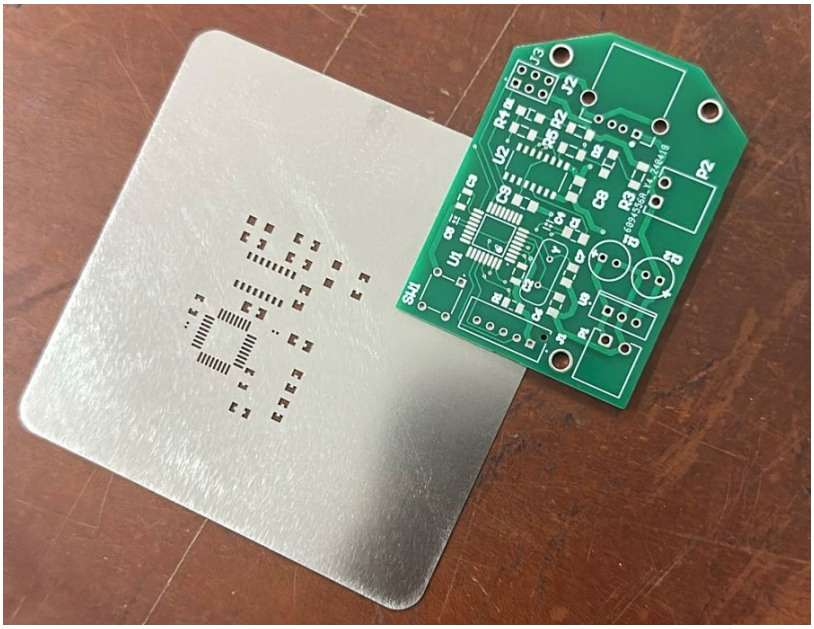


Figure 8: Manufactured PCB

4.3.3 Soldered PCB

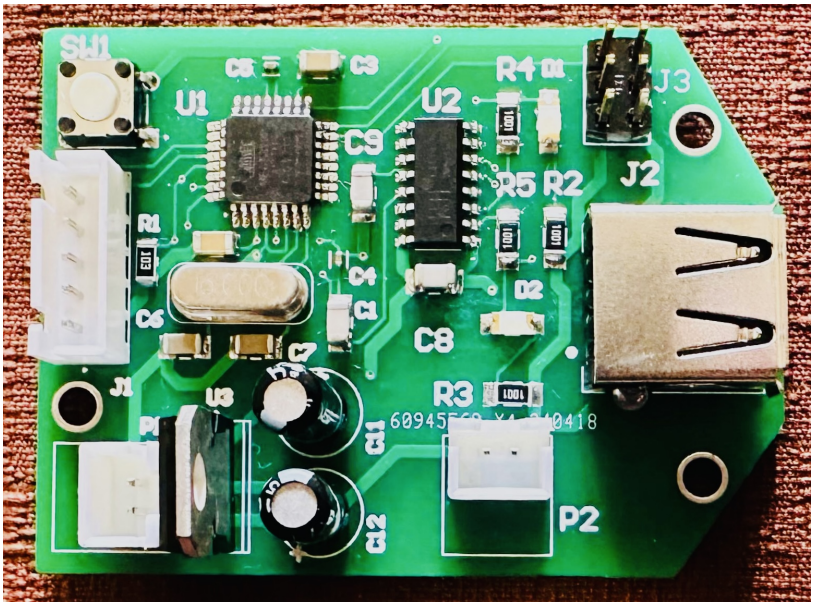


Figure 9: Top View - Soldered PCB

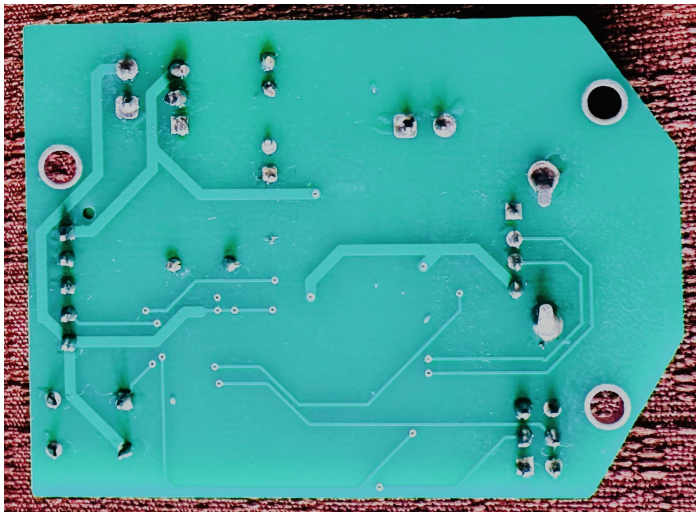


Figure 10: Bottom View - Soldered PCB

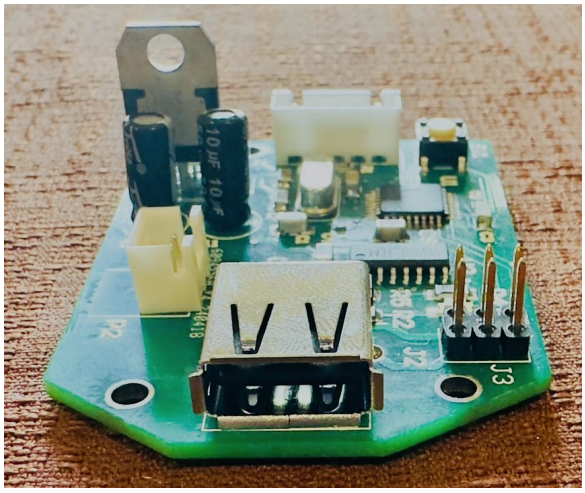


Figure 11: a. Side Views - Soldered PCB

Figure 12: b. Side Views - Soldered PCB

5 SolidWorks Design

Our SolidWorks design consists of two primary components: the Main Controller Unit and the Accelerometer Holding Part. Both designs are moldable and have been meticulously created to ensure functionality and ease of manufacturing. Below are the details and design trees for both components.

5.1 Overview and Specification

5.1.1 Main Controller Unit

The Main Controller Unit part is designed to house the main microcontroller and associated electronic components. The design includes a base and a lid, both moldable for efficient manufacturing.

5.1.2 Accelerometer Holding Part

The Accelerometer Holding Part is designed to securely hold the accelerometer in place while allowing for precise measurements. It consists of a base and a lid.

Both designs ensure that the components are securely housed and protected while allowing for efficient assembly and maintenance. The moldable design of these parts facilitates cost-effective manufacturing and ensures durability and reliability in their application.

5.2 Design Specifications

The design specifications for the enclosure are centered around the functional requirements and operational environment of the Vibration Damping System.

5.2.1 Dimensions and Compactness

- Compact Design: The enclosure is designed to be as compact as possible, ensuring minimal space consumption within the machine's tool environment.
- Dimensions:
  - Main Controller Unit – 130.3 mm x 87.0 mm x 35.0 mm
  - Accelerometer Holding Part – 40.0 mm x 36.0 mm x 24.0 mm

5.2.2 Lightweight

- Material Selection: Lightweight materials such as PLA+ (Polylactic Acid Plus) are chosen to reduce the overall weight of the system. A lightweight enclosure facilitates easier handling and installation.
- Weight: The initial enclosure weighs 42g.

5.2.3 Hardness

- Material Hardness: The enclosure materials are selected for their high hardness and durability. PLA+ (Polylactic Acid Plus) provides excellent resistance to mechanical impact and wear.
- Solid Contact: The enclosure must be as hard as possible to ensure there is no soft interface between the machine tools and the accelerometer. This hardness ensures that vibrations are not absorbed or dampened by the enclosure material itself, maintaining a solid, hard contact between the DAQ device and the machine surface.

5.2.4 Moldable

- Both the Main Controller Unit and Accelerometer Holding Part are designed to be moldable, facilitating mass production and ensuring consistency in quality.

5.3 Design Drawings

5.3.1 Main Controller Unit

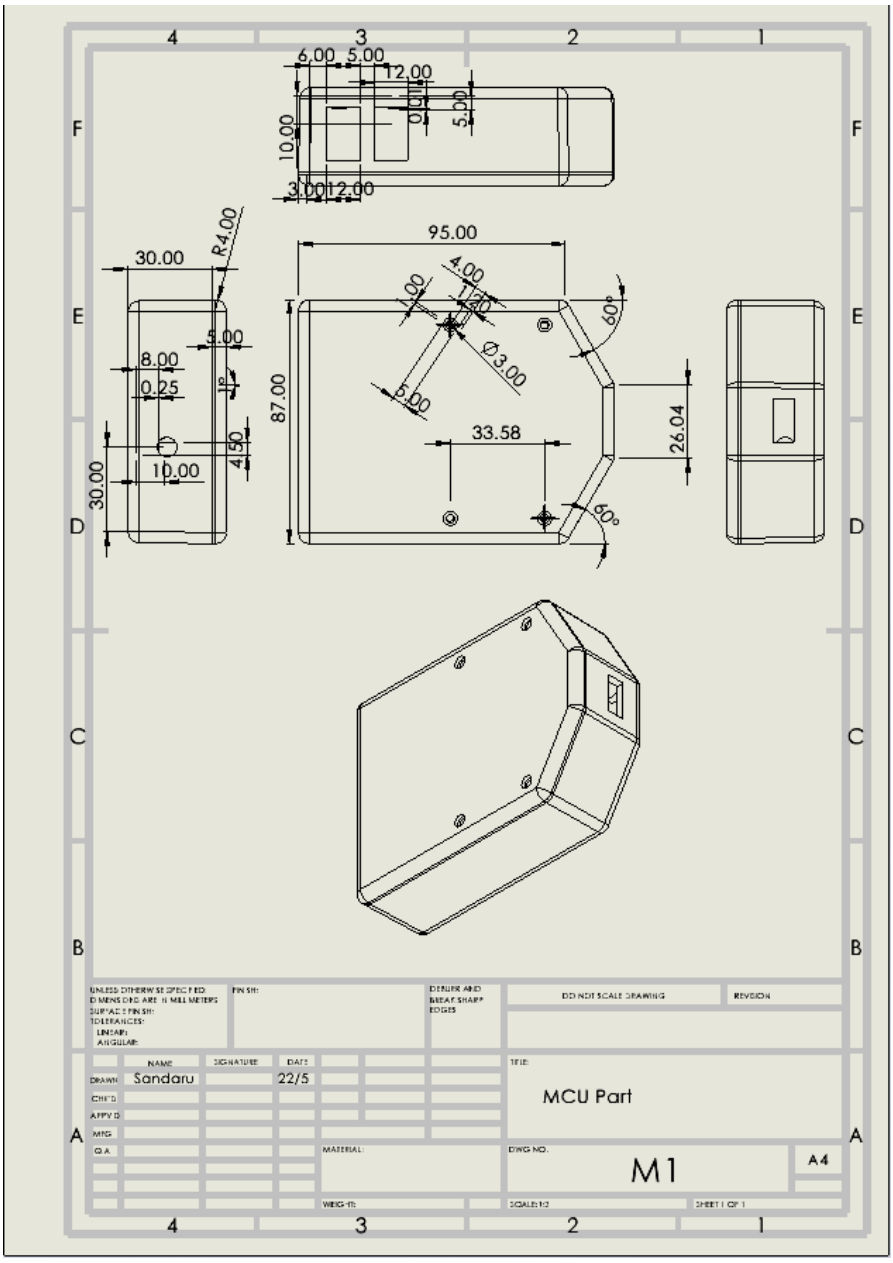


Figure 13: Main Controller Part-Design Drawing

5.3.2 Accelerometer Holding Part

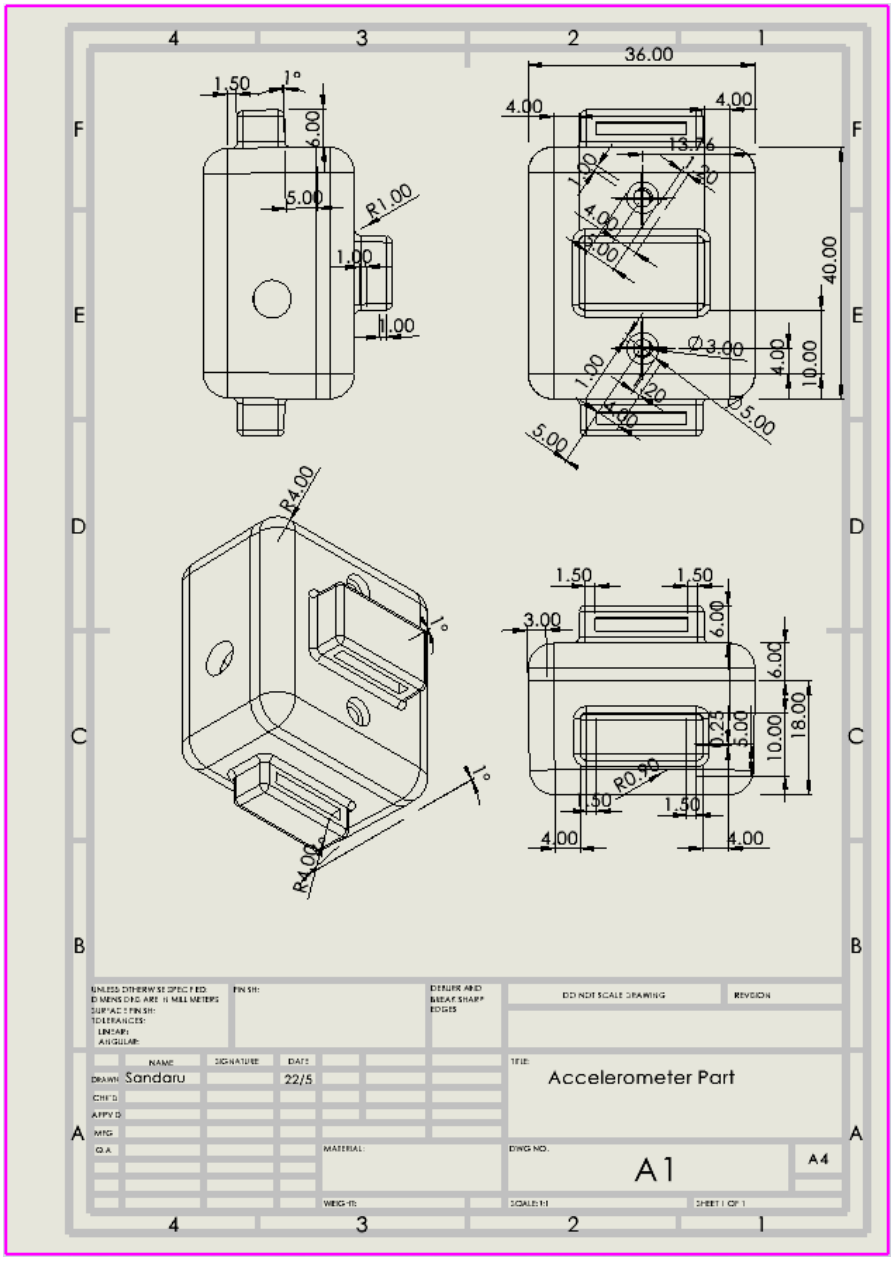


Figure 14: Accelerometer Holding Part-Design Drawing

5.4 3D Model of Enclosure Designs

The 3D model for the enclosure of the Vibration Damping System is designed using SolidWorks 2020, a common and parametric 3D CAD tool known for its robust product data management capabilities. This software allows for seamless updates to the design, ensuring that modifications can be made without compromising the initial specifications.

5.4.1 CAD Tool: SolidWorks 2020

Description: SolidWorks 2020 is utilized for designing the enclosure due to its advanced parametric modeling features and comprehensive product data management. This choice allows for efficient design iterations and ensures that any changes can be easily managed and tracked.

Advantages:

- Parametric Design: Facilitates easy modifications by adjusting design parameters.
- Product Data Management: Ensures consistency and traceability of design changes, maintaining the integrity of the original specifications.

5.4.2 Material: PLA+

Description: The enclosure is made from PLA+ (Polylactic Acid Plus), a durable and eco-friendly plastic known for its enhanced properties compared to standard PLA. It provides the necessary strength while being lightweight.

Specifications:

1. Weight: The initial enclosure weighs 42g
2. Dimensions:
   - Main Controller Unit – 130.3 mm x 87.0 mm x 35.0 mm
   - Accelerometer Holding Part – 40.0 mm x 36.0 mm x 24.0 mm
3. Draft Angles: The enclosure features 1-degree draft angles to facilitate easy removal from molds during manufacturing.
4. Thickness: The design maintains a minimum thickness of 3 mm, which is greater than the moldable thickness range for PLA+.

5.4.3 Main Controller Unit

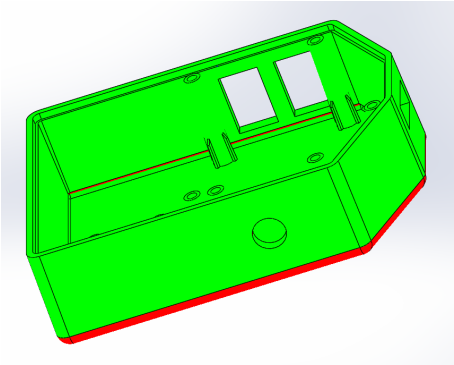


Figure 15: Draft Analysis-Base

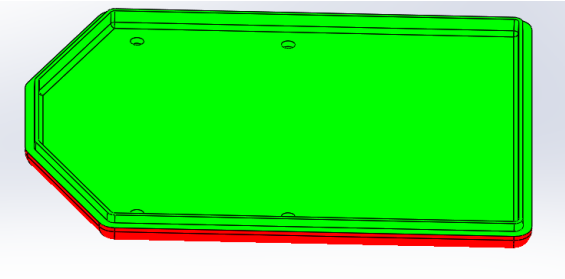


Figure 16: Draft Analysis-Lid

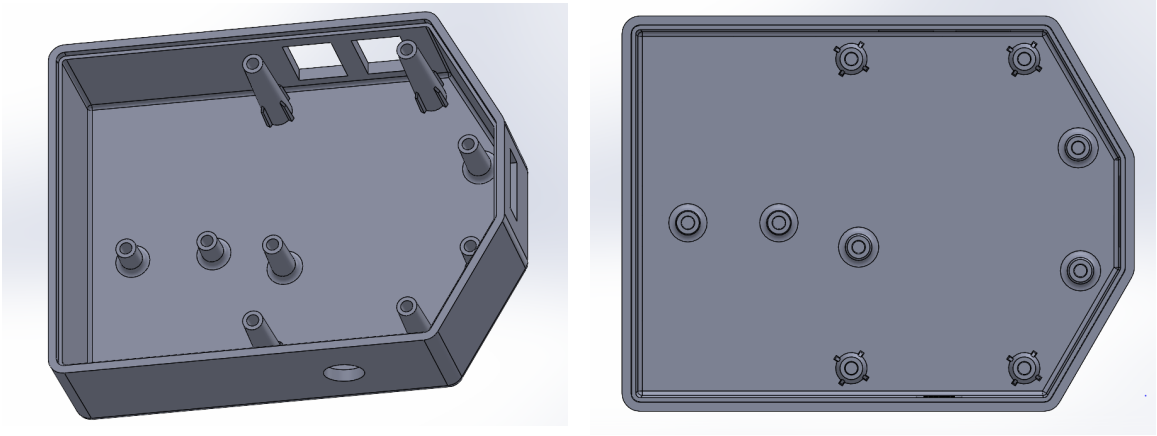


Figure 17: Main controller Unit Base

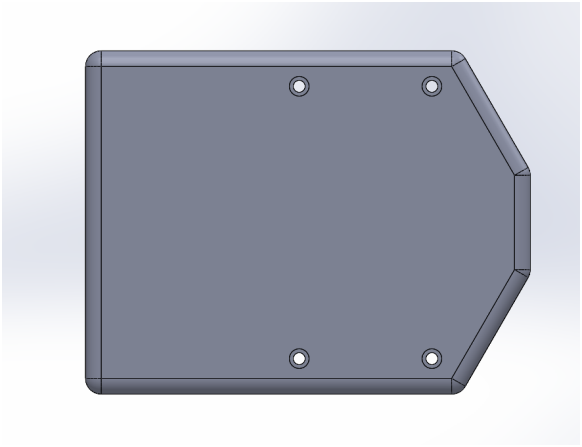


Figure 18: Main Controller Unit Lid

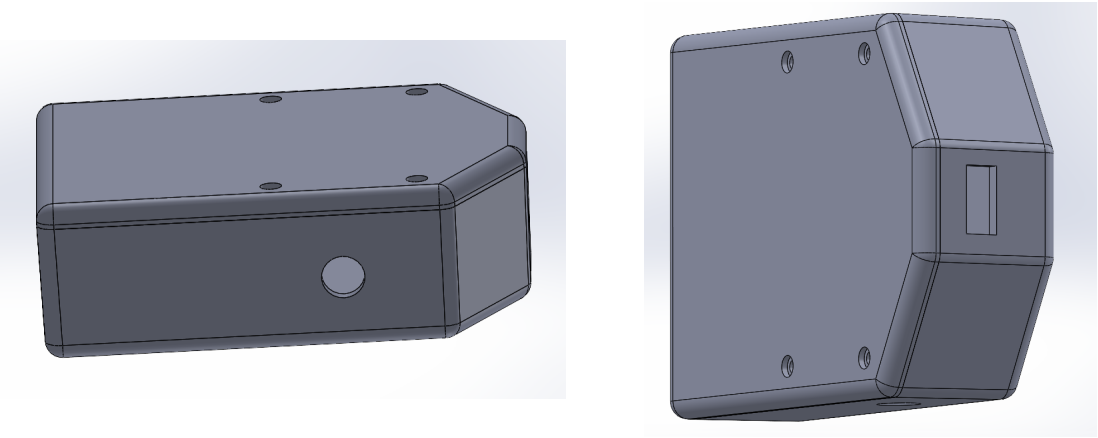


Figure 19: Main Controller Unit Assembly

5.4.4 Accelerometer Holding Part

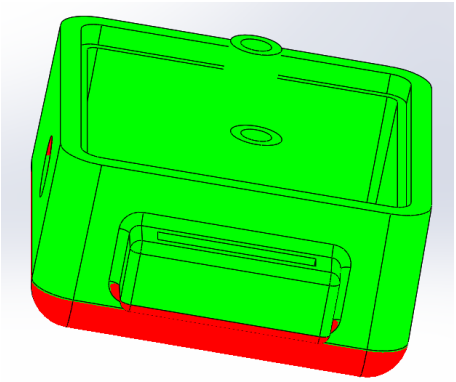


Figure 20: Draft Analysis-Base

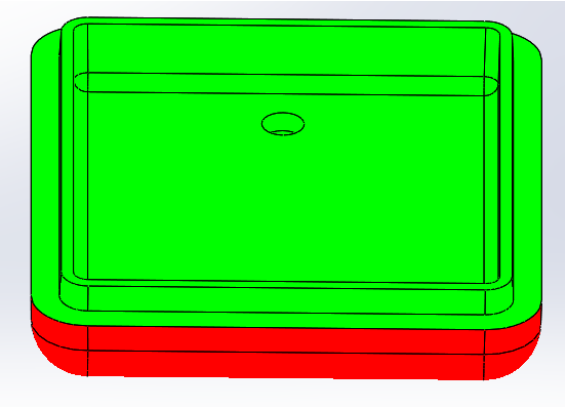


Figure 21: Draft Analysis-Lid

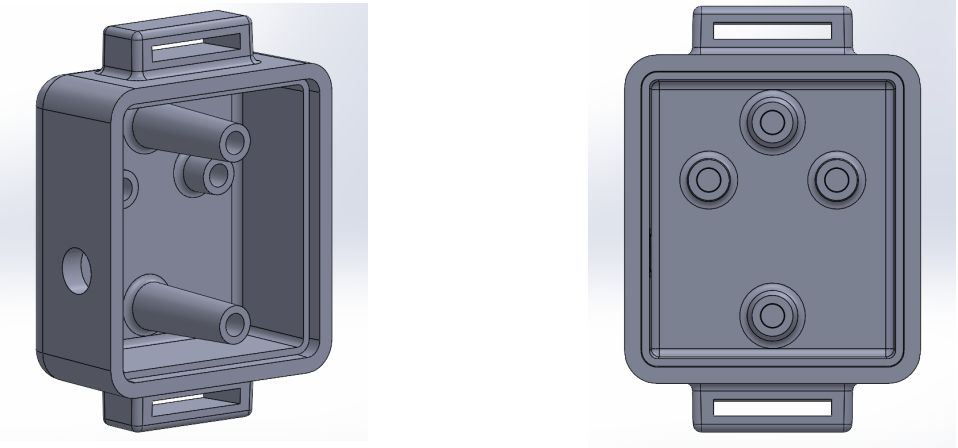


Figure 22: Accelerometer Holding Part Base

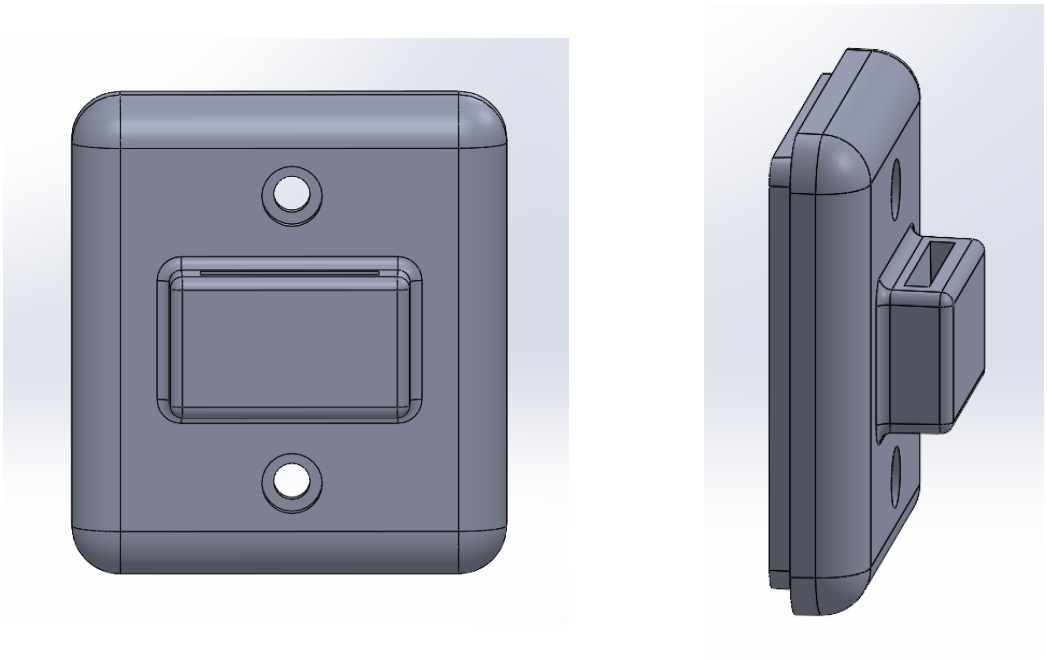


Figure 23: Accelerometer Holding Part Lid

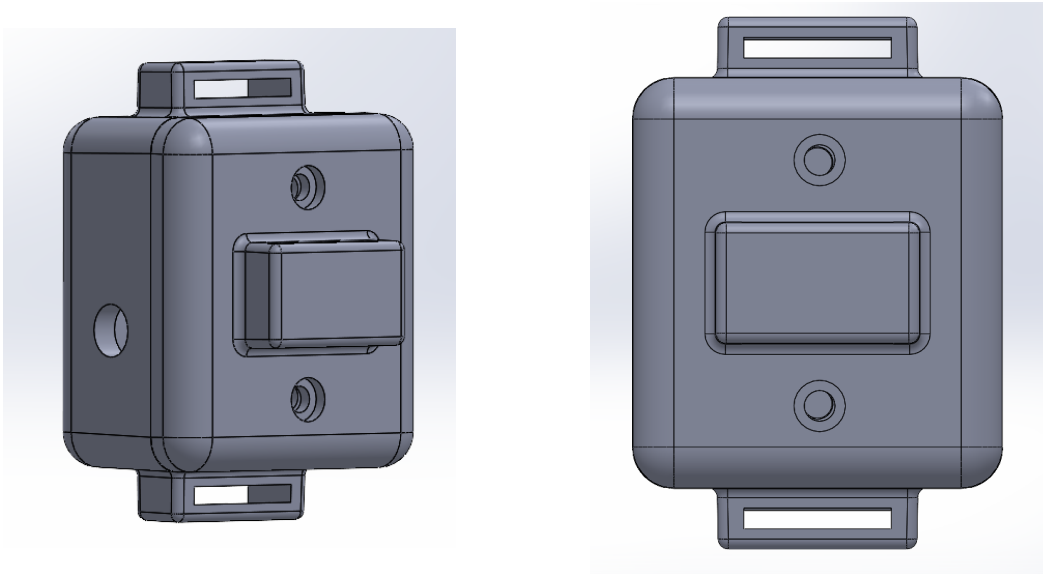


Figure 24: Accelerometer Holding Part Assembly

5.5 Printed Enclosure Design

5.5.1 Main Controller Unit

Figure 25: Inside View

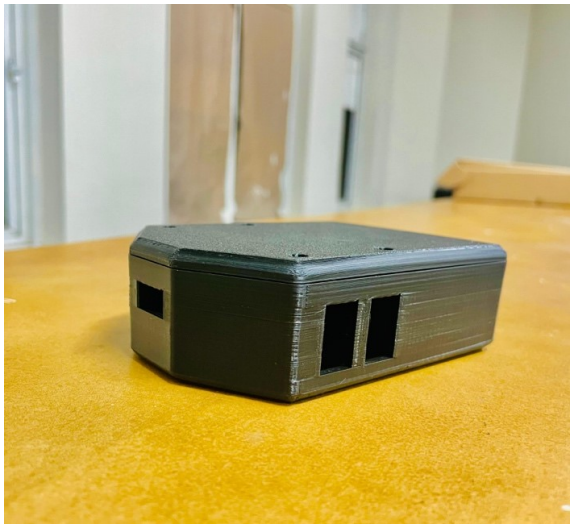


Figure 26: Front View

Figure 27: Top View

Figure 28: Side View

5.5.2 Accelerometer Unit

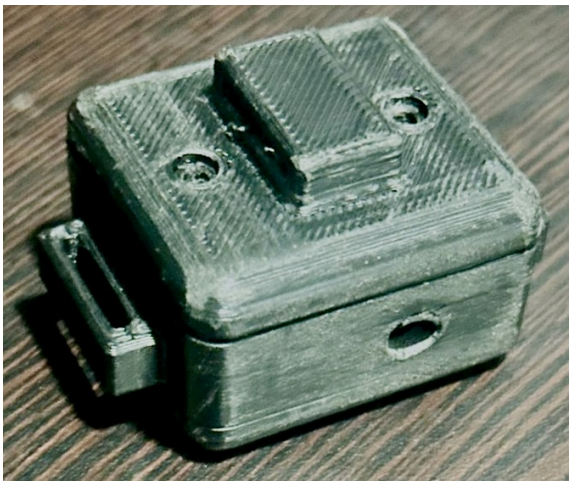


Figure 29: Side View

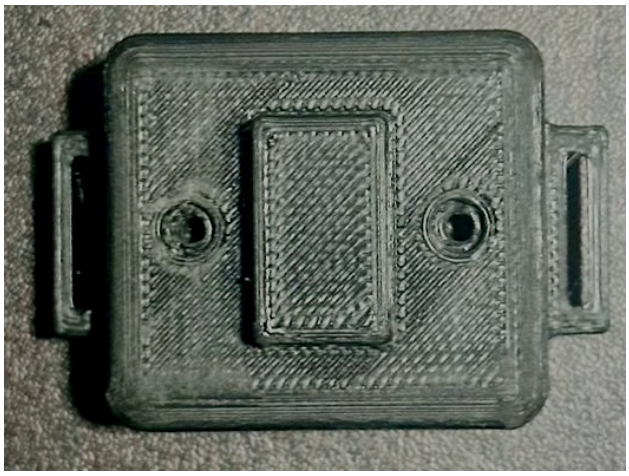


Figure 30: Top View

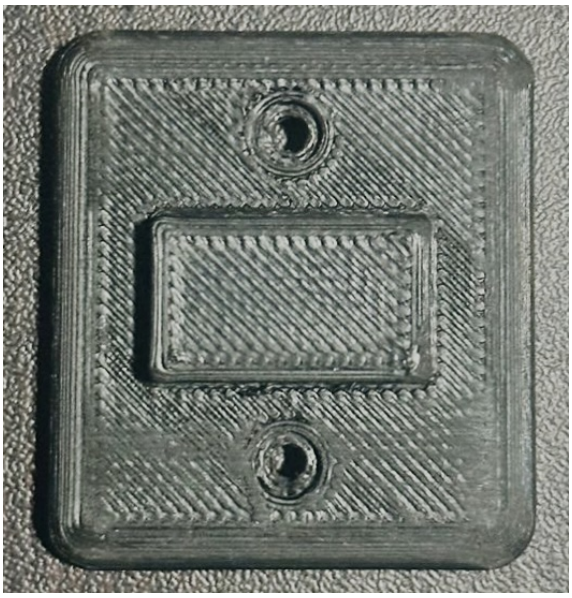


Figure 31: Top View - Lid

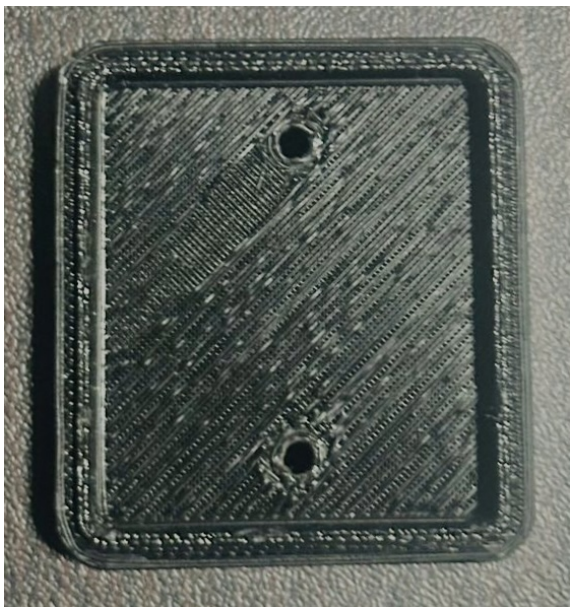


Figure 32: Bottom View - Lid

Figure 33: Side View - Base

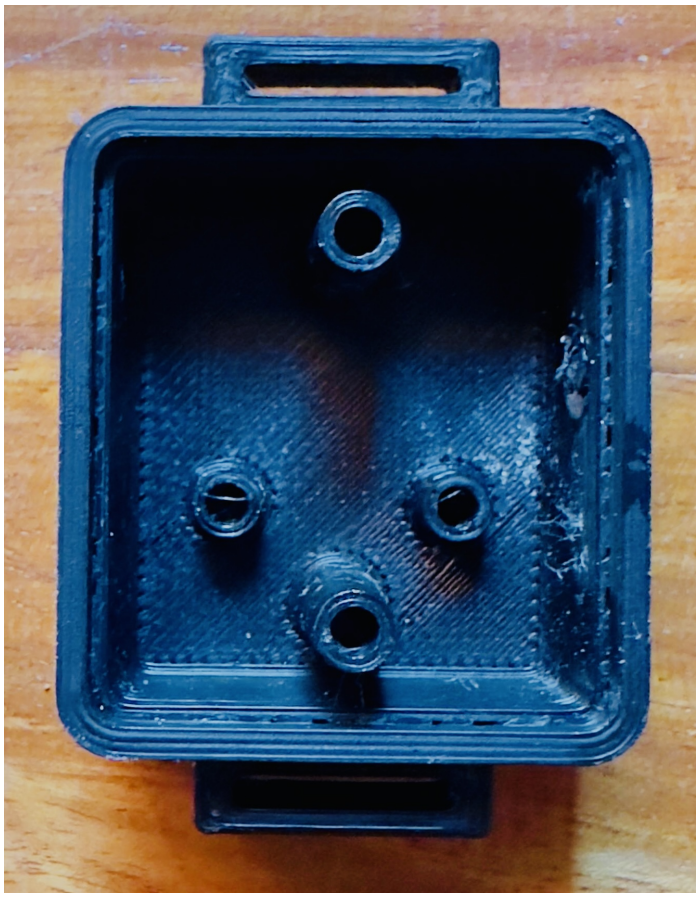


Figure 34: Top View - Base

6 Software Details

The software component of the Vibration Damping System for Machine Tools is integral to the successful acquisition, processing, and analysis of vibrational data. This section provides an overview of the software architecture, the code used for accelerometer readings, and the MATLAB code for data analysis.

6.1 Software Architecture and Overview

The software architecture leverages direct port manipulation and MATLAB to achieve seamless data acquisition and analysis. The ATmega328P microcontroller is programmed using C++ to interface with the MPU6050 accelerometer and capture vibrational data. This data is then transmitted to a computer where MATLAB is used for detailed analysis and calculations.

6.1.1 Microcontroller Programming

- The ATmega328P microcontroller is programmed using Microchip Studio which allows precise control over hardware resources and efficient data handling.
- Instead of relying on Arduino libraries, direct port manipulation is employed to communicate with the MPU6050 accelerometer, ensuring more control and customization of the data acquisition process.

6.1.2 MATLAB for Data Analysis

- **Data Transfer**: Captured data from the microcontroller is sent to a computer and imported into MATLAB.
- **Analysis and Calculations**: MATLAB is used to perform advanced calculations, visualize data, and verify the results. Its powerful computational capabilities ensure accurate analysis of the vibrational data.

6.2 Measuring Acceleration Using MPU6050 Accelerometer

The MPU6050 accelerometer is essential for measuring the vibrations in the vibration damping system. It records acceleration across three axes: X, Y, and Z. The ATmega328P microcontroller uses the I2C (TWI) protocol to communicate with the MPU6050 sensor. The collected data is processed to extract acceleration values in $m\,s^{-2}$, which is then saved into a CSV file for further analysis in MATLAB.

6.2.1 Code for Accelerometer Reading

The following `main.c` file is responsible for setting up the MPU6050 sensor, configuring Timer1 for periodic interrupts, reading the acceleration data, and transmitting it via UART for further processing.

```c
/**
 * @file MPU_Accel_Measurement.c
 * @brief Implements accelerometer data reading using MPU6050 and TWI.
 *
 * This file contains the main logic to initialize the MPU6050 sensor, configure
 * Timer1 for periodic interrupts, and read and process accelerometer data.
 *
 * @Achira Hansindu
 * @date 14-Aug-24
 * @version 1.0
 */


#include "uart.h"                 // Include UART library for serial communication
#include "twi_master.h"           // Include TWI (I2C) library for communication with MPU6050
#include <avr/io.h>               // Include AVR I/O library for accessing registers
#include <avr/interrupt.h>        // Include AVR interrupt library for handling interrupts
#include <stdio.h>                // Include standard I/O library for printf function

#define MPU_ADDRESS       0x68    // I2C address of the MPU6050 sensor
#define ACCEL_X_H         0x3B    // Register address for high byte of X-axis accelerometer data
#define ACCEL_X_L         0x3C    // Register address for low byte of X-axis accelerometer data
#define ACCEL_Y_H         0x3D    // Register address for high byte of Y-axis accelerometer data
#define ACCEL_Y_L         0x3E    // Register address for low byte of Y-axis accelerometer data
#define ACCEL_Z_H         0x3F    // Register address for high byte of Z-axis accelerometer data
#define ACCEL_Z_L         0x40    // Register address for low byte of Z-axis accelerometer data
#define PWR_MGMT          0x6B    // Register address for power management

typedef struct
{
  float x_axis;                 // Floating point variable to store X-axis acceleration
  float y_axis;                 // Floating point variable to store Y-axis acceleration
  float z_axis;                 // Floating point variable to store Z-axis acceleration
} accel_t;

```

```
volatile uint32_t millis_count = 0;    // Counter for milliseconds
volatile uint8_t sec_count = 0;        // Counter for seconds
volatile uint8_t min_count = 0;        // Counter for minutes
volatile uint8_t read_flag = 0;        // Flag to indicate that it's time to read data

// Function to configure Timer1 for 1ms intervals
void timer1_config(void)
{
  TCCR1B |= (1 << WGM12);              // Set Timer1 to CTC mode
  TIMSK1 |= (1 << OCIE1A);             // Enable interrupt on compare match
  OCR1A = 249;                         // Set compare value for 1ms interval
  TCCR1B |= (1 << CS11) | (1 << CS10); // Start Timer1 with prescaler 64
}

// Function to handle errors by checking return codes
void handle_error(ret_code_t code)
{
  if (code != SUCCESS)                 // If the code is not SUCCESS, there is an error
  {
    printf(BR "Error! Code = 0x%02X\n" RESET, code); // Print the error code
    while (1);                         // Infinite loop to halt execution on error
  }
}

// Function to initialize the MPU6050 sensor
void mpu_initialize(void)
{
  ret_code_t code;
  uint8_t config_data[2] = {PWR_MGMT, 0}; // Data to wake up the MPU6050 (write 0 to PWR_MGMT register)
  code = tw_master_transmit(MPU_ADDRESS, config_data, sizeof(config_data), false); // Send data over I2C
  handle_error(code);                  // Check if the transmission was successful
}

// Function to read raw accelerometer data from the MPU6050
void get_accel_raw(accel_t* accel_data)
{
  ret_code_t code;
  uint8_t data_buffer[6];              // Buffer to store the raw data from the sensor

  data_buffer[0] = ACCEL_X_H;          // Start reading from the X-axis high byte register
  code = tw_master_transmit(MPU_ADDRESS, data_buffer, 1, true); // Send register address
```

```
77    handle_error(code);               // Check if the transmission was
       successful
78
79    code = tw_master_receive(MPU_ADDRESS, data_buffer, sizeof(
       data_buffer)); // Read data from MPU6050
80    handle_error(code);               // Check if the reception was
       successful
81
82    // Combine high and low bytes and convert to floating-point values
83    accel_data->x_axis = (int16_t)(data_buffer[0] << 8 | data_buffer
       [1]) / 16384.0;
84    accel_data->y_axis = (int16_t)(data_buffer[2] << 8 | data_buffer
       [3]) / 16384.0;
85    accel_data->z_axis = (int16_t)(data_buffer[4] << 8 | data_buffer
       [5]) / 16384.0;
86  }
87
88  // Function to convert raw accelerometer data to physical units (m/s
       ^2)
89  void convert_accel_data(accel_t* accel_data)
90  {
91    get_accel_raw(accel_data);        // Call function to get raw data
92    accel_data->x_axis *= 9.81;       // Convert X-axis data to m/s^2
93    accel_data->y_axis *= 9.81;       // Convert Y-axis data to m/s^2
94    accel_data->z_axis *= 9.81;       // Convert Z-axis data to m/s^2
95  }
96
97  // Function to format a timestamp as a string
98  void format_timestamp(char *buffer)
99  {
100   uint32_t milliseconds;
101   uint8_t seconds, minutes, hours;
102
103   cli();                            // Disable interrupts to prevent
       data inconsistency
104   milliseconds = millis_count;      // Get current milliseconds count
105   seconds = sec_count;              // Get current seconds count
106   minutes = min_count;              // Get current minutes count
107   sei();                            // Re-enable interrupts
108
109   hours = minutes / 60;             // Calculate hours from minutes
110   minutes = minutes % 60;           // Calculate remaining minutes
111
112   // Format the timestamp as "HH.MM.SS.mmm"
113   sprintf(buffer, "%02u.%02u.%02u.%03lu", hours, minutes, seconds,
       milliseconds % 1000);
114 }
115
116 // Interrupt Service Routine (ISR) for Timer1 Compare Match A
117 ISR(TIMER1_COMPA_vect)
118 {
119   millis_count++;                   // Increment milliseconds counter
120
```

```
121   if (millis_count >= 1000) {       // If 1000 milliseconds have
       passed
122     millis_count = 0;             // Reset milliseconds counter
123     sec_count++;                  // Increment seconds counter
124
125     if (sec_count >= 60) {        // If 60 seconds have passed
126       sec_count = 0;            // Reset seconds counter
127       min_count++;              // Increment minutes counter
128
129       if (min_count >= 60) {  // If 60 minutes have passed
130         min_count = 0;      // Reset minutes counter (one hour passed
       )
131       }
132     }
133   }
134
135   read_flag = 1;                    // Set flag to indicate it's time
       to read data
136 }
137
138 int main(void)
139 {
140   uart_init(250000);                // Initialize UART with a baud
       rate of 250000
141   cli_reset();                      // Clear interrupt flag and reset
       configuration
142   puts(BY "MPU Accelerometer Measurement Started...\n" RESET); //
       Print start message
143
144   tw_init(TW_FREQ_400K, true);      // Initialize TWI (I2C) with 400
       kHz frequency and internal pull-up resistors
145   mpu_initialize();                 // Initialize the MPU6050 sensor
146   timer1_config();                  // Configure Timer1 for timing
       operations
147   sei();                            // Enable global interrupts
148
149   accel_t accel_data;               // Create a variable to store
       accelerometer data
150   char time_buffer[20];             // Buffer to store the timestamp
151
152   puts(BG CURSOR_RIGHT("14")
153   "--------------- Application Running ---------------\n" RESET); //
       Print running message
154
155   while (1)
156   {
157     if (read_flag)                // Check if the read_flag is set
158     {
159       read_flag = 0;            // Clear the read_flag
160       convert_accel_data(&accel_data); // Convert raw accelerometer
       data to physical units
161
162       // Print the accelerometer data
163       printf("AX %5.2f ", accel_data.x_axis);
```

```
164         printf("AY %5.2f ", accel_data.y_axis);
165         printf("AZ %5.2f ", accel_data.z_axis);
166 
167         format_timestamp(time_buffer);  // Create a formatted
    timestamp
168         printf("%s\n", time_buffer);    // Print the timestamp
169       }
170     }
171 }
```

Listing 1: Main C file for reading accelerometer data from the MPU6050 using the ATmega328P microcontroller.

The data is captured and processed periodically using interrupts generated by Timer1. The I2C communication is handled by the `twi_master.c` file, and UART transmission is handled by `uart.c`, which are detailed below:

6.2.2 I2C Communication - `twi_master.c`

This file implements the I2C protocol for communicating with the MPU6050 sensor to read the accelerometer data. It provides functions for initiating, reading, and writing over the I2C bus.

```
1  /*
2  * twi_master.c
3  *
4  * Created: 09-Jun-19 11:20:17 AM
5  *  Author: TEP SOVICHEA
6  */
7 
8  #include "twi_master.h"
9 
10 static ret_code_t tw_start(void)
11 {
12   /* Send START condition */
13 #if DEBUG_LOG
14   printf(BG "Send START condition..." RESET);
15 #endif
16   TWCR =  (1 << TWINT) | (1 << TWEN) | (1 << TWSTA);
17 
18   /* Wait for TWINT flag to set */
19   while (!(TWCR & (1 << TWINT)));
20 
21   /* Check error */
22   if (TW_STATUS != TW_START && TW_STATUS != TW_REP_START)
23   {
24 #if DEBUG_LOG
25     printf("\n");
26 #endif
27     return TW_STATUS;
28   }
29 
30 #if DEBUG_LOG
31   printf("SUCCESS\n");
```

```
#endif
  return SUCCESS;
}


static void tw_stop(void)
{
  /* Send STOP condition */
#if DEBUG_LOG
  puts(BG "Send STOP condition." RESET);
#endif
  TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWSTO);
}


static ret_code_t tw_write_sla(uint8_t sla)
{
  /* Transmit slave address with read/write flag */
#if DEBUG_LOG
  printf(BG "Write SLA + R/W: 0x%02X..." RESET, sla);
#endif
  TWDR = sla;
  TWCR = (1 << TWINT) | (1 << TWEN);

  /* Wait for TWINT flag to set */
  while (!(TWCR & (1 << TWINT)));
  if (TW_STATUS != TW_MT_SLA_ACK && TW_STATUS != TW_MR_SLA_ACK)
  {
#if DEBUG_LOG
    printf("\n");
#endif
    return TW_STATUS;
  }

#if DEBUG_LOG
  printf("SUCCESS\n");
#endif
  return SUCCESS;
}


static ret_code_t tw_write(uint8_t data)
{
  /* Transmit 1 byte*/
#if DEBUG_LOG
  printf(BG "Write data byte: 0x%02X..." RESET, data);
#endif
  TWDR = data;
  TWCR = (1 << TWINT) | (1 << TWEN);

  /* Wait for TWINT flag to set */
  while (!(TWCR & (1 << TWINT)));
  if (TW_STATUS != TW_MT_DATA_ACK)
  {
```

```
86 #if DEBUG_LOG
87     printf("\n");
88 #endif
89     return TW_STATUS;
90   }
91 
92 #if DEBUG_LOG
93   printf("SUCCESS\n");
94 #endif
95   return SUCCESS;
96 }
97 
98 
99 static uint8_t tw_read(bool read_ack)
100 {
101   if (read_ack)
102   {
103     TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWEA);
104     while (!(TWCR & (1 << TWINT)));
105     if (TW_STATUS != TW_MR_DATA_ACK)
106     {
107       return TW_STATUS;
108     }
109   }
110   else
111   {
112     TWCR = (1 << TWINT) | (1 << TWEN);
113     while (!(TWCR & (1 << TWINT)));
114     if (TW_STATUS != TW_MR_DATA_NACK)
115     {
116       return TW_STATUS;
117     }
118   }
119   uint8_t data = TWDR;
120 #if DEBUG_LOG
121   printf(BG "Read data byte: 0x%02X\n" RESET, data);
122 #endif
123   return data;
124 }
125 
126 
127 void tw_init(twi_freq_mode_t twi_freq_mode, bool pullup_en)
128 {
129   DDRC  |= (1 << TW_SDA_PIN) | (1 << TW_SCL_PIN);
130   if (pullup_en)
131   {
132 #if DEBUG_LOG
133     puts(BG "Enable pull-up resistor." RESET);
134 #endif
135     PORTC |= (1 << TW_SDA_PIN) | (1 << TW_SCL_PIN);
136   }
137   else
138   {
139     PORTC &= ~((1 << TW_SDA_PIN) | (1 << TW_SCL_PIN));
```

```
140   }
141   DDRC  &= ~((1 << TW_SDA_PIN) | (1 << TW_SCL_PIN));
142
143   switch (twi_freq_mode)
144   {
145     case TW_FREQ_100K:
146     /* Set bit rate register 72 and prescaler to 1 resulting in
147     SCL_freq = 16MHz/(16 + 2*72*1) = 100KHz */
148     TWBR = 72;
149     break;
150
151     case TW_FREQ_250K:
152     /* Set bit rate register 24 and prescaler to 1 resulting in
153     SCL_freq = 16MHz/(16 + 2*24*1) = 250KHz */
154     TWBR = 24;
155     break;
156
157     case TW_FREQ_400K:
158     /* Set bit rate register 12 and prescaler to 1 resulting in
159     SCL_freq = 16MHz/(16 + 2*12*1) = 400KHz */
160     TWBR = 12;
161     break;
162
163     default: break;
164   }
165 }
166
167
168 ret_code_t tw_master_transmit(uint8_t slave_addr, uint8_t* p_data,
        uint8_t len, bool repeat_start)
169 {
170   ret_code_t error_code;
171
172   /* Send START condition */
173   error_code = tw_start();
174   if (error_code != SUCCESS)
175   {
176     return error_code;
177   }
178
179   /* Send slave address with WRITE flag */
180   error_code = tw_write_sla(TW_SLA_W(slave_addr));
181   if (error_code != SUCCESS)
182   {
183     return error_code;
184   }
185
186   /* Send data byte in single or burst mode */
187   for (int i = 0; i < len; ++i)
188   {
189     error_code = tw_write(p_data[i]);
190     if (error_code != SUCCESS)
191     {
192       return error_code;
```

```
193     }
194   }
195
196   if (!repeat_start)
197   {
198     /* Send STOP condition */
199     tw_stop();
200   }
201
202   return SUCCESS;
203 }
204
205
206 ret_code_t tw_master_receive(uint8_t slave_addr, uint8_t* p_data,
      uint8_t len)
207 {
208   ret_code_t error_code;
209
210   /* Send START condition */
211   error_code = tw_start();
212   if (error_code != SUCCESS)
213   {
214     return error_code;
215   }
216
217   /* Write slave address with READ flag */
218   error_code = tw_write_sla(TW_SLA_R(slave_addr));
219   if (error_code != SUCCESS)
220   {
221     return error_code;
222   }
223
224   /* Read single or multiple data byte and send ack */
225   for (int i = 0; i < len-1; ++i)
226   {
227     p_data[i] = tw_read(TW_READ_ACK);
228   }
229   p_data[len-1] = tw_read(TW_READ_NACK);
230
231   /* Send STOP condition */
232   tw_stop();
233
234   return SUCCESS;
235 }
```

Listing 2: I2C communication functions in twi_master.c.

6.2.3 UART Communication - `uart.c`

This file contains the implementation for UART communication, allowing the ATmega328P to send accelerometer data to a computer or external device for visualization or logging. Below is the code for reading accelerometer data using the MPU6050.

```
/*
* ----------------------------------------------------------------------------
* "THE BEER-WARE LICENSE" (Revision 42):
* <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
* can do whatever you want with this stuff. If we meet some day, and you think
* this stuff is worth it, you can buy me a beer in return.
* Joerg Wunsch
* ----------------------------------------------------------------------------
*
* Stdio demo, UART implementation
*
* $Id: uart.c,v 1.1 2005/12/28 21:38:59 joerg_wunsch Exp $
*
* Mod for mega644 BRL Jan2009
* Mod for mega328p  TS  Feb2019
*/


#include "uart.h"

char *cp, *cp2;
char b[RX_BUFSIZE];

/*
* Setup standard io stream to use UART functions
*/

FILE uart_io = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

/*
* Initialize the UART to tx/rx, 8N1.
*/
void uart_init(uint32_t baudrate)
{
  stdout = stdin = &uart_io;

  float ubrr_tmp = (float)F_CPU/16.0/baudrate-1;
  uint16_t ubrr = round(ubrr_tmp);

  UBRR0H = ubrr >> 8;
  UBRR0L = ubrr;

  /* Enable double speed */
  //UCSR0A |= (1 << U2X0);

  /* Enable receiver and transmitter */
  UCSR0B |= (1 << RXEN0) | (1 << TXEN0);
```

```
47
48    /* Set frame format: 8 data bits, parity None, 1 stop bit */
49    UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00);
50
51    // initialize rx pointer
52    cp = b;
53  }
54
55  /*
56  * Send character c down the UART Tx, wait until tx holding register
57  * is empty.
58  */
59  int uart_putchar(char c, FILE *stream)
60  {
61    if (c == '\a')
62    {
63      fputs("*ring*\n", stderr);
64      return 0;
65    }
66
67    if (c == '\n')
68    uart_putchar('\r', stream);
69    loop_until_bit_is_set(UCSR0A, UDRE0);
70    UDR0 = c;
71
72    return 0;
73  }
74
75  /*
76  * Receive a character from the UART Rx.
77  *
78  * This features a simple line-editor that allows to delete and
79  * re-edit the characters entered, until either CR or NL is entered.
80  * Printable characters entered will be echoed using uart_putchar().
81  *
82  * Editing characters:
83  *
84  * . \b (BS) or \177 (DEL) delete the previous character
85  * . ^u kills the entire input buffer
86  * . ^w deletes the previous word
87  * . \t will be replaced by a single space
88  *
89  * All other control characters will be ignored.
90  *
91  * The internal line buffer is RX_BUFSIZE (80) characters long, which
92  * includes the terminating \n (but no terminating \0).  If the
       buffer
93  * is full (i. e., at RX_BUFSIZE-1 characters in order to keep space
       for
94  * the trailing \n), any further input attempts will send a \a to
95  * uart_putchar() (BEL character), although line editing is still
96  * allowed.
97  *
98  * Input errors while talking to the UART will cause an immediate
```

```
99  * return of -1 (error indication).  Notably, this will be caused by
        a
100 * framing error (e. g. serial line "break" condition), by an input
101 * overrun, and by a parity error (if parity was enabled and
        automatic
102 * parity recognition is supported by hardware).
103 *
104 * Successive calls to uart_getchar() will be satisfied from the
105 * internal buffer until that buffer is emptied again.
106 */
107
108
109 int uart_getchar(FILE *stream)
110 {
111   uint8_t c;
112
113   loop_until_bit_is_set(UCSR0A, RXC0);
114   if (UCSR0A & _BV(FE0)) return _FDEV_EOF;
115
116   if (UCSR0A & _BV(DOR0)) return _FDEV_ERR;
117
118   c = UDR0;
119
120   /* behaviour similar to Unix stty ICRNL */
121   if (c == '\r') c = '\n';
122
123   if (c == '\n')
124   {
125     *cp = c;
126     uart_putchar(c, stream);
127     // add null character to terminate the string
128     cp[+1] = '\0';
129     // reset the pointer
130     cp = b;
131     return c;
132   }
133   else if (c == '\t') c = ' ';
134
135   if ((c >= (uint8_t)' ' && c <= (uint8_t)'\x7e') ||
136   c >= (uint8_t)'\xa0')
137   {
138     if (cp == b + RX_BUFSIZE - 1)
139     uart_putchar('\a', stream);
140     else
141     {
142       *cp++ = c;
143       uart_putchar(c, stream);
144     }
145   }
146
147   switch (c)
148   {
149     case '\b':
150     case '\x7f':
```

```
151     if (cp > b)
152     {
153       uart_putchar('\b', stream);
154       uart_putchar(' ', stream);
155       uart_putchar('\b', stream);
156       cp--;
157     }
158     break;
159 
160     case 'c' & 0x1f:
161     // reset the pointer
162     cp = b;
163     return -1;
164     break;
165 
166     case 'u' & 0x1f:
167     while (cp > b)
168     {
169       uart_putchar('\b', stream);
170       uart_putchar(' ', stream);
171       uart_putchar('\b', stream);
172       cp--;
173     }
174     break;
175 
176     case 'w' & 0x1f:
177     if (cp[-1] == ' ')
178     {
179       uart_putchar('\b', stream);
180       uart_putchar(' ', stream);
181       uart_putchar('\b', stream);
182       cp--;
183     }
184     else
185     {
186       while (cp > b && cp[-1] != ' ')
187       {
188         uart_putchar('\b', stream);
189         uart_putchar(' ', stream);
190         uart_putchar('\b', stream);
191         cp--;
192       }
193     }
194     break;
195   }
196 
197   return c;
198 }
199 
200 /*
201 * Check rx buffer
202 */
203 int uart_available(void)
204 {
```

```
205   if (UCSR0A & (1 << RXC0)) return 1;
206   return 0;
207 }
208
209 int gets_nb(char* buf)
210 {
211   int c = getchar();
212
213   if (c == -1)
214   {
215     return -1;
216   }
217
218   if (c == '\n')
219   {
220     char* ptr = b;
221     while (*ptr != '\0')
222     {
223       *buf = *(ptr++);
224       buf++;
225     }
226     // add null terminator back to string
227     *buf = '\0';
228     return 1;
229   }
230   return 0;
231 }
232
233 int cli_flag = 0;
234 void cli_print(void)
235 {
236   if (cli_flag == 0)
237   {
238     printf(BY ">> " RESET);
239     cli_flag = 1;
240   }
241 }
242
243 void cli_done(void)
244 {
245   cli_flag = 0;
246 }
247
248 void cli_reset(void)
249 {
250   printf(CLEAR_SCREEN CLEAR_SCROLL);
251 }
```

Listing 3: UART communication functions in uart.c.

6.2.4 Header Files

The accompanying header files, `twi_master.h` and `uart.h`, are critical to define constants, macros, and function prototypes used by the respective source files.

Header File - `twi_master.h`

This file defines constants, macros, and function prototypes used for I2C communication between the microcontroller and the MPU6050 accelerometer.

```
/*
* twi_master.h
*
* Created: 09-Jun-19 11:20:04 AM
*  Author: TEP SOVICHEA
*/


#ifndef TWI_MASTER_H_
#define TWI_MASTER_H_

#include <avr/io.h>
#include <util/twi.h>
#include <stdbool.h>

#define DEBUG_LOG     0
#define SUCCESS       0

#define TW_SCL_PIN      PORTC5
#define TW_SDA_PIN      PORTC4

#define TW_SLA_W(ADDR)    ((ADDR << 1) | TW_WRITE)
#define TW_SLA_R(ADDR)    ((ADDR << 1) | TW_READ)
#define TW_READ_ACK     1
#define TW_READ_NACK    0

typedef uint16_t ret_code_t;

typedef enum {
  TW_FREQ_100K,
  TW_FREQ_250K,
  TW_FREQ_400K
} twi_freq_mode_t;

void tw_init(twi_freq_mode_t twi_freq, bool pullup_en);
ret_code_t tw_master_transmit(uint8_t slave_addr, uint8_t* p_data,
    uint8_t len, bool repeat_start);
ret_code_t tw_master_receive(uint8_t slave_addr, uint8_t* p_data,
    uint8_t len);

#endif /* TWI_MASTER_H_ */
```

Listing 4: Header file twi$_m$*aster.h*.

Header File - `uart.h`

This header file contains function declarations and macros used for initializing and managing UART communication on the ATmega328P microcontroller.

```
/*
* ----------------------------------------------------------------------------
* "THE BEER-WARE LICENSE" (Revision 42):
* <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
* can do whatever you want with this stuff. If we meet some day, and you think
* this stuff is worth it, you can buy me a beer in return.
* Joerg Wunsch
* ----------------------------------------------------------------------------
*
* Stdio demo, UART declarations
*
* $Id: uart.h,v 1.1 2005/12/28 21:38:59 joerg_wunsch Exp $
*/


#ifndef UART_H_
#define UART_H_

#define F_CPU 16000000UL

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <avr/io.h>

#define ESC           "\033"
#define RESET         ESC "[0m"
#define CLEAR_SCREEN  ESC "c"
#define CLEAR_SCROLL  ESC "[3J"
#define CLEAR_LINE    ESC "[2K"

#define CURSOR_RIGHT(col)   ESC "[" col "C"
#define CURSOR_LEFT(col)    ESC "[" col "D"

#define BY          ESC "[33;1m"  // Light Yellow.
#define BG          ESC "[32;1m"  // Light Green.
#define BR          ESC "[31;1m"  // Light Green.

/*
* Perform UART startup initialization.
*/
void uart_init(uint32_t baudrate);
```

```
43
44 /*
45 * Send one character to the UART.
46 */
47 int uart_putchar(char c, FILE *stream);
48
49 /*
50 * Size of internal line buffer used by uart_getchar().
51 */
52 #define RX_BUFSIZE 80
53 #define TX_BUFSIZE 80
54
55 /*
56 * Receive one character from the UART.  The actual reception is
57 * line-buffered, and one character is returned from the buffer at
58 * each invocation.
59 */
60
61 int uart_getchar(FILE *stream);
62
63 /*
64 * Check if UART buffer is available
65 */
66 int uart_available(void);
67
68 /*
69 * gets_nb is similar to the function gets provided by stdlib.h
70 * However, it provides non-blocking functionality so that the CPU
71 * doesn't stall for input, instead it will only read a character
72 * when UART is available and give result once it receive '\n'
    character
73 */
74 int gets_nb(char* buf);
75
76 /*
77 * Print console start indicator ">>" only once
78 */
79 void cli_print(void);
80
81 /*
82 * Tells the console it's ready to take another command
83 */
84 void cli_done(void);
85
86 /*
87 * Send an ANSI escape sequence to clear the terminal
88 */
89 void cli_reset(void);
90
91 #endif /* UART_H_ */
```

Listing 5: Header file uart.h.

6.3 MATLAB Analysis

The MATLAB analysis component is critical for extracting and analyzing the vibration data from the machine. The acceleration data, recorded from the MPU6050 accelerometer, is processed to generate the frequency spectra and to compute essential parameters for designing a suitable shock absorber. This section explains the data processing steps and the calculations performed in MATLAB.

6.3.1 MATLAB Code for Data Analysis

Below is the MATLAB code for performing FFT and visualizing the frequency components of the acceleration data and the raw acceleration data:

```
% Read the table
datatable = readtable('before_shock.csv');

% Convert the table to an array
data = table2array(datatable);

% Separate the data into x, y, and z components
acc_x = data(:, 1);
acc_y = data(:, 2);
acc_z = data(:, 3);

% Convert the time data to seconds
time = data(:, 4)*3600 + data(:, 5)*60 + data(:, 6) + data
    (:, 7)/1000; % Time in seconds

% Create time vector assuming a constant sampling rate
Fs = 1 / mean(diff(time));  % Mean sampling interval
T = 1/Fs; % Sampling period
L = length(acc_x); % Length of signal
t = (0:L-1)*T; % Time vector

% Function to compute single-sided amplitude spectrum
compute_spectrum = @(signal) abs(fft(signal)/L);
get_single_sided = @(P2) P2(2:floor(L/2) + 1); % Exclude
    the first element (0 Hz)
adjust_amplitude = @(P1) [2*P1(1:end-1); P1(end)];

% Frequency vector
f = Fs*(1:floor(L/2))/L; % Exclude the first frequency (0
    Hz)

% Compute single-sided amplitude spectrum for each axis
P2_x = compute_spectrum(acc_x);
P1_x = adjust_amplitude(get_single_sided(P2_x));

P2_y = compute_spectrum(acc_y);
P1_y = adjust_amplitude(get_single_sided(P2_y));

P2_z = compute_spectrum(acc_z);
```

```
P1_z = adjust_amplitude(get_single_sided(P2_z));

% Calculate y-axis limits separately for each plot
max_amplitude_x = max(P1_x);
max_amplitude_y = max(P1_y);
max_amplitude_z = max(P1_z);

% Plot the acceleration in time domain for X-axis
figure;
plot(t, acc_x);
title('Acceleration in Time Domain - X-axis');
xlabel('Time (s)');
ylabel('Acceleration - X-axis');
grid on;

% Plot the acceleration in time domain for Y-axis
figure;
plot(t, acc_y);
title('Acceleration in Time Domain - Y-axis');
xlabel('Time (s)');
ylabel('Acceleration - Y-axis');
grid on;

% Plot the acceleration in time domain for Z-axis
figure;
plot(t, acc_z);
title('Acceleration in Time Domain - Z-axis');
xlabel('Time (s)');
ylabel('Acceleration - Z-axis');
grid on;

% Plot the single-sided amplitude spectrum for X-axis
figure;
plot(f, P1_x);
title('Single-Sided Amplitude Spectrum of X-axis
    Acceleration');
xlabel('Frequency (Hz)');
ylabel('|P1(f)|');
ylim([0 max_amplitude_x * 1.1]);
grid on;

% Plot the single-sided amplitude spectrum for Y-axis
figure;
plot(f, P1_y);
title('Single-Sided Amplitude Spectrum of Y-axis
    Acceleration');
xlabel('Frequency (Hz)');
ylabel('|P1(f)|');
ylim([0 max_amplitude_y * 1.1]);
grid on;

% Plot the single-sided amplitude spectrum for Z-axis
figure;
plot(f, P1_z);
```

```
89 title('Single-Sided Amplitude Spectrum of Z-axis
       Acceleration');
90 xlabel('Frequency (Hz)');
91 ylabel('|P1(f)|');
92 ylim([0 max_amplitude_z * 1.1]);
93 grid on;
94
95 % --- Calculation of Natural Frequency, Damping Ratio,
       Spring Constant, and Damping Coefficient using Y-axis
       data ---
96
97 % Find the natural frequency (peak in FFT) for the Y-axis
98 [~, idx_y] = max(P1_y);
99 natural_frequency_y = f(idx_y);
100 fprintf('Natural frequency: %.3f Hz\n', natural_frequency_y
       );
101
102 % Prompt for mass input
103 mass = input('Enter the mass (kg): ');
104
105 % Calculate the spring constant (k)
106 k = mass * (2 * pi * natural_frequency_y)^2;
107 fprintf('Spring constant: %.3f N/m\n', k);
108
109 % Calculate the damping ratio  using the half-power
       bandwidth method
110 % Assuming FWHM (Full Width at Half Maximum) method is used
111 bandwidth_y = sum(P1_y >= (max(P1_y)/sqrt(2))) * (Fs/L);
112 damping_ratio_y = bandwidth_y / (2 * natural_frequency_y);
113 fprintf('Damping ratio: %.10f\n', damping_ratio_y);
114
115 % Calculate the damping coefficient (c)
116 damping_coefficient_y = 2 * damping_ratio_y * sqrt(k * mass
       );
117 fprintf('Damping coefficient: %.3f Ns/m\n',
       damping_coefficient_y);
118
119 % --- Calculation of Dynamic Viscosity  ---
120
121 % Prompt for piston diameter input
122 piston_diameter = input('Enter the piston diameter (m): ');
123
124 % Calculate the effective piston area A
125 A = pi * (piston_diameter/2)^2;
126
127 % Calculate the dynamic viscosity
128 eta = damping_coefficient_y / A;
129 fprintf('Dynamic viscosity : %.6f Pa.s\n', eta);
130
131 % --- Calculation of Wire Diameter d ---
132
133 % Prompt for inputs related to spring design
134 D = input('Enter the mean coil diameter (m): ');
135 G = input('Enter the shear modulus (Pa): ');
```

```
n = input('Enter the number of active coils: ');

% Calculate the wire diameter d
d = ((8 * k * D^3 * n) / G)^(1/4);
fprintf('Wire diameter d: %.6f m\n', d);
```

Listing 6: MATLAB Code

MATLAB Code Explanation

This section provides an overview of the key parts of the code and their functions.

Step 1: Data Import and Preprocessing

The first step involves reading the acceleration data from a CSV file that contains time-stamped acceleration values for the X, Y, and Z axes. The data is converted into a format suitable for analysis.

```
% Read the table
datatable = readtable('/Users/achirahansindu/Desktop/ACA
    Sem 4/Electronics design realization/Sandaru/
    before_shock.csv');

% Convert the table to an array
data = table2array(datatable);

% Separate the data into x, y, and z components
acc_x = data(:, 1);
acc_y = data(:, 2);
acc_z = data(:, 3);

% Convert the time data to seconds
time = data(:, 4)*3600 + data(:, 5)*60 + data(:, 6) + data
    (:, 7)/1000; % Time in seconds
```

Here, the data is read from a CSV file and separated into the X, Y, and Z-axis components for further analysis. The time data is converted into seconds to accurately plot the acceleration against time.

Step 2: Time-Domain Visualization

After separating the acceleration data into the three components (X, Y, Z), the data is visualized in the time domain. This helps in understanding the raw acceleration signals over time.

```
% Plot the acceleration in time domain for X, Y, and Z axes
figure;
plot(t, acc_x);
title('Acceleration in Time Domain - X-axis');
xlabel('Time (s)');
ylabel('Acceleration - X-axis');
grid on;
```

The above code visualizes the time-domain signal for the X-axis acceleration. Similar plots are generated for the Y and Z axes. This allows for observing the acceleration behavior before proceeding to frequency analysis.

Step 3: Frequency-Domain Analysis Using FFT

To analyze the frequency components of the acceleration data, the Fast Fourier Transform (FFT) is performed on the X, Y, and Z acceleration signals. The resulting single-sided amplitude spectrum is plotted for each axis.

```
% Compute single-sided amplitude spectrum for each axis
P2_x = compute_spectrum(acc_x);
P1_x = adjust_amplitude(get_single_sided(P2_x));

% Plot the single-sided amplitude spectrum for X-axis
figure;
plot(f, P1_x);
title('Single-Sided Amplitude Spectrum of X-axis
    Acceleration');
xlabel('Frequency (Hz)');
ylabel('|P1(f)|');
ylim([0 max_amplitude_x * 1.1]);
grid on;
```

The FFT results reveal the prominent frequencies in the system, helping identify the natural frequency of the machine's vibration. This is important for designing the damping system.

Step 4: Calculation of System Parameters

Once the frequency analysis is completed, the parameters of the system are calculated. This includes the natural frequency and damping ratio.

Natural Frequency: The peak in the frequency spectrum corresponds to the natural frequency of the system, which is identified from the Y-axis data.

```
% Find the natural frequency (peak in FFT) for the Y-axis
[~, idx_y] = max(P1_y);
natural_frequency_y = f(idx_y);
fprintf('Natural frequency: %.3f Hz\n', natural_frequency_y
    );
```

Damping Ratio: The damping ratio ζ is computed using the half-power bandwidth method based on the Y-axis frequency spectrum.

```
% Calculate the damping ratio
bandwidth_y = sum(P1_y >= (max(P1_y)/sqrt(2))) * (Fs/L);
damping_ratio_y = bandwidth_y / (2 * natural_frequency_y);
fprintf('Damping ratio: %.10f\n', damping_ratio_y);
```

Step 5: Inputs for the System

To calculate additional system parameters like the spring constant and damping coefficient, the following inputs are required:

- Mass (kg)
- Piston Diameter (m)
- Mean Coil Diameter (m)
- Shear Modulus (Pa)
- Number of Active Coils

Spring Constant: The spring constant is calculated using the mass and natural frequency of the system. It quantifies the stiffness of the spring in the shock absorber.

```
% Prompt for mass input
mass = input('Enter the mass (kg): ');

% Calculate the spring constant (k)
k = mass * (2 * pi * natural_frequency_y)^2;
fprintf('Spring constant: %.3f N/m\n', k);
```

Damping Coefficient: The damping coefficient is computed using the spring constant and the damping ratio. It represents the resistance force generated by the damper in response to the motion.

```
% Calculate the damping coefficient (c)
damping_coefficient_y = 2 * damping_ratio_y * sqrt(k * mass
    );
fprintf('Damping coefficient: %.3f Ns/m\n',
    damping_coefficient_y);
```

Step 6: Calculation of Shock Absorber Parameters

The shock absorber parameters are essential for designing or selecting the right spring and damper for the system. These parameters are calculated using the values derived from the system and additional user inputs.

Dynamic Viscosity: Dynamic viscosity is a measure of the fluid's resistance to flow inside the shock absorber. It is calculated using the damping coefficient and the effective area of the piston.

```
% Prompt for piston diameter input
piston_diameter = input('Enter the piston diameter (m): ');

% Calculate the effective piston area A
A = pi * (piston_diameter / 2)^2;

% Calculate the dynamic viscosity
eta = damping_coefficient_y / A;
fprintf('Dynamic viscosity : %.6f Pa.s\n', eta);
```

Wire Diameter: The wire diameter of the spring is calculated based on the spring constant, coil diameter, shear modulus, and number of active coils. This helps in determining the material specifications for the spring used in the shock absorber.

```
% Prompt for inputs related to spring design
D = input('Enter the mean coil diameter (m): ');
G = input('Enter the shear modulus (Pa): ');
n = input('Enter the number of active coils: ');
% Calculate the wire diameter d
d = ((8 * k * D^3 * n) / G)^(1/4);
fprintf('Wire diameter d: %.6f m\n', d);
```

This MATLAB code performs a detailed frequency analysis of the vibration data using FFT and calculates key parameters for shock absorber design, including the spring constant, dynamic viscosity, and wire diameter. These parameters are crucial for the selection or customization of an effective damping solution to reduce vibrations in the system. The visualizations and calculations provide insight into the system's vibration behavior before and after applying a damping solution.

6.3.2 Acceleration Data Visualization

The raw acceleration data for the X, Y, and Z axes are plotted to provide a visual representation of the vibrational signals. This initial step helps in understanding the nature and characteristics of the vibrations.

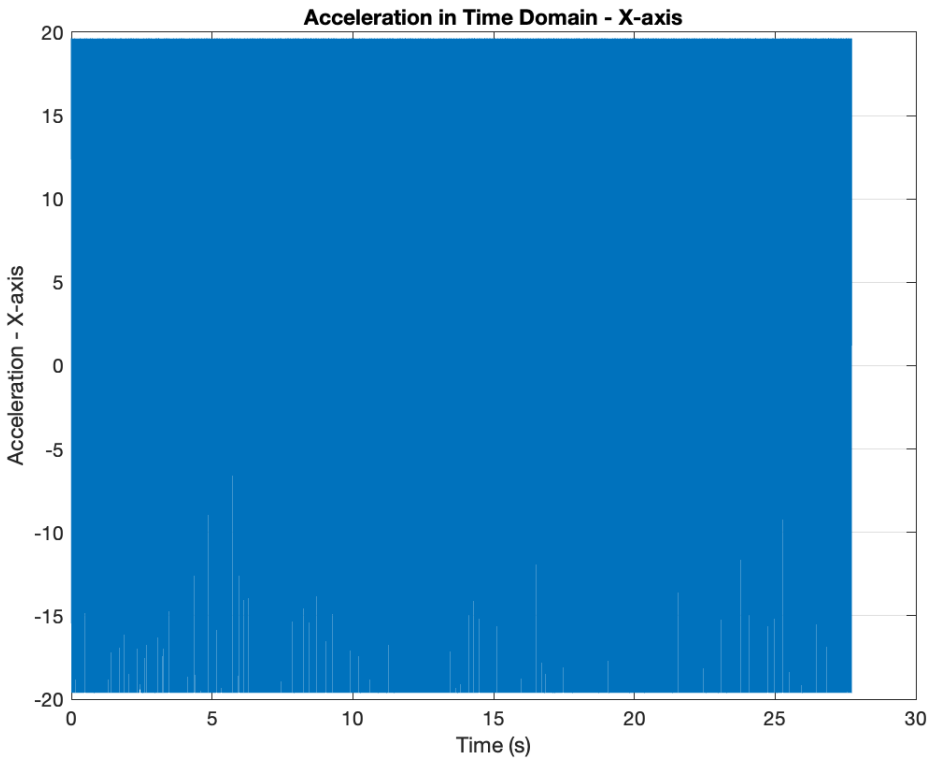


Figure 35: Acceleration along X-axis

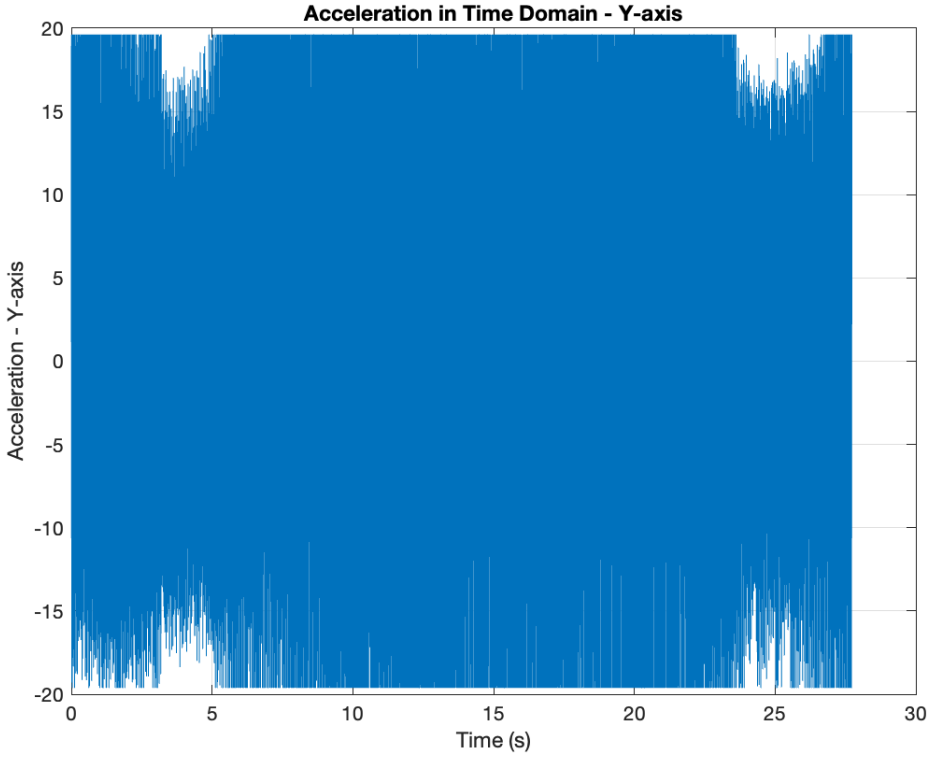


Figure 36: Acceleration along Y-axis

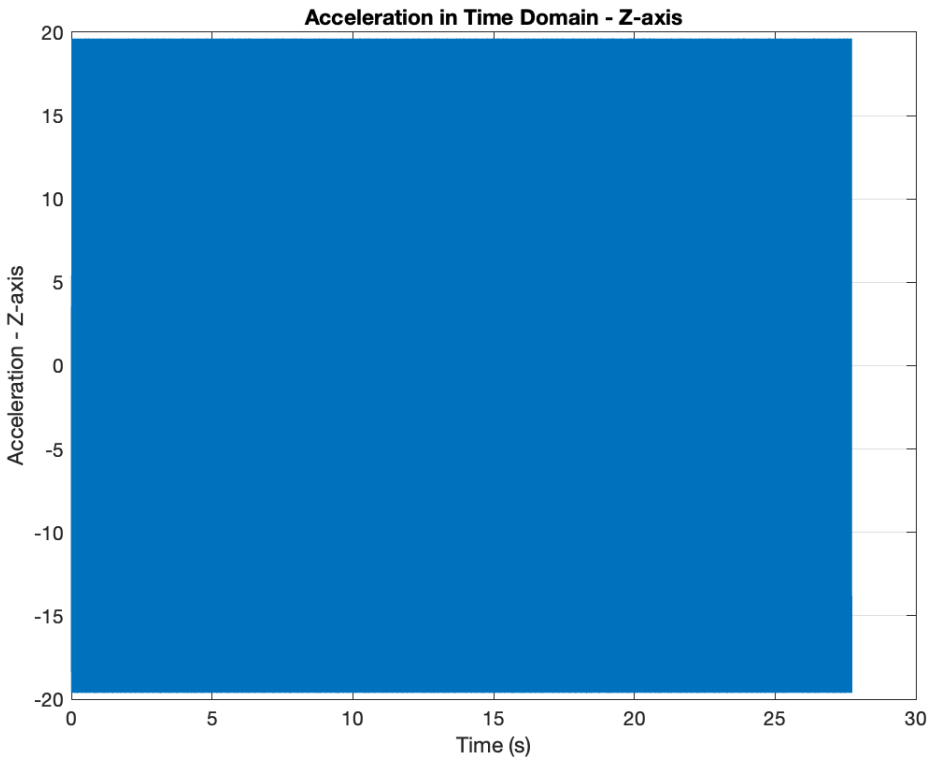


Figure 37: Acceleration along Z-axis

6.3.3 Frequency Analysis Using FFT

To analyze the frequency components of the vibration signals, the Fast Fourier Transform (FFT) is used. The FFT is a powerful mathematical tool that transforms time-domain signals into the frequency domain, allowing us to observe the different frequencies present within the signal. This analysis is crucial for identifying key characteristics, such as the natural frequency of the system and any dominant harmonics.

The FFT process for the analysis of vibration data involves the following key steps:

Sampling Frequency

The sampling frequency f_s is calculated from the time data obtained during the measurements using the time vector created by time-stamps due to the non-uniform sampling frequency. The time intervals between each recorded data point are used to determine the sampling rate, ensuring that the signal is captured with sufficient resolution.

$$f_s = \frac{1}{\Delta t}$$

where Δt is the mean time difference between successive data points.

Frequency Span

According to the Nyquist Theorem, the maximum frequency f_{span} that can be recorded is half of the sampling frequency. This is known as the Nyquist frequency, and it defines the range of frequencies that can be accurately analyzed.

$$f_{\text{span}} = \frac{f_s}{2}$$

The Nyquist frequency represents the highest frequency component that can be identified from the sampled data.

Single-Sided Amplitude Spectrum

The FFT provides a full spectrum of frequencies, including both positive and negative frequencies. However, for practical purposes, we analyze only the positive frequencies, which form the single-sided amplitude spectrum. This spectrum gives us the magnitude of each frequency component, allowing us to identify the significant frequencies present in the signal.

The FFT spectrum reveals peaks corresponding to the system's natural frequencies and harmonic frequencies. These frequencies are critical for understanding the system's vibrational behavior.

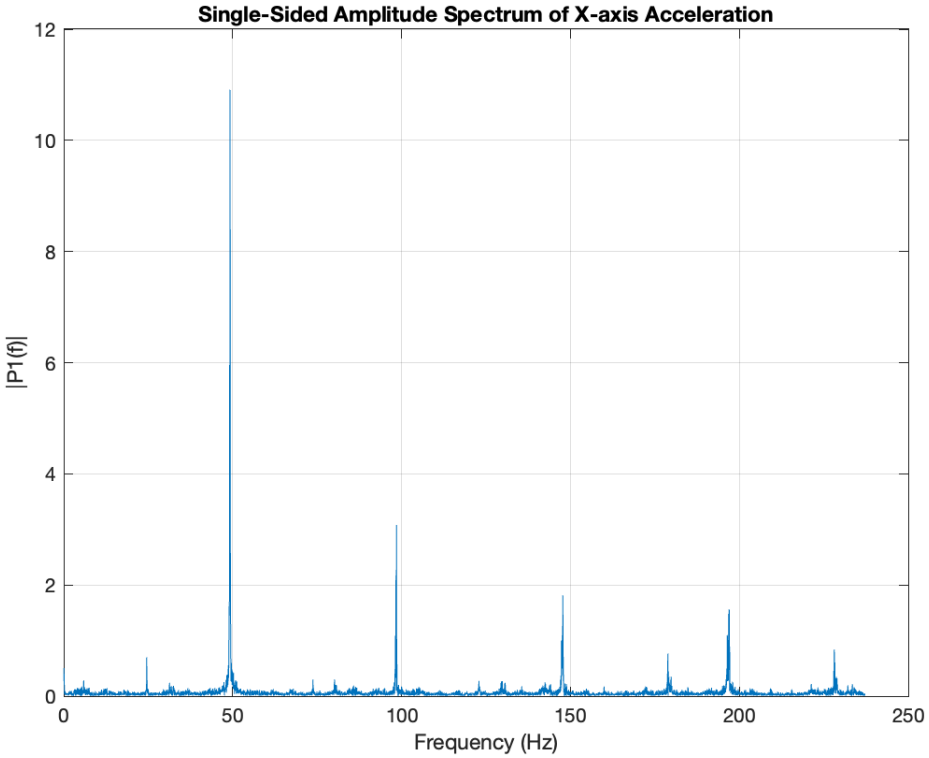


Figure 38: Single-Sided Amplitude Spectrum of X-axis Acceleration

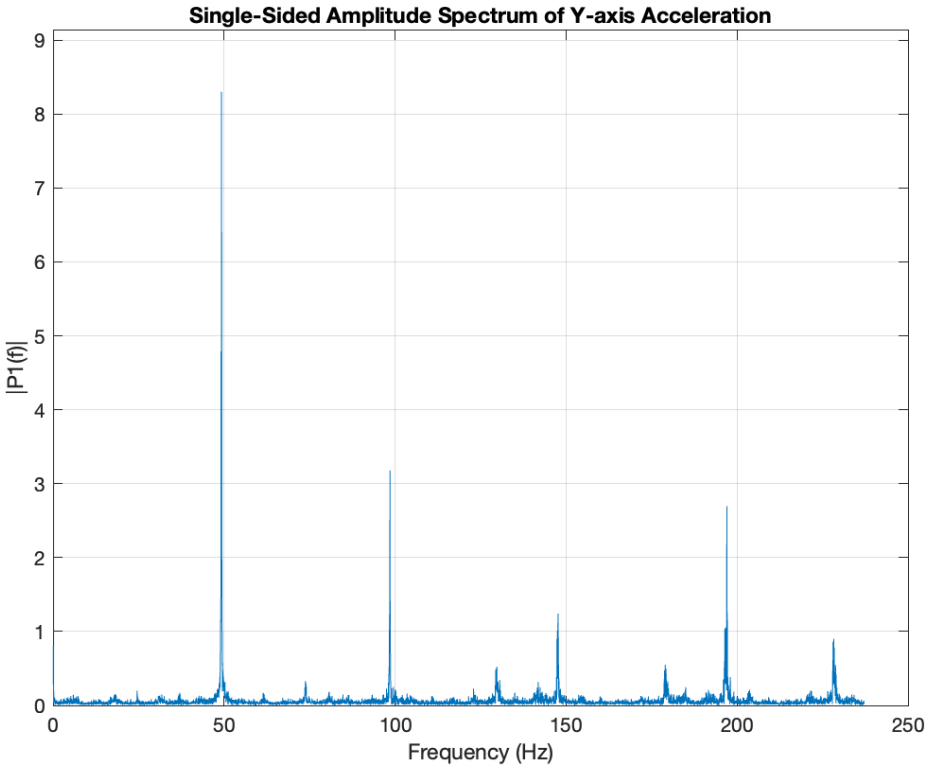


Figure 39: Single-Sided Amplitude Spectrum of Y-axis Acceleration

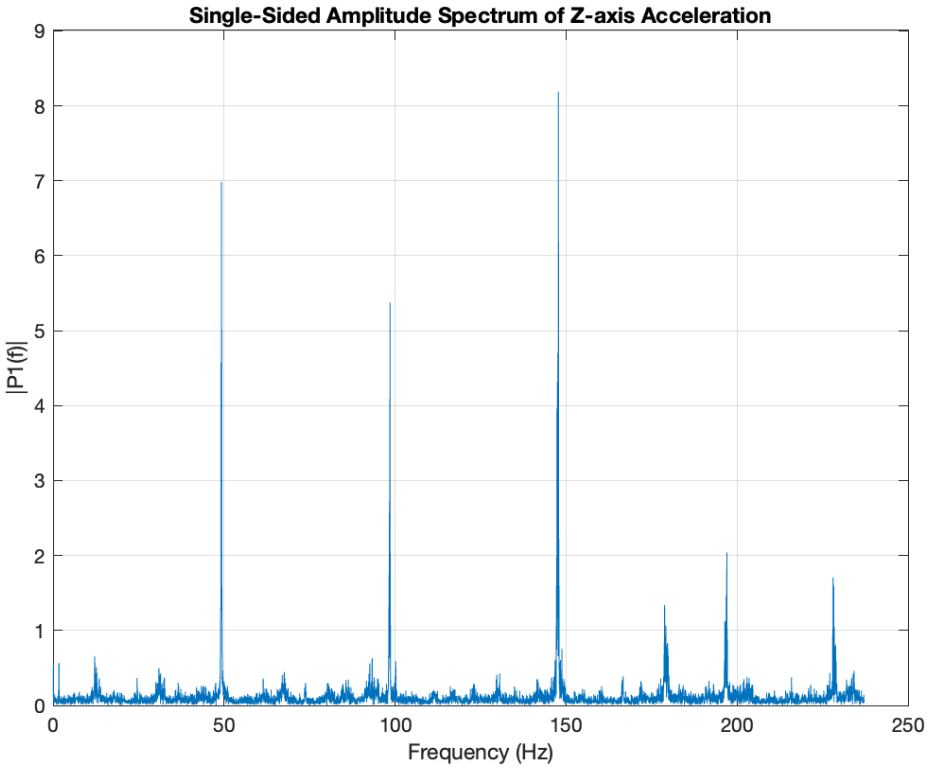


Figure 40: Single-Sided Amplitude Spectrum of Z-axis Acceleration

6.3.4 Calculation of Shock Absorber Parameters

The design and optimization of a shock absorber involve several key parameters: spring constant, damping coefficient, dynamic viscosity, and wire diameter of the spring. These parameters are calculated based on the frequency response of the system and the mechanical properties of the materials involved.

Step 1: Natural Frequency Calculation

The Fast Fourier Transform (FFT) is first applied to the acceleration data captured from the system. This transforms the time-domain vibration signals into their frequency components, allowing for the identification of peaks corresponding to the natural frequency of the system. The natural frequency f_n is identified as the frequency at the peak amplitude in the frequency spectrum.

$$f_n = \text{Peak frequency in FFT (Hz)}$$

Step 2: Damping Ratio Calculation

The damping ratio ζ is computed using the Full Width at Half Maximum (FWHM) method. In this approach, the bandwidth Δf at half of the maximum amplitude in the FFT spectrum is determined, and the damping ratio is calculated as:

$$\zeta = \frac{\Delta f}{2 f_n}$$

Where:

- Δf is the bandwidth at half power frequencies.
- f_n is the natural frequency.

Step 3: Transfer Function for the Shock Absorber

The transfer function of the shock absorber can be derived as a second-order system transfer function, which describes the relationship between the input and output in terms of the damping and natural frequency. The transfer function can be written as:

$$G(s) = \frac{\eta}{s^2 + 2\sigma s + \rho}$$

Here:

- $\eta = \frac{1}{m}$

- s is the Laplace variable.
- $2\sigma = \frac{c}{m}$
- $\rho = \frac{K}{m}$

Check the C Appendix for the derivations of the transfer function. By comparing this with the second-order generic transfer function:

$$G(s) = \frac{k}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

We can derive the following formulas.

Spring Constant (K)

The spring constant is derived from the natural frequency of the system. Using ω_n natural frequency, the spring constant K is given by:

$$\begin{aligned} \omega_n^2 &= \rho \\ \omega_n^2 &= \frac{K}{m} \\ K &= m\omega_n^2 \\ K &= m(2\pi f_n)^2 \end{aligned}$$

We can substitute the natural frequency (f_n) found from the FFT and the mass to find the spring constant.

Damping Coefficient (c)

The damping coefficient is calculated using the damping ratio (ζ) and natural frequency (f_n). By comparing with the generic formula:

$$\begin{aligned} 2\sigma &= 2\zeta\omega_n \\ \frac{c}{m} &= 2\zeta\omega_n \\ c &= 2\zeta m\omega_n \\ c &= 2m\zeta\sqrt{\frac{K}{m}} \\ c &= 2\zeta\sqrt{Km} \end{aligned}$$

We can substitute the natural frequency (f_n), damping ratio (ζ), and the mass to find the damping coefficient.

Step 4: Calculation of Shock Absorber Parameters

Dynamic Viscosity (η)

The damping coefficient is a parameter that quantifies the damping effect provided by a fluid within a shock absorber. It can be expressed as:

$$c = \eta A$$

Where:

- η is the dynamic viscosity of the fluid,
- A is the area of the piston being dampened.

The dynamic viscosity is a measure of a fluid's resistance to flow and can be calculated by rearranging the damping coefficient formula:

$$\eta = \frac{c}{A}$$

This parameter is crucial for the shock absorber's ability to dissipate energy.

Wire Diameter (d)

The wire diameter for the spring is calculated based on the required spring constant using the formula:

$$k = \frac{Gd^4}{8D^3N}$$

Where:

- G is the shear modulus of the material,
- d is the wire diameter,
- D is the mean diameter of the spring,
- N is the number of active coils.

This formula shows that the spring constant is dependent on the material's properties and the geometric configuration of the spring. The formula can be rearranged to calculate the required wire diameter for the spring production:

$$d = \left(\frac{8kD^3N}{G}\right)^{1/4}$$

This expression provides a method to determine the necessary wire diameter to achieve a desired spring constant, taking into account the spring's dimensions and the material properties.

This systematic approach to calculating the parameters of the shock absorber ensures that the designed system effectively reduces vibrations.

6.3.5 Comparison of Vibration Data Before and After Implementing a Shock Absorber System

To assess the effectiveness of a newly implemented shock absorber system, a comparative analysis of the vibration data before and after the installation is essential. This comparison is facilitated through the analysis of the frequency components of vibration signals, using the Fast Fourier Transform (FFT).

Vibration data is acquired from the system before and after the shock absorber installation. This data includes acceleration measurements along the x, y, and z axes. A single-sided amplitude spectrum is computed for each axis to identify dominant frequencies. Users can select the axis (x, y, or z) for which they wish to view the vibration analysis. The system processes data for the selected axis and plots the FFT for both pre- and post-installation states. The natural frequency and its corresponding amplitude are calculated for both datasets. An improvement percentage is computed based on the reduction in amplitude at the natural frequency, providing a clear measure of the damper's effectiveness.

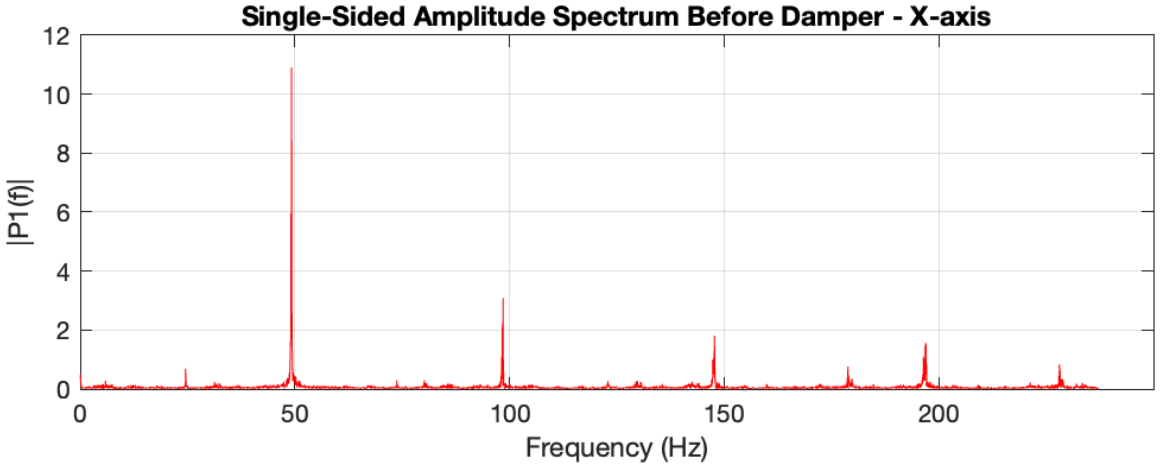


Figure 41: FFT before implementing the damper

The following MATLAB code is used for the comparisons:

```
% MATLAB code for the comparison of the system before and
    after implementing the damper

% Load pre-installation and post-installation vibration
    data
```

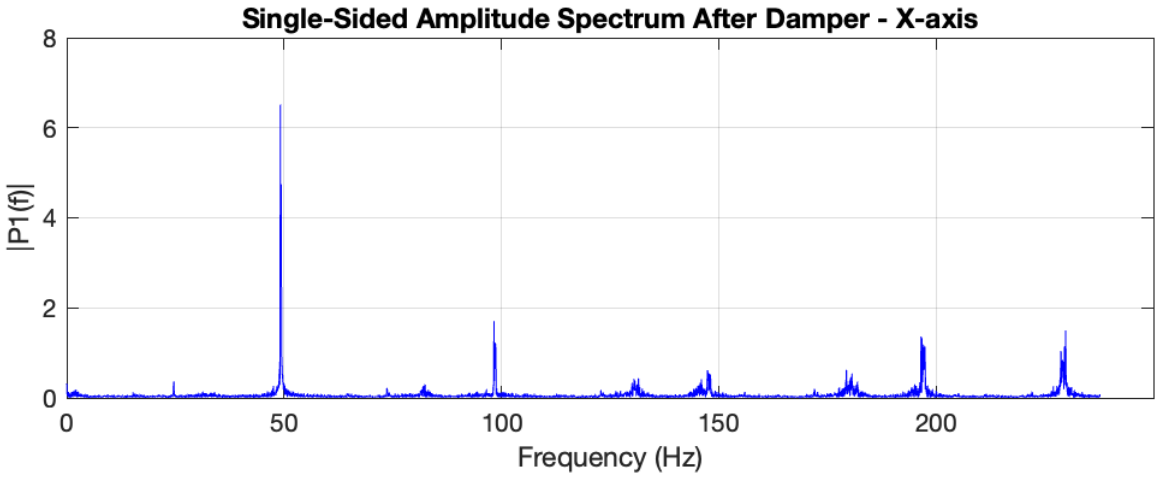


Figure 42: FFT after implementing the damper

```
data_before = load('vibration_before.mat');
data_after = load('vibration_after.mat');

% Define the sampling frequency
fs = 1000; % Example sampling frequency in Hz

% Perform FFT on pre-installation data
N_before = length(data_before.acceleration);
Y_before = fft(data_before.acceleration);
f_before = (0:(N_before/2)-1)*(fs/N_before);
P_before = abs(Y_before(1:N_before/2));

% Perform FFT on post-installation data
N_after = length(data_after.acceleration);
Y_after = fft(data_after.acceleration);
f_after = (0:(N_after/2)-1)*(fs/N_after);
P_after = abs(Y_after(1:N_after/2));

% Plot FFT before and after implementing the damper
figure;
subplot(2,1,1);
plot(f_before, P_before);
title('FFT Before Implementing the Damper');
xlabel('Frequency (Hz)');
ylabel('Amplitude');

subplot(2,1,2);
plot(f_after, P_after);
title('FFT After Implementing the Damper');
xlabel('Frequency (Hz)');
ylabel('Amplitude');

% Compute natural frequency and amplitude for both datasets
[~, idx_before] = max(P_before);
f_n_before = f_before(idx_before);
```

```
amp_before = P_before(idx_before);

[~, idx_after] = max(P_after);
f_n_after = f_after(idx_after);
amp_after = P_after(idx_after);

% Calculate improvement percentage
improvement = ((amp_before - amp_after) / amp_before) *
    100;

fprintf('Natural Frequency Before: %.2f Hz\n', f_n_before);
fprintf('Amplitude Before: %.2f\n', amp_before);
fprintf('Natural Frequency After: %.2f Hz\n', f_n_after);
fprintf('Amplitude After: %.2f\n', amp_after);
fprintf('Improvement: %.2f%%\n', improvement);
```

7 Front-End Software: UI for Data Acquisition System

The data acquisition system for vibration analysis interfaces the DAQ device with a computer via a USB connection. The data received at the computer's serial port is read by custom software developed using Python and PyQt5. This software provides a graphical user interface (GUI) to facilitate user interaction for starting, stopping, and saving the recorded vibration data.

7.1 Detailed Explanation of UI

1. **Initial Launch Screen:**
   Upon launching the application, the user is greeted with the initial screen. The user must press "Next" to proceed.

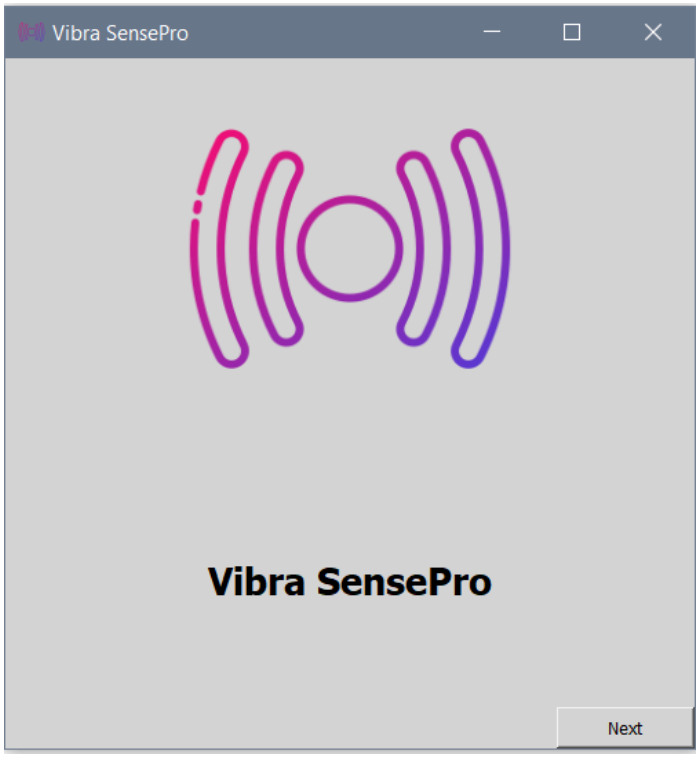


Figure 43: The initial launch screen of the Vibration Data Analyser.

2. **COM Port and Baud Rate Selection:**
   In this window, the user selects the appropriate COM port and baud rate for communication. The "Refresh" button updates the list of available ports, and the "Connect" button initiates the connection.

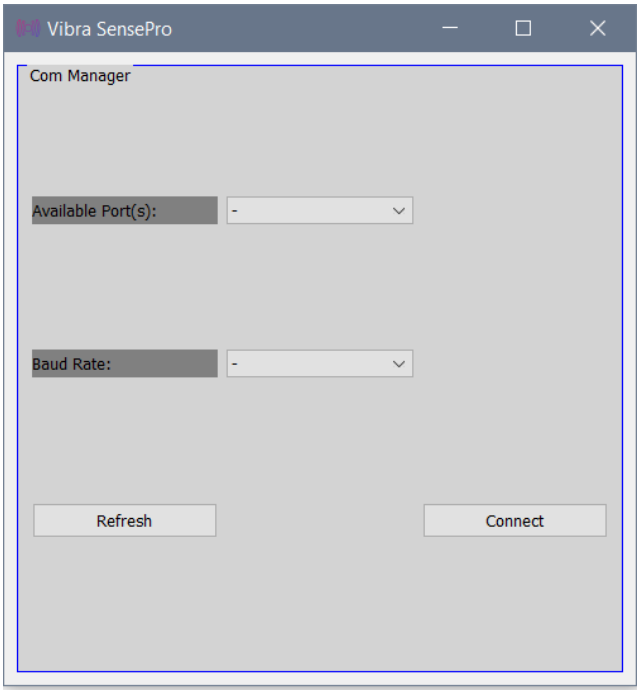


Figure 44: The screen for selecting the available COM port and baud rate.

3. **COM Port and Baud Rate Connected:**
   Once the correct COM port and baud rate are selected, pressing the "Connect" button connects the device, and the user is taken to the data collection window.

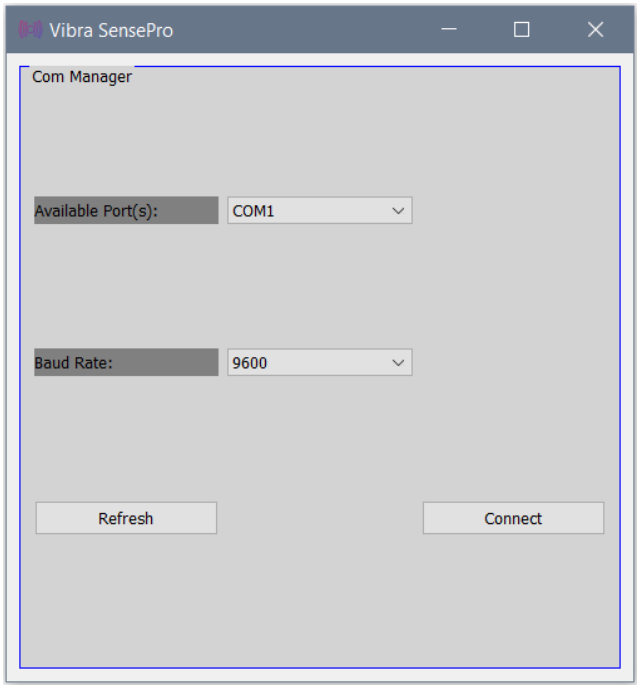


Figure 45: The screen showing the selected COM port and baud rate.

4. **Data Collection Controls:**
   In this window, the user can start, stop, and save the recording of acceleration data. The "Start" button begins data collection, the "Stop" button halts it, and the "Save" button opens a dialog to save the recorded data.

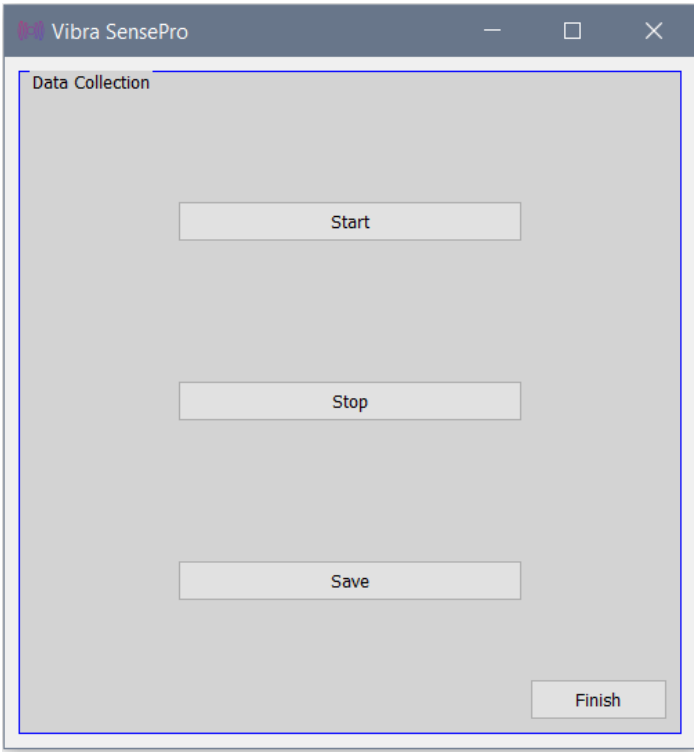


Figure 46: The screen for controlling data collection.

5. **Start Collecting Data:**
   When the user presses "Start," the system begins collecting data, as indicated by the message "Start collecting data..."

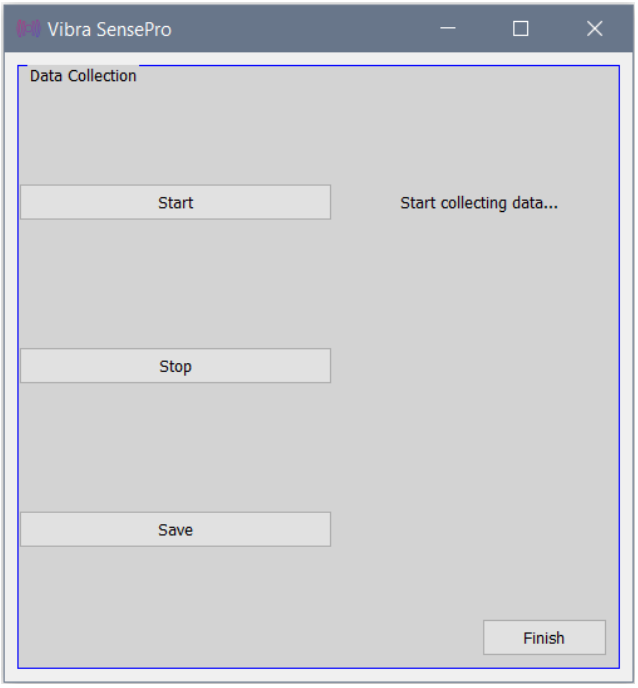


Figure 47: The screen indicating data collection has started.

6. **Stop Collecting Data:**
   Pressing "Stop" halts the data collection, and the message "Stop collecting data..." appears on the screen.

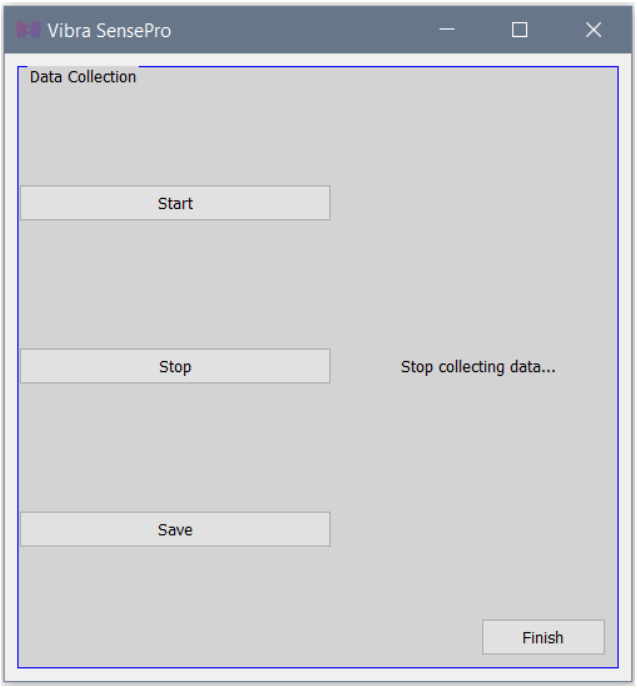


Figure 48: The screen indicating data collection has stopped.

7. **Saving Data:**
   After stopping the data collection, the user can save the data by pressing the "Save" button. This opens a dialog where the user can specify the file name and location.

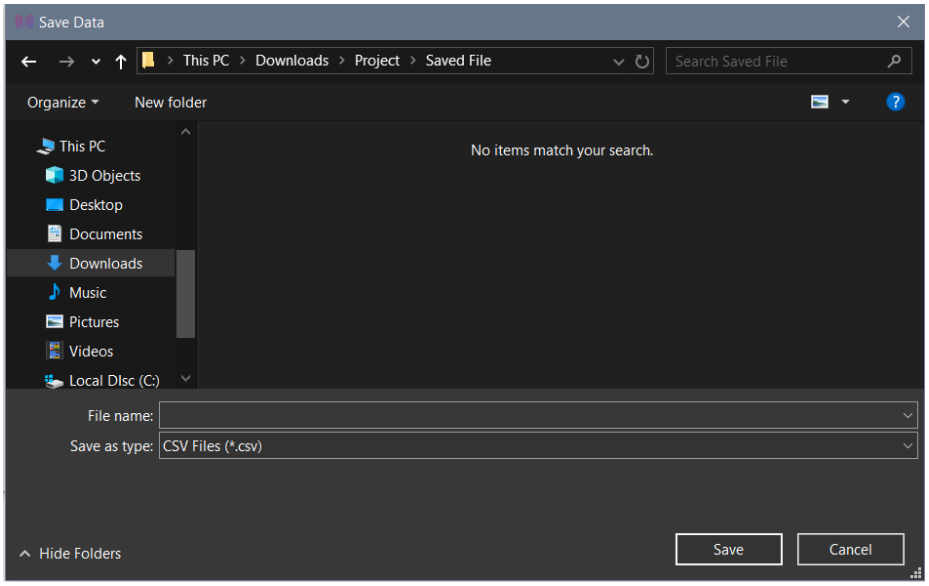


Figure 49: The save dialog for saving the collected data as a CSV file.

8. **Success Message:**
   Upon successfully saving the data, a confirmation message appears, indicating the location of the saved file.

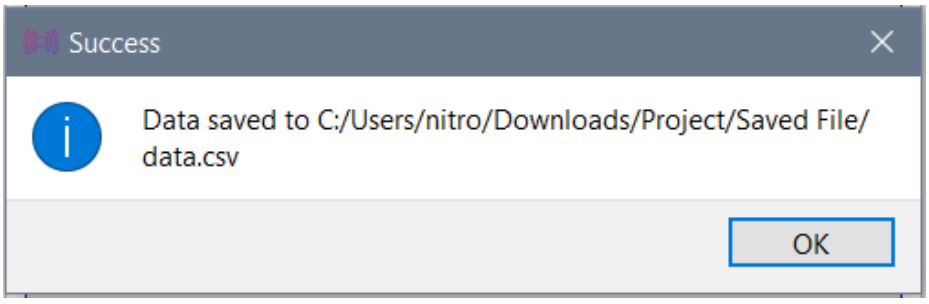


Figure 50: The success message confirming that the data has been saved.

9. **Saved File Location:**
   The saved file can be found in the specified save location.

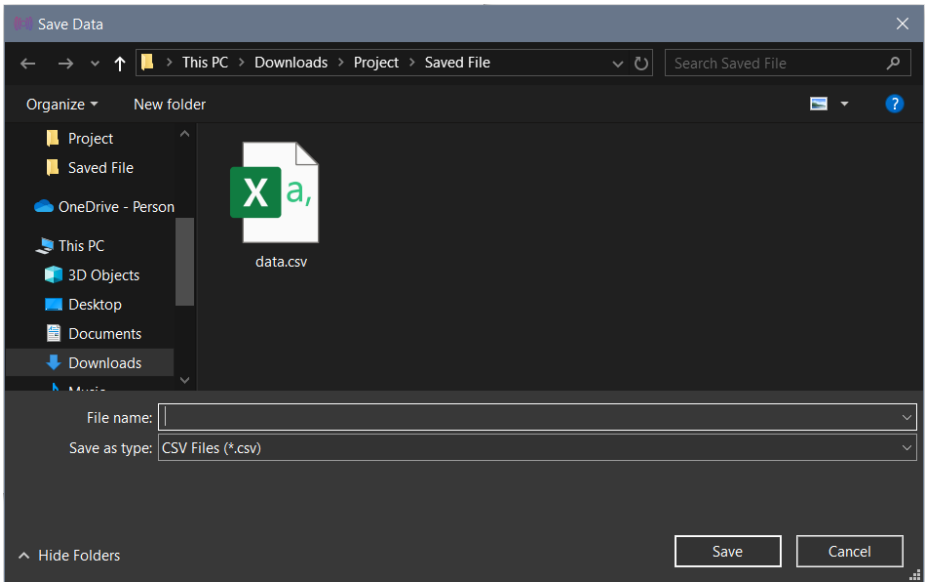


Figure 51: The saved data file in the specified location.

10. **Close the Windows:**
    When the "Finish" button is clicked, all the opened windows are closed.

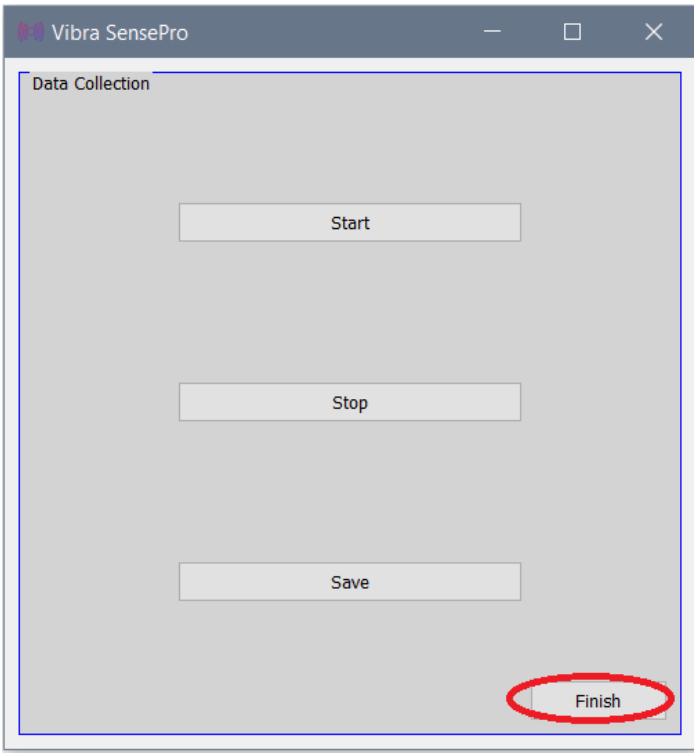


Figure 52: Click the Finish Button to close all Windows

7.2 Python Codebase Using PyQt5

Below is the Python codebase developed using PyQt5 for the Vibra SensePro. This code defines the GUI and the functions for reading data from the serial port, starting and stopping the data collection, and saving the recorded data into a CSV file.

```
import sys
import serial
import serial.tools.list_ports
import csv

from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QVBoxLayout, QWidget, QGroupBox, QGridLayout, QPushButton, QComboBox, QFileDialog, QMessageBox
from PyQt5.QtGui import QIcon, QFont
from PyQt5.QtCore import Qt, QThread, pyqtSignal

class App(QWidget):
    def __init__(self):
        super().__init__()
        self.title = 'Vibra SensePro'
        self.left = 100
        self.top = 100
        self.width = 500
        self.height = 500
        self.initUI()

    def initUI(self):
        self.setWindowTitle(self.title)
        self.setGeometry(self.left, self.top, self.width, self.height)
        self.setStyleSheet("background-color: lightgray;")
        self.setWindowIcon(QIcon('icon.png'))

        self.layout = QVBoxLayout()

        self.iconLabel = QLabel(self)
        self.iconLabel.setPixmap(QIcon('icon.png').pixmap(250, 250))
        self.iconLabel.setAlignment(Qt.AlignCenter)

        self.nameLabel = QLabel('Vibra SensePro', self)
        self.nameLabel.setTextFormat(Qt.RichText)
        font = QFont('Cabiri', 16)
        font.setBold(True)
        self.nameLabel.setFont(font)
        self.nameLabel.setAlignment(Qt.AlignCenter)

        self.layout.addWidget(self.iconLabel)
        self.layout.addWidget(self.nameLabel)

        self.setLayout(self.layout)

```

```
44          self.btn_next = QPushButton('Next', self)
45          self.btn_next.setGeometry(400, 470, 100, 30)
46          self.btn_next.clicked.connect(self.next)
47
48      def next(self):
49          self.dialog = ComManager()
50          self.dialog.show()
51
52  class ComManager(QWidget):
53      def __init__(self):
54          super().__init__()
55          self.initUI()
56          self.com_refresh()
57
58      def initUI(self):
59          self.setWindowTitle("Vibra SensePro")
60          self.setGeometry(100, 100, 500, 500)
61          self.setWindowIcon(QIcon('icon.png'))
62
63          self.frame = QGroupBox("Com Manager", self)
64          self.frame.setStyleSheet("QGroupBox { background-
    color: lightgray; border: 1px solid blue; }")
65          self.frame.setGeometry(10, 10, 480, 480)
66
67          self.label_com = QLabel("Available Port(s):", self.
    frame)
68          self.label_com.setStyleSheet("background-color:
    gray;")
69
70          self.label_bd = QLabel("Baud Rate:", self.frame)
71          self.label_bd.setStyleSheet("background-color: gray
    ;")
72
73          self.drop_com = QComboBox(self.frame)
74          self.drop_com.setMinimumWidth(100)
75          self.drop_com.addItem('-')
76
77          self.drop_baud = QComboBox(self.frame)
78          self.drop_baud.setMinimumWidth(100)
79          self.drop_baud.addItem('-')
80          self.drop_baud.addItems(['9600', '19200', '38400',
    '57600', '115200', '250000'])
81
82          self.btn_refresh = QPushButton("Refresh", self.
    frame)
83          self.btn_refresh.clicked.connect(self.com_refresh)
84          self.btn_connect = QPushButton("Connect", self.
    frame)
85          self.btn_connect.clicked.connect(self.connect)
86
87          self.publish()
88
89      def publish(self):
90          self.layout = QGridLayout(self.frame)
```

```
        self.layout.addWidget(self.label_com, 1, 1)
        self.layout.addWidget(self.label_bd, 2, 1)
        self.layout.addWidget(self.drop_com, 1, 2)
        self.layout.addWidget(self.drop_baud, 2, 2)
        self.layout.addWidget(self.btn_refresh, 3, 1)
        self.layout.addWidget(self.btn_connect, 3, 3)
        self.frame.setLayout(self.layout)

    def com_refresh(self):
        self.drop_com.clear()
        self.drop_com.addItem('-')
        ports = serial.tools.list_ports.comports()
        if ports:
            for port in ports:
                self.drop_com.addItem(port.device)
        else:
            self.drop_com.addItems(['COM1', 'COM2'])
        self.drop_com.setCurrentIndex(0)

    def connect(self):
        self.dialog1 = MainWindow(self.drop_com.currentText
    (), self.drop_baud.currentText())
        self.dialog1.show()

class SerialThread(QThread):
    data_received = pyqtSignal(str)

    def __init__(self, port, baudrate):
        super().__init__()
        self.port = port
        self.baudrate = baudrate
        try:
            self.ser = serial.Serial(port, baudrate)
        except serial.SerialException as e:
            self.ser = None
            self.error_message = str(e)

    def run(self):
        if not self.ser:
            return
        self.running = True
        while self.running:

            if self.ser.in_waiting:
                try:
                    data = self.ser.readline().decode('utf
    -8').strip()
                    self.data_received.emit(data)
                except UnicodeDecodeError:
                    pass
                except TypeError:
                    pass
    def stop(self):
        if self.ser:
```

```
143             self.running = False
144             self.ser.close()
145 
146 class MainWindow(QMainWindow):
147     def __init__(self, port, baudrate):
148         super().__init__()
149         self.port = port
150         self.baudrate = baudrate
151 
152         self.initUI()
153 
154     def initUI(self):
155         self.setWindowTitle("Vibra SensePro")
156         self.setGeometry(100, 100, 500, 500)
157         self.setWindowIcon(QIcon('icon.png'))
158 
159         self.frame = QGroupBox("Data Collection", self)
160         self.frame.setStyleSheet("QGroupBox { background-
    color: lightgray; border: 1px solid blue; }")
161         self.frame.setGeometry(10, 10, 480, 480)
162 
163         self.start_button = QPushButton("Start", self)
164         self.start_button.setFixedSize(250, 30)
165         self.start_button.clicked.connect(self.
    start_data_collection)
166 
167         self.stop_button = QPushButton("Stop", self)
168         self.stop_button.setFixedSize(250, 30)
169         self.stop_button.clicked.connect(self.
    stop_data_collection)
170 
171         self.save_button = QPushButton("Save", self)
172         self.save_button.setFixedSize(250, 30)
173         self.save_button.clicked.connect(self.save_data)
174 
175         self.start_label = QLabel("Start collecting data
    ...", self)
176         self.start_label.setAlignment(Qt.AlignCenter)
177         self.start_label.hide()  # Initially hide the start
     label
178 
179         self.stop_label = QLabel("Stop collecting data...",
     self)
180         self.stop_label.setAlignment(Qt.AlignCenter)
181         self.stop_label.hide()  # Initially hide the stop
    label
182 
183         self.layout = QGridLayout(self.frame)
184         self.layout.addWidget(self.start_button, 1, 1)
185         self.layout.addWidget(self.start_label, 1, 2)
186         self.layout.addWidget(self.stop_button, 2, 1)
187         self.layout.addWidget(self.stop_label, 2, 2)
188         self.layout.addWidget(self.save_button, 3, 1)
189         #self.layout.addWidget(self.finish_button, 4, 2)
```

```
        self.frame.setLayout(self.layout)

        container = QWidget()
        container.setLayout(self.layout)
        self.setCentralWidget(container)

        self.finish_button = QPushButton('Finish', self)
        self.finish_button.setGeometry(380, 450, 100, 30)
        self.finish_button.clicked.connect(QApplication.
    instance().quit)

        self.data_thread = None
        self.data = []

    def start_data_collection(self):
        if self.port and self.baudrate:
            self.data_thread = SerialThread(self.port, int(
    self.baudrate))
            if not self.data_thread.ser:
                QMessageBox.critical(self, "Error", f"
    Failed to open serial port: {self.data_thread.
    error_message}")
                return
            self.data_thread.data_received.connect(self.
    update_data)
            self.data_thread.start()
            self.start_label.show()
            self.stop_label.hide()

    def stop_data_collection(self):
        if self.data_thread:
            self.data_thread.stop()
            self.start_label.hide()
            self.stop_label.show()

    def update_data(self, data):
        print(f"Received data: {data}")  # Debug print
        self.data.append(data)
        # Assuming you have a QLabel to display data.
    Create one if not
        if not hasattr(self, 'data_label'):
            self.data_label = QLabel(self)
            self.layout.addWidget(self.data_label, 5, 1, 1,
     2)
        self.data_label.setText(f"Data: {data}")


    def save_data(self):
        self.stop_label.hide()
        file_path, _ = QFileDialog.getSaveFileName(self, "
    Save Data", "", "CSV Files (*.csv)")
        if file_path:
            with open(file_path, 'w', newline='') as
    csv_file:
```

```
                writer = csv.writer(csv_file)
                writer.writerow(["AX", "AY", "AZ", "Hours",
 "Minutes", "Seconds", "Milliseconds"])  # Updated CSV
Header
                for line in self.data:
                    parts = line.split()  # Split the line
by whitespace
                    if len(parts) == 7:  # Ensure that we
have the expected number of parts
                        ax = parts[1]  # Extract AX value
                        ay = parts[3]  # Extract AY value
                        az = parts[5]  # Extract AZ value
                        timestamp = parts[6]  # Extract
Timestamp as a single part

                        # Split the timestamp into hours,
minutes, seconds, milliseconds
                        time_parts = timestamp.split('.')
                        if len(time_parts) == 4:  # Ensure
correct timestamp format
                            hours = time_parts[0]
                            minutes = time_parts[1]
                            seconds = time_parts[2]
                            milliseconds = time_parts[3]

                            # Write data to CSV with the
split timestamp
                            writer.writerow([ax, ay, az,
hours, minutes, seconds, milliseconds])

            QMessageBox.information(self, "Success", f"Data
 saved to {file_path}")




if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = App()
    window.show()
    sys.exit(app.exec_())
```

Listing 7: Python Code for Vibra SensePro

This Python code creates the UI for the Vibra SensePro, allowing users to interact with the system for data acquisition, recording, and saving vibration data. The GUI provides a user-friendly interface for seamless operation, enhancing the overall user experience.

8 MATLAB Application for Data Analysis

The MATLAB Application for Data Analysis is used for detailed processing and visualization of the collected vibrational data. The application has a user-friendly interface that allows users to visualize time-domain data, analyze frequency-domain data using Fast Fourier Transform (FFT), and calculate system parameters necessary for designing a vibration-damping solution. The application also provides a comparison of system performance before and after the installation of the damping solution

.

8.1 Data Visualization - Acceleration in Time Domain

In the first window, the acceleration data along the X, Y, and Z axes is plotted. This data is extracted from the CSV file obtained from the DAQ system. The selected file can be loaded, and the acceleration over time is visualised for all three axes.

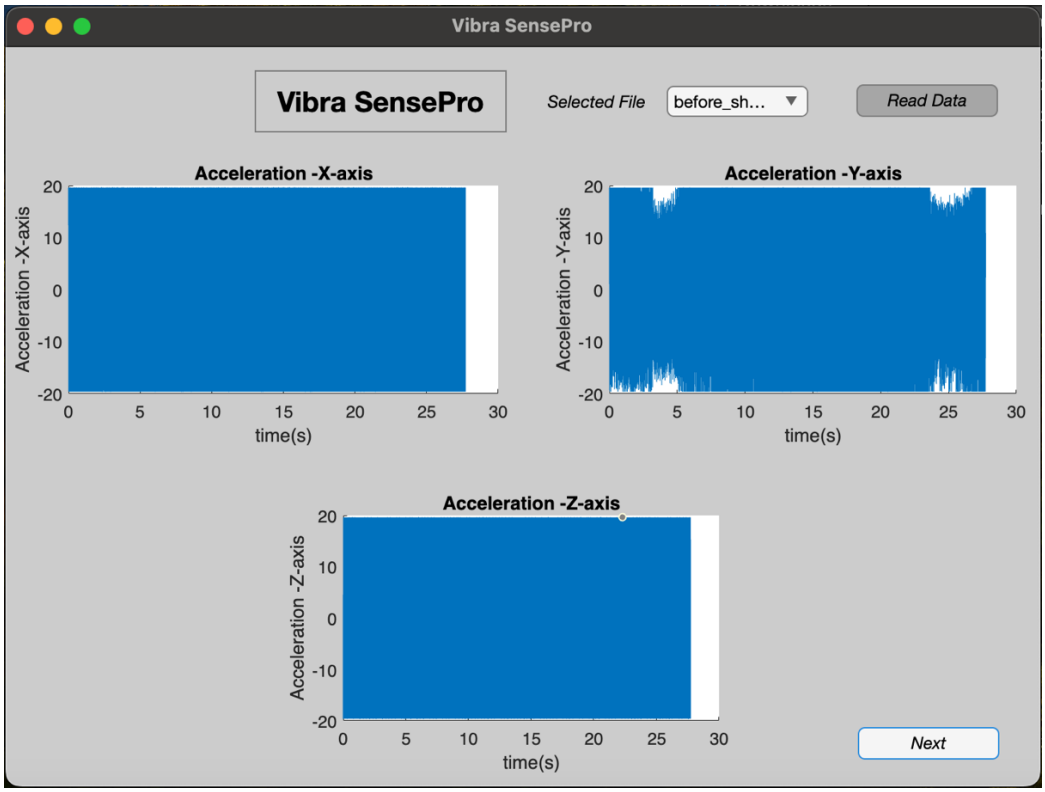


Figure 53: Time-domain acceleration plot for X, Y, and Z axes.

8.2 Frequency Spectrum - FFT Analysis

The second window provides a frequency domain analysis using FFT. The FFT plots display the frequency components of the vibration signals for

the X, Y, and Z axes, allowing the identification of peaks corresponding to the natural frequency and its harmonics. This is crucial for further analysis and design of the damping system.

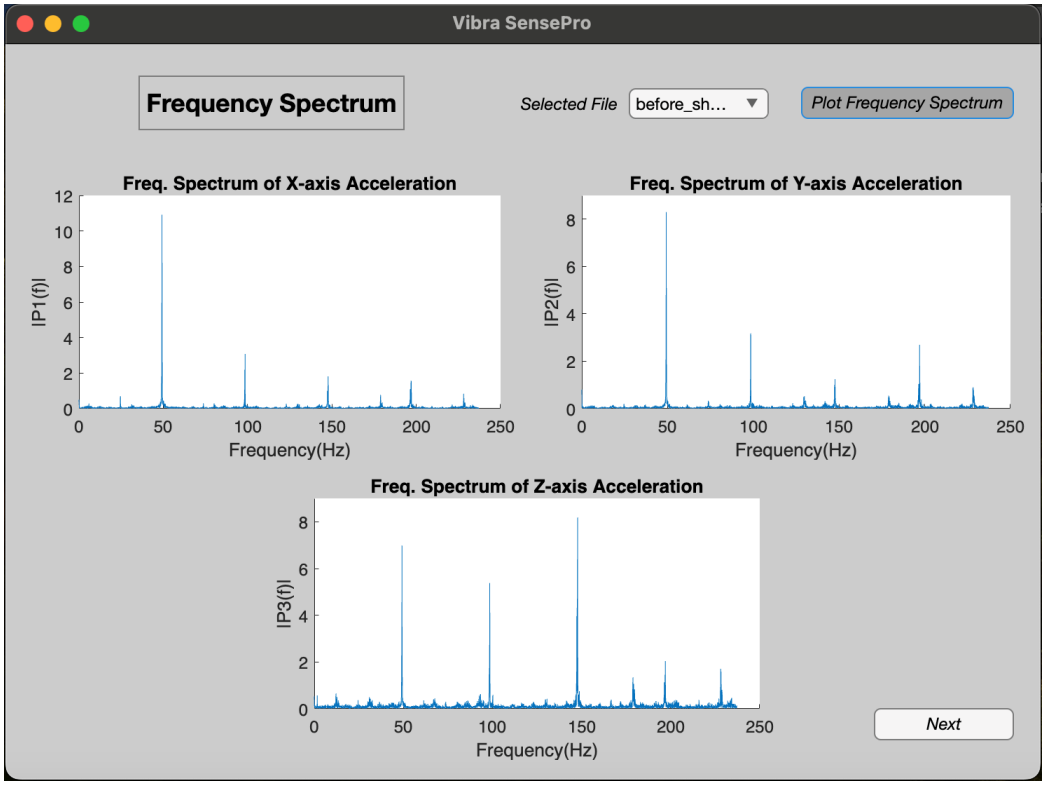


Figure 54: Frequency spectrum (FFT) for X, Y, and Z axes, showing dominant frequency peaks.

8.3 System Parameters Calculation

In the third window, the system parameters related to the shock absorber are calculated. The user first loads a CSV file containing the vibration data, and the application automatically extracts key parameters such as the natural frequency and damping ratio obtained from the FFT analysis.

By entering additional known parameters related to the system, such as the mass, piston diameter, mean coil diameter, shear modulus, and the number of active coils in the spring, the application calculates important values such as the spring constant, dynamic viscosity, and wire diameter of the spring. These parameters are critical for selecting or designing an appropriate shock absorber tailored to the system's specific requirements.

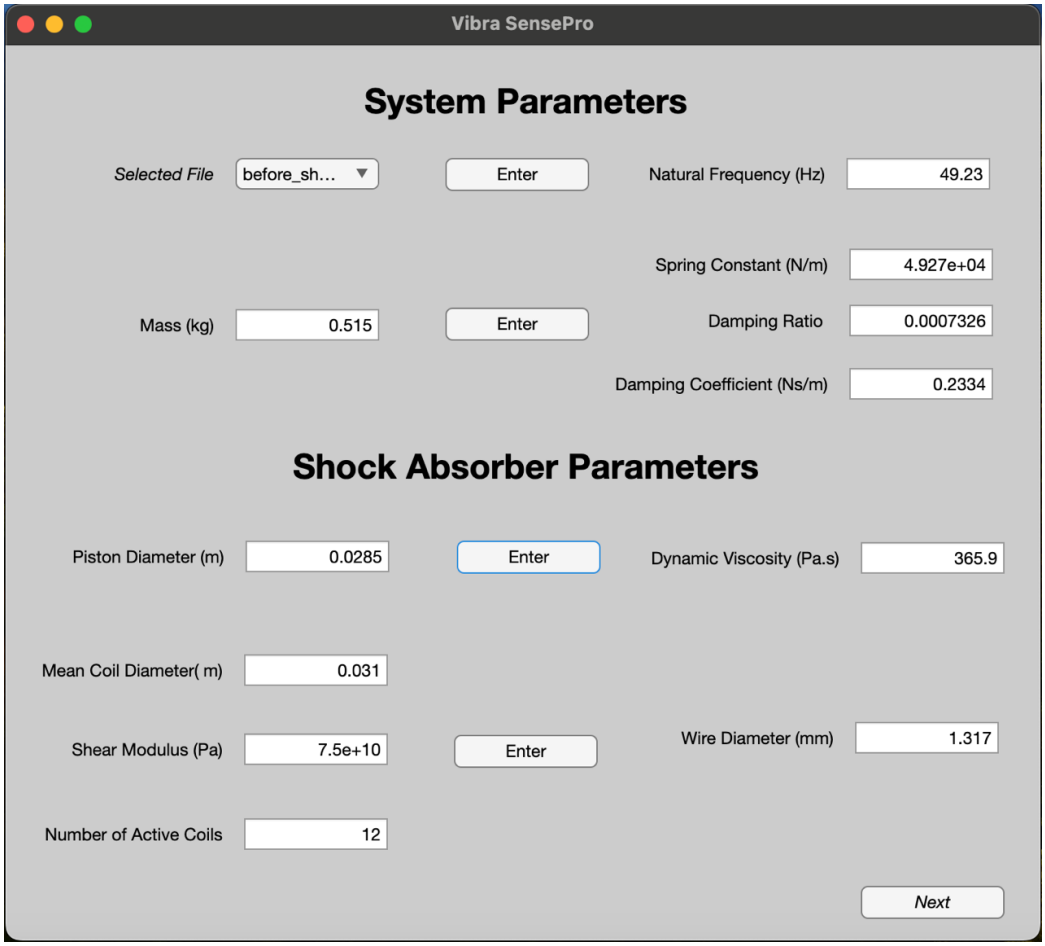


Figure 55: System parameters calculation window for the shock absorber.

8.4 System Comparison - Before and After Damping

The fourth window allows users to load the CSV files of the system's acceleration data recorded before and after the installation of the damping solution. After selecting the files, the frequency spectra of the accelerations are plotted for comparison across all three axes (X, Y, and Z). This comparison highlights the reduction in vibration amplitude and any shifts in the natural frequency. The application also quantifies the damping effect by calculating the overall vibration reduction as a percentage.

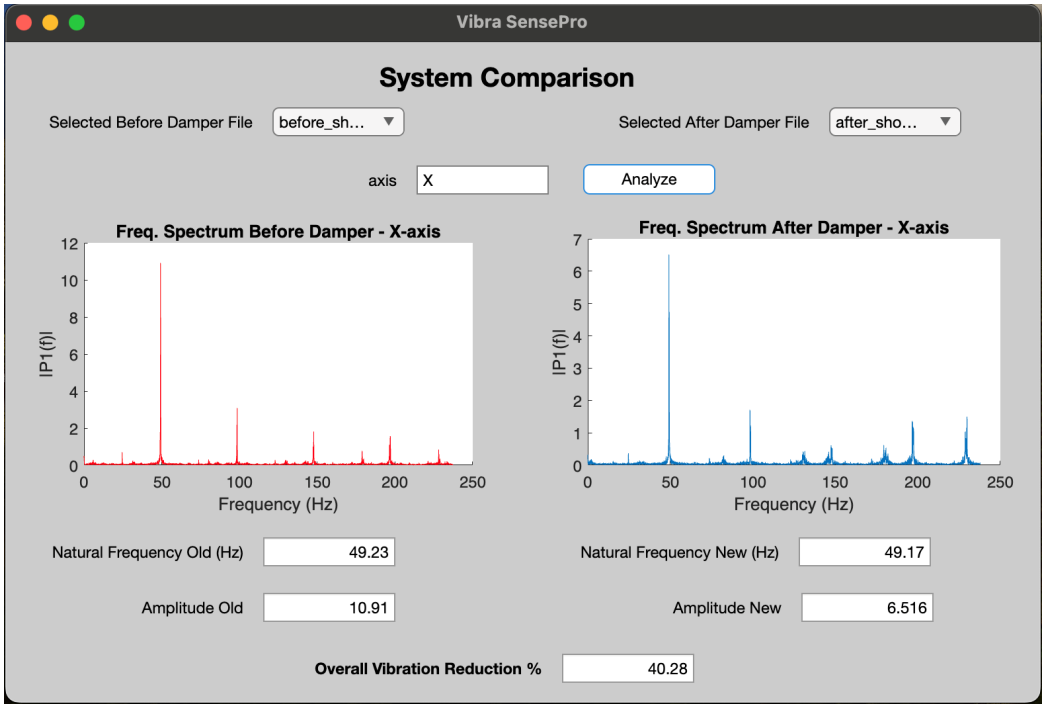


Figure 56: Comparison of frequency spectra before and after shock absorber installation.

This feature provides a comprehensive visualization of how the shock absorber improves the system's performance by reducing vibrations across all axes.

This MATLAB application facilitates not only data analysis but also aids in the design and validation process of the vibration-damping system. The combination of time-domain and frequency-domain analysis, along with system parameter calculation, ensures a comprehensive solution for vibration analysis and shock absorber design.

9 System Integration

The strategic integration of the data acquisition system provides a reliable and precise solution for acquiring and analyzing vibration data to produce damping systems for managing machine tool vibrations.

Figure 57: Data Acquisition System connected to the computer.

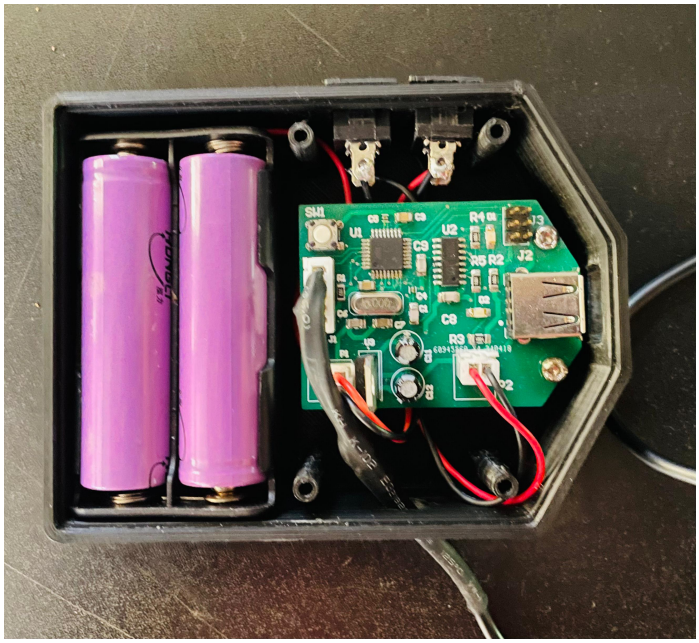


Figure 58: System Integration showing the internal wiring.

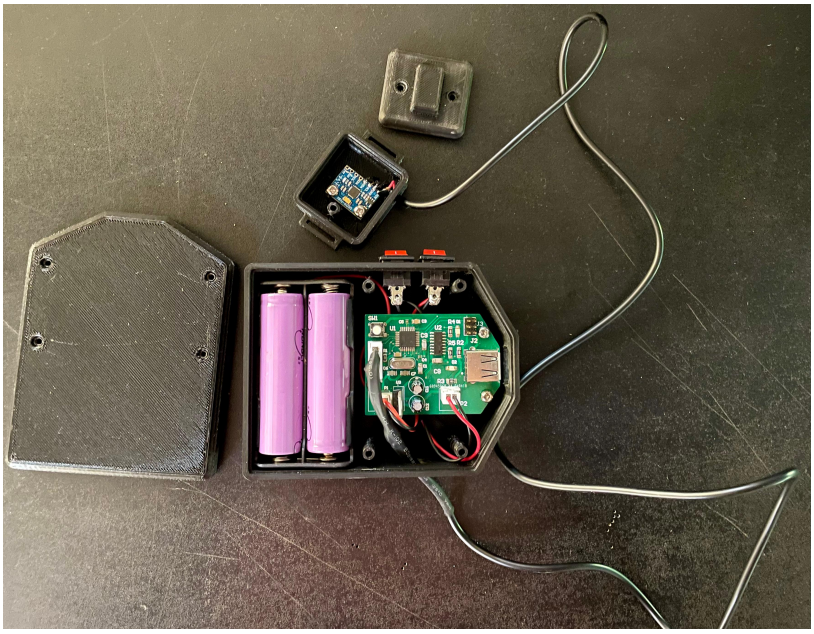


Figure 59: Data Acquisition Device.

Figure 60: Data Acquisition Device and the Shock Absorber made for the Bench Grinder.

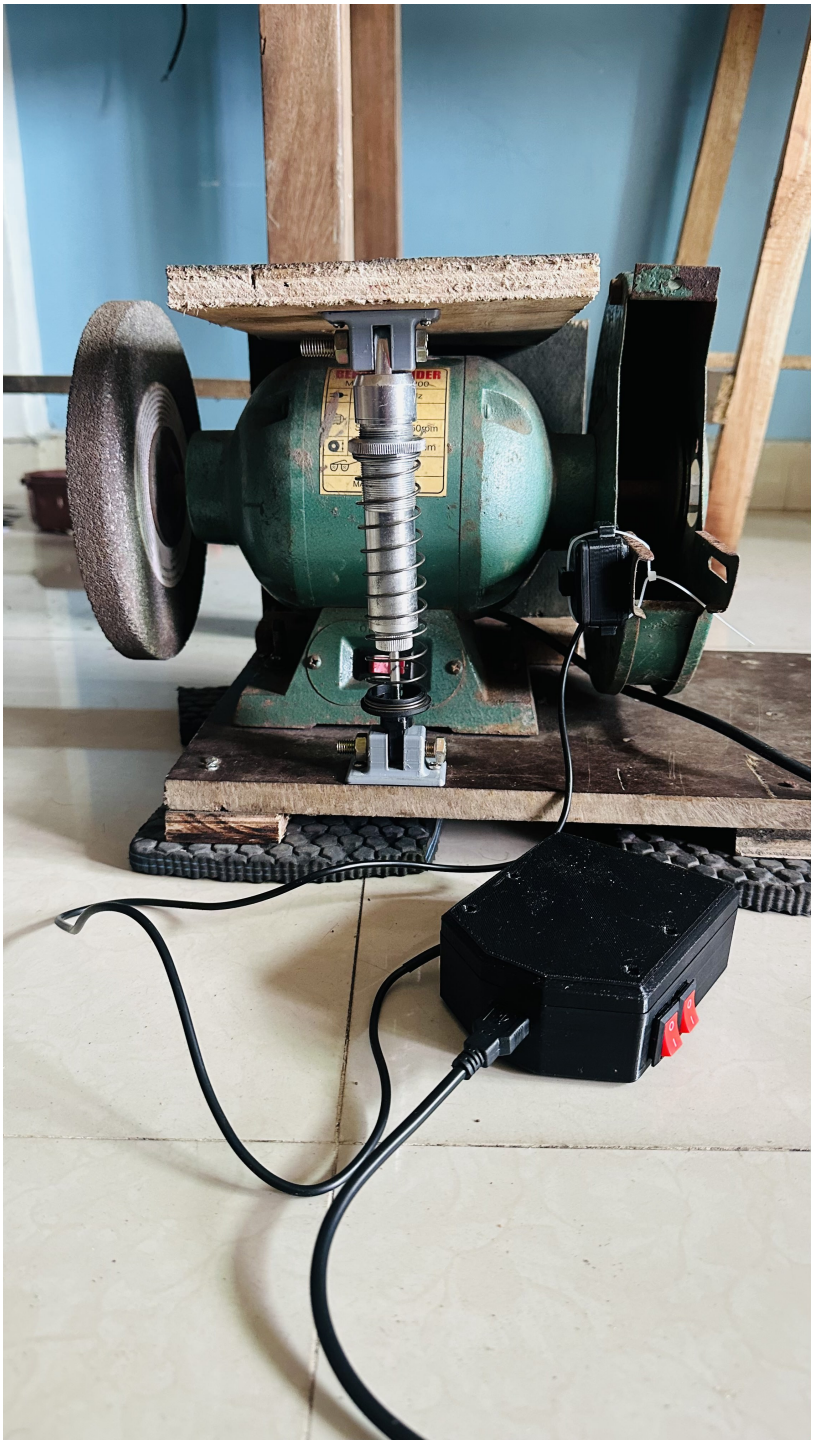


Figure 61: Fully functional Prototype.

A Appendix: Daily Log Entries

A.1 26 February – 3 March

A.1.1 Research Phase

- **Conducted Research:** Reviewed extensive literature on vibration damping and analyzed existing products in the industry to understand the current landscape.
- **Industry Standards:** Assessed the prevailing industry standards and cutting-edge technologies for efficient and precise vibration damping systems.
- **System Design Decision:** Based on the gathered insights, decided to focus on developing a system integrating a data acquisition device and a spring and damper system for vibration control.

A.2 4 March – 10 March

A.2.1 Conceptual Design Phase

- **Brainstorming Sessions:** Organized multiple brainstorming sessions to foster innovative ideas for accurately capturing vibrational data from machine tools.
- **Concept Development:** Created four different conceptual designs, each rigorously evaluated for its functionality, precision, cost-effectiveness, and ease of implementation.

A.3 11 March – 17 March

A.3.1 Design Evaluation Phase

- **Detailed Evaluation:** Conducted a comprehensive evaluation of the four conceptual designs, considering technical feasibility, functionality, precision, ease of assembly, serviceability, and overall performance.
- **Concept Selection:** After thorough analysis and deliberation, selected the most promising design for further development and prototyping.

A.4 18 March – 24 March

A.4.1 Component Selection Phase and Feasibility Check

- **Component Analysis:** Examined various accelerometers for data acquisition and microcontrollers for data collection and storage. Reviewed component options for the PCB design.

- **Feasibility Check:** Tested accelerometer readings using the I2C protocol to confirm the feasibility of the selected components.
  Navigate- 6.2 Measuring Acceleration Using MPU6050 Accelerometer

- **Equipment Finalization:** Finalized the selection of components through collaborative discussions and technical assessments to ensure they meet the project requirements.
  Navigate- 4.1 Component Selection and Justifications

- **Project Plan Development:** Developed a detailed project plan outlining key milestones, tasks, and a timeline to guide project progression.

A.5 25 March – 31 March

A.5.1 Design Phase

- **PCB Design:** Designed the PCB for the data acquisition system, ensuring compatibility with selected components.
  Navigate- 4.2 Schematics Design

- **Prototype Design:** Developed designs for the main unit of the data acquisition system and the accelerometer holding part using SolidWorks.
  Navigate- 5 SolidWorks Design

Design Tree

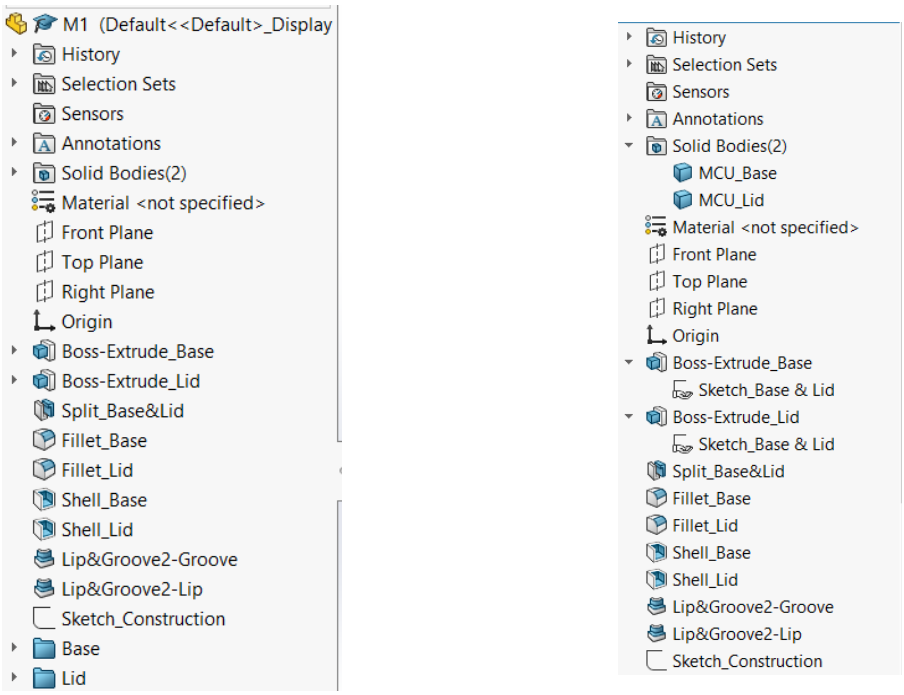


Figure 62: Design Tree of Main Controller Unit

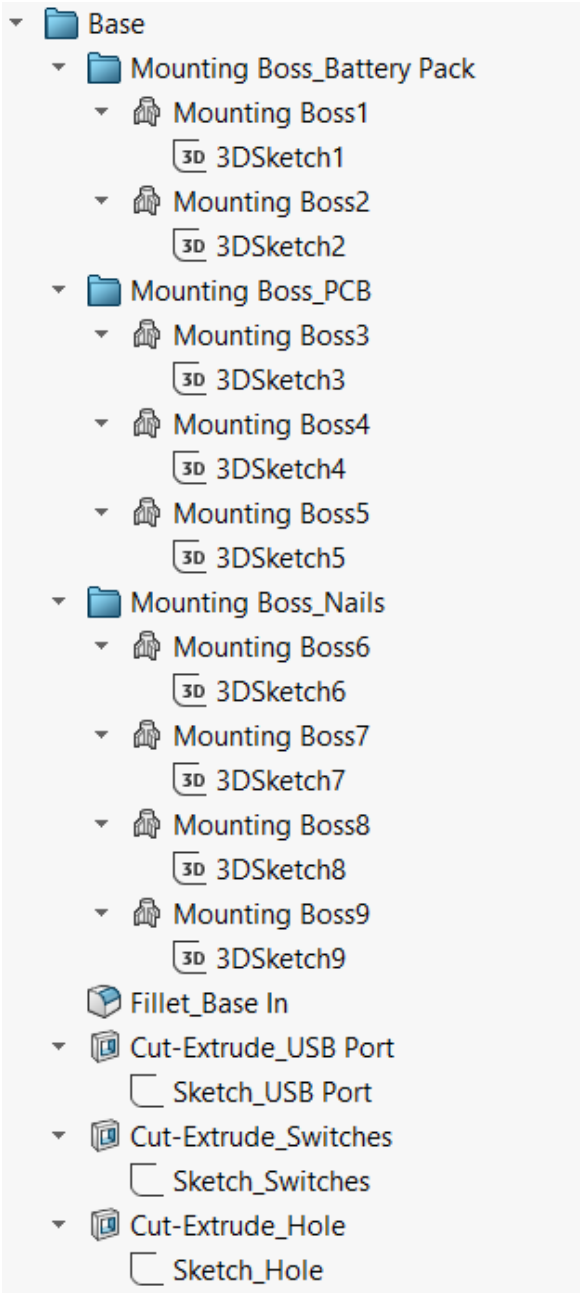


Figure 63: Design Tree of Main Controller Unit Base

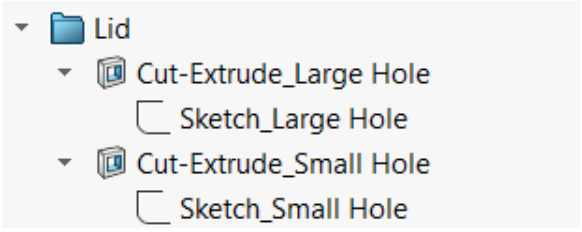


Figure 64: Design Tree of Main Controller Unit Lid

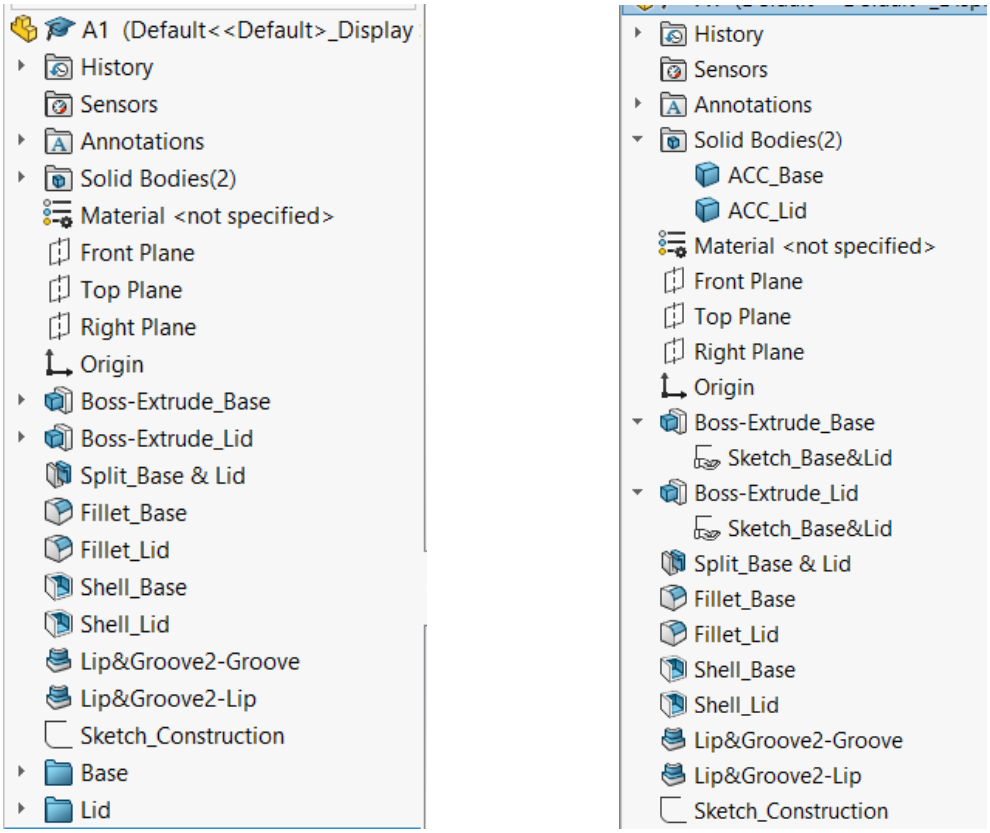


Figure 65: Design Tree of Accelerometer

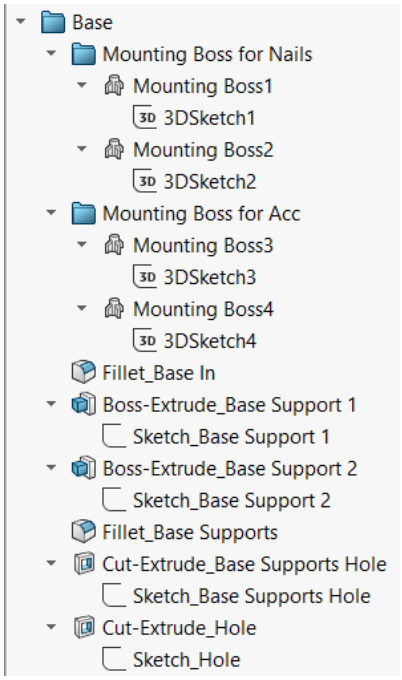


Figure 66: Design Tree of Accelerometer Base

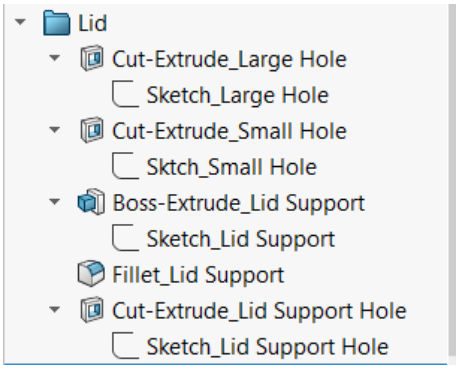


Figure 67: Design Tree of Accelerometer Lid

A.6 1 April – 7 April

A.6.1 PCB Finalization

- **PCB Finalization:** Completed the final design of the PCB and sent it for manufacturing at JLC PCB.

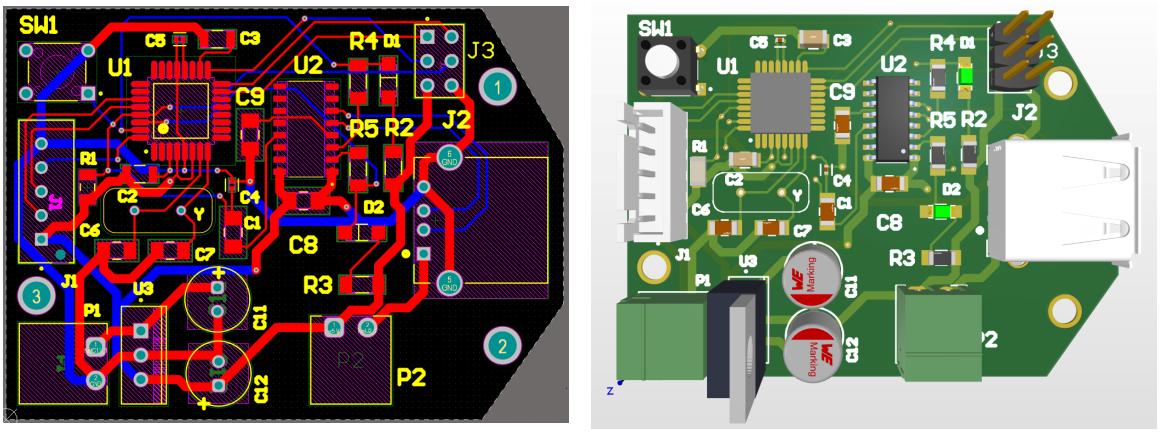


Figure 68: 2D and 3D View

A.6.2 Initial Experiments

- **Initial Experiments:** Conducted preliminary experiments with the accelerometers to gather data and perform Frequency Response Function (FRF) analysis.

A.7 8 April – 14 April

A.7.1 Data Processing using MATLAB

- **Detailed Analysis:** Performed a comprehensive analysis of the vibrational data collected using MATLAB to calculate the FRF and determine the necessary parameters for the damping system.
  Navigate- 6.3 MATLAB Analysis

A.8 15 April – 21 April

A.8.1 Component Arrival & Preparation

- **Component Receipt:** Received the fabricated PCBs and all other ordered components.

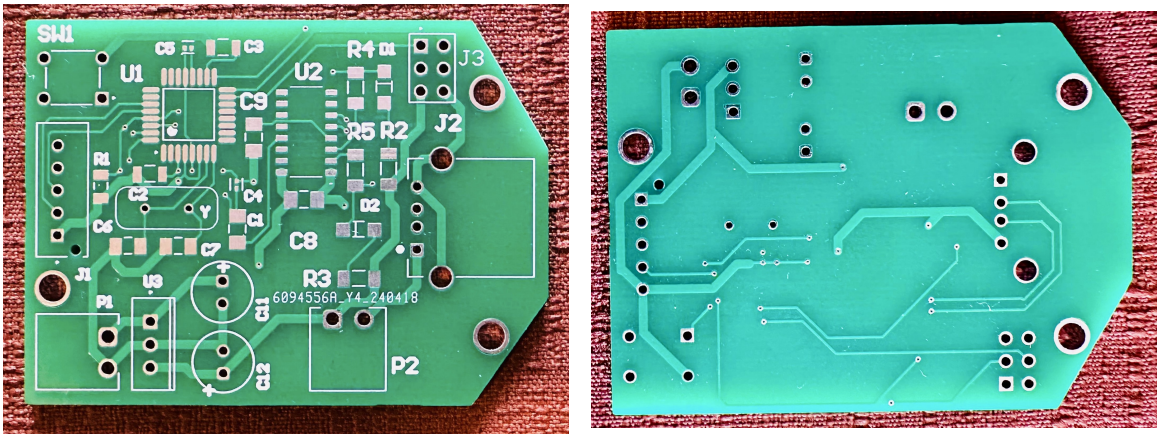


Figure 69: Bare PCB

Figure 70: Manufactured PCB

- **Quality Inspection:** Inspected the PCBs for quality assurance and compliance with the design specifications.
- **Organization:** Sorted and organized components to streamline the assembly process.

A.9 22 April – 28 April

A.9.1 PCB Assembly & Soldering

- **Component Placement:** Began placing and soldering components on the PCB, starting with smaller components and progressing to larger ones.
- **Inspection:** Used a microscope to inspect solder joints and ensure proper connections.

- **Completion:** Completed the assembly and soldering of the PCB, followed by cleaning to remove any flux residue. Conducted final inspections to address any issues.

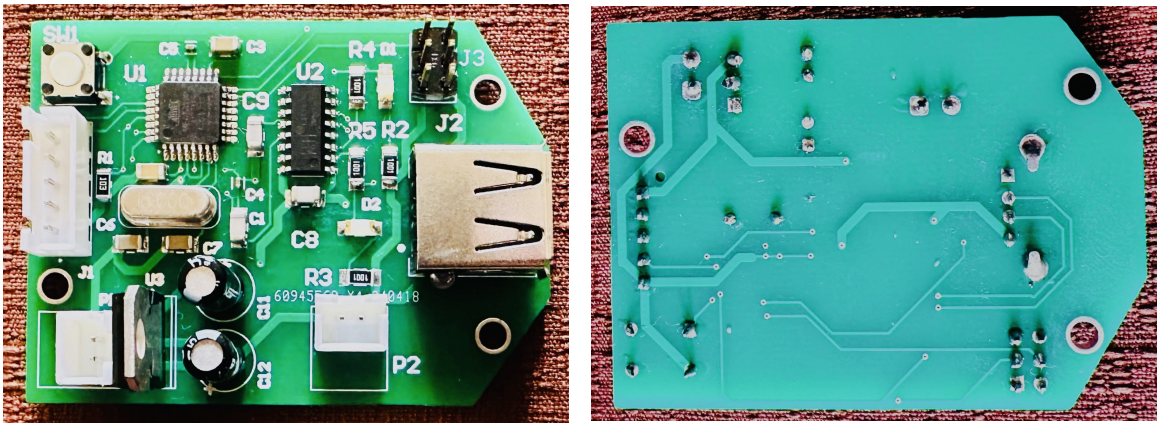


Figure 71: Soldered PCB

Figure 72: Soldered PCB

A.10 29 April – 5 May

A.10.1 Testing & Troubleshooting

- **Initial Testing:** Conducted initial power-up tests for the assembled PCB.
- **Functional Testing:** Performed detailed functional tests, including voltage and current measurements, to verify the system's performance.
- **Communication Verification:** Ensured proper communication between the microcontroller unit (MCU) and the computer.

A.11 6 May – 12 May

A.11.1 Final Adjustments

- **Software and Hardware Adjustments:** Made final adjustments to both software and hardware based on the results from the testing phase.

A.11.2 Enclosure Finalization

- **3D Printing and Integration:** 3D printed the enclosure and integrated it with the assembled PCB and wiring to complete the housing.

Figure 73: Main Controller Unit Enclosure

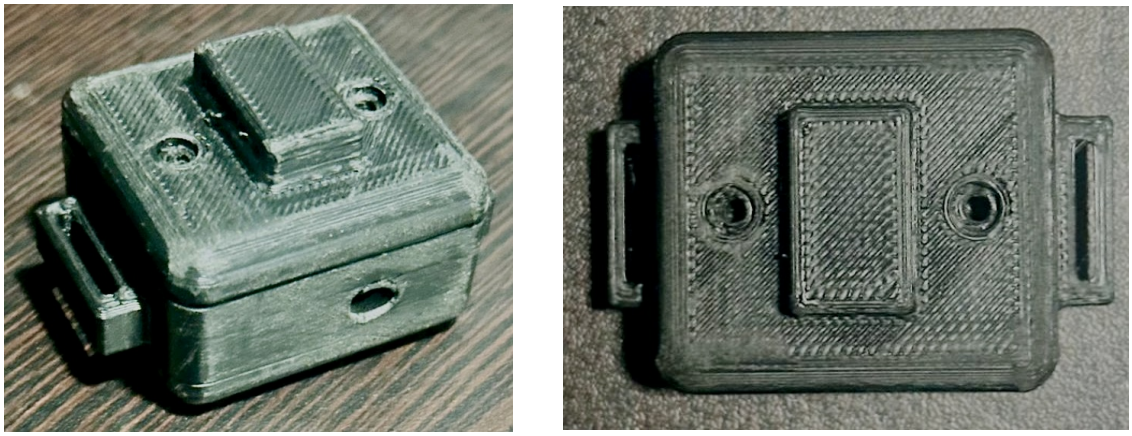


Figure 74: Accelerometer Unit Enclosure

- **Assembly:** Ensured a secure fit and proper mounting of the PCB within the enclosure.

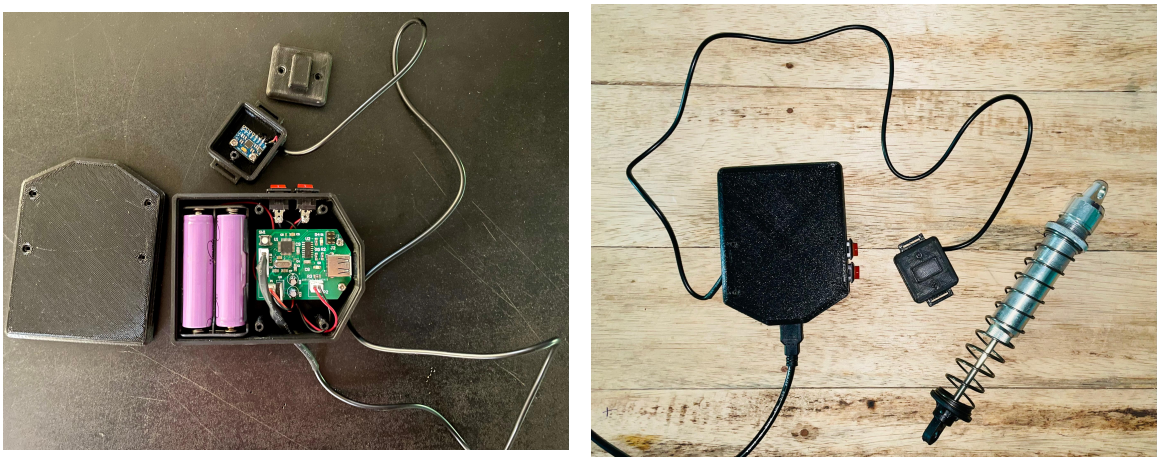


Figure 75: Fully Functional Data Acquisition Device.

A.12 13 May – 19 May

A.12.1 Shock Absorber Manufacturing

- **Data Analysis:** Analyzed vibration data from the Data Acquisition Device tailored for the bench grinder.
- **Spring Manufacturing and Damping Fluid Replacement:** Manufactured a custom spring with the calculated optimal diameter for the shock absorber based on this data. Selected and replaced the damping fluid in the shock absorber to achieve the desired viscosity and damping properties, enhancing the system's performance.

A.12.2 System Integration and Testing

- **Shock Absorber Assembly:** Integrated the newly manufactured spring and optimized damping fluid into the shock absorber system.
- **Performance Validation:** Conducted rigorous testing to validate the improvements in vibration reduction and efficiency, confirming the shock absorber's effectiveness post-optimization.

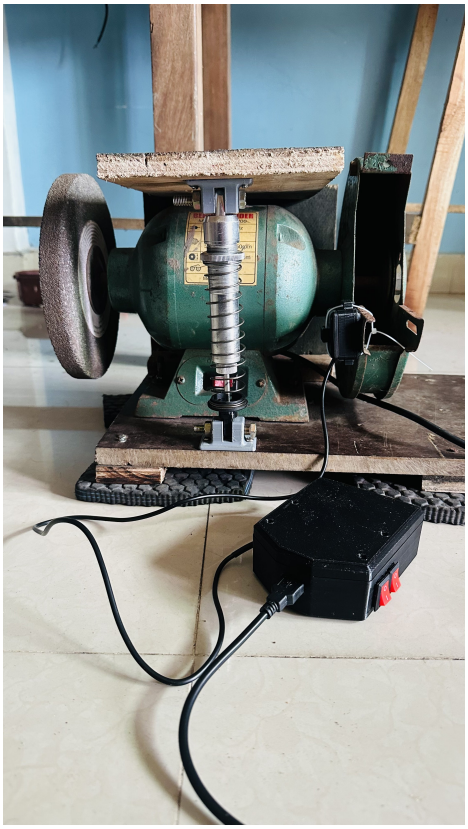


Figure 76: Shock Absorber Integration.

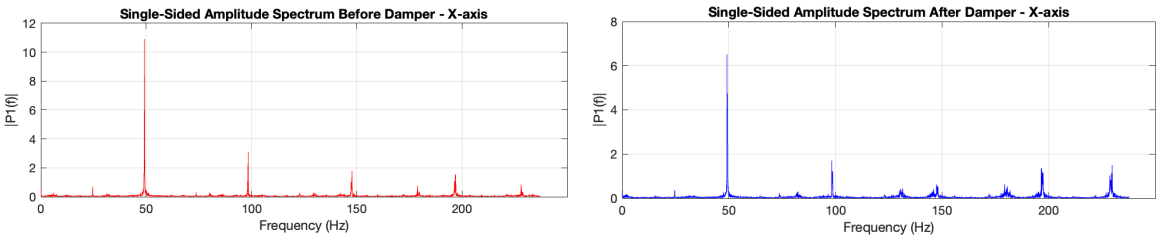


Figure 77: Performance Validation.

A.13 20 May – 15 June

A.13.1 Final Report Preparation

- **Report Compilation:** Compiled and finalized the project reports, including the Design Methodology Document and the Design Details Document. Detailed the design process, challenges encountered, and solutions implemented throughout the project.

B Appendix: Arduino code for the MPU6050 accelerometer used earlier

Below is the Arduino code used earlier for reading accelerometer data using the MPU6050 .

```
#include <Wire.h>
#include <MPU6050.h>

// Create an instance of the MPU6050
MPU6050 accelgyro;

// Variables to store accelerometer readings
int16_t ax, ay, az;

// Function to initialize the MPU6050
void initializeMPU6050() {
  Serial.println("Initializing MPU6050...");
  accelgyro.initialize();
  if (!accelgyro.testConnection()) {
    Serial.println("MPU6050 connection failed");
    while (1);
  }
  Serial.println("MPU6050 initialized successfully");
}

void setup() {
  Serial.begin(9600);
  Wire.begin();
  initializeMPU6050();
}

void loop() {
  // Read accelerometer values
  accelgyro.getAcceleration(&ax, &ay, &az);

  // Get the current timestamp
  unsigned long timestamp = millis();

  // Send the accelerometer data over the serial port
  Serial.print(timestamp);
  Serial.print(",");
  Serial.print(ax);
  Serial.print(",");
  Serial.print(ay);
  Serial.print(",");
  Serial.println(az);

  delay(10);
}
```

Listing 8: Arduino code for MPU6050 Accelerometer Data Acquisition

C Appendix: Derivation of the Transfer Function for a Shock Absorber

The mechanical system of a shock absorber can be modeled as a mass m attached to a spring and a damper. The input force $f(t)$ causes displacement $y(t)$ of the mass.

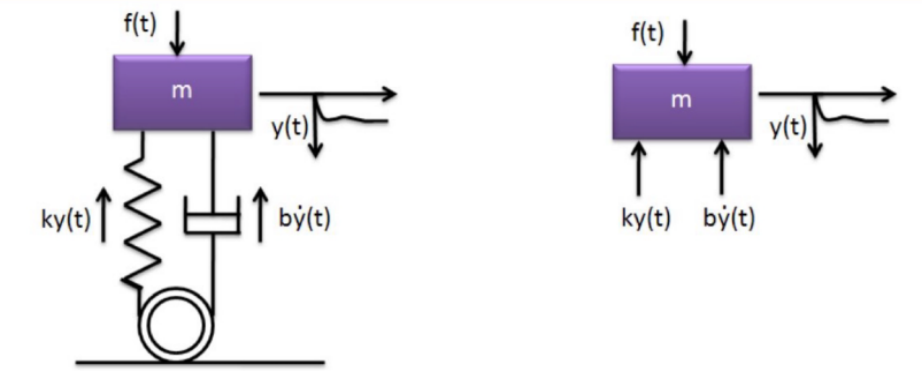


Figure 78: Representation of a shock absorber system

Differential Equation:

The motion of the mass is governed by the equation:

$$m\ddot{y}(t) = f(t) - ky(t) - b\dot{y}(t)$$

where:

- k is the spring constant
- b is the damping coefficient

Rearranging:

Dividing by m and rearranging the terms, we get:

$$\ddot{y}(t) + \frac{b}{m}\dot{y}(t) + \frac{k}{m}y(t) = \frac{1}{m}f(t)$$

Let:

- $\rho = \frac{k}{m}$
- $2\sigma = \frac{b}{m}$
- $\eta = \frac{1}{m}$

The equation simplifies to:

$$\ddot{y}(t) + 2\sigma\dot{y}(t) + \rho y(t) = \eta f(t)$$

Laplace Transform:

Applying the Laplace transform (assuming zero initial conditions for simplicity), the equation becomes,

$$s^2Y(s) + 2\sigma sY(s) + \rho Y(s) = \eta F(s)$$

Rearranging gives:

$$(s^2 + 2\sigma s + \rho)Y(s) = \eta F(s)$$

Transfer Function:

The transfer function $G(s)$ relating the output $Y(s)$ to the input $F(s)$ is then,

$$G(s) = \frac{Y(s)}{F(s)} = \frac{\eta}{s^2 + 2\sigma s + \rho}$$

This transfer function represents the behavior of the shock absorber system in the frequency domain, capturing the effects of both the spring and the damper.

References

[1] Microchip Technology, 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash. Version 7810D–AVR–01/15. Available at: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

[2] Nanjing Qinheng Microelectronics, USB to Serial Port Chip. Sheet version 2C. Available at: https://www.mpja.com/download/35227cpdata.pdf

[3] TDK InvenSense, MPU-6050 Six-Axis (Gyro + Accelerometer) MEMS Motion-Tracking™ Device. Available at: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

[4] Mouser Electronics. Available at: https://www.mouser.com/

[5] JLCPCB. Available at: https://jlcpcb.com/

Declaration of Review

Reviewed by Wijesinghe C. D.

From group *Vibration damping system for surgical Robot arms*

Review comments

- The heatsink placement near some capacitors might impact their performance and negatively affect thermal management. This issue should be addressed to ensure optimal functioning of the capacitors.
- The report offers thorough frequency analysis and includes FFT for better understanding. It is well-organized, easy to handle, and comprehensively documented.

I hereby declare that I have review the design documentation for the project *Vibration Damping System for Machine Tools*, with thorough inspection of the schematic and mold designs. I confirm that the comments made by above are my own and reflects unbiased critiques of the device design methodology.

Date: 2024/06/10 **Signature:** 

Rupasinghe N.P.S.S. (210549D)
From Group K - 30

I have gone through the documentation of the Vibration Damping System for Machine Tools. The document consists of all the required items related to the project. With the necessary codes, graphs, pictures, and 3D views, they completed the document well.
They have hierarchical schematic design and documented enclosure design, ensuring comprehensive and detailed coverage of the project.

checked by
N.P.S.S. Rupasinghe
210549D.

Signature