Project Report

**Topic**: Musical Note Recognition

Table of Contents

[**Synopsis**](#_1rjg9rw4qbcm)1

[**Techniques Used and Flowcharts**](#_30j0zll)3

[A – determining the volume](#_1fob9te) 3

[B – Note Classification](#_somgdfyqmh8a) 4

[**User Interface**](#_cl7pq312hgp2) **6**

[**Program**](#_epp0uads83mp)7

[note\_start\_detector](#_o4fdc261r37t) 8

[graph\_plotter](#_hmjx9ab0q9wc) 8

[frequency\_func](#_xag3rw7ag98f) 9

[freq\_to\_note](#_qetiswg3673l) 9

[note\_classifier](#_6q23p7pd3ghl) 9

[calculate\_distance](#_7az9t5vcjax7) 9

[note\_predicter](#_srnw5fd6g3fv) 10

[main](#_qzocfaayr1cs) 10

[**Results**](#_6odblb96px9u) **11**

[**Conclusion**](#_6odblb96px9u) **13**

[**Bibliography**](#_amcd20vmqmoa) **14**

# Synopsis

Our main goal from the start was to apply the concepts of the Fourier Transform in such a way that it would be useful to a real-world scenario. Although there were many projects one could do with this concept after pondering through some of them, we finally decided to go with note recognition as this project not only tests our knowledge to a great extent but also uses other concepts taught to us while studying signals and systems.

The whole concept behind the project is to locate the time at which a particular note is played and then using that information and other techniques to predict which note was played. Moreover, the project is divided into two steps:

1. Detect the location of the note in the audio file
2. Check and analyze the audio file at that point to detect what note was played at that point.

The analysis here depends a lot on the recording, whether there were any background noises or has some high frequency or low-frequency noise pollution in it. Measures are taken in the program to distinguish between the important frequencies and the frequencies generated due to the noise in the background. The major libraries of python used here are **PyDub, NumPy and SciPy and python-Levenshtein.** The whole process after being divided to explain, spans over the two topics note location detection and not classification. This project helps in understanding PyDub as a library and also at the same time teaches how to take an audio sample and perform a practical Fast Fourier transform over it. Of course, this is just the synopsis and the real understanding can be achieved in the latter part of this report, but it would be suggested that the reader has knowledge about the basics of python and at the same time a basic knowledge about music and notes.

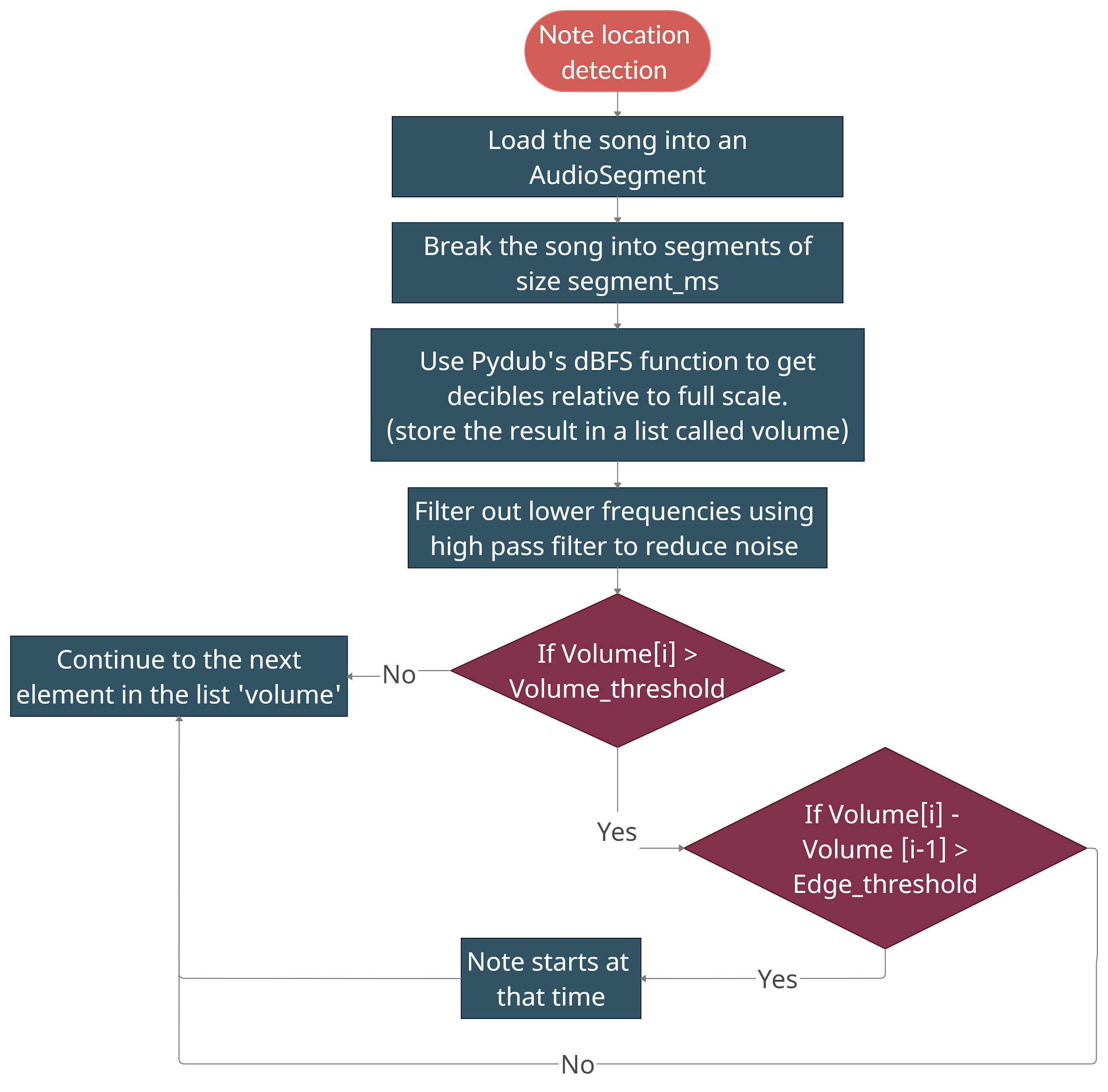
# Techniques Used and Flowcharts

## A – determining the volume

PyDub is a library which can be used for importing audio files and doing basic audio processing on that file. Using this library, we can divide this audio into segments, edit them, get the volume of those segments etc. It starts with loading the song into segments which can be done by basic array slicing. Then, to find the volume of these segments, we use PyDub’s dBFS function on each segment. Then, to reduce the noise in the audio segment, we pass that audio through a high pass filter to filter out frequencies below 80 Hz. Upon analyzing it, one can decipher that the starting of each note results in the sudden increase of volume but at the same time, it would be hard to differentiate as sometimes the succeeding note might be less loud or there is a less substantial increase than some of the noise before the first note. But this identification can be improved by removing the frequencies under 80 Hz with a high pass filter as it helps in reducing the excessive noise significantly.

Then a concept of the cliff is introduced via code where if there is at least an increase of 5dB from one single sample to the next, so for the given condition the cliffs were taken into account and at the same time, this has a minimum threshold for the top of the cliff (-35 dBFS).

Then a basic rule is added that the notes have to be a minimum distance apart from each other, to introduce a gap between two notes. So this whole gives us a substantially precise starting time of every note and at the same time the volume of that note. The flowchart below summarises this entire process.

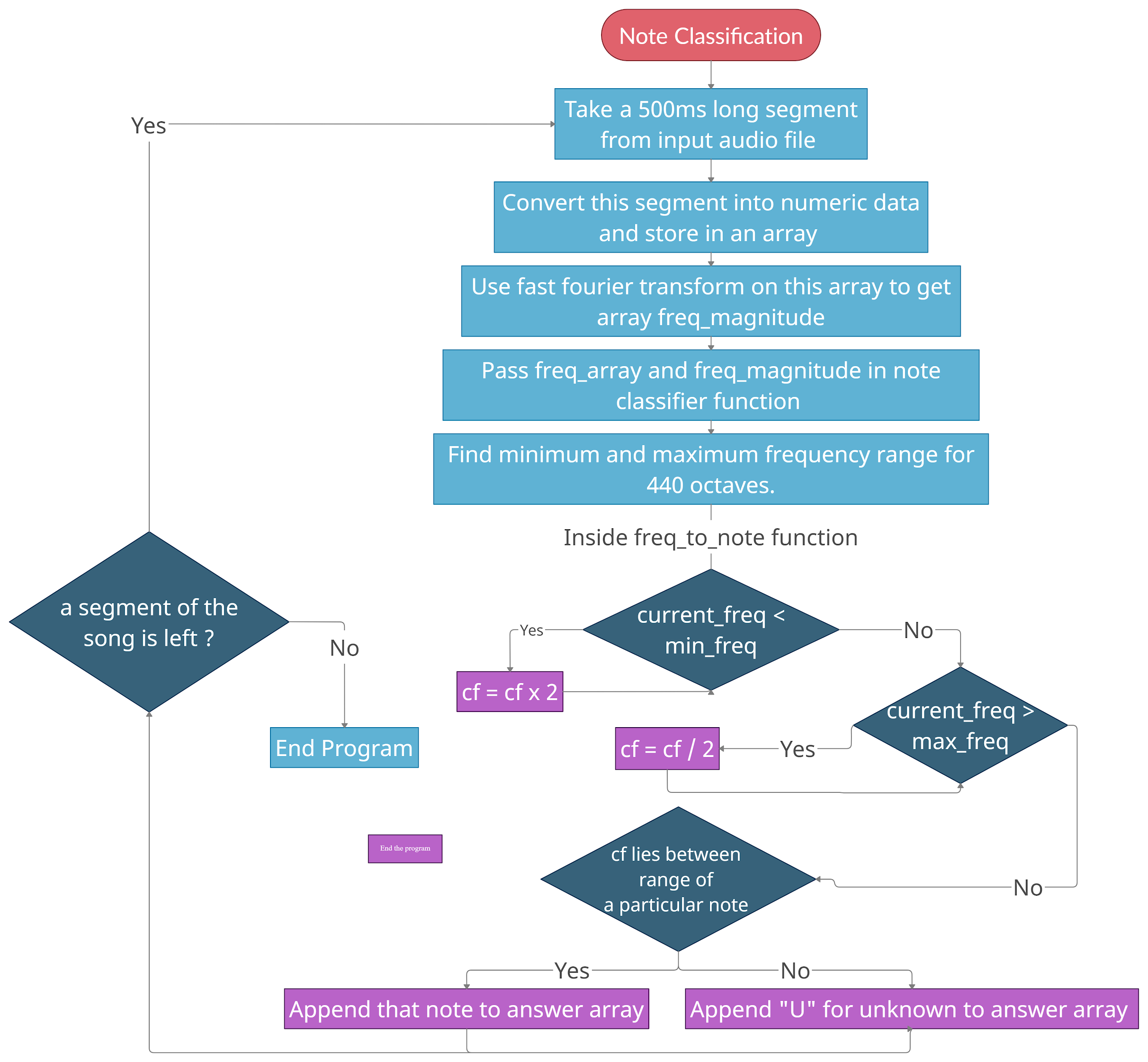


## 

## B – Note Classification

To classify the notes, we use techniques such as fast Fourier transform, converting song data to numeric data with the help of NumPy and SciPy, scaling the frequencies of the song etc. Firstly, we take a 500-millisecond long sample from the audio. Then we convert the audio data in that sample to numeric data.

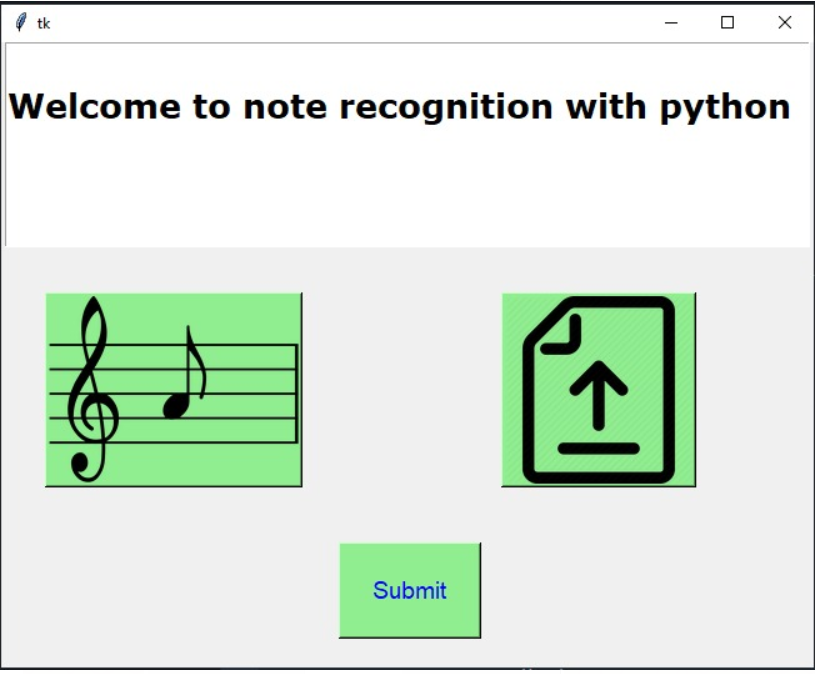
After passing this numeric data in the frequency\_func function, we get the frequency array and frequency magnitude array. To get these, we use fast Fourier transform and a concept called “zero-centring” (subtracting the mean of the array from all the individual elements). Then we iterate over the frequencies in the frequency array and pass them in the freq\_to\_note function. In this function, we first scale these frequencies to bring them into the 440 octaves. Then we check whether these frequencies lie in the tolerance range of any note. If they do, we have the predicted note!! We repeat this process until there are no more audio samples left to take.



# 

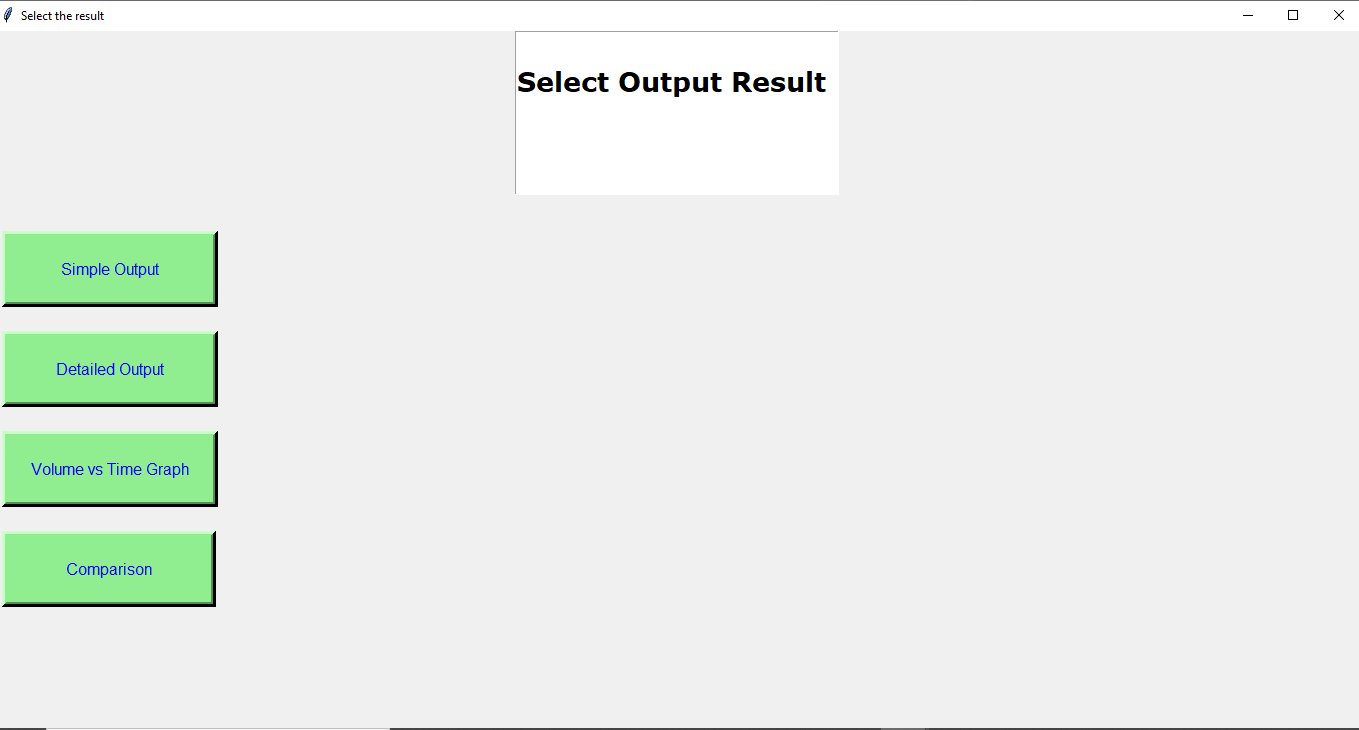
# User Interface

The User Interface was implemented from scratch using the python Tkinter library. The UI is user friendly and can be understood very easily.



The button on the left gives support for uploading the musical file, and the button on the right supports for uploading the text file for comparison with the note predicted from our program.

After clicking on the submit button, the user gets 4 choices.



As you can see, these 4 choices are:

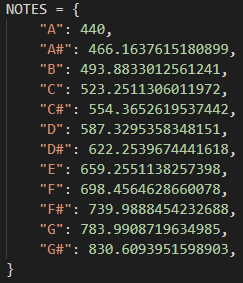
1. Simple Output: Shows the predicted notes
2. Detailed Output: Shows detailed analysis of each note
3. Volume vs Time Graph: Shows the dBFS vs time graph
4. Comparison: Compares the predicted notes with the actual notes

# 

# Program

**Note:** All the functions explained below have been modified in the latest version of the program to integrate the program with the user interface.

The following function is used in the program to find out the starting times of all the notes played.



We use the above dictionary to map notes to their corresponding frequencies.

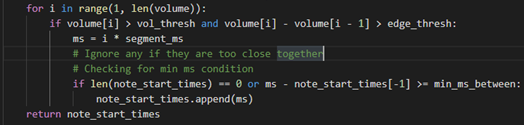
Functions used in the program are described below.

## note\_start\_detector

The key objective of this function is to find the starting times of notes played in the audio song uploaded to the program. The key logic used here is: **Whenever a note is played, there is a sudden increase in volume.**

So, to find out the starting times of all the notes, we set an edge threshold = 5 (edge\_thresh in the program) and a volume threshold = -35 dBFS so that no points having dBFS less than that can be considered to be starting points of notes.

We decided the values of volume threshold and edge threshold after doing some trial and error. Hence, if the increase in the volume is more than 5 and the value of dBFS is more than -35, it will be considered to be a note. Look at the following piece of code for better understanding.



## graph\_plotter

This function aimed to generate a graph such that the function is dynamic, furthermore, the graph should be saved in a way that even if multiple graphs are printed the function can differentiate between them.

It is also very important that notes are visible but also to colour code every note such that one could easily differentiate between them. Lastly, there is a legend provided in the graph such that the user could see them and understand which note is played at which time.

## frequency\_func

This function takes an input of a sample (a segment) of the song (which is an audio sample). Then, it converts this sample into numeric data.

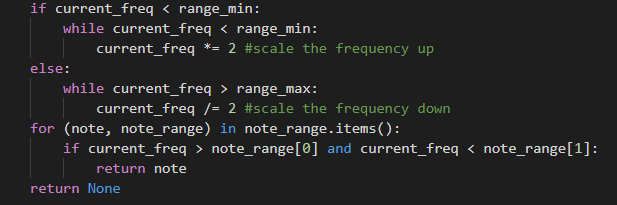
Then we subtract the mean of this numeric data from all the elements (this is called zero-centring). Then we go ahead to compute the Fast Fourier Transform of this numeric audio data and normalize it later. This gives us the frequency magnitude.

## freq\_to\_note

In this function, we check whether the given frequency is within the tolerance range of a note. First of all, we set a tolerance range for all the notes.

Firstly, the program checks whether the given frequency is inside the 440-octave range. If it is not, the note is scaled up/down until it is in this range.

Now, if the frequency is within the tolerance range of any note, we return that note. Otherwise, we return ‘None’. Look at the code snippet below for better understanding.



## note\_classifier

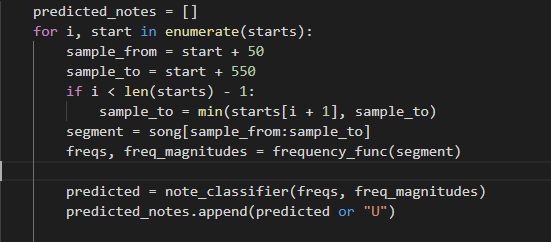
This function takes the above-mentioned input of a list of frequencies and a list of frequency magnitudes. While iterating through the list of frequency magnitudes, if the frequency magnitude is greater than 0.01, we use the freq\_to\_note function to get the note corresponding to that frequency.

## calculate\_distance

This function is used to calculate the difference between the actual notes (taken as the input) and the predicted notes. If there are x notes different in the predicted notes and actual notes, the “distance” will be x/total\_notes. To make the task easier, we simply used the distance() function from the python-Levenshtein library to check how different the predicted and actual notes are.

## note\_predicter

This function uses all of the above-mentioned functions to calculate the final note prediction. First, we declare an empty list called predicted\_notes. Here we use the list of starting times and draw samples of length approximately 500 ms. Then we use the frequency\_func to get arrays of frequencies and frequency magnitudes by passing in the data of that segment. Using these arrays, we use the note\_classifier function to find out which note is being played in those 500 milliseconds.

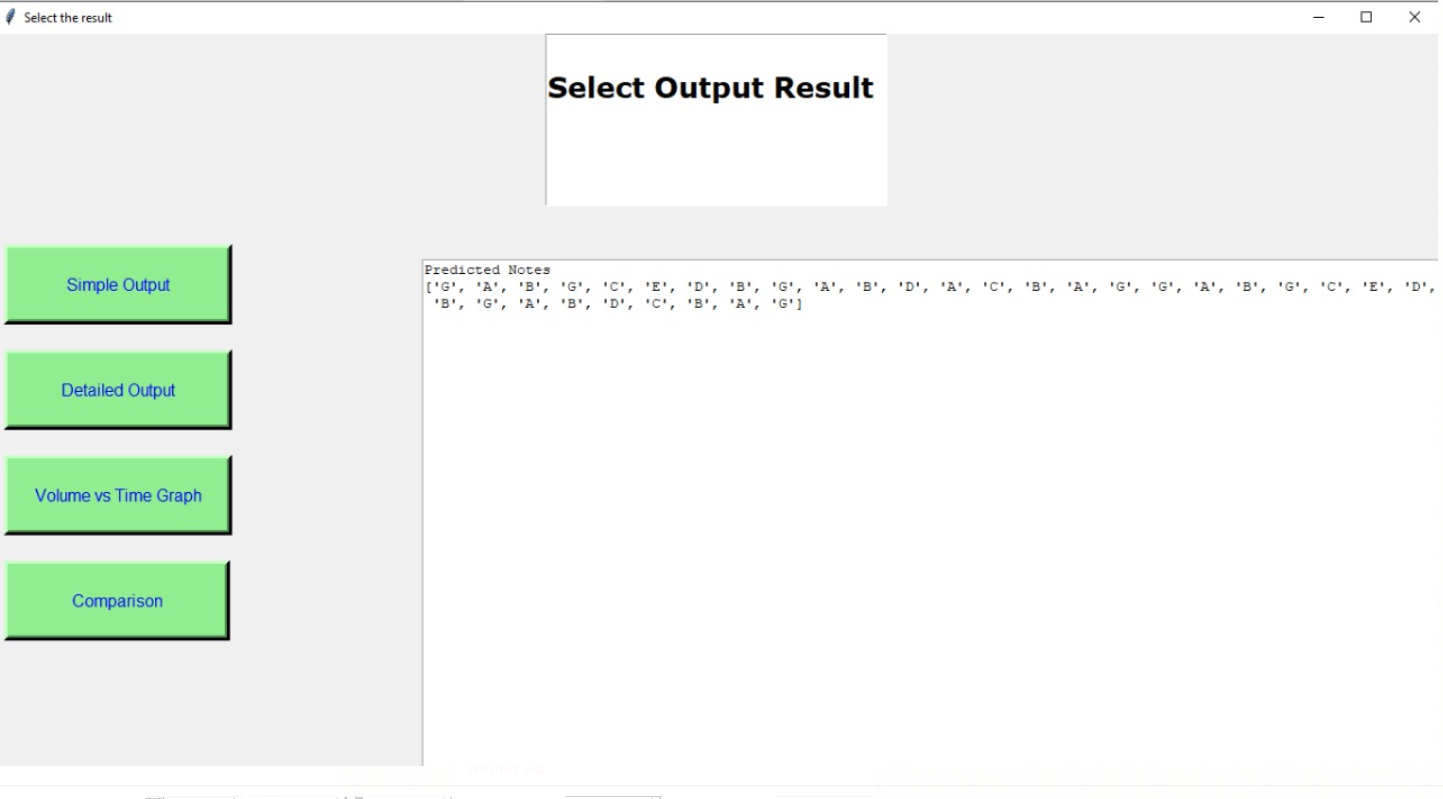


## main

In this function, we take input of the song by using PyDub’s AudioSegment. Then we pass this song through a high pass filter to remove frequencies less than 80 Hz. Then we use the note\_start\_detector function to get a list of starting times of notes. Then we use the note\_predicter function to find out the predicted notes.

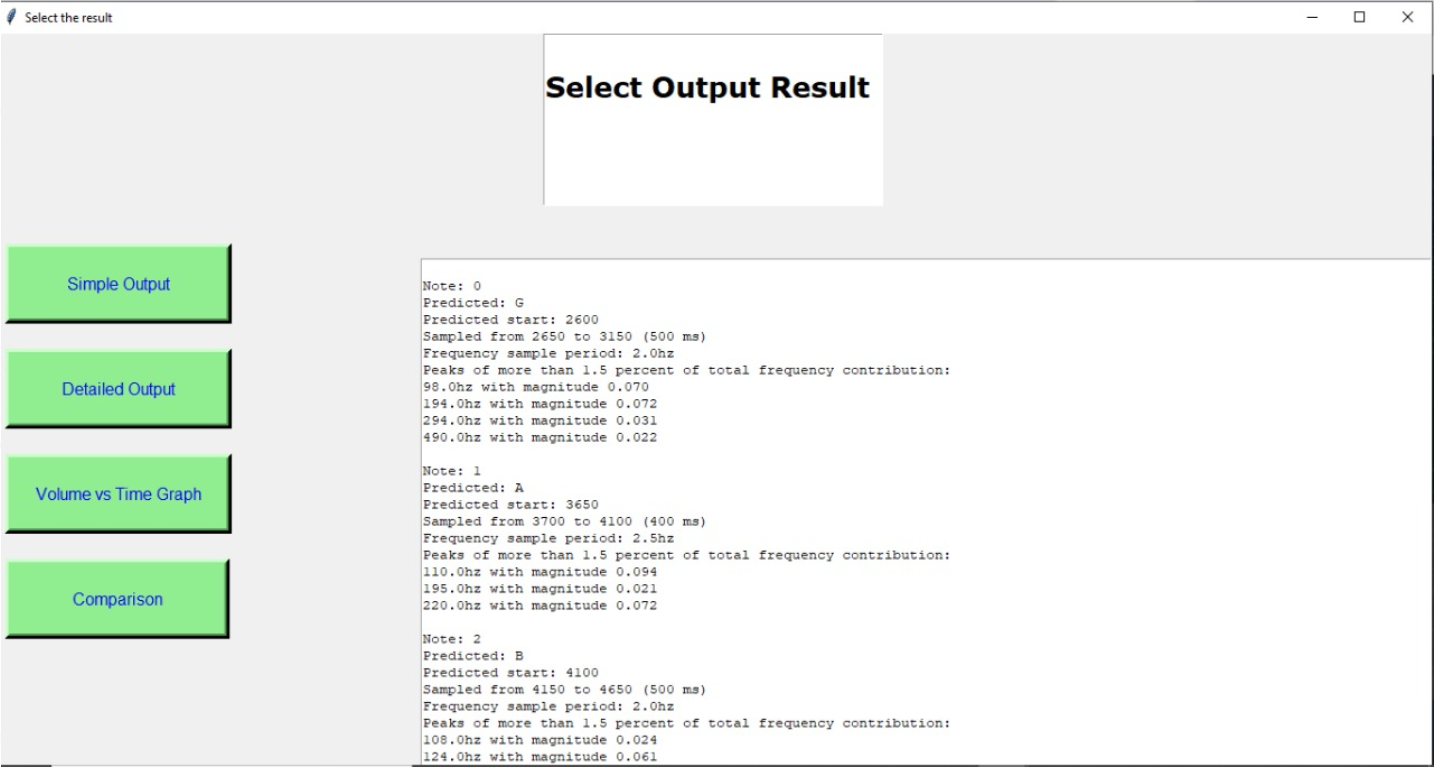
Throughout the entire program, we write all of the outputs in different files like “Notes.txt”, “Simple.txt” and “Compare.txt”. These are the files we use to display the Detailed output, simple output and comparing output respectively.

# Results

By clicking on the Simple Output button the user gets a minimised version of our note prediction, just showing notes at distinct time intervals.

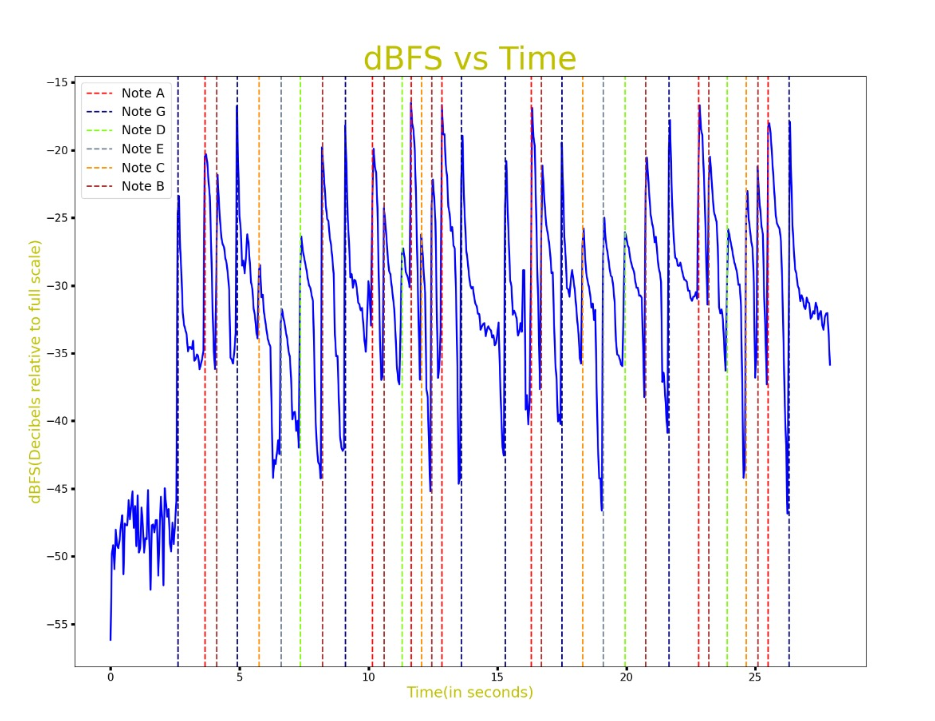
The output window on the right shows the simple output as asked by the user.

On clicking the Detailed Output button the user gets the full details of the audio file uploaded. Here we get analysis for each note played. We get the starting time of the note, how long was the note sampled for in the program, magnitude and frequency of different spikes in the magnitude vs frequency graph of the sample taken of that note. With every note displayed with the frequency and the time interval in which it has been played.



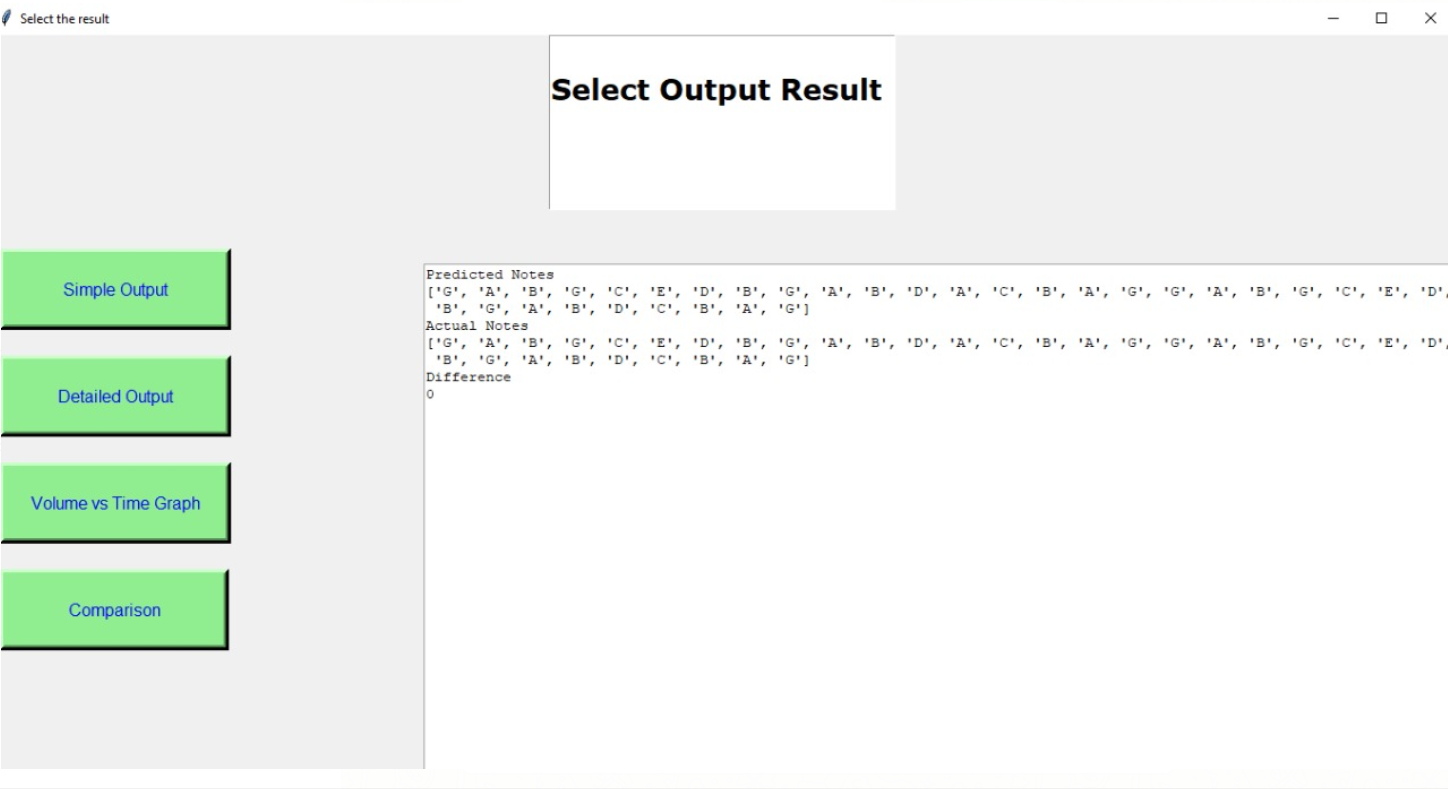
The output window on the right shows the detailed output as asked by the user.

On clicking the Volume vs Time Graph a new window pops up on the device which shows the output graph of dBFS (decibels relative to Full Scale) vs time.



As you can see, the graph has been colour coded concerning the note played in the graph.

The last button shows the comparison between the notes predicted with our program with the text file uploaded by the user.



Upon comparing the two, in this case, we see that the difference or the “distance” between the two notes is 0. This shows how accurate our program is.

# Bibliography

Matplotlib. (2020, January 5). *Magnitude Spectrum*. matplotlib. https://matplotlib.org/3.1.1/api/\_as\_gen/matplotlib.pyplot.magnitude\_spectrum.html

Michigan Technological University. (2008, March 7). *Physics of Music - Notes*. mtu. https://pages.mtu.edu/~suits/notefreqs.html

Patel, J. K., & Gopi, E. S. (2015, August 21). Musical Notes Identification using Digital Signal Processing. *Procedia Computer Science*, *57*(2015), 876-884. https://www.sciencedirect.com/science/article/pii/S1877050915020281

pydub. (2020, August 25). *API Documentation*. pydub. https://github.com/jiaaro/pydub/blob/master/API.markdown

SciPy. (2020, November 4). *Fourier Transforms*. SciPy. https://docs.scipy.org/doc/scipy/reference/tutorial/fft.html

# 