2018

NEA

Contents

Analysis

Problem

Statement that describes the problem and solution

The result of not mixing/equalising the sound played through speakers can have mainly 2 effects: have the sound/music being projected sound strange or off; which when loud and with higher frequencies amplified can cause ear damage, and having feedback loops which when loud can also cause ear damage at certain but common frequencies.

I plan to design and make a program that plays a sound through the user's pa systems or speakers, records the feedback from the sound played through the speakers, then finally inform the user about the sound that is being played and how they can adjust it.

Methods and sources

How the problem was researched

Who the problem was being solved for

In my previous school, I and a handful or peers along with a couple of teachers ran the entire school's pa; including drama performances, masses, plays, or any performances. As a result, we would have to do the mixing as almost all of the DJ's, weather student/teachers/others as they usually wouldn't know how to properly and quickly balance their sound for music and when using the microphone. I also found this problem occurring with DJ's that were hired for events in and out of the school, and also currently in the normal pa used for assemblies.

I am trying to help the people under control of the sound with the equalising as to solve the problem for the people that have to listen to the sound.

I will try to target the program towards people with little knowledge on PA and speaker systems as they are the type of people to most likely have this problem. This means giving explanations on the their sound in layman's terms, and giving the program default settings to be able to run with little input from the user, unless desired by the user.

Identifying a third party

Detail for a third party to understand

My third part will be the other pupils that usually play the music at socials as they are able to give me feedback on the software when testing the pa systems before an event.

I would have them playback their sound and use the mic with a flat equaliser on the mixer/computer, let them run the program, equalise the sound based on what the program outputs, and play back their sound from their input and microphone, then get feedback on how useful or not the program was and how easy or difficult the program is to run and/or understand.

Research

The reason that certain frequencies at the same volume can cause ear damage

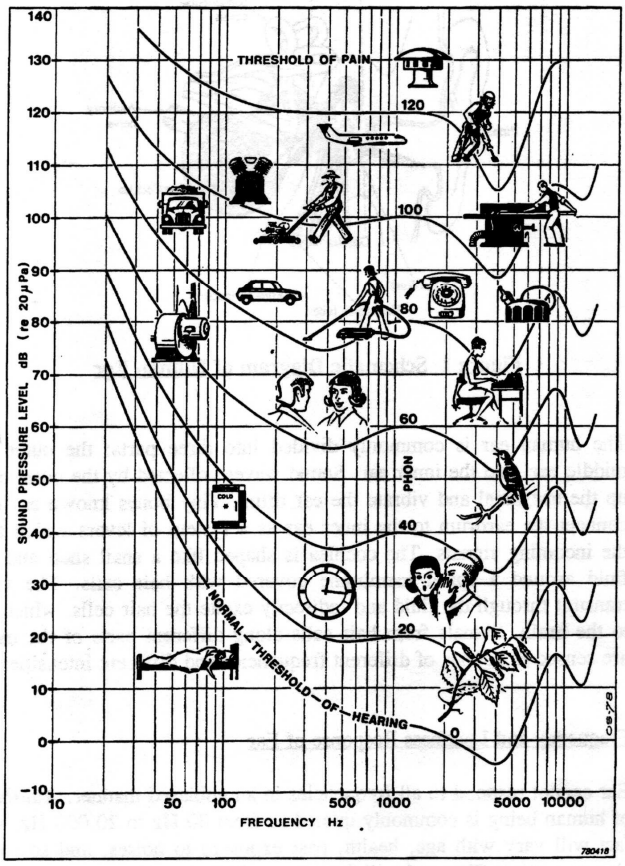
When playing a sound, there are certain frequencies that the person playing the sound mostly looks out for.

I've notices that in mostly hip hop/rap/electronic music, people generally look out for the low ends that give the kick of the bass that they enjoy.

What they don't realise is that when they want more low end, instead of equalising the sound to have more low end, they opt out to increase the volume altogether.

This in turn brings up some of the higher frequencies that is easier replicated by speakers and are more sensitive to the ear (due to ears not having a neutral frequency response (tends to be more 500-6khz), peaking at around 4khz as we can see in the image to the right).

The result of this is really loud, often painful treble that is usually dampened to the person playing the audio due to them usually being behind or to the sides of the speaker and not using monitors to truly listen to what there are outputting.



[Image above showing how the human's ear filters out certain frequencies. The lower the band of frequency, the less we filter out the frequency, the louder it is]

The reason that feedback loops occur

A Feedback loop occurs when a microphone that is being played back through a speaker in real time starts picking up its own recording from the speaker, sending it right back to itself many times per second which amplify itself and can drown out all the other frequencies and won't stop until the microphone's output/input are decreased or until the certain frequency that is looping is cut back a bit. As the frequency loop can amplify itself, it could get very loud and potentially damage the ears of people listening, especially for frequencies above 2000 Hz.

[1]This is a demonstration I made using headphones, and a sensitive mic with the 'listen to this device' option activated in Windows' microphone properties. To identify the frequencies in real time I used a program called 'Friture'.

With my microphone and headset frequency response and the (almost, mostly dominant on 43 hz) white noise made by a desktop fan, I got 2 dominant feedback loops of around 4 kHz and 6 kHz

[1] [Please refer to the video titled 'demonstration']

Prototyping and critical path

-List of measurable, specific objectives, covering all required functionality of the solution

-Model of the problem that informs the design stage.

Program requirements:

Must:

- Record audio

- play audio of a given frequency

- analyse audio

- Display a graph of the audio that the user can read and understand

- Be self-explanatory to use.

- Give an explanation of how the can equalise their sound

- On the default run, take no more than 30 seconds from the time the program starts to the time the graph is built up and shown on the user interface

Should:

- When played through a PA system with a mic inline, or when the mic's output is being played back real time, point out a dominant frequency and be able to suggest if it is going through a feedback loop.

- explain to the user what to do to get rid of the feedback loop depending on their setup.

- adjust the graph's output to the sound response of a human's ear.

-be able to save the user's settings

Could have:

- An optional safety feature which reads the amplitude of the recording in real time and based off of that, when the output of the sound is over some threshold, interrupt the playback as to protect the user.

- The ability to display the user's microphone input in real time for the user to know if the mic is connected properly or not.

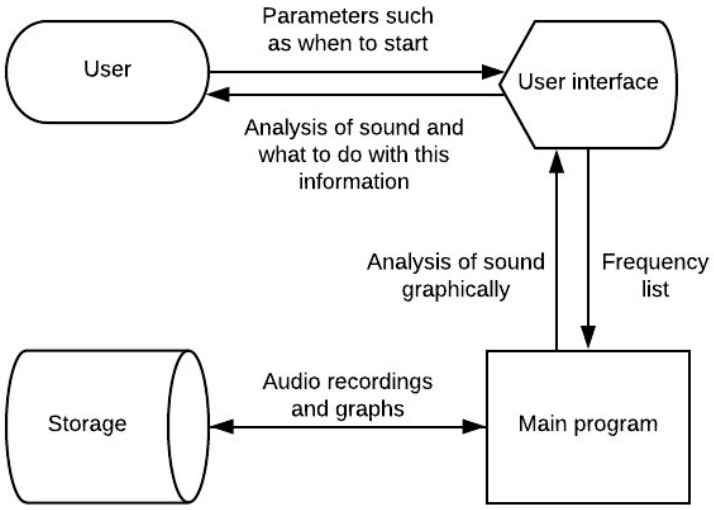
- The option to change the mic's input from the default input

- The graph adjust itself to the sound response of the microphone

Documented Design

Sequencing

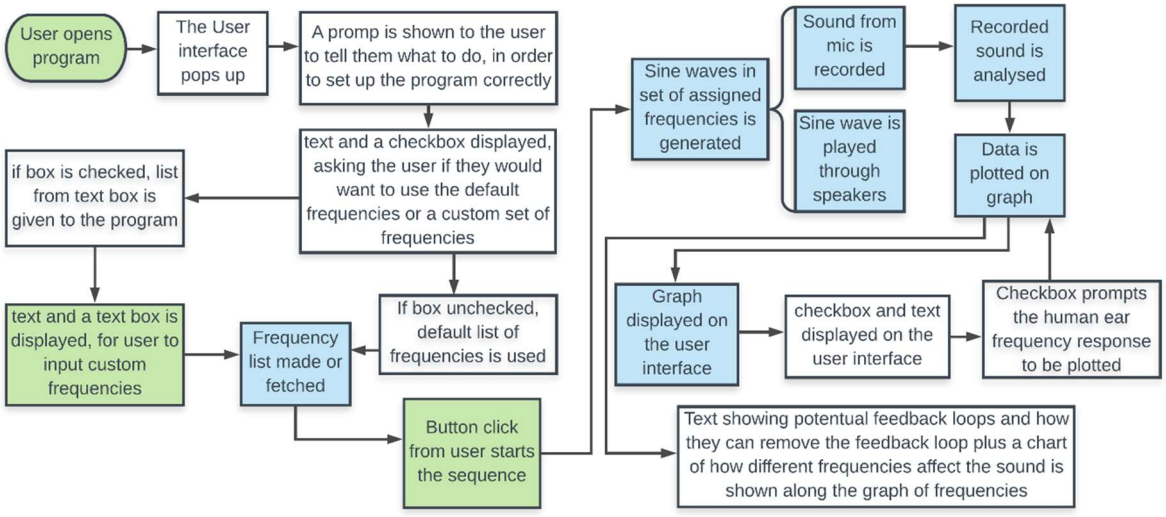
Initial Data Flow Diagram



Potential Data types to use

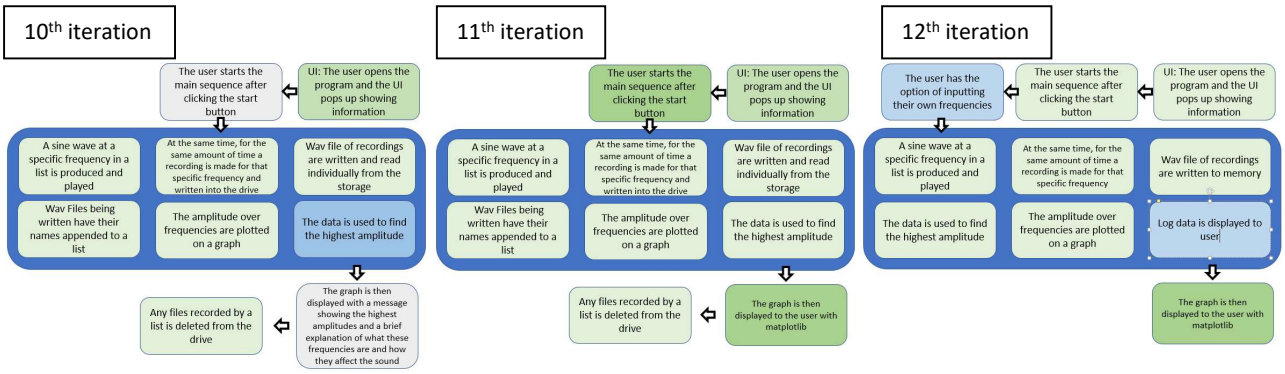
Type of data	Reason
Vector graphs	To show the analysis of the audio
Lists	For the set of frequencies the program will run
Text	To give the user and information before/after the main program has run
WAV	For any of the sound files as they are an easy format to work with.
PNG	For any extra charts to use in the explanation of the user before and/or after the main program has run

Initial High level overview

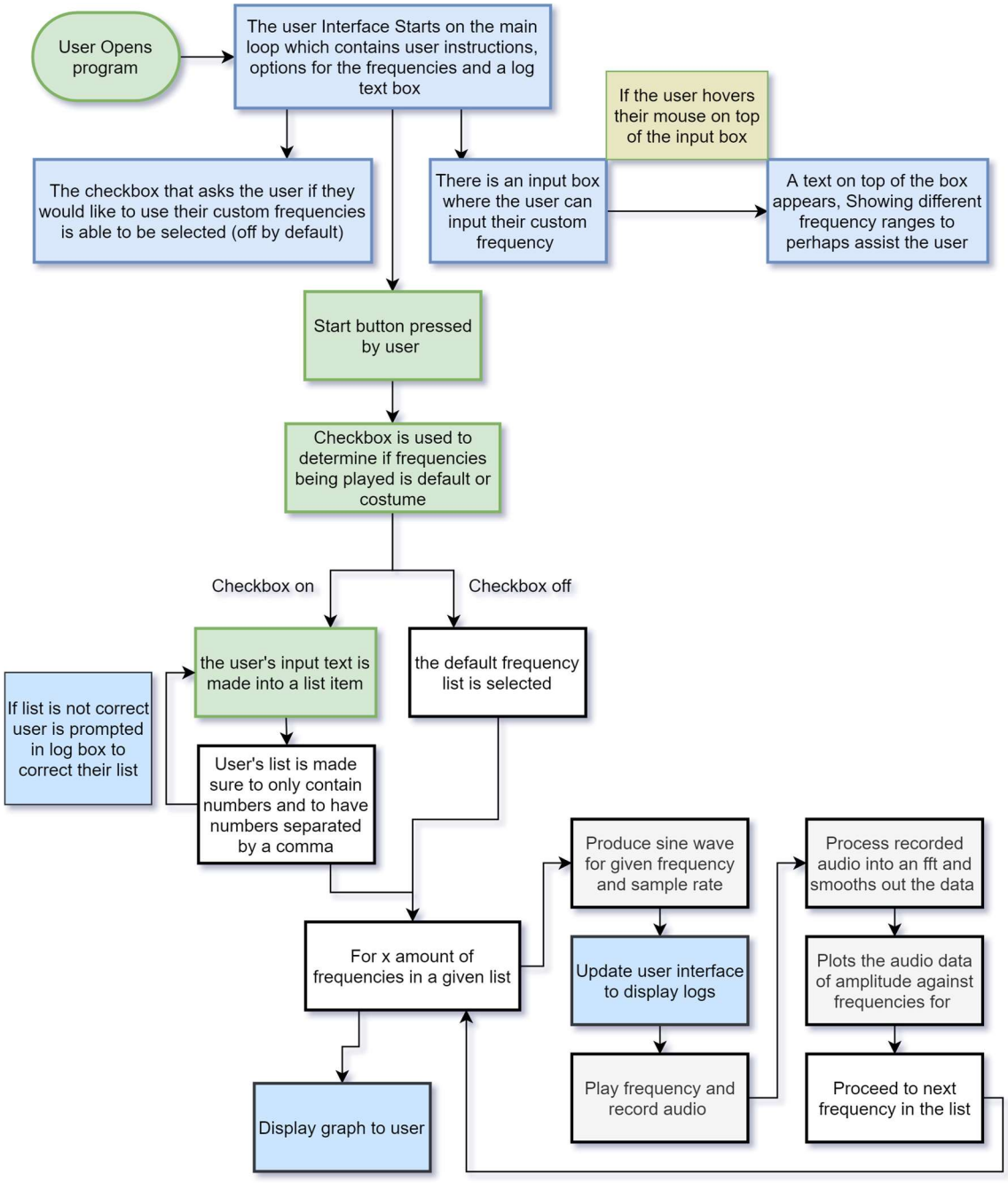


Documented design

In this project, I use the rapid prototyping method and constantly develop the program, making changes in each iteration. I do this to make sure that I don't get carried away doing parts of the program and can set myself achievable milestones that so I can stay on track of my time. Also, so that I get a very strong understanding of how the program and modules works and will work. This might also result in a more stable program overall as I can constantly add improvements in methods to make new things work, and also to optimise some of the processes within the program to make it run more efficiently.



Final high level overview



Technical Solution

Guide

Line 1-7: Import of the modules that I use within the program. The imported libraries are further explained along the document.

Line 9: Initialised the default_frequency_list which is used as the default frequency list of the program and contains the same frequencies as the windows equaliser settings.

Line 10: Initialised the sample_rate to be 48000 as it is very commonly used with digital audio and follows the Nyquist theorem. It lets me sample from 0-22kHz and I doubt would have any problems with consumer most audio equipment.

Lines 11-12: Used the value of the sample rate to set the default sample rate for the sounddevice library and also to be the num_samples (number of samples) as I would only want to play and record 1 second of audio per frequency played.

Line 13: Set the default channel for the sounddevice library to 1 channel for less overall memory use and processing as the program wouldn't have to deal with multi-channel audio.

Lines 14 - 16: Initialised the Tkinter module to be called root, and initialised the user's inputs in the user interface to later be passed on by the user. These inputs are an integer which has a range of 0 to 1 which acts as a Boolean when a checkbox is used and the other input is input consists of characters which is how data is captured when the user writes their frequency list.

Lines 19 - 107: A class named 'UI' is created and initialised. These lines are almost completely dependent of the Tkinter library as it allows me to create the look and function of the user interface. The user interface is made by a main frame that sits within the root frame and is broken up into 3 sub frames which are the: top frame, mid frame and the bottom frame. Each sub frame is then further divided into their own frames such as the input frame, start frame, log frame etc. which contain their own labels and widgets which make the look and function of the user interface. The UI contains a 3 step instruction to assist the user in correctly using the program, A checkbox for allowing the user to play their own frequencies, an input box where the user can write in their own frequencies, a small explanation that states the default frequencies, a log box that shows the user what is happening and finally, a start button to run the main sequence. I also added a title to the user interface, a set size of 720x325pixels when the user interface pops up and the ability to resize the user interface from a maximum size of 800x400 pixels to a minimum size of 550x300 pixels.

Lines 113 - 117: There is a function print_to_gui which captures all of the print statements from the program and inserts the statements into the log text box within the user interface. The function also makes the scrolls the scrollbar to reveal the bottom of the log box after each print statement.

Lines 119 - 127: There are two functions named 'input_box_on_hover()' and 'input_box_off_hover()'. The function 'input_box_on_hover()' function is called when the courser hovers over the input box where the user inputs their custom frequency. When the 'input_box_on_hover()' is called, it replaces the text above the input box to have a small message that informs the user about the different spectrum of frequencies and a small examples of where the frequency may be present in music. This is there to help out the user in making their custom list of frequencies to test out. Once the user removes their mouse from the text box, 'input_box_off_hover()' is called and the text in the box reverts back to its original text

Lines 129 - 144: There is the function 'check_validity_of_user_frequency_list()' which is a point where the user's input frequency list can be checked to make sure it does not contain any letters and to make sure that the list is not empty when the checkbox to use the custom frequencies is ticked. A Boolean variable 'list_is_valid' is initiated and equals to False. There is an 'if' statement that uses regex to make sure the user's list contain no letters and there is an 'elif' statement that makes sure the user's list is not empty. If the user's list contains letters or is empty, the function returns 'False'. If the list has made it past these checks, the function returns 'True'

Lines 146 - 152: The is the function 'start_button()' which has an 'if' statement that calls the function 'check_validity_of_user_frequency_list' and is 'True' is returned, the frequency analysing portion of the program starts. Else, if the 'check_validity_of_user_frequency_list()' returns anything but 'True', the function is passed and the user is left with the warning messages from the 'check_validity_of_user_frequency_list' function.

Lines 155 - 158: The class called 'Mainclass' is made and when called, passes and returns nothing. The functions in this class are static and the reason for the class being there is more for organisation rather than any practical application.

Line 160 - 177: There is the function 'make_user_frequency_list()' which gets the user's input list as text from the user interface and converts it to a list of integers. Firstly, the function split's the user's text into a list of text separated by a comma and initialises 'correct_user_list' to equal 'True'. Then the function starts a 'for' loop which tries to convert each variable in the list from text to integer form and if its attempt fails, It gets a value error and displays a prompt to the user which asks them to only use certain characters in their list and then sets 'correct_user_list' to equal 'False'. If the 'correct_user_list' still equals to 'True', the user's frequency list is returned, if the 'correct_user_list' is equal to anything but 'True', the user's frequency list is given the frequency of 1hz and is returned.

Lines 179 - 188: There is a function ''determine_frequency_list()' which uses 'if' statements to return the list of frequencies the user is going to be using in their run. If the checkbox is not checked, the frequency list returned is the default list and if not, the function 'make_user_frequency_list()' is called to make the user's frequency list.

Lines 190 - 193: There is a function 'produce_sinewave()' that for a given frequency needed , sample rate and sample size, returns a sine wave. I used numpy's sin function to make the sine waves.

Lines 195 – 200: There is a function 'play_audio_and_record_audio()' which uses the a function in the sounddevice library 'ds.playrec()' that is able to record audio whilst playing audio. I use this to play the required frequency whilst recording the audio from the microphone in int32 form (which gave me the least amount of problems processing). I also use a sounddevice function 'sd.wait()' which gives a brief pause to other operations before moving onto other frequencies as they might interfere with each other. The function then returns recorded audio.

Lines 202 – 210: There is a function called 'process_input_audio()' which gets raw audio data, unpacks the data, arrays the data, carries out the fast Fourier transform (fft) onto the data, gets the absolute values for the data, converts the data to an n-dimensional fft so I can then divide the data by 1.0*10^6 to make the data legible and then finally, apply the inverse function of the n-dimensional fft data and return the processed data as 'smoothed audio data'. In other words, I convert the audio data into a form where I can apply functions to the data that make the data readable on a graph. I then smooth out the data and return that data. I use the struck library to unpack the audio data, and then used Numpy to find absolute values of the data and then convert the data into the fft and rfft and back to fft form.

Lines 112 – 223: There is a function called 'plot_fft_graph()' which has the processed audio as an input and constructs a graph around that data. The graph is plotted with both the x and y axis logarithmically scaling as amplitude is logarithmic and to compress the 0-20kHz band of frequencies comfortably in a graph. The function uses Numpy's 'np.argmax' function to find the maximum frequencies recorded during the playtime of a specified frequency and displays it to the user in order to find any feedback or noise during the session. For each frequency, only a range of frequencies of [input frequency + or – input frequency/10] is plotted. This range is made from 'min_plot_for_frequency' and the 'max_plot_for_frequency' as to make the graph far less messy when plot. I use matplotlib pyplot functions to plot the data in the range of the minimum and maximum and then place a colour coded label for each frequency on the top of the graph and lables for the amplitude and frequency.

Lines 225 – 231: There is a function called 'display_graph()' which sets the range of the viewable graph on the x axis based off of the list of frequencies played which shows the highest and lowest frequency played on the same graph. The function then displays the graph to the user via a pop up matplotlib window.

Lines 233 -255: There is a function called 'start_analysing()' which is called when the start button is pressed on the user interface and is used to call all of the functions necessary to start the process of the audio analysing. Firstly, the 'determine_frequency_list()' function is called to make the list of frequencies to be played. Then, in a for loop tried for each frequency: the sinewave is produced using 'produce_sinewave()', the user interface is then updated to show the progress of the program using 'root.update_idletasks()', the audio is played and recording is recoded using the function 'play_audio_and_record_microphone()', the recorded audio data is then processed using the function 'process_audio_data()' and the processed audio is then plotted using the function 'plot_fft_data'. There is an exception to an 'sd.PortAudioError' which occurs when a microphone is not properly plugged into the computer. This exception prompts the user to plug in their microphone and restart the program. The other exception is there for a 'ValueError' which occurs when an improper list is made by the user that gets through the exception handling in the 'check_validity_of_user_frequency_list()' function. When the 'ValueError' occurs, the user is prompted to correct their list, using only numbers followed by a comma. At the end of the 'start_analysing()' function, the graph is displayed to the user using the 'display_graph()' function.

Lines 258 -261: The **'**UI**'** class is called and put into a loop in order to display the user interface. I am not sure how or why Tkinter works like this but to keep the user interface in a loop, I was advised to use this method in the Tkinter tutorials.

Libraries used

Sound device

-Sound device provided a function to play audio whilst recording at the same time in different file types such as int16/32, float (sd.playrec). Also, Sound device is able to record the audio at different sample rates and channels. This helped me as I could reduce the amount of data I would need to collect and thus save time for the convenience of the user. I managed to save time by setting the default sample rate to 48000 samples/second and channel to mono. I chose to keep the sample rate at 48000 samples/second as it is commonly used, follows the Nyquist theorem and I probably wouldn't need to increase the sample rate because of aliasing problems.

```
recorded_audio = sd.playrec(input_audio, sample_rate, channels=1,
                            dtype='int32')
```

Numpy

-I used Numpy for its function to apply the fast Fourier transform algorithm (np.fft()) to change the input audio signal into its frequency components. The fast Fourier transform Algorithm is perfect for my program as it only has the time complexity of O(nlog(n)) where n is the size of my data. I have not attempted using other algorithms as the fast Fourier transform is a convenient function in Numpy

I used Numpy for the function to find the absolute values for the data (np.abs()); for the so that I in a way separate the amplitude from my data to be able to plot the amplitude in the Y axis.

I then used Numpy for the inverse of the fast Fourier transform data function (np.fft.rfft()) and (np.fft.irfft()) which basically smoothens out the data which helps with the plotting.

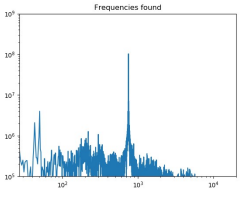
I also used Numpy to find the maximum values for the processed audio data to then show the user if there is any feedback or if there is too much noise using the (np.argmax()) function

Lastly, I used Numpy to create a sine wave for a given sample rate and frequency that I would like to generate using the (np.sin())

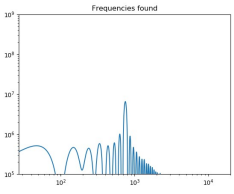
```
audio_data = np.array(audio_data)
audio_data_fft = np.fft.fft(audio_data)
audio_data_absolute = np.abs(audio_data_fft)
audio_data_rfft = np.fft.rfft(audio_data_absolute)/1000000
smoothed_audio_data = np.fft.irfft(audio_data_rfft)


np.argmax(audio_data)

produced_sinewave = [np.sin(2 * np.pi * frequency * x / sample_rate) for x
in range(sample_rate)]
```



[Graph with no smoothing]



[Graph with smoothing]

Matplotlib.pyplot

-used for its data plotting and vector graph functions (plt. [plot, title, xlim] etc.), also for its ability to save graphs as images that are simple to display on the user interface

Tkinter

-used for its many function to make and display an interactive user interface. It could also provide the user with progress during the program's main sequence by updating the interface in-between tasks, giving the user information about the loudest frequency when a certain frequency was played to find feedback loops.

```
root.update_idletasks()

Def print_to_gui(printed_statements):
    self.logtext_box.insert(INSERT,
    printed_statements)
```

Struct

-used for its function (struct.unpack) for string to be opened as a packed binary data format to then be read as an array.

Libraries used in development

Threading

I had a few attempts at threading as it would allow for the library that I use to run my UI (tkinter) to stop becoming unresponsive and get Window's unresponsive program message. Also so that the UI updates the Log text box so that the user is able to see the progress of the program as the program is run.

My idea was to open tkinter on one thread and have the main sequence that analyses frequencies start on a separate thread. I eventually ended my attempts as tkinter is not thread safe and my program was starting to get very complicated in design as I continued to make the program threaded.

```
plot_fft_graph_thread=threading.Thread(target
=Mainclass.plot_fft_graph(self,
input_frequency=frequency_list[i],
audio_data=audio_data_frames))

plot_fft_graph_thread.start()
```

Perhaps If I planned on making the program threaded earlier in the lifecycle, I would have been able to implement it.

To solve my problem, I ended up using a tkinter's (update_idletasks) that temporarily pauses my frequency analyser sequence before moving onto the next task for just enough time to update the tkinter interface

These libraries further down were critical at one point when my audio data and picture data was written to the computer's hard drive rather than memory.

Wave

-used for its function (wave.open) to read .wav files as frames

Os

-used for its function (os.remove) to delete files from a drive to clean up any files left from the program.

Scipy.io

-used for its function (wav.write) to write data as an audio file (.wav) and save it onto the user's compute

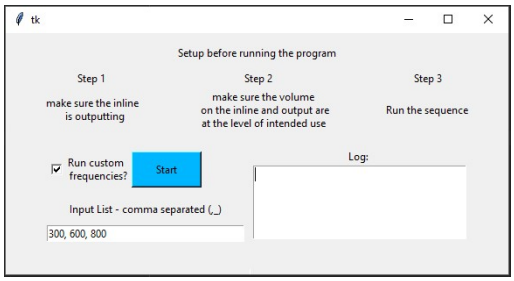
Data structures

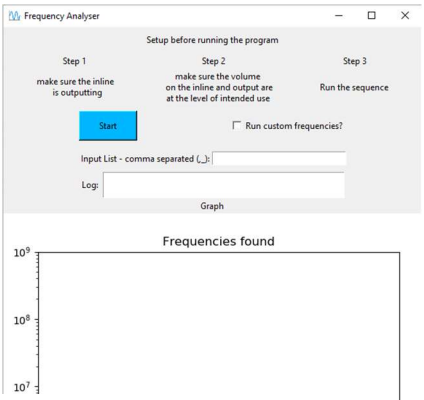
The only data structures I hold in memory are lists that hold integer values from the default frequency list and from the user's frequency list

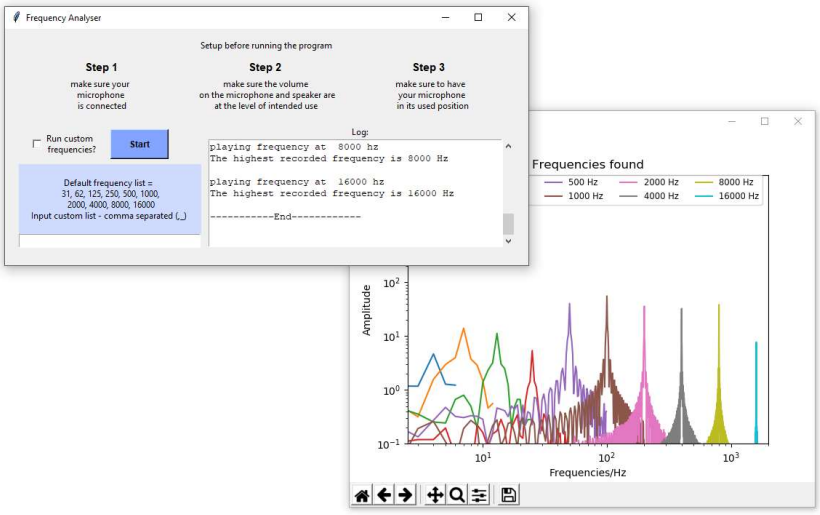
File structure and organisation

Initially, Wav files were written to the same path of where the program is stored and were deleted after each session. This took up more processing power and could be bottlenecked by the disk speed but was easy to implement. There was a naming scheme for each consecutive Wav file for a given frequency which was ‘sinewave at (frequency)hz.wav’. After I was able to write the audio data to memory, I did not have to make a list of all the recordings as I did not need to save the audio data for reading and writing

Also, initially, to display the graphs I planned on only saving the matplotlib graphs and displaying the image to the User after the frequencies are plotted. File names for the graphs would simply be ‘graph(test number).png’. This method would re-write graphs with the same name as each graph did not have a unique name. Saving the graphs as an image and displaying the image would give me the potential to edit the images which might have had its advantages but I changed to just displaying the graphs to that the user would have more control with the graph as matplotlib has several features that can even let you save your own graph with its own name







I tried to design the User interface to be as simple as possible to make it as user friendly as possible. To do this, I had several different iterations of the User interface and got regular feedback from my client about colours, log text, sizes icon placement and potential helpful features that would help the user.

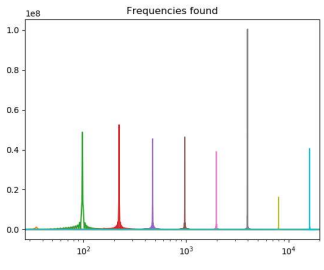
The start button was made big and light blue/purple for the convenience and ease of the end user. This start button, without any other parameters filled, starts the main sequence using default values to be used. Also, the graph adjusts itself to the frequencies plotted to abstract unused data after a run, are all labelled and each signal has its own colour.

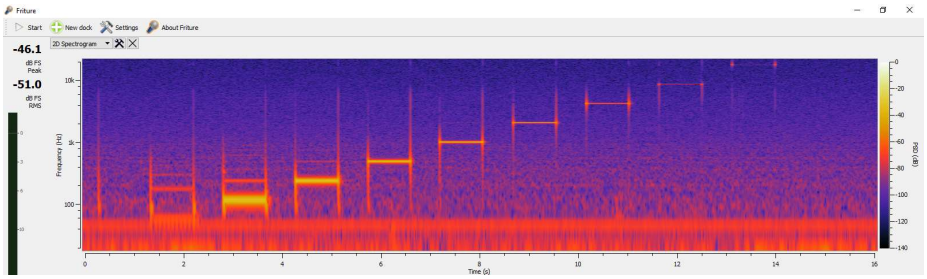
Tests

Explanation of some of the tests I ran

For parts of the program such as the analysis of the frequencies, I did a lot of my testing whilst writing the program as I wanted to make sure it worked as well as I could get it to work at the time to at least some level of usability before moving on. This is it's the focal point of the program and the program at the very least had to be able to do it.

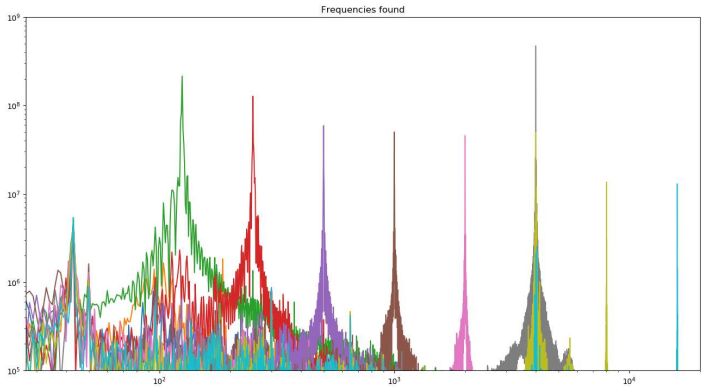
To test that I was playing the correct frequencies, I used a program called 'Friture' which I also used to make feedback loops as Friture would allow me to play specific frequencies which I could then compare to the frequencies I was playing to the user. I plotted the frequency axis (x axis) logarithmically against the y axis linearly





After looking at their spectrogram, I could conclude that my the frequencies I was producing were correct for the sample rate

To test if my program was able to show feedback loops, I also re-created my initial feedback loop demonstration whilst gathering data and plotted the x axis (frequency) and the y axis (amplitude) logarithmically and I got a graph that is:



The graph strongly suggests a feedback loop occurring at around the 4 kHz range and which continues playing when 8 kHz and 16 kHz is played. This is shown by the mic picking up a 4 kHz signal when it is playing 8 kHz and 16kHs from the extra yellow and blue lines directly under the 4 kHz signal grey line

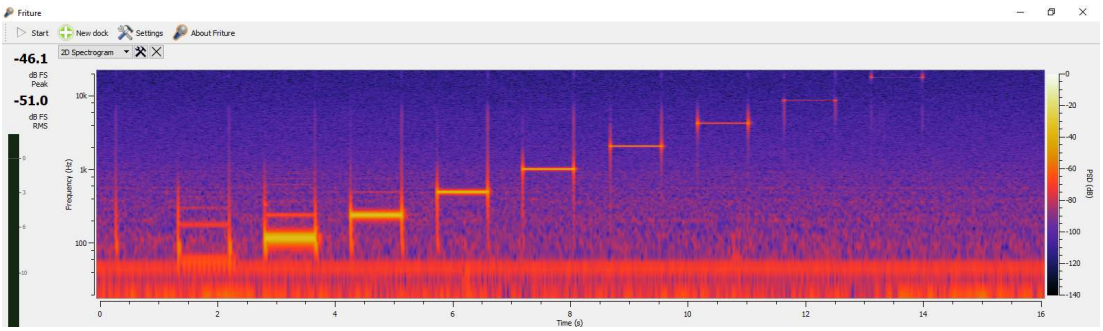
I also ran a few tests to see whether or not changing my sound source (in this case my headphones) Equaliser would influence the amplitude of the frequencies picked up by the microphone. The headphone I used were again the Audio-Technica ATH-M30x as they provide a reasonably flat sound response which would make the comparison of the amplitude of different sounds a bit easier and for the mic, I used a Tonor condenser Microphone as it was the only microphone I had available that I think wouldn't clip, distort and affect the data as long as the volume was reasonably low. I also played the same default set of frequencies for each test.

The tests against the requirements that I was able to test.

Must:

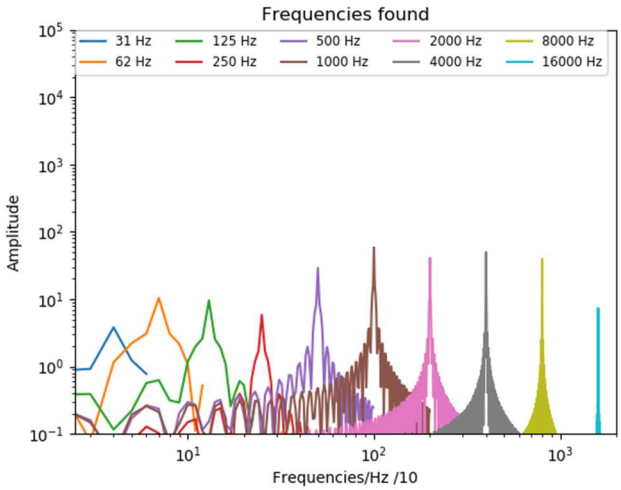
- **Record audio**
  - Test
    - Use units in program to recorded and write a 1 second audio file which I clap (to make sure it picked out the correct sounds) into storage.
  - Expected results
    - A .wav file of written to where the program is saved locally with expected noise that I made when recording the audio.
  - Given result
    - A 1 second .wav file was saved in the python file's location which contains audio data of the clap I made whilst recording.

- **play audio of a given frequency**
  - Test
    - I would use my program to produce a sinewave of different given frequencies and play them. Then, using a software called 'Friture', in real time display the spectrogram of frequencies played, Frequencies = [31, 62, 125, 250, 500, 1000, 2000, 4000, 8000, 16000]
  - Expected results
    - The spectrogram to display dark lines spanning 1 second each the same frequencies I intended to play.
  - Given result



    - The spectrogram showed 9 clear lines of varying intensity and also harmonics of the frequencies usually above their respective frequencies. The first 31 HZ frequency is drowned out by other noise at the 10-40 HZ range which could be due to my microphone or, more likely from the low frequency hum when my computer's fans when they spin at idle temperatures (~30^{o}). Intense vertical lines can also be seen on the ends of the frequencies played. This is due to a pop that plays before and after the frequency is played. This should not be a problem though as audio data is only captured during the playing time of the frequencies.

- **analyse audio**
  - Test
    - Use my program to plot a matplotlib graph with the given audio data from the recording made within the program. The same audio used was the confirmed correct audio played in my previous test with the same frequencies and same conditions.
  - Expected results
    - A graph of amplitude against frequncies of the recordings made
  - Given result
    - A graph of amplitude against frequencies of the recordings

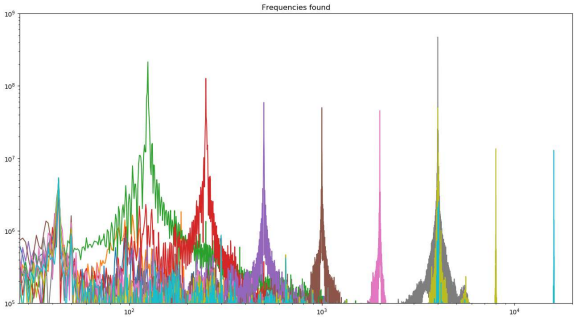


- **Display a graph of the audio that the user can read and understand**
  - Test
    - Let my client run the program and assess how the graph looks
  - Expected results
    - My client to have some sort of understanding as to the amplitude of the frequencies.
  - Given result
    - My client did understand which frequency was which using the labels and how loud they were.
- **Be self-explanatory to use**
  - Test
    - Let my client open and operate the program to play default frequencies and also their own frequencies.
  - Expected results
    - The client to be able to play the default frequencies and their own frequencies.
  - Given result
    - My client was able to play the default frequencies but did have difficulties using the input frequencies with problems such as forgetting to tick the box which runs the costume frequencies. Also My client only clicked on hovered over the input box for a sort amount of time before typing the input list meaning that they did not notice the text displaying information about different frequencies until I explained.

- **Give an explanation of how the can equalise their sound**
    - Test
        - Let my client use the program and ask if they had an idea of how to then equalise their sound directly from the computer
    - Expected results
        - My client would have a third party application they use to equalise the output sound of their computer or at least use Window's equaliser settings.
    - Given result
        - My client did not know how to equalise their sound and did not feel that they would be confident equalising their sound with my brief explanation given by the user interface.
- **On the default run, take no more than 30 seconds from the time the program starts to the time the graph is built up and shown on the user interface**
    - Test
        - Start the default set of frequencies and time from when the start button is pressed to when the graph is displayed. I used AMD's a screen capture software to do the timing
    - Expected results
        - The sequence to take around 20 second to accomplish the 8 different frequencies played
    - Given result
        - The sequence only took 12 seconds to play, record, analyse, and display the graph in 12 seconds

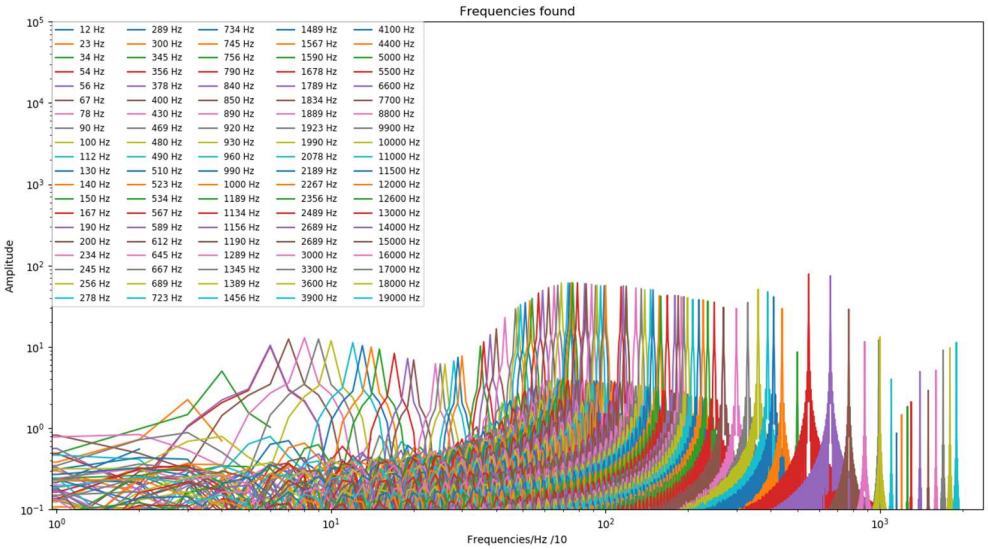
Should:

- **When played through a PA system with a mic inline, or when the mic's output is being played back real time, point out a dominant frequency and be able to suggest if it is going through a feedback loop.**
    - Test
        - Use Window's audio settings to listen to the mic. This setting played back the microphones input sound through the speakers. I forced a feedback loop my keeping my microphone close to the source of audio and with high gain.
    - Expected results
        - The microphone to have a feedback loop starting at the 1000Hz to the 16000Hz range and to be seen by frequencies being carried on and showing in runs of other frequencies. In the graph, this might look like a certain frequency might have lines of different colour overlap at high amplitudes.
    - Given result
        - Some frequencies such as 125Hz and 250Hz had high amplitudes due to them having a feedback loop within their own playing time but did not carry over between other tests. As for the 4000Hz test, I found that the amplitude was very large and that it carried on even after playing the 8000Hz and the 16000Hz part of the test as seen by their yellow and blue colours under the main grey 4000Hz test. My program is only able to tell the highest played frequency for a given frequency which did not help much when finding the Feedback loops.



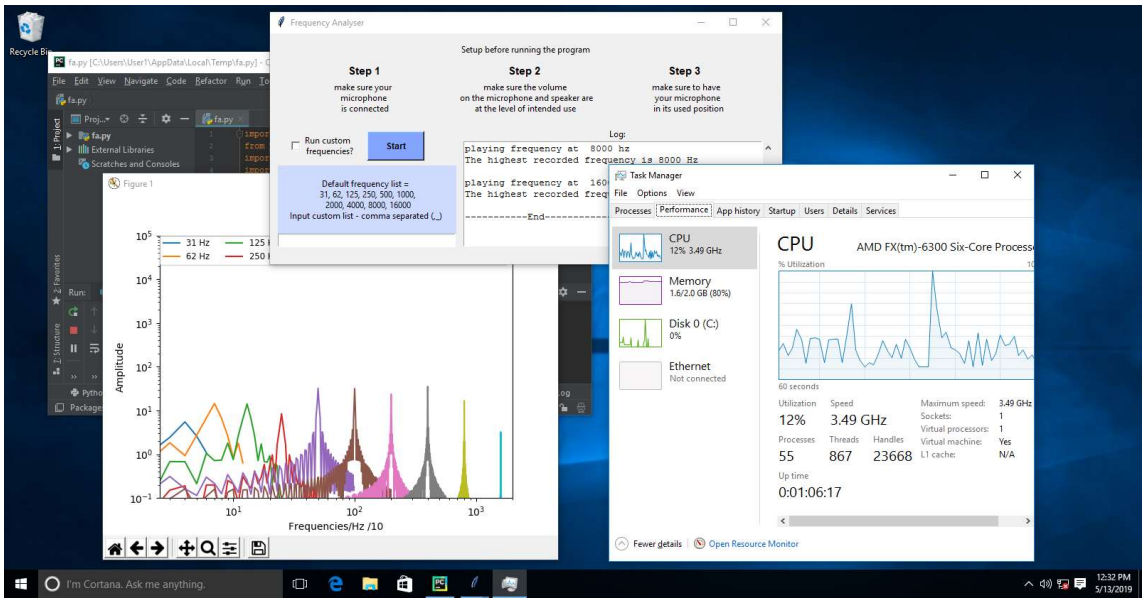
I also Ran a few tests to find if my program would handle playing 100 different frequencies without problems to test the robustness of the system. I made a list of 100 different frequencies ranging from 20 Hz to the 19000 Hz range. During the test, I also timed the runs using the same method of recording my screen to find the average processing time per frequency played to test the efficiency of my methods.

This is what the result of the 100 tests looks like:



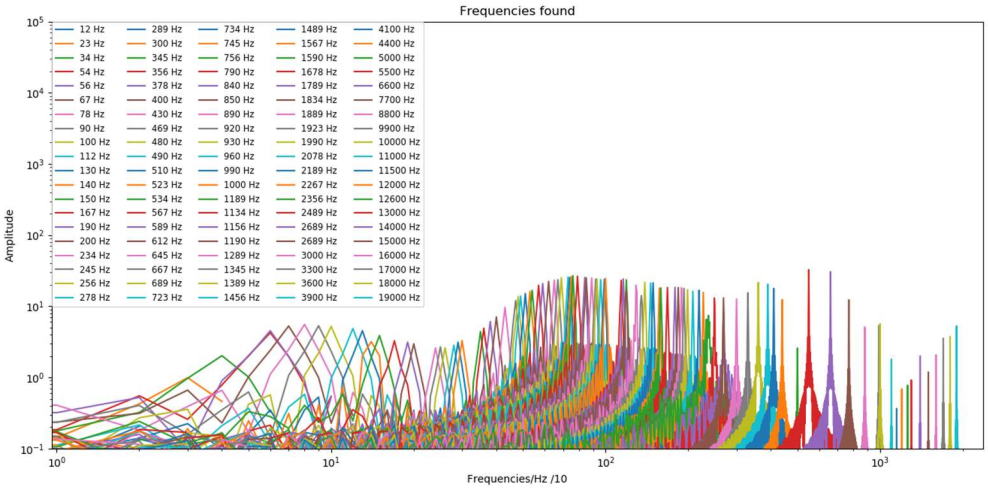
The test took a total of 136 second to run on my machine which meant that there was only 36 extra seconds outside of the playing and recording of processing time. The program took a maximum of 250 MB of RAM during the process and there were no problems except for Tkinter hanging up half the way through and it stopped displaying the log box until the process ended.

I then ran my program on a Virtual Machine that I configured using Oracle's VM VirtualBox software running a clean installation of Windows 10 64 bit and gave the system 2 gigabytes of ram clocked at 1866Mhz and running dual channel, and a single 3.5GHz core with no turboboost from the same AMD fx 6300 processor I used for the tests for the Client. The program worked well except for when random Windows tasks using up all of the CPU which gave a low cracking noise instead of pausing.



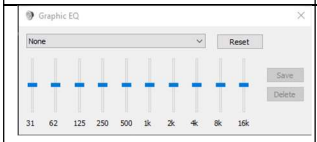
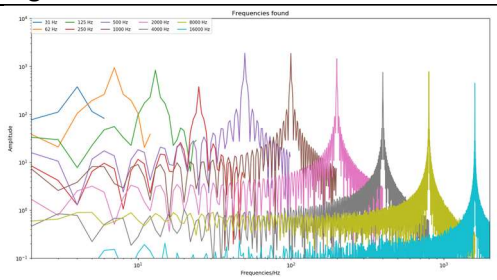
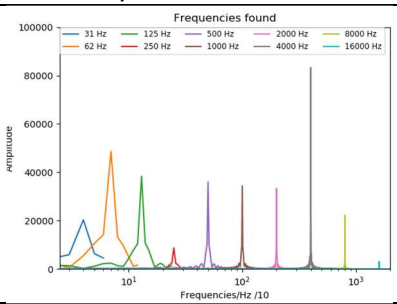
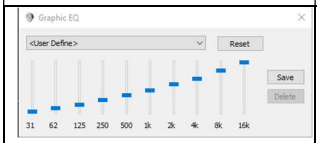
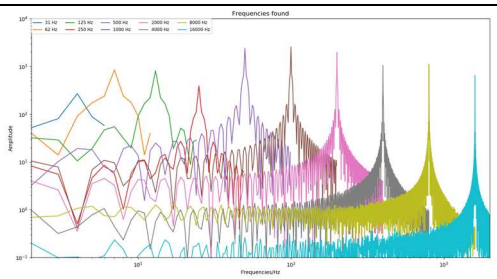
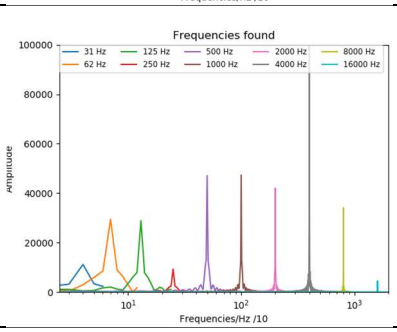
I then ran the program again with the 100 separate frequencies in the virtual machine to see how much the lack of another core to handle the operating system takes a toll on the program runtime.

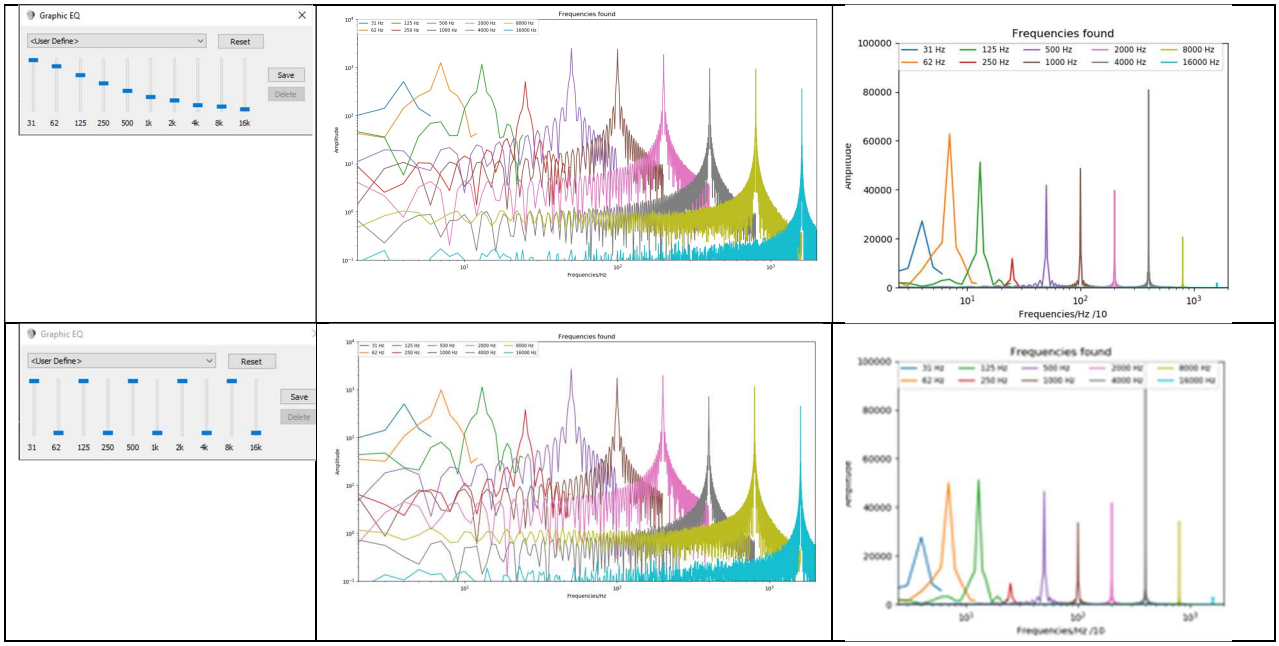
This was the graph that ended up being made:



The program took a total of 145 seconds to fully run with the same frequencies used which means that the total processing time was 45 seconds which is 25% slower than the with multiple cores but I would say is still usable.

Eq test: I also ran a few tests to show how much Window's equaliser settings affect the amplitude of their corresponding frequency. I made 2 tests. One shows the amplitude logarithmically which is closer to how sound is perceived to humans and in the other column, I show the same test with amplitude being linear to excaudate the amplitudes so changes can be seen much easier.

Equaliser Configuration	Graph plotted with amplitude logarithmic	Graph with graph plotted in a linear way
		
		



I am able to see how the Window's equaliser is able to affect the amplitudes of the corresponding frequencies plotted on the graph. Although, I do see that the amplitudes do not change drastically as the change is somewhere within the realm of +- 5 to 10 decibels. That shows that larger differences may be seen by people with dedicated equalisers on a sound board or when using another piece of software to equalise their sound.

Evaluation

My program accomplished most of what I expected technically that I needed in order to solve the problem that I initially tried to investigate. Technical problems such as the recording of audio, the processing of audio and display of a user interface that can get parts of the program running and display information to the user.

After writing the program and setting up a system for my client to run it on; a computer with a speaker and a microphone in a sort of podcast setting, I set out to get some feedback.

The positive raised by my client was:

- The user interface is not too clustered and they liked the use of the colours to make it obvious on how to start the analysing.
- They could understand the messages displayed in the log box were enough to get an idea of what the program is doing and an idea of how to troubleshoot expectedly common problems. Such as mistakes in their input lists of frequencies.
- The program quick and snappy to run and complete a cycle of frequencies (the processor and memory used by system the user tested on contained an AMD 'fx 6300' processor running at 3.5 GHz and the memory used was a configuration of 2x 'HyperX Fury' DDR3 4GB modules running at 1866MHz dual channel)

The feedback I got in order to improve on my program:

- The explanation of the different frequencies in the user interface is not very extensive and is too hidden.
    - Solution: Have a brightly coloured button in the user interface that opens a pop up window with a more detailed and extensive guide on what different range of frequencies sound like and what they mean in audio.
- It would be easier and quicker to understand and the graph of recorded frequencies at a glance if it were plotted as a bar chart and not a line graph.
    - Solution: Separate the visible frequencies in the graph and make the bar chart with the highest amplitude being the top of the bar. Keep the bar coloured to distinguish one frequency for another.
- For people that might need to keep volume under a threshold, if the graph plotted had amplitude in terms of decibels rather than some random value
    - Solution: use an algorithm that could give the amplitude of the fft signal in decibels from the recording of the microphone.
- Reading the graph would also be easier if the labels were plotted directly on top of the graph
    - Solution: I could have done this using one of matplotlib's functions to attach the label to the top of the peak of the data for frequency played on the graph which could be static or be shown when the user hovers over the plotted data.

- Controlling the equaliser would be much more convenient on the computer if the an equaliser could be put on the user interface itself rather than digging into Microsoft's Window's audio settings to get to them
    - Solution: to program an in built equaliser which has all computer data pass through the program, or a more feasible solution being some button that when clicked, has Window's equaliser pop up.

Other problems could realistically be improved on or implemented are:

- Have the program scale to the resolution of the display to stay about one size and to not loose text resolution when scaled up.
    - Solution: I am not sure how to approach this problem at the moment but as far as I know. it might require me to use another library to make a User interface with rather than Tkinter
- The ability to filter or get rid of noise from other sources whilst the program is recording.
    - Solution: Have a function which records the room whilst not playing any sounds, gather the frequencies playing in the background and subtract their amplitudes from the final displayed graph with the frequencies recorded.
- The ability to automatically recognise feedback loops and inform the user.

    - Solution: check the 2 highest amplitudes when a frequency is played. Is the second highest amplitude is not noise, has been played or is a harmonic of the frequency, prompt the user that they might have a feedback loop

- The ability to recognise the sound profile of a given type of microphone to adjust and correct the recorded audio data.
    - Solution: Before starting the program, the user could either enter the sound profile of their microphone as stated by the manufacturer on a slider if they know it which would then adjust the amplitude values for the frequencies recorded or the user can at least state the type of microphone they are using or, perhaps the manufacturer of the microphone they are using (for example a condenser microphone from Behringer) and some assumptions can be made about the sound profile that could then be adjusted on the graph

- A safety feature that could stop the program after the reaching a maximum amplitude set by the user
    - Solution: The user could have a box where they can input their maximum amplitude for a frequency in decibels and whilst the mic is recorded, the audio processing could happen in real time and stop the program if the maximum volume set by the user is reached.

My program for the most part covers the fundamentals of what it needs to do I could have implemented more important features into the program if I did not spend a lot of time getting stuck programming small features such as the auto-scroll in the log box within the user interface. Although I did make an effort to stick to pep8 programming standards, I still did not plan the names I should have used for functions and variables and I also have functions that do a lot of things that might be unexplained which did cause difficulties in writing the code after periods of time between programming sessions. If I were to start again, I would prioritise time in developing the important parts of the program first and then plan my code out before jumping in writing. This would also save me time from programming methods that I would change further down the line in development. I used GitHub to record my progress and to save my repository so that I could develop my program when I did not have access to my computer and to remind myself of things that have and have not sorted out in my code. My Repository is at https://github.com/WhicheverCub05/CS-NEA.