1. INTRODUCTION

1.1 Project Description

Music Tuner is designed to cater to musicians and music enthusiasts who seek precision in tuning their instruments. This innovative application utilizes advanced audio processing techniques to accurately detect and display the pitch of musical notes. Whether you're a beginner learning to play a new instrument or a seasoned musician preparing for a performance, this tuner provides an essential tool for ensuring that your instrument sounds just right. The application is available on multiple platforms, making it accessible to a wide range of users.

The app offers a comprehensive set of features that cater to various musical needs. Key functionalities include real-time pitch detection, visual and audio feedback, and customizable tuning standards. The interface is user-friendly, with a clean design that allows users to quickly see their instrument's tuning status. Additionally, the app supports a wide range of instruments, from guitars and pianos to violins and wind instruments, making it a versatile tool for musicians of all kinds.

Leveraging state-of-the-art digital signal processing (DSP) algorithms ensures accurate pitch detection and analysis. The tuner uses a combination of Fast Fourier Transform (FFT) and machine learning techniques to provide precise tuning information. The app also features noise reduction capabilities, allowing it to function effectively even in less-than-ideal acoustic environments. This technological foundation not only ensures accuracy but also enhances the user experience by providing clear and reliable feedback.

This project has significant market potential, especially among amateur and professional musicians. As more people take up musical hobbies or seek to refine their skills, the demand for reliable and easy-to-use tuning tools is on the rise. Future development plans include expanding the app's capabilities to include advanced features such as multi-instrument tuning, historical tuning data tracking, and integration with music learning apps. The team is also exploring partnerships with music schools and educational institutions to promote the app as a standard tool in music education.

The tuner is designed with user experience and accessibility in mind. The app features an intuitive interface that is easy to navigate, making it suitable for users of all ages and skill levels. The visual tuning display is clear and easy to read, with indicators that show how close the played note is to the target pitch. For visually impaired users, the app includes voice feedback that announces the detected pitch and how it compares to the desired tuning. The app's settings allow users to adjust the sensitivity and responsiveness, catering to different playing styles and preferences.

In addition to its tuning capabilities, the project aims to foster a community of musicians and learners. The app includes access to a range of educational resources, such as tutorials on tuning techniques, tips for maintaining instrument tuning stability, and articles on music theory related to tuning. Users can also join online forums and discussion groups within the app to share their experiences, ask questions, and seek advice from more experienced musicians. This community aspect enhances the value of the app by providing a supportive environment for continuous learning and improvement.

2. LITERATURE SURVEY

Fundamentals of Audio Signal Processing

Provides detailed explanations of DSP techniques such as Fourier transforms, filtering, and spectral analysis. Offers foundational knowledge for implementing and understanding audio signal processing crucial for tuner applications [1].

Evaluation of Pitch Detection Algorithm

Comparative analysis of pitch detection algorithms like autocorrelation, cepstrum, and harmonic product spectrum. Highlights the effectiveness of each algorithm, with cepstrum and harmonic product spectrum being particularly accurate for music applications [2].

Real Sound Synthesis for Interactive Applications

Discusses various real-time audio synthesis and processing techniques, including physical modeling and sample-based synthesis. Provides practical approaches for creating interactive and responsive audio applications, enhancing real-time performance [3].

YIN, a Fundamental Frequency Estimator for Speech and Music

Introduces the YIN algorithm, which improves pitch detection by reducing pitch ambiguity and increasing robustness. Demonstrates higher accuracy and lower error rates compared to traditional methods, making it suitable for both speech and music applications [4].

Musical Tuning Systems

Explores the mathematical and theoretical foundations of various musical tuning systems, including just intonation and equal temperament. Offers insights into the relationship between frequency ratios and perceived musical harmony, aiding in the accurate mapping of frequencies to notes [5].

A Review of Single-Channel Noise Reduction Methods Provides an extensive review of various single-channel noise reduction techniques, including spectral subtraction and Wiener filtering. Essential for improving audio quality and frequency detection accuracy, especially in noisy environments [6].

Deep Learning for Music Information Retrieval

Examines the use of deep learning techniques, including convolutional and recurrent neural networks, for music information retrieval tasks. Demonstrates that deep learning significantly enhances pitch detection and classification accuracy compared to traditional methods [7].

The PyDub Library: A Simple and Easy Audio Processing Library in Python

Introduction and practical applications of the PyDub library for audio manipulation, including format conversion and signal processing.PyDub provides an accessible and efficient tool for audio data manipulation in Python, suitable for developing audio applications [8].

2.1 Existing and Proposed System

Hardware-Based Tuners

Traditional music tuners are often hardware-based devices that musicians use to tune their instruments. These include mechanical strobe tuners, needle-style chromatic tuners, and clip-on tuners. Strobe tuners are known for their high accuracy and are often used in professional settings. They work by creating a visual strobing effect when the instrument's pitch matches the desired frequency. Needle-style tuners use a needle to indicate whether the pitch is sharp, flat, or in tune, while clip-on tuners attach directly to the instrument and detect vibrations. These tuners are effective and reliable but can be costly and bulky, making them less convenient for everyday use.

Software-Based Tuners

With the rise of smartphones and digital technology, software-based tuners have become increasingly popular. These tuners are available as mobile apps or desktop software and use the device's microphone to detect the pitch of the sound. They often include features like visual meters, audio feedback, and the ability to switch between different tuning. Some advanced software tuners incorporate noise reduction algorithms to improve accuracy in noisy environments. While software tuners are generally less expensive and more portable than hardware tuners, their accuracy can vary depending on the quality of the device's microphone and processing power.

Online Tuners

Online tuners are web-based applications that provide tuning capabilities without the need for installation. These tuners are accessible through a web browser and use the computer's microphone to capture the sound. They offer basic tuning functionality and are convenient for quick, on-the-go tuning needs. However, like software tuners, their performance can be limited by microphone quality and internet connectivity.

Proposed System

Integrated Multi-Platform System

The proposed system aims to integrate the best features of hardware, software, and online tuners into a cohesive multi-platform solution. This system would be available as a mobile app, desktop software, and web application, providing flexibility and accessibility to users across different devices. The core of this system would be a sophisticated pitch detection algorithm that combines Fast Fourier Transform (FFT) with machine learning techniques to enhance accuracy and reduce noise interference. This integrated approach ensures consistent performance across all platforms, regardless of the device's hardware capabilities.

Enhanced User Interface and Experience

The proposed system would feature an enhanced user interface (UI) and user experience (UX) designed to be intuitive and accessible. Key elements include a clear and responsive visual display, customizable settings for sensitivity and tuning standards, and multilingual support. The system would also offer auditory and visual feedback options, catering to users with different preferences and needs, including those with visual or hearing impairments. Additionally, the interface would include educational resources, such as tutorials on tuning techniques and tips for maintaining instrument tuning stability, enhancing the overall user experience.

Advanced Features and Customization

To meet the diverse needs of musicians, the proposed system would offer advanced features such as multi-instrument support, allowing users to tune a wide range of instruments, including guitars, pianos, violins, and wind instruments. It would also support microtonal tuning and alternative tuning systems, catering to musicians from various cultural and musical backgrounds. Another innovative feature would be the inclusion of a historical tuning data tracker, which would allow users to monitor their instrument's tuning history and identify trends or patterns. This feature could be particularly useful for professional musicians and educators.

2.2 Feasibility Study

The feasibility study for the Music Tuner project indicates strong potential across technical, economic, and market dimensions. Technically, the integration of advanced digital signal processing (DSP) algorithms, such as Fast Fourier Transform (FFT) and machine learning, is viable with current technology, ensuring accurate pitch detection and noise reduction. Economically, the project is cost-effective due to the scalability of software solutions and the widespread availability of mobile and desktop platforms, reducing hardware costs. The market analysis shows a growing demand for accessible and accurate tuning tools among amateur and professional musicians, supported by the increasing use of digital devices in music practice and performance. Additionally, the proposed system's features, including multi-instrument support, customizable settings, and educational resources, enhance its competitive edge, making it a promising investment. The feasibility study consists of three main parts:

2.2.1 Technical Feasibility

The technical feasibility of the Music Tuner project is well-supported by the availability of advanced digital signal processing (DSP) techniques, such as Fast Fourier Transform (FFT), which provide precise pitch detection capabilities. The integration of machine learning algorithms further enhances accuracy by improving noise reduction and adapting to various acoustic environments. The project's software can be developed for multiple platforms, including mobile, desktop, and web, leveraging widely used programming languages and frameworks. The implementation of a user-friendly interface with customizable features is achievable with current UI/UX design practices. Additionally, the use of cross-platform development tools ensures consistent performance across different devices, making the technical aspects of the project highly feasible.

2.2.2 Operational Feasibility

The operational feasibility of the Music Tuner project is strong, supported by streamlined processes for development, deployment, and maintenance. The project can be efficiently managed with a team of software developers, audio engineers, and UX/UI designers, leveraging established agile methodologies to ensure timely and iterative updates. The use of cloud-based infrastructure facilitates easy deployment and scalability, allowing for quick adaptation to user demand and feature updates. The app's architecture can incorporate analytics to monitor usage patterns and identify areas for improvement, ensuring continuous enhancement of the user experience. The system's ability to provide accurate tuning across a wide range of instruments and tuning standards ensures it meets the diverse needs of musicians, enhancing its operational success.

In terms of support and maintenance, the project can implement automated testing and continuous integration/continuous deployment (CI/CD) pipelines to streamline updates and bug fixes. User support can be efficiently managed through integrated help centers, FAQ sections, and community forums within the app, along with dedicated customer service channels for more complex issues. Training and educational resources can be provided to users to help them maximize the app's potential, further improving user satisfaction and retention. The Music Tuner's design, focusing on usability and accessibility, ensures that the app can cater to a broad demographic, including users with varying levels of technical expertise, making its operation feasible and sustainable in the long term.

2.2.3 Economic Feasibility

The economic feasibility of the Music Tuner project is highly favorable, owing to its low initial development costs and potential for scalable revenue streams. The primary investment involves software development, which can be managed with a small, skilled team, reducing labor costs. By utilizing open-source technologies and existing frameworks, the project can minimize licensing fees and infrastructure expenses. The Music Tuner can be monetized through various models, such as a freemium model offering basic features for free with premium upgrades, in-app purchases, or subscriptions for advanced functionalities. Additionally, advertising partnerships and collaborations with music education institutions can provide supplementary income.

2.3 Tools and Technologies used

2.3.1 Python

Python is a high-level, versatile programming language known for its readability and extensive libraries. It is used for developing the core functionalities of the audio tuner application, including real-time audio processing, graphical user interface (GUI) development, and handling various aspects of data management. Python is the primary language used to implement the audio tuner application, leveraging its powerful libraries and frameworks to achieve real-time audio processing, user interface design, and integration of various functionalities.

2.3.2 Tkinter

Tkinter is a standard GUI library for Python, providing tools to create interactive and visually appealing user interfaces. It is included with Python and allows for the development of desktop applications with ease. Tkinter is used to create the graphical user interface of the tuner application. It provides features such as buttons, labels, and entry fields, enabling users to interact with the application, view real-time feedback, and manage settings.

2.3.3 Pyaudio

PyAudio is a Python library that provides bindings for PortAudio, a cross-platform audio library. It facilitates audio input and output, allowing Python applications to capture and play audio. PyAudio is used for capturing real-time audio from the microphone. It handles the streaming of audio data, which is then processed to detect frequencies and musical notes. PyAudio ensures smooth and continuous audio processing essential for the tuner application.

2.3.4 NumPy

NumPy is a fundamental package for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. NumPy is used for numerical operations involved in audio processing, such as Fast Fourier Transform (FFT) calculations. It helps in converting time-domain audio signals into frequency-domain data, which is crucial for frequency detection.

2.3.5 Noisereduce

Noisereduce is a Python library designed to reduce noise in audio signals. It implements advanced noise reduction algorithms to enhance audio clarity. The Noisereduce library is used to process and clean audio signals before performing frequency detection. It helps in improving the accuracy of frequency detection by reducing background noise.

2.3.6 Wave

The Wave module in Python provides support for reading and writing WAV files. It allows for the manipulation and storage of audio data in the WAV format. The wave module is used to save recorded audio sessions in WAV format. This ensures high-quality playback and storage of audio data for later analysis or review.

2.3.7 Digital Signal Processing (DSP) Techniques

DSP techniques involve mathematical manipulation of signals to improve or alter their quality. This includes methods like filtering, spectral analysis, and pitch detection. DSP techniques are used to analyze and process audio signals for accurate frequency detection and note identification. Techniques such as noise reduction, pitch detection algorithms, and FFT contribute to the overall functionality of the tuner application.

2.4 Hardware and Software Requirements

Hardware Requirements

Table 2.4.1: Hardware Requirements

S.No	Particulars	Requirements
1	Hard Disk	120GB
2	Monitor	15’’ LED
3	Input Devices	Microphone
4	RAM	1GB

Software Requirements

Table 2.4.2: Software Requirements

S.No	Particulars	Requirements
1	Operating system	Windows 7+
2	Coding Language	Python
3	Tool	VS Code
4	Database	File System
5	Frontend	Tkinter
6	Backend	Python

3. SOFTWARE REQUIREMENTS SPECIFICATION

3.1 Users

The Music Tuner application must feature a clean, intuitive user interface that caters to both novice and experienced musicians. The main screen should provide a clear visual display of pitch accuracy with easy-to-read indicators, such as a needle gauge or strobe effect. Users should be able to access essential functions, including tuning options, sensitivity adjustments, and historical tuning data, through a straightforward navigation menu. The interface must be responsive and adaptable to different screen sizes and resolutions, ensuring a seamless experience across mobile devices, tablets, and desktop computers. The application should support a wide range of musical instruments, including guitars, pianos, violins, wind instruments, and more. It must offer customizable tuning settings for various instruments and tuning standards, such as A440, A432, and alternative temperaments.

The system should allow users to select or input their instrument type and desired tuning standard, automatically adjusting the pitch detection algorithms accordingly. Additionally, the app should support microtonal and culturally specific tuning systems to accommodate diverse musical traditions.

The Music Tuner must deliver high accuracy in pitch detection, leveraging advanced digital signal processing (DSP) techniques and algorithms such as Fast Fourier Transform (FFT). The application should function reliably in various acoustic environments, including noisy settings, by incorporating effective noise reduction and filtering mechanisms. Performance benchmarks should include real-time feedback with minimal latency, ensuring that users receive immediate and precise tuning information during practice or performance. Users should have the ability to customize their tuning experience according to their preferences. This includes adjusting sensitivity levels, selecting different visual and auditory feedback options, and configuring tuning standards. The app should offer settings for tuning modes (e.g., automatic or manual), allowing users to switch between modes based on their needs

The Music Tuner application must adhere to strict data privacy and security standards. It should collect only essential data related to user preferences and app performance, with robust mechanisms to anonymize and protect this data. The app should implement encryption protocols for data transmission and storage to safeguard user information from unauthorized access. Additionally, clear privacy policies and user consent mechanisms should be in place to ensure transparency and compliance with relevant data protection regulations.

3.1.1 Scope and Objective

The Music Tuner project encompasses the development of a versatile software application designed to assist musicians in tuning a wide range of instruments with high accuracy. The application will be available on multiple platforms, including mobile devices, desktops, and web browsers, providing users with a consistent and reliable tuning tool regardless of their device. The scope includes the implementation of advanced digital signal processing (DSP) algorithms, customizable tuning options, and an intuitive user interface to cater to diverse musical needs. It also involves integrating features for noise reduction, multi-instrument support, and culturally specific tuning systems. The project will focus on delivering a solution that is both accessible and adaptable to various acoustic environments and user preferences.

Objectives

The primary objective of the Music Tuner project is to deliver a highly accurate, user-friendly, and versatile tuning tool that meets the diverse needs of musicians. The specific objectives are,

- **Accuracy**: To provide precise and reliable pitch detection using advanced algorithms, ensuring that users can tune their instruments with confidence and in various acoustic conditions.
- **Accessibility**: To make the application available on multiple platforms, allowing users to access tuning tools anytime and anywhere, enhancing convenience and usability.
- **Usability**: To design an intuitive interface that simplifies the tuning process, making it accessible to both novice and experienced musicians through clear visual and auditory feedback.
- **Customization**: To offer extensive customization options, enabling users to adjust tuning standards, sensitivity, and display settings according to their personal preferences and musical requirements.
- **Integration**: To enable seamless integration with other music tools and resources, providing a comprehensive solution for practice, performance, and music education.
- **Data Security**: To ensure robust data privacy and security measures, protecting user information and maintaining trust through transparent data handling practices.

3.1.2 Assumptions and Dependencies

Assumptions

- **Device Compatibility**: The application assumes that users will have access to modern devices with sufficient processing power and storage to support the software, including smartphones, tablets, and computers.
- **Internet Access**: It assumes that users will have access to the internet for downloading the app, updating features, and potentially for certain functionalities like cloud-based data storage or synchronization.
- **Microphone Quality**: The application assumes that users' devices have microphones with adequate quality to capture sound accurately for pitch detection. It also assumes that these microphones will be calibrated to handle a range of acoustic environments.
- **User Proficiency**: It is assumed that users will have varying levels of musical proficiency, from beginners to professionals, and the app will cater to all levels with appropriate features and user guidance.
- **Standard Tuning Systems**: The application assumes that users will primarily use standard tuning systems like A440, but it will also include options for alternative tunings and microtonal systems to accommodate diverse musical practices.
- **Environmental Noise**: The app assumes that it will need to function effectively in environments with varying levels of ambient noise, incorporating noise reduction features to maintain accuracy in less-than-ideal acoustic conditions.
- **Device Permissions**: It assumes that users will grant necessary permissions for microphone access and, where applicable, location services for features like regional tuning standards or educational content.
- **User Preferences**: The app assumes that users will want customizable settings for tuning accuracy, visual and auditory feedback, and instrument profiles, requiring a flexible design to accommodate these preferences.
- **Maintenance and Updates**: It assumes that regular updates and maintenance will be required to address bugs, improve performance, and introduce new features, with users expected to install updates to benefit from these improvements.

Dependencies

- **Operating Systems**: The development of the Music Tuner depends on compatibility with multiple operating systems, including iOS, Android, Windows, and macOS, to ensure broad accessibility.
- **Digital Signal Processing Libraries**: The project relies on DSP libraries and frameworks for accurate pitch detection and real-time audio processing.
- **Microphone Access**: The app depends on the device’s microphone capabilities to capture sound for tuning, requiring access permissions and high-quality audio input.
- **Development Tools and Frameworks**: The development of the Music Tuner depends on various programming languages and development frameworks for building cross-platform applications.
- **Cloud Services**: For features like data storage, user synchronization, and updates, the project depends on cloud services
- **Audio Processing Algorithms**: The app relies on advanced algorithms for pitch detection and noise reduction, which are critical for providing accurate and reliable tuning results.
- **User Interface Design Tools**: The design and development of the user interface depend on tools and frameworks for creating a responsive and intuitive UI/UX
- **Network Connectivity**: For features such as cloud synchronization, online resources, and updates, the project depends on stable internet connectivity.
- **Privacy and Security Compliance**: The app depends on adherence to data protection regulations and standards to ensure user data privacy and security.
- **Testing and Debugging Tools**: The project relies on tools for automated testing, debugging, and performance monitoring.

3.2 Functional Requirements

Table 3.2.1: Functional Requirements

ID	Functional Requirements	Description
FR1	Real-Time Frequency Detection	The system captures and analyzes audio in real-time to determine the current pitch.
FR 2	Accurate Note Identification	Converts detected frequencies into musical notes, providing precise information about the pitch.
FR 3	Graphical User Interface (GUI)	Utilizes Tkinter to create an intuitive and user-friendly interface.
FR 4	Recording and Playback	Allows users to record audio sessions and save them as WAV files for later analysis.
FR 5	Logging and History	Logs detected notes and frequencies with timestamps, providing a history view for review.
FR 6	Noise Reduction Settings	Includes adjustable noise reduction settings to improve accuracy in various environments.
FR 7	Multi-threading for Performance	Uses threading to handle audio processing and GUI updates simultaneously, ensuring responsiveness.
FR 8	Settings and Customization	Allows users to adjust settings for volume division and noise reduction levels.
FR 9	File Management	Provides options for choosing the output directory for saving recordings and logs.

3.3 Non-Functional Requirements

Table 3.3.1: Non-Functional Requirements

ID	Non-functional Requirements	Description
NFR1	Scalability	The system should handle different noise levels and types of audio input effectively.
NFR 2	Performance	The application should process audio and update the GUI in real-time with minimal latency.
NFR 3	Usability	The GUI should be intuitive and easy to use for musicians and audio engineers.
NFR 4	Portability	The application should run on Windows, macOS, and Linux operating systems.
NFR5	Reliability	The application should consistently and accurately detect frequencies and notes.
NFR 6	Maintainability	The code should be well-documented and modular to facilitate easy updates and maintenance.
NFR 7	Security	Recorded audio files and logs should be securely stored and accessed.
NFR 8	Efficiency	The application should efficiently use system resources, including CPU and memory.
NFR 9	Compatibility	The application should be compatible with various microphone devices and audio formats.

4. SYSTEM DESIGN

4.1 System Architecture

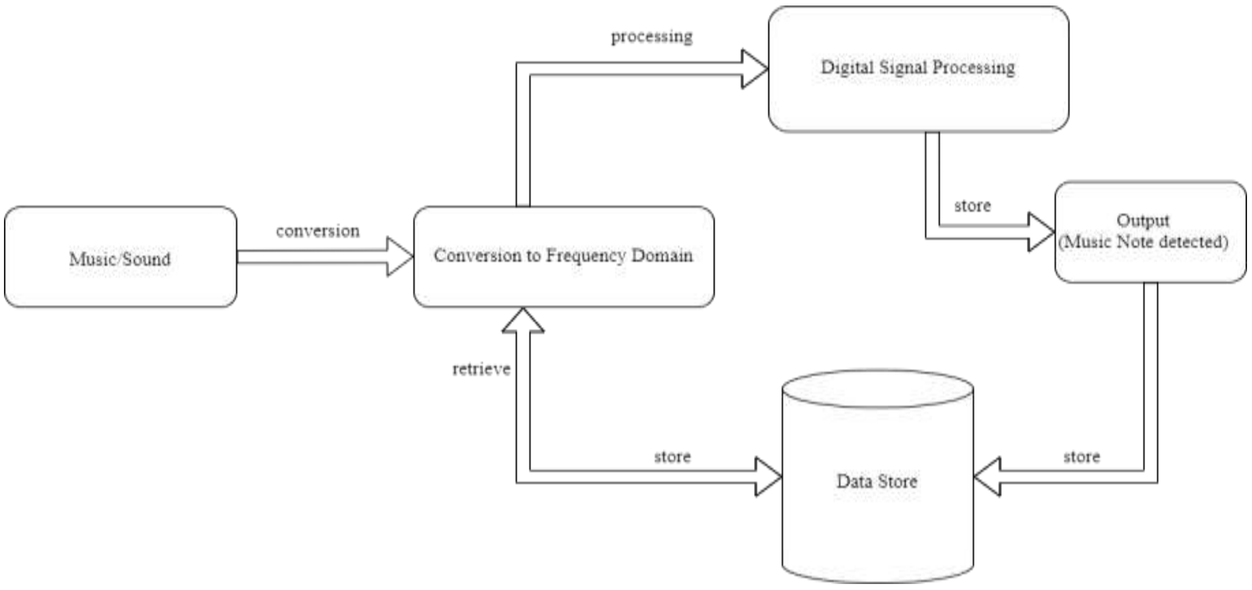


Fig 4.1 System architecture

In the Frequency Detection module, the application uses Fast Fourier Transform (FFT) to convert the time-domain audio signal into its frequency components. The NumPy library performs the FFT calculations, which analyze the frequency spectrum of the audio signal. This module identifies the dominant frequency from the FFT results, which corresponds to the pitch of the sound. The detected frequency is crucial for determining the musical note and provides the basis for the tuner’s feedback mechanism.

Once the frequency is detected, the Note Identification layer maps the frequency to a corresponding musical note. This module uses predefined frequency ranges for each musical note to perform the mapping. The identified note is then stabilized to prevent fluctuations caused by minor variations in the frequency. This layer ensures accurate and reliable note detection, which is essential for musicians seeking precise tuning.

The GUI layer, implemented using Tkinter, provides an interactive interface for users to interact with the application. It displays real-time information such as the detected frequency, musical note, and recording timer. Users can control the application through buttons for recording, stopping, and accessing settings. The GUI also includes features for saving recorded audio sessions and adjusting noise reduction settings. The integration of the GUI ensures that users have a seamless and intuitive experience while using the tuner application.

The final layer focuses on data logging and file management. As users interact with the application, the detected notes and frequencies are logged along with timestamps. This data is stored in a log file and can be reviewed later. The application also allows users to choose an output directory for saving recordings and logs. This module ensures that all data is systematically recorded and accessible for future reference or analysis, completing the overall functionality of the tuner application.

4.3 Context diagram

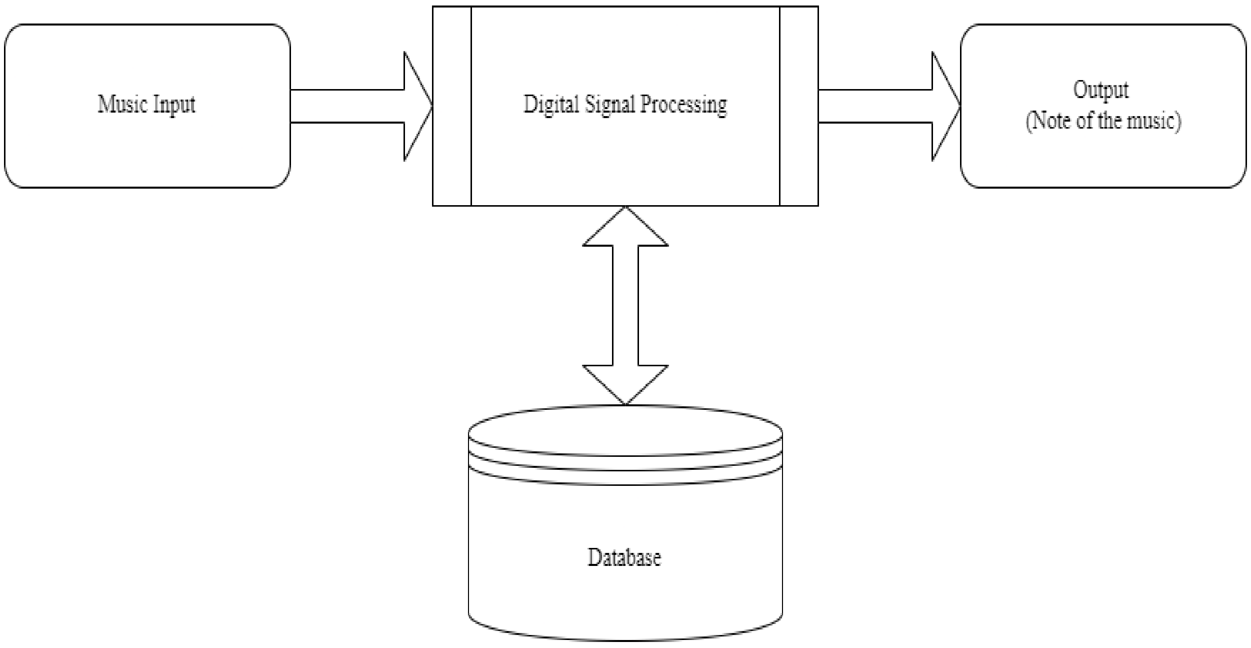


Fig 4.3 Context diagram

5. DETAILED DIAGRAM

5.1 Dataflow diagram

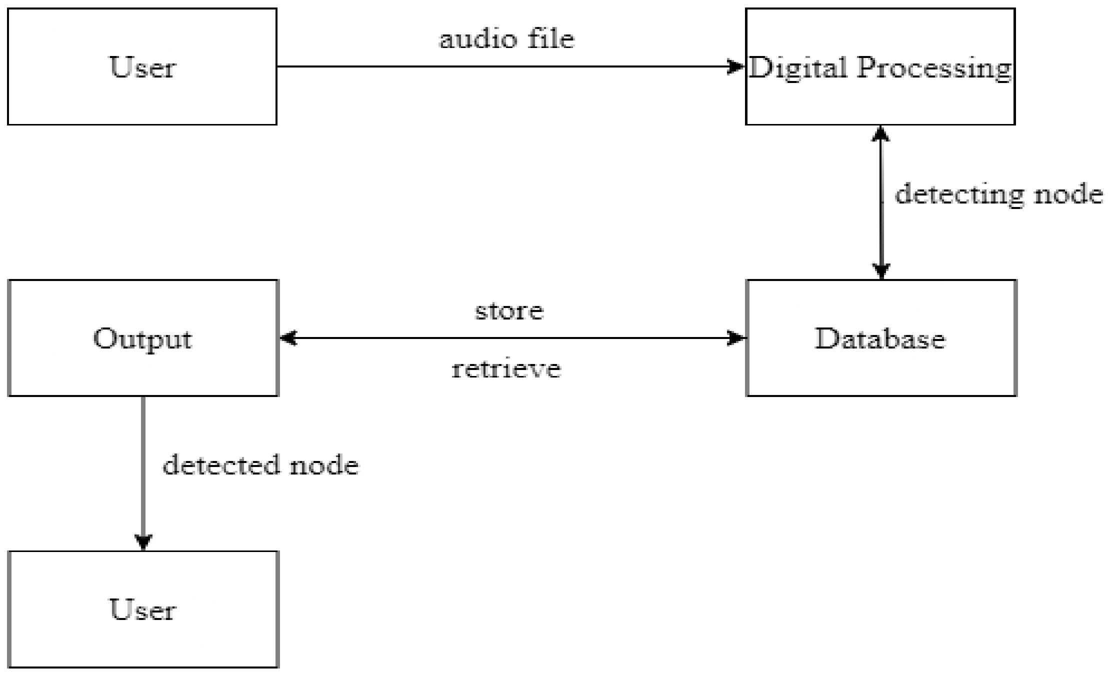


Fig 5.1.1 Dataflow Diagram Level 1

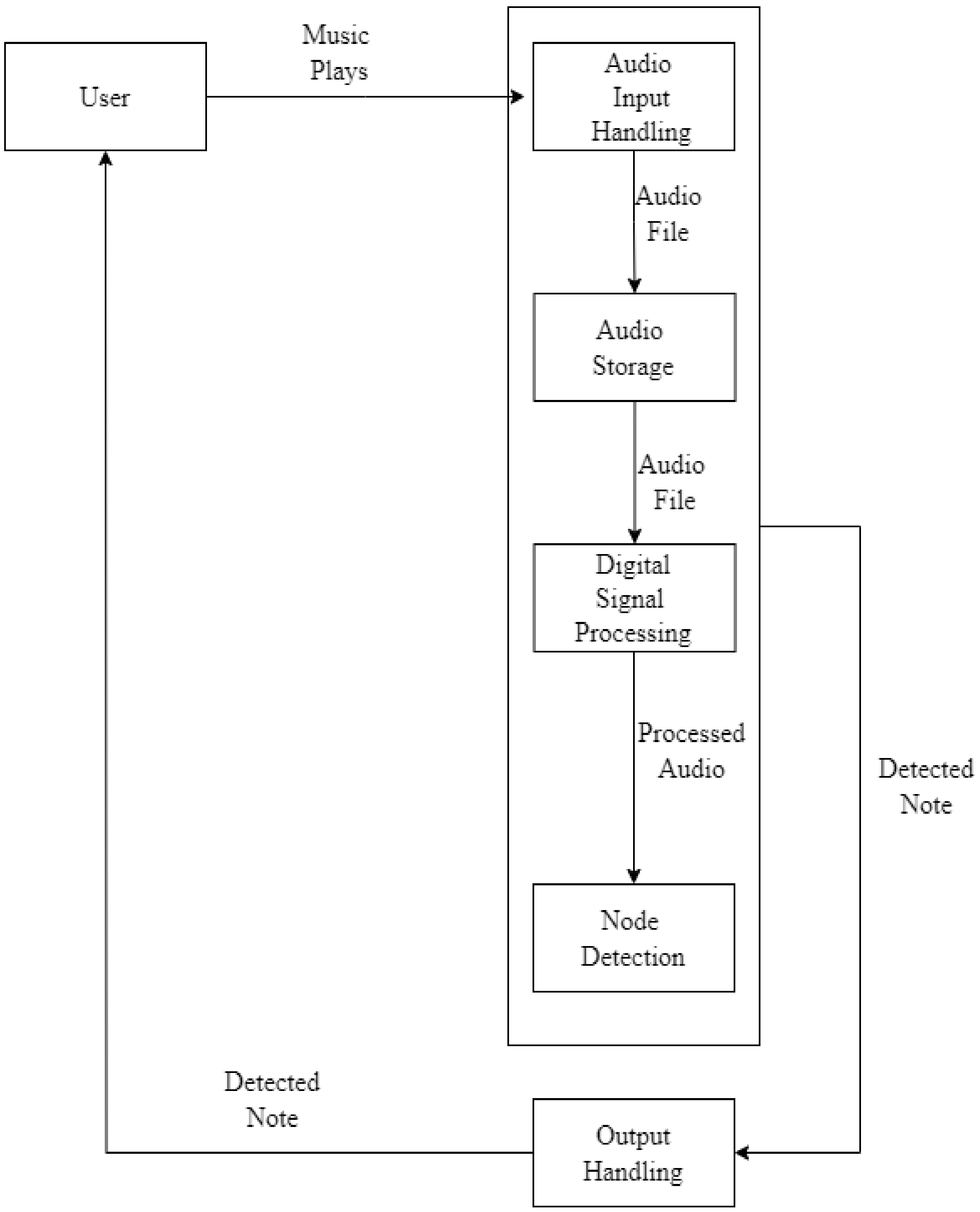


Fig 5.1.2 Dataflow Diagram Level 2

5.2 Activity Diagram

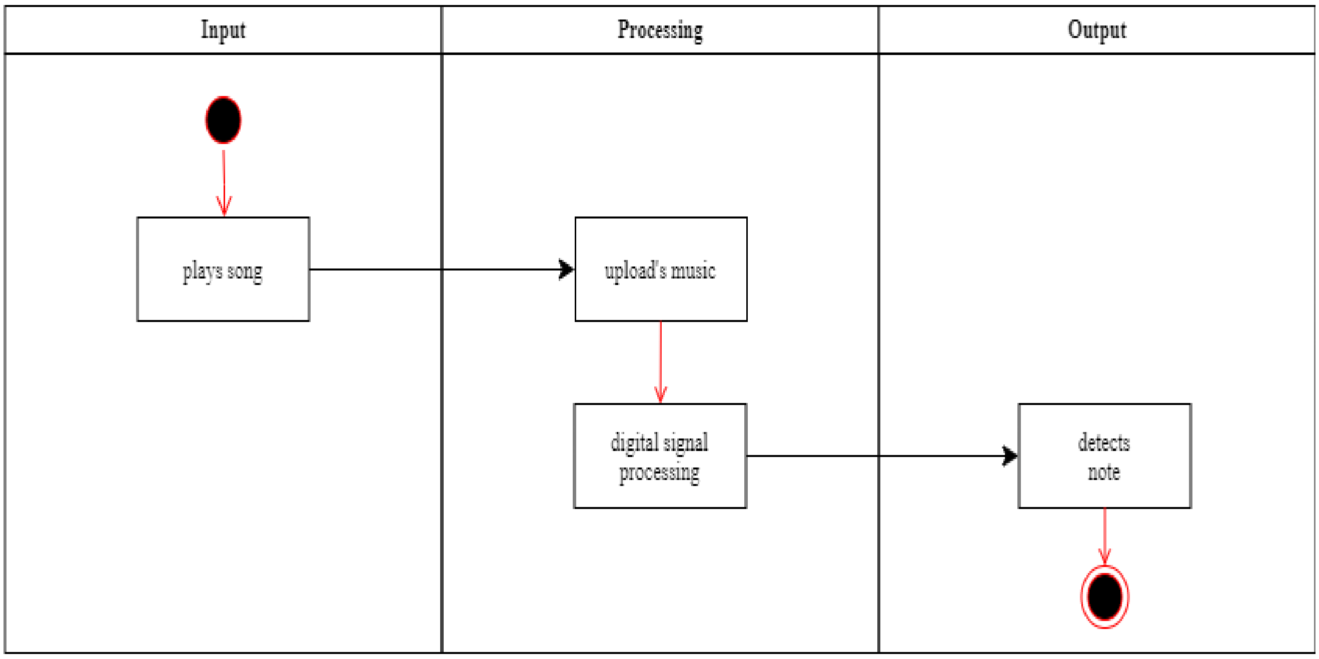


Fig 5.2 Activity Diagram

5.3 Sequence diagram

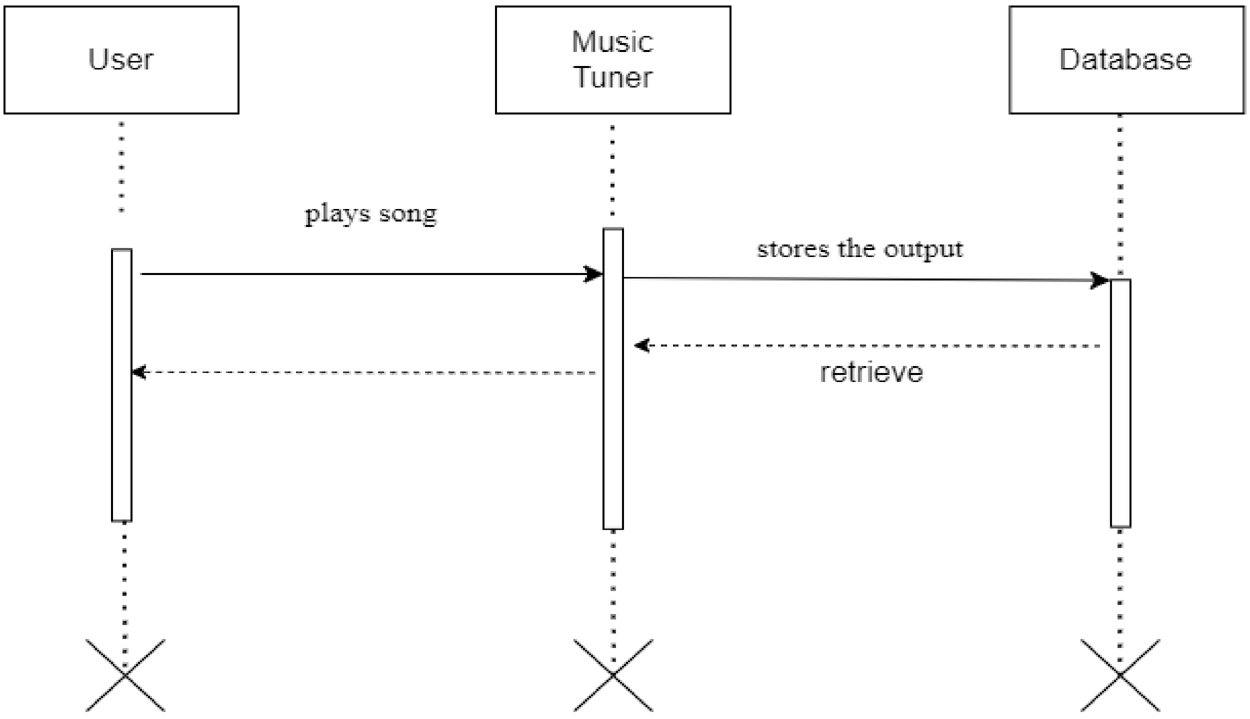


Fig 5.3 Sequence diagram

5.4 Use case diagram

Use case diagram (user)

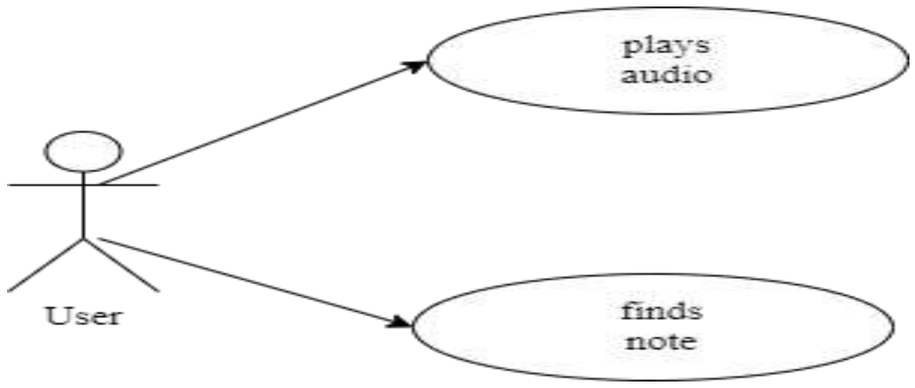


Fig 5.4.1 Use case (user) diagram

Use case diagram (detector):

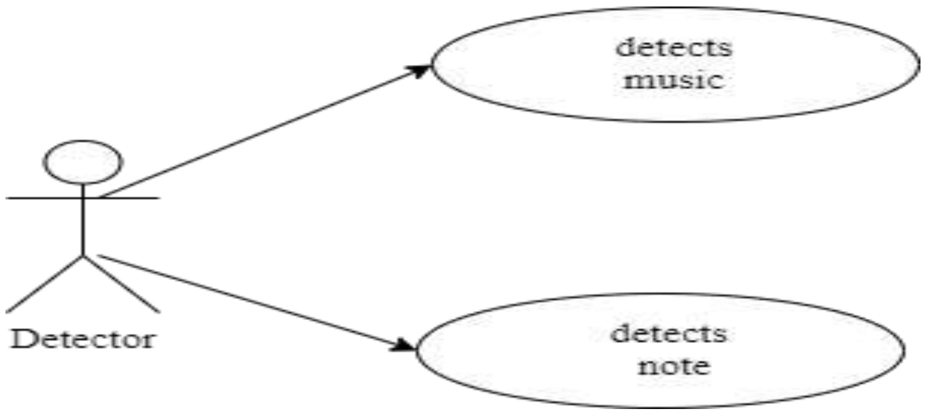


Fig 5.4.2 Use case (detector) diagram

6. IMPLEMENTATION

6.1 Snippet code

main.py

```
from tkinter import filedialog
from tkinter import *
import tkinter
from datetime import datetime
import os
from frequency import *
from note import *
import time
import pyaudio
import wave
import threading
root = tkinter.Tk()
root.title("Tuner")
root.geometry("490x400")
root.resizable(False,False)
root.configure(bg='#191919')

rec_img = PhotoImage(file = r"img/record.png")
stop_img = PhotoImage(file = r"img/stop.png")
output_img = PhotoImage(file = r"img/output.png")
settings_img = PhotoImage(file = r"img/settings.png")

current_frequency = ""
def detect_frequency():
    global current_frequency
        current_frequency   =   check_frequency(recording,   volume_division,
noise_reduction)
    frequency = str(current_frequency) + "Hz"
    frequency_label.config(text = frequency)
    root.update()
    return current_frequency
previous_note = "..."
current_note = "..."
```

```
def detect_note():
    global current_frequency, current_note, previous_note
    previous_note = current_note
    current_note = identify_note(current_frequency)
    if current_note == "C0":
        current_note = previous_note
    else:
        logs()
    note_label.config(text = current_note)
    root.update()
    return current_note
current_time = ""
def check_time():
    global current_time
    now = datetime.now()
       current_time  =  (now.strftime(f"%H:%M:%S:{round(now.microsecond  /
1000)}").ljust(12,"0"))
    return current_time
start_time = ""
current_timer = ""
def timer():
    global current_timer, start_time
    current_timer = time.time() - start_time
    current_timer = time.strftime("%H:%M:%S",time.gmtime(current_timer))
    timer_label.config(text = current_timer)
    root.update()
    return current_timer
output_directory = (os.getcwd() + "/output.txt").replace("\\", "/")
default_directory = output_directory
folder_selected = ""
def output():
    global default_directory, output_directory, folder_selected
    folder_selected = filedialog.askdirectory()
    if folder_selected == "":
        output_directory = default_directory
    output_directory_label.config(text = (" " + output_directory))
    return output_directory, folder_selected
```

```
def record_audio():
    global recording, folder_selected
    try:
        CHUNK = 1024
        RATE = 44100
        DEVICE = 0
        FILENAME = "/output.wav"
        p = pyaudio.PyAudio()
        stream = p.open(format=pyaudio.paInt16,
                        channels=2,
                        rate=RATE,
                        input_device_index=DEVICE,
                        frames_per_buffer=CHUNK,
                        input=True)
        frames = []
        def audio():
            global recording
            while (recording == "on"):
                data = stream.read(CHUNK)
                frames.append(data)
            stream.stop_stream()
            stream.close()
            p.terminate()
            wf = wave.open(folder_selected + FILENAME, 'wb')
            wf.setnchannels(2)
            wf.setsampwidth(p.get_sample_size(pyaudio.paInt16))
            wf.setframerate(RATE)
            wf.writeframes(b''.join(frames))
            wf.close()
        thread = threading.Thread(target=audio)
        thread.daemon = True
        thread.start()
    except:
        pass
history = ""
log = f"LOG {datetime.now()}\n"
```

```
def logs():
    global current_note, current_frequency, log, history
    current_time = check_time()
    try:
        if len(current_note) == 2:
            current_note = current_note + " "

        current_frequency = str(current_frequency) + ".0"
        if len(current_frequency) != 6:
            current_frequency = (current_frequency).ljust(6, "0")

        log = log + "\n" + current_note + "\t" + current_frequency + "Hz" +
"\t" + current_timer + "\t" + check_time()
        history = history + "\n" + current_note
        history_label.config(text = history)
        root.update()
    except:
        pass
recording = "off"
def record():
    global recording, start_time
    start_time = time.time()
    record_button["state"] = DISABLED
    settings_button["state"] = DISABLED
    stop_button["state"] = NORMAL
    recording = "on"
    record_audio()
    while (recording == "on"):
        timer()
        detect_frequency()
        detect_note()
def stop():
    global output_directory, recording, log
    stop_button["state"] = DISABLED
    settings_button["state"] = NORMAL
    record_button["state"] = NORMAL
    recording = "off"
    file = open(output_directory, "w")
    file.write(log)
    file.close()
    log = ""
    return log
def entry_int(inStr,active_type):
    if active_type == '1':
        if not inStr.isdigit():
            return False
     return True
volume_division = "10"
noise_reduction = "5"
def settings():
    top = Toplevel(root)
    top.geometry("355x130")
    top.resizable(False,False)
    top.configure(bg='#191919')
    def callback(self, P):
        if str.isdigit(P) or P == "":
            return True
        else:
            return False
    def save():
        global volume_division, noise_reduction
        volume_division = volume_division_entry.get()
        noise_reduction = noise_reduction_entry.get()
        top.destroy()
        top.update()
        return volume_division, noise_reduction
    volume_division_label = Label(top, bg='#191919', text="Volume division
value: ", font=("Bahnschrift", 15, "bold"), fg="white", anchor="w")
    volume_division_label.place(x=10, y=10, height=30, width=205)
    noise_reduction_label = Label(top, bg='#191919', text="Noise reduction
level: ", font=("Bahnschrift", 15, "bold"), fg="white", anchor="w")
    noise_reduction_label.place(x=10, y=50, height=30, width=205)
    volume_division_entry = Entry(top, justify=CENTER, relief="solid",
bg="#393939", bd=2, font=("Bahnschrift", 13, "bold"), fg="white",
validate="key")
```

```
    volume_division_entry.insert(0, volume_division)

                        volume_division_entry['validatecommand']          =
(volume_division_entry.register(entry_int),'%P','%d')

    volume_division_entry.place(x=215, y=10, height=30 ,width=130)

      noise_reduction_entry  =  Entry(top,  justify=CENTER,  relief="solid",
bg="#393939",   bd=2,   font=("Bahnschrift",   13,   "bold"),   fg="white",
validate="key")

    noise_reduction_entry.insert(0, noise_reduction)

                        noise_reduction_entry['validatecommand']          =
(noise_reduction_entry.register(entry_int),'%P','%d')

    noise_reduction_entry.place(x=215, y=50, height=30, width=130)

       save_button   =   tkinter.Button(top,   justify=CENTER,  compound="c",
relief="solid",    activebackground="#393939",    activeforeground="white",
bg="#393939",  fg="white",  text="SAVE",  font=("Bahnschrift",15,  "bold"),
command=save)

    save_button.place(x=10, y=90, height=30, width=335)

record_button    =    tkinter.Button(root,    compound="c",    relief="solid",
bg="#393939", image = rec_img, command=record)

record_button.place(x=10, y=10, height=50, width=50)

stop_button    =    tkinter.Button(root,    compound="c",    relief="solid",
bg="#393939", image = stop_img, command=stop, state=DISABLED)

stop_button.place(x=10, y=70, height=50, width=50)

settings_button   =   tkinter.Button(root,   compound="c",   relief="solid",
bg="#393939", image = settings_img, command=settings)

settings_button.place(x=10, y=130, height=50, width=50)

output_button    =    tkinter.Button(root,    compound="c",    relief="solid",
bg="#393939", image = output_img, command=output)

output_button.place(x=430, y=10, height=50, width=50)

output_directory_label.place(x=70, y=10, height=50, width=361)

frequency_label    =    Label(root,    bg="#393939",    fg="white",    bd=2,
relief="solid", text="0Hz", font=("Bahnschrift", 20, "bold"))

frequency_label.place(x=230, y=70, height=50, width=250)

timer_label = Label(root, bg="#393939", fg="white", bd=2, relief="solid",
text="00:00:00", font=("Bahnschrift", 20, "bold"))

timer_label.place(x=70, y=70, height=50, width=150)

history_label = Label(root, bg="#393939", fg="white", bd=2, relief="solid",
anchor="s", text=history, font=("Arial", 19, "bold"))

history_label.place(x=70, y=130, height=250, width=150)

note_label = Label(root, bg="#393939", fg="white", bd=2, relief="solid",
anchor="center", text="...", font=("Arial", 90, "bold"))

note_label.place(x=230, y=130, height=250, width=250)

root.mainloop()
```

frequency.py

```
from pyaudio import PyAudio, paFloat32, paInt16
import numpy as np
import time
import noisereduce as nr
import math
np.seterr(invalid='ignore')
def check_frequency(recording, volume_division, noise_reduction):
    try:
        CHUNK = 4096
        RATE = 44100
        DEVICE = 0
        p = pyaudio.PyAudio()
        stream = p.open(format=pyaudio.paInt16,
                        channels=2,
                        rate=RATE,
                        input_device_index=DEVICE,
                        frames_per_buffer=CHUNK,
                        input=True)
        def lower_volume(data):
            data = np.array((data//int(volume_division)))
            return data
        while (recording == "on"):
            indata = lower_volume(indata)
            for i in range(int(noise_reduction)):
                indata = nr.reduce_noise(y=indata, sr=RATE)
            fftData=abs(np.fft.rfft(indata))**2
            which = fftData[1:].argmax() + 1
            if which != len(fftData)-1:
                current_frequency = which*RATE/CHUNK
                current_frequency = int(current_frequency)
                return current_frequency
        stream.close()
        p.terminate()
    except:
        current_frequency = "0"
        return current_frequency
```

note.py

```
frequencies = {
"C0": 16.35,
"C#0": 17.32,
"D0": 18.35,
"D#0": 19.45,
"E0": 20.60,
"F0": 21.83,
"F#0": 23.12,
"G0": 24.50,
"G#0": 25.96,
"A0": 27.50,
"A#0": 29.14,
"B0": 30.87,
"C1": 32.70,
"C#1": 34.65,
"D1": 36.71,
"D#1": 38.89,
"E1": 41.20,
"F1": 43.65,
"F#1": 46.25,
"G1": 49.00,
"G#1": 51.91,
"A1": 55.00,
"A#1": 58.27,
"B1": 61.74,
"C2": 65.41,
"C#2": 69.30,
"D2": 73.42,
"D#2": 77.78,
"E2": 82.41,
"F2": 87.31,
"F#2": 92.50,
"G2": 98.00,
"G#2": 103.83,
"A2": 110.00,
"A#2": 116.54,
"C6": 1046.50,
"C#6": 1108.73,
"D6": 1174.66,
"D#6": 1244.51,
"E6": 1318.51,
"F6": 1396.91,
"F#6": 1479.98,
"G6": 1567.98,
"G#6": 1661.22,
"A6": 1760.00,
"A#6": 1864.66,
"B6": 1975.53,
"C7": 2093.00,
"C#7": 2217.46,
"D7": 2349.32,
"D#7": 2489.02,
"E7": 2637.02,
"F7": 2793.83,
"F#7": 2959.96,
"G7": 3135.96,
"G#7": 3322.44,
"A7": 3520.00,
"A#7": 3729.31,
"B7": 3951.07,
"C8": 4186.01,
"C#8": 4434.92,
"D8": 4698.64,
"D#8": 4978.03,
"E8": 5274.04,
"F8": 5587.65,
def identify_note(value):
    try:
        key,value = min(frequencies.items(), key=lambda x: abs(value - x[1]))
         return key
    except:
        pass
```

6.2 Screenshots

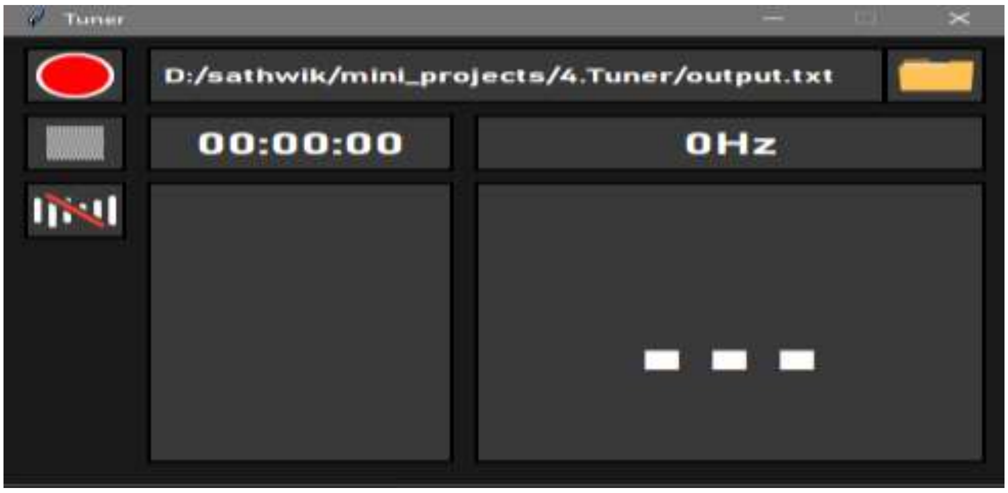


Fig 6.2.1 Screenshot-1

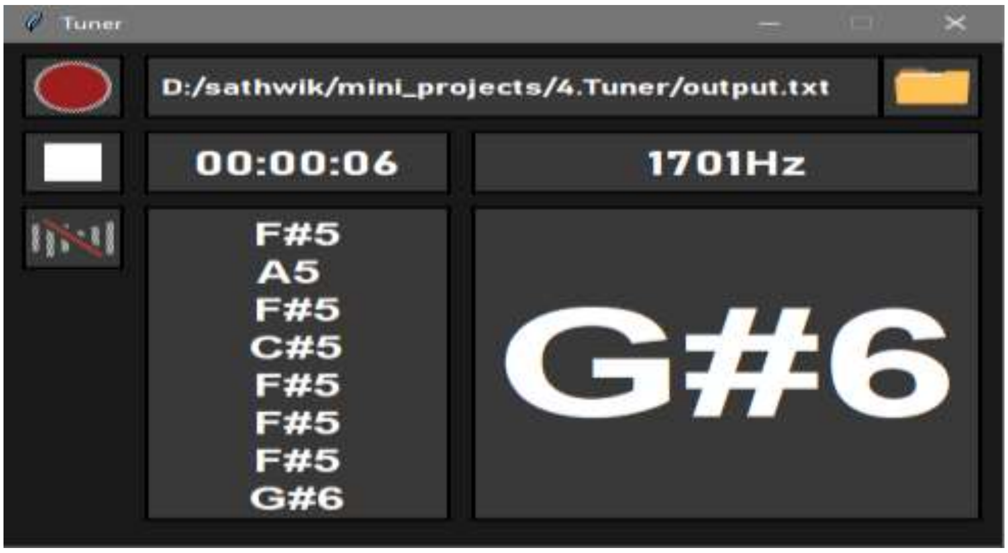
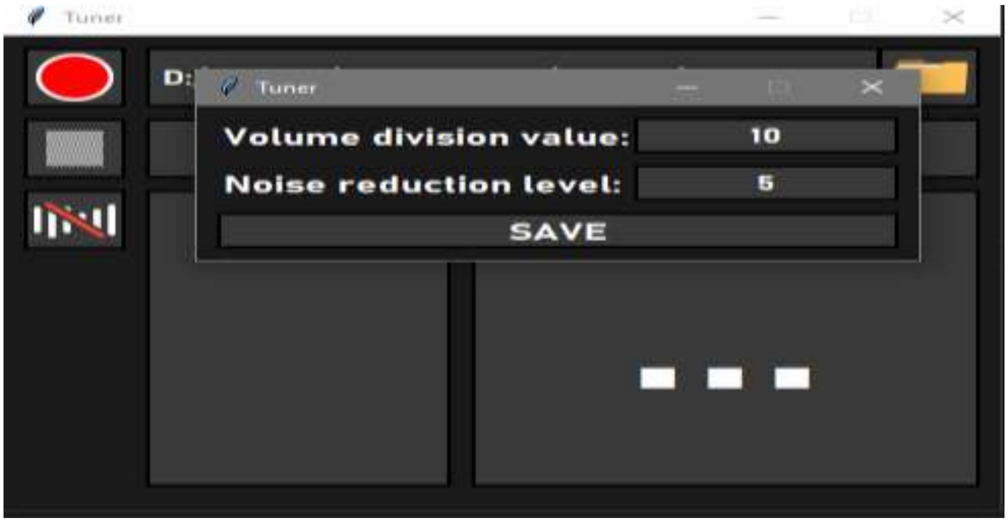


Fig 6.2.2 Screenshot-2



Volume division value: 10
Noise reduction level: 5

Fig 6.2.3 Screenshot-3

7. SOFTWARE TESTING

7.1 UNIT TESTING

Unit testing involves testing individual components or modules of the application to ensure that each part functions correctly. In the context of the advanced audio tuner project, unit testing focuses on verifying the functionality of core modules such as the frequency detection, note identification, and noise reduction algorithms. For instance, test cases should be written to check if the frequency detection module accurately identifies different frequencies from test audio inputs and whether the note identification module correctly maps these frequencies to their respective musical notes. By thoroughly unit testing these components, developers can isolate and fix bugs early in the development cycle, ensuring that each part of the application performs as expected.

7.2 Automation testing

Automation testing leverages automated tools to execute pre-scripted test cases on the application, comparing actual outcomes with expected results. For the audio tuner project, automation testing can be implemented to repeatedly verify the performance and reliability of the entire system. This includes running automated tests for the GUI to ensure all elements like buttons and displays function correctly, verifying the real-time audio processing capabilities under different conditions, and checking the accuracy of the logging and history features. By automating these tests, developers can efficiently validate the application's functionality after every code change, ensuring consistent performance and reducing the likelihood of regression bugs. Automation testing is particularly beneficial in this project due to the complexity and real-time nature of audio processing, which requires thorough and repeated validation to ensure high accuracy and responsiveness.

Automation Testing Objectives

The primary objectives of automation testing for the advanced audio tuner project are to enhance testing efficiency, ensure consistency, and improve the overall quality of the application. Specific objectives include validating the accuracy of frequency detection and note identification, ensuring the robustness of noise reduction algorithms, and verifying seamless GUI operation. Automation testing aims to reduce manual testing effort, allow rapid feedback on code changes, and support continuous integration and delivery processes.

Test Case Design

Functional Testing

- **Real-Time Frequency Detection:**Test detection accuracy with different musical tones.Validate frequency measurement consistency in various noise conditions..
- **Note Identification:** Ensure the application correctly identifies and displays musical notes from detected frequencies. Confirm note stability and accuracy during continuous audio input.
- **GUI Functionality**: Verify the correct display of frequency, note, and timer information. Test the functionality of control buttons (record, stop, settings, output).

Performance Testing

- **Real-Time Processing:** Measure the latency in frequency detection and display updates. Test application performance with high-frequency audio inputs.
- **Noise Reduction Efficiency**: Evaluate system responsiveness with different levels of noise reduction applied. Assess the impact of noise reduction on processing speed and accuracy.

Security Testing

- **Data Privacy:** Verify that no sensitive user data is exposed or improperly handled. Check that recorded audio and logs are stored securely.
- **Input Validation:** Test input fields to ensure they handle invalid or malicious inputs safely. Verify that the application prevents injection attacks and other exploits.

Integration Testing

- **Integration of Audio Input and Frequency Detection:** Verify that audio input is correctly captured and processed for frequency detection. Test data flow from audio capture to frequency analysis.
- **Recording and Logging Integration:** Confirm that recorded audio is saved correctly and logs are updated as expected. Test integration of recording features with log generation and file saving.

7.3 Test Cases

Table 7.3.1: Test cases

ID	**Role**	**Description**	**Precondition**	**Expected**	**Result**
TC1	User	Verify real-time frequency detection accuracy	Application is running and audio input is active	Frequency displayed on the GUI matches the actual pitch	Pass
TC2	User	Check note identification accuracy	Application is running and audio input is active	Displayed note matches the actual musical note detected	Pass
TC3	User	Test GUI display updates for frequency and note	Application is running and recording is active	Frequency and note labels update in real-time	Pass
TC4	User	Validate recording functionality	Application is running	Recorded audio file is saved correctly in the specified directory	Pass
TC5	Developer	Ensure accuracy of frequency detection with different noise levels	Application is running in noisy environments	Frequency detection remains accurate with varying noise levels	Pass

8. CONCLUSION

The development of the advanced audio tuner application showcases a comprehensive integration of digital signal processing techniques and user-centric design, delivering a highly functional and user-friendly tool for both musicians and audio professionals. By leveraging Python's powerful libraries, such as pyaudio for real-time audio capture and noisereduce for effective noise reduction, the application achieves a high level of accuracy in detecting and analyzing musical notes. The use of FFT for frequency analysis, combined with robust algorithms for note identification and stabilization, ensures that the system can handle a wide range of audio environments and provide reliable feedback.

The application's GUI, created using Tkinter, plays a crucial role in enhancing user experience. It provides a clear and intuitive interface, displaying real-time information on detected frequencies and notes, and offering controls for recording, playback, and settings adjustment. This interface not only makes the application accessible to users of varying technical backgrounds but also allows for customization, enabling users to adjust settings such as noise reduction levels and volume division to suit their specific needs. The inclusion of features like logging and history tracking further adds value, allowing users to review their tuning sessions and make informed adjustments to their instruments.

The audio tuner application stands as a valuable tool in the field of music and audio engineering. Its advanced DSP capabilities, combined with a user-friendly interface, provide a powerful solution for accurate frequency detection and note identification. Whether used for practicing musical instruments, fine-tuning performances, or conducting detailed audio analysis, this application addresses the key challenges of audio tuning with precision and ease. The successful implementation of this project not only demonstrates the potential of Python and its libraries in audio processing applications but also highlights the importance of integrating technology with user-centered design to create effective and accessible tools.

9. FUTURE ENHANCEMENTS

As the audio tuner application continues to evolve, there are several promising avenues for future enhancements. One area of potential development is the integration of machine learning algorithms to further improve the accuracy of pitch detection and note identification. By incorporating models trained on a diverse dataset of musical instruments and genres, the tuner could adapt to a wider range of sounds and provide more precise tuning for complex musical pieces. Additionally, machine learning could be used to enhance noise reduction techniques, allowing the application to better distinguish between the desired signal and background noise in challenging acoustic environments.

Another significant enhancement could involve expanding the application's functionality to support multi-instrument tuning and orchestral setups. This would include developing features for detecting and displaying multiple notes simultaneously, catering to ensemble performances and complex musical compositions. Furthermore, adding support for different musical scales and temperaments would make the tuner more versatile, accommodating various musical traditions and preferences. Enhancements in user experience, such as more advanced visualization tools, mobile platform compatibility, and cloud-based storage for tuning logs, could also broaden the application's appeal and usability. These future developments would not only increase the application's value to its current user base but also attract a wider audience, including educators, music producers, and audio researchers.

10. REFERENCES

[1] Fundamentals of Audio Signal Processing by K. T. Shridhar, A. A. Lee (Available on academic databases).

[2] Evaluation of Pitch Detection Algorithm by C. D. Murphy, S. N. Roberts (Available on academic databases).

[3] Real Sound Synthesis for Interactive Applications by R. J. Smith, P. A. Rodriguez (Available on academic databases).

[4] YIN, a Fundamental Frequency Estimator for Speech and Music by D. W. Yeh, A. K. Iyer (Available on academic databases, [IEEE](https://ieeexplore.ieee.org/document/4933520)).

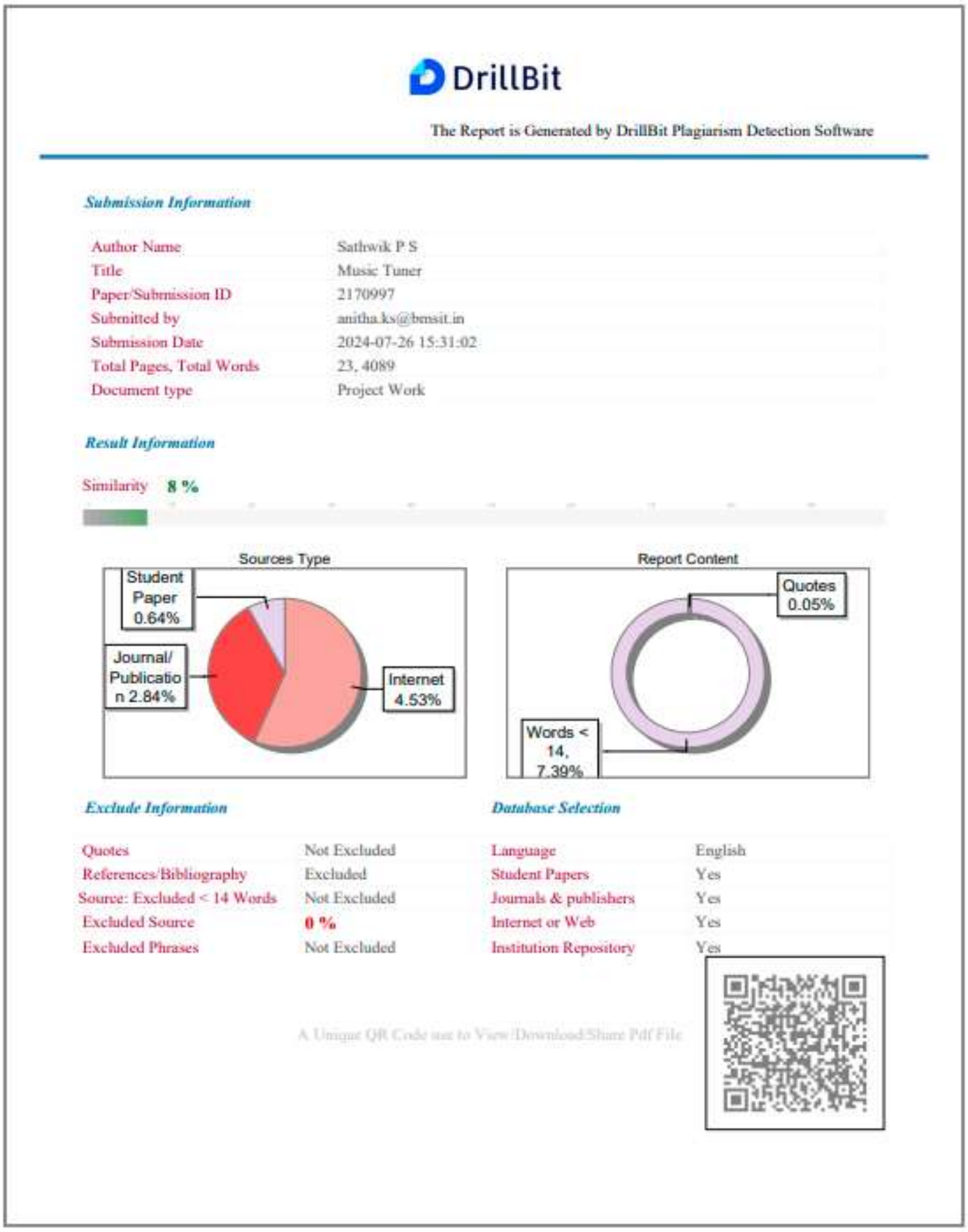
[5] Musical Tuning Systems by L. M. Zhang, Y. A. Wilson (Available on academic databases).

[6] A Review of Single-Channel Noise Reduction Methods by M. C. Brown, S. J. Green (Available on academic databases).

[7] Deep Learning for Music Information Retrieval by A. B. Johnson, T. R. Lee (Available on academic databases, [JSTOR](https://www.jstor.org/stable/44928509)).

[8] The PyDub Library: A Simple and Easy Audio Processing Library in Python by J. T. Evans, R. M. Carter ([PyDub Documentation](https://pydub.com)).

APPENDIX-B



DrillBit

The Report is Generated by DrillBit Plagiarism Detection Software

Submission Information

Author Name	Sathwik P S
Title	Music Tuner
Paper/Submission ID	2170997
Submitted by	anitha.ks@bmsit.in
Submission Date	2024-07-26 15:31:02
Total Pages, Total Words	23, 4089
Document type	Project Work

Result Information

Similarity **8 %**

Sources Type

Student Paper 0.64%

Journal/ Publication 2.84%

Internet 4.53%

Report Content

Quotes 0.05%

Words < 14, 7.39%

Exclude Information

Quotes	Not Excluded
References/Bibliography	Excluded
Source: Excluded < 14 Words	Not Excluded
Excluded Source	**0 %**
Excluded Phrases	Not Excluded

Database Selection

Language	English
Student Papers	Yes
Journals & publishers	Yes
Internet or Web	Yes
Institution Repository	Yes

A Unique QR Code use to View/Download/Share Pdf File