An Insight into Audio Signal Processing for Musical Information Retrieval

Aditya Tatwawadi - Wren's

Westminster School
Computational Engineering Project
John Locke
August 2021

Contents

List of Python Code:

Foreword:

My main goal of this John Locke has been to make an application with Python. Please reach out if you would like a live demonstration. Much of this report is supplentary as an overview of Audio Signal Processing. I also have very little to no background in music. Consider accessing GitHub repository here [1]

An Insight into Audio Signal Processing

1 Introduction

Audio Signal Processing (ASP) has addressed many problems previously faced across various industries. Being able to identify songs was one such notoriously difficult issue. Companies such as *Shazam* have found ingenious methods to solve this and create a user-friendly product. Today, *Shazam* is valued at over $400 million, and has been acquired by *Apple* [2]. Modern day applications of ASP include data-compression, speech processing, noise-cancellation and more. This is an industry to look out for.

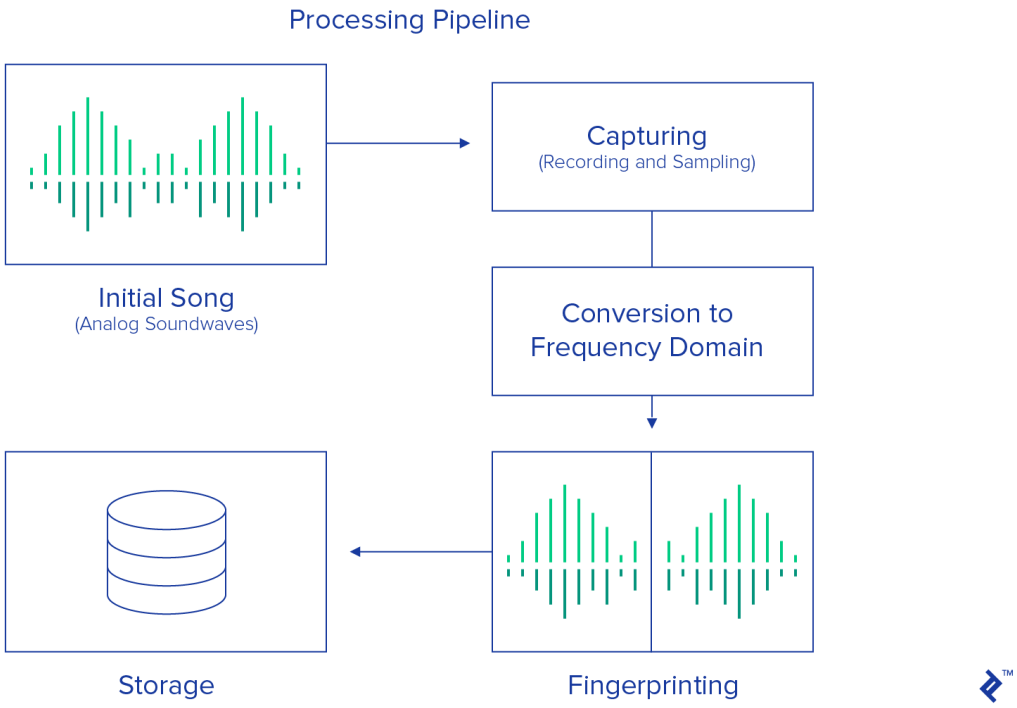

Figure 1: Shazam application timeline [3]

The theory and applications behind ASP have historically been championed at *Bell Labs* since the start of the 20th century after the introduction of telephones, radios and phonographs. Previously, analogue signals - which relied on continuous signals being "analogous" to the sound waves in air - needed to be physically altered by electric circuits. This made the process cumbersome and cost inefficient. Thanks to the advent of computational power, digital signals can now be used to express audioforms in binary numbers. This makes modern day signal processing more powerful, reliable and cheaper to implement [4].

In this report, the focus will be placed on music information retrieval, as well as the insights into the mathematics and physics to achieve this. Additionally, I have also built my own application to give song predictions using the *Python 3* programming language. This can be found referring to section 2.2 on page 3.

2 The Parson's Code

2.1 Theory

Developed by Denys Parsons in his 1975 book *The Directory of Tunes and Musical Themes*, the Parson's code is a surprisingly simple yet accurate way of identifying melodies; this makes it well-suited towards music information retrieval [5].

One of the key benefits of using the Parson's code is that no technical music knowledge is needed to implement it. It works through melodic motion: that is by identifying movements of pitch varying up and down. The first tone of reference is denoted with an asterix: *. If the following note is higher, a U (up) is used. If the following note is lower a D (down) is used. If the following note is repeated, a "R" (repeat) is used.

e.g. The Parson's code to display *Fur Elise* would be:

*DUDUDUDDDUUU.

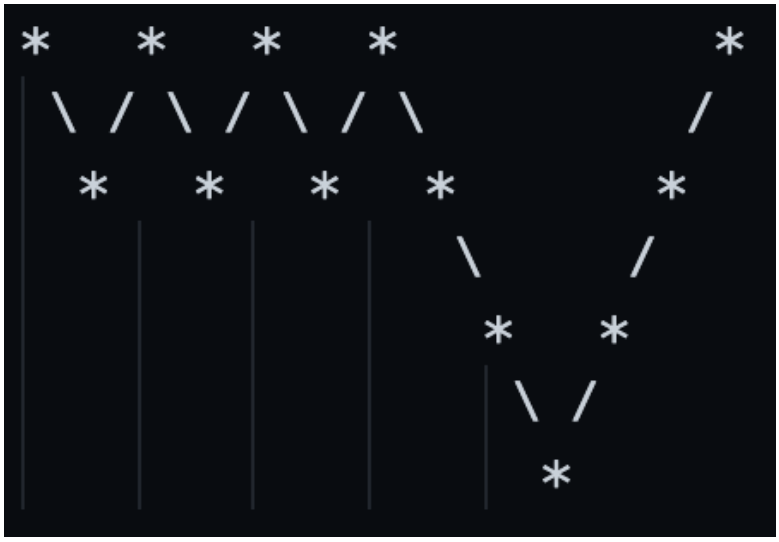

Figure 2: Parson's contour for "Fur Elise" using my application

The Parson's code is only reliable in music retrieval if there is an identifiable, but not simultaneously varied pitch. Due to this, extraction is only feasible for classical/ folk/ distinct melodies, and not so much for hip-hop/ jazz songs or anything containing mixed timbres. Other limitations include difficulty in storing data, due to the Parson's code being relatively unknown and not well documented.

2.2 Practical Implementation of Parson's Code

On page 7 is the code of my project written to suggest to a user the top 3 most likely song predictions of a given MIDI audio file. See Listings 1,2,3 from Appendix. To access a cleaner version of my code, consider forking my repository on *Github* [1]. An explanation can be found on page 4.

2.3 Brief Overview of Application

There are 3 main files which run the code - main.py, compute.py and webscrape.py.

- The main.py file imports the code from the other two files, and produces an image from *Musipedia*: which has the information about the top 3 most likely song names and their composer.
- The compute.py file is an algorithm to extract the Parson's code from a MIDI file, produce a visual contour, and store it as a string to be accessed later.
- The webscrape.py file hits the internet and uses the *Musipedia* database to extract the songs corresponding to the identfied Parson's Code.

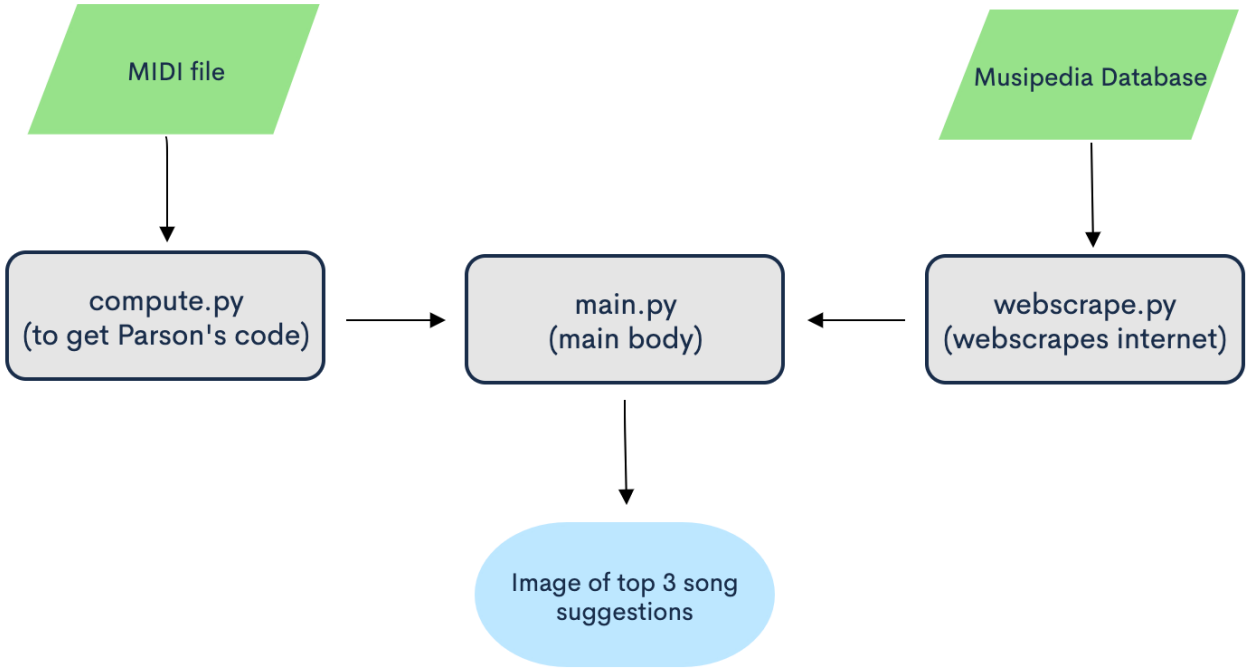

Figure 3: Flowchart of events

It is important to note that this is a somewhat abstraction of what is actually happening - the true computation lies in the *Mido* python library. The MidiFile object is able to read, write and play back any MIDI files. Although humans cannot hear .mid files like they would a .wav file, midi files are more accurate in transferring information between different places.

The next section will consider some of the mathematics and physics behing how exactly an audio file is decomposed, and an insight into how more sophisticated methods are used to generalize audio extraction to any type of music.

3 Scientific Methods

3.1 Physics behind ASP

What is sound? Sound can be described as a vibration of pressure that propagates as longitudinal waves. After hitting the eardrum, the vibration is then transmitted to hair cells in the cochlea, which produce electrical signals to the brain.

This process of converting to an electrical signal from the pressure of a sound wave (continuous signals) is imitated by recording devices. In a microphone, the Operational Amplifer (Op-Amp) translates this into an analog voltage signal. For the continuous signal to be of use, it must be translated into a discrete signal that can be stored digitally. This is done by capturing a digital value that represents the amplitude of the signal. The quantization of the input means that an analog-to-digital converter performs many conversions on very small pieces of the signal is a process known as sampling [6].

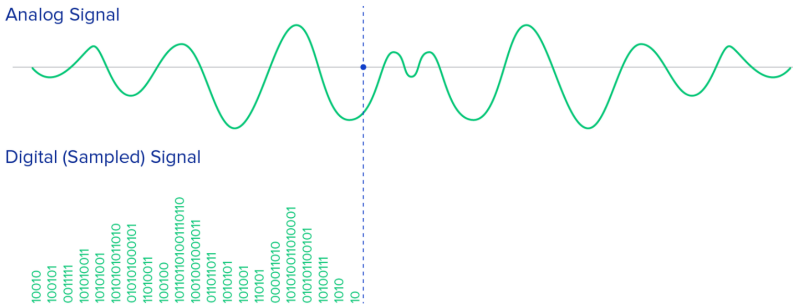

Figure 4: Change of signals [3]

Every sound in the world can be represented as the sum of multiples of pure tones in different amplitudes. As tones are expressed in sinusoidal forms, every sound in the world can therefore be expressed in terms of sinusoidal waves. Music is played by various instruments and singers, producing a combination of sinewaves at varying frequencies, creating an overall more obscure sine wave. This can be visualized using a Spectrogram: a 3D graph. Time on the horizontal X axis. Frequency on the vertical Y axis. A colour representing the amplitude of a defined frequency (in decibels, dB) as the 3rd dimension. Spectrograms can be computed using Fourier Transforms [7].

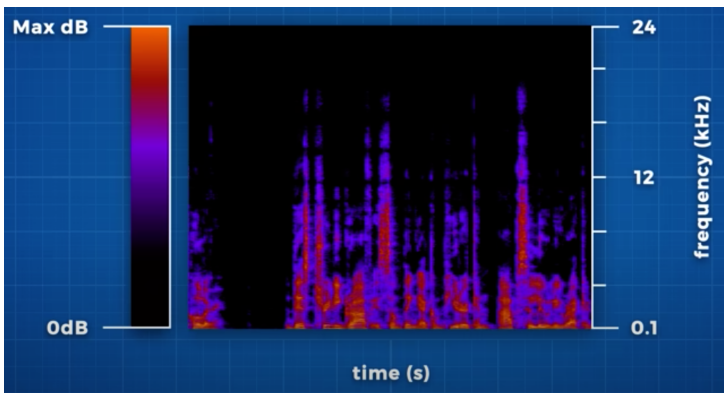

Figure 5: Spectrogram [8]

3.2 Mathematics behind ASP

Sound is made up of a lot of complex wavefunctions. One analogy which could be used to represent the essence of ASP is like blending up a smoothie, and then trying to reverse-engineering the process to get back its original ingredients.

Though of course impossible in real life, in the 19th century Jean-Baptiste Joseph Fourier made a trailblazing advancement that the time domain can be represented as the sum of multiple sinusoidal signals (as aformentioned on page 5). This is known as the Fourier Series.

The Fourier Series is used to represent a periodic function by a discrete sum of complex exponentials, while the Fourier transform is then used to represent a general, non-periodic function by an integral of complex exponentials. The Fourier Transform can be viewed as the limit of the Fourier series of a function as the period approaches infinity, so the limits of integration are $(-\infty, \infty)$. Instead of trying to crudely fit a signal to model a wave as $A\sin(2\pi(ft+\phi))$, any complex wavefunction can be represeted as the sum of sines and cosines [9]. It is important to note that this is one of the most elegant, yet conceptually difficult mathematical topics to understand, and so cannot be given full justice in just one section of a report. We are concerned with Discrete FT in ASP.

The Fourier Transform (FT) applies to continuous time giving a continuous spectrum:

$$\hat{\mathcal{F}}(\xi) = \int_{-\infty}^{+\infty} f(t)e^{-2\pi i\xi t}\,dt = \int_{-\infty}^{+\infty} f(x)e^{-i\omega t}\,dt$$

where $\omega \in (-\infty, +\infty)$

The Discrete Fourier Transform (DFT) applies to finite time giving a discrete spectrum:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-i(2\pi kn/N)}$$

where $n = 0, 1, ..., N-1$

N is the number of samples that composed the signal. X(K) represents the kth bit of frequencies. x(n) is the nth sample of the audio sample.

The use of the complex number $i = \sqrt{-1}$ is purely graphical: the algebra tends to be nicer with geometrically related problems to do with winding or rotation. This can be visualized on the Argand Diagram; instead of x,y axis on the Cartesian Plane, there are real and imaginary axis. As the Fourier Transform pertains to circular paths, Euler's Formula is a suitable way to generate one:

$$e^{i\theta} = \cos(\theta) + i\sin(\theta)$$

In industry, applications like *Shazam* need to optimise processing time to compute a spectrogram, and produce a match to the recorded audio, all within the constraints of the phone's processor. For this reason, the Fast Fourier Transform (FFT) - which serves the same function as the DFT, but only much faster, - is used. The simplest and most efficient version of the FFT (also used in the numpy package in the implementation of my code in page 7) is the Cooley-Tukey algorithm. The FFT has a time-complexity of $O(nlog(n))$ which is far superior to DFT's $O(n^2)$ [10]:

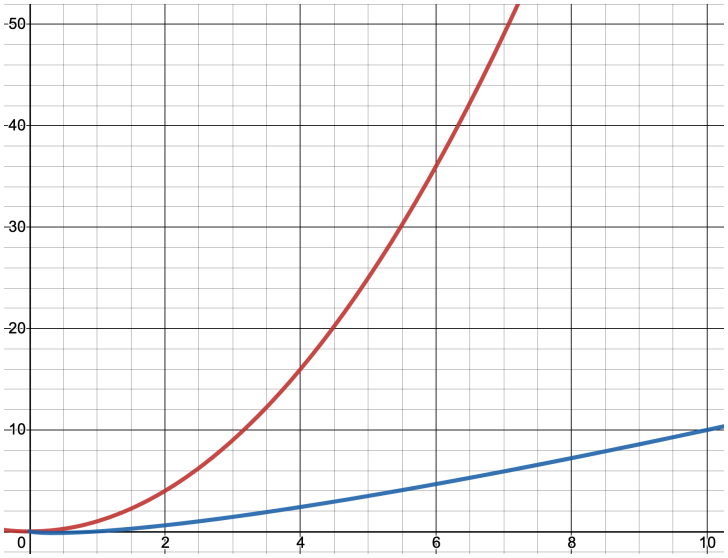

Figure 6: DFT - Red - $O(n^2)$ —— FFT - Blue - $O(nlog(n))$

Such is the importance of the FFT, that the Institue of Electrical & Electronics Engineers (IEEE) deemed it as "the most ubiquitous algorithm in use to analyze digital data", and has been heralded as the top 10 most influential algorithms of the century by countless publishers and universities [11].

3.3 Practical Implementation of Fourier Transform

Python Code which performs an FFT can be located in the Appendix under Listing 4 on page 13. This takes a time domain signal, applies a FFT and converts it into a frequency domain signal. Like my previous application, this can be found on my GitHub repository.

3.4 Database

As a sidenote, it is intriguing to consider how *Shazam* stores this data. For each individual song, after the FFT is applied and a spectrogram in computed, a *fingerprint* - i.e. a starmap is used to reduce the data from 3D to 2D. Hash functions and tables are then used in searching algorithms. This goes beyind the scope of this report, but more detail can be located in this article by the founder of Shazam - Avery Wang [12].

4 Appendix

A compilation of all code. Everything hosted on GitHub [1].

Listing 1: main.py

```
"""
Aditya Tatwawadi - Summmer 2021 - Westminster School - John Locke

Application to take a song in a .mid file, convert it into Parson's Code
and display a photo of possible song suggestions

n.b.
- Generally only successful for Classical music with a single melody line in isolation (Though does work for most songs on musipedia.
- .mid file should be very accurate (ideally taken from musipedia).

Some test songs & their corresponding Parson's codes:

Fur Elise               | DUDUDUDDDUUU      | Giscard Rasquin & Ludwig van Beethoven
Eine kleine Nachtmusik | DUDUDUUUDDUDUDDUD | Wolfgang Amadeus Mozart
Jana Gana Mana          | UURRRRR           | Rabindranath Tagore
"""

#Import general dependency
import time

#Import Pillow to show screenshot (will need to be done with HomeBrew)
from PIL import Image


def show_suggestions():
    music_suggestions = Image.open("musipedia_recommendations.png")
    music_suggestions.show()

if __name__ == "__main__":

    #Import file written to compute Parson's Code
    import compute

    #Delay needed to read txt file
    time.sleep(1)

    #Import file written to scrape Musipedia
    import webscrape
    time.sleep(1)
    show_suggestions()
```

Listing 2: compute.py

```
#Import dependency to read a .mid file. RECOMMENDED - mido documentation
import mido

#Enter path of file
path_name_of_mid_file = "//Users/adityatatwawadi/Desktop/Projects/Python/JohnLocke2021/Midi Files/FurElise.mid"

#Compute stuff
class ComputeParsons:

    def __init__(self):
        pass

    def midi_to_parsons(midifile, limit=13, offset=0):

        """
        Takes a midi file path, outputs a string
        """

        #Info to be added to empty variable string parsons
        global parsons
        count = 0
        parsons = ""

        #Reads midifile - a mid file is a series of 8-bit bytes
        for message in mido.MidiFile(midifile):

            if "note_on" in str(message) and offset == 0:

                if parsons == "":
                    # initialise list
                    prev = message.note
                    parsons += "*"

                #comparison to determine u d or r (As per Parson's Code)
                elif message.note > prev:
                    parsons += "u"
                    prev = message.note
                elif message.note < prev:
                    parsons += "d"
                    prev = message.note
                elif message.note == prev:
                    parsons += "r"
                    prev = message.note

                #increment count
                count += 1
                if count >= limit:
```

```
                    break

            elif "note_on" in str(message):
                offset -= 1

        return(f"The Parson's code for the file you have inputted
            with the path {path_name_of_mid_file} is {parsons}")

    def contour(code):

        """
        Creates a contor image using the Parson's code
        Output: UI display
        """

        #Not needed - only for individual file testing
        if code[0] != "*":
            raise ValueError("Parsons Code must start with '*'")

        #Initialize an empty dictionary
        contour_dict = {}
        pitch = 0
        index = 0

        max_pitch = 0
        min_pitch = 0

        contour_dict[(pitch, index)] = "*"

        for character in code:
            if character == "r":
                index += 1
                contour_dict[(pitch, index)] = "-"

                index += 1
                contour_dict[(pitch, index)] = "*"
            elif character == "u":
                index += 1
                pitch -= 1
                contour_dict[(pitch, index)] = "/"

                index += 1
                pitch -= 1
                contour_dict[(pitch, index)] = "*"

                if pitch < max_pitch:
                    max_pitch = pitch
            elif character == "d":
                index += 1
                pitch += 1
                contour_dict[(pitch, index)] = "\\"
```

```
                index += 1
                pitch += 1
                contour_dict[(pitch, index)] = "*"

                if pitch > min_pitch:
                    min_pitch = pitch

        for pitch in range(max_pitch, min_pitch+1):
            line = [" " for _ in range(index + 1)]
            for pos in range(index + 1):
                if (pitch, pos) in contour_dict:
                    line[pos] = contour_dict[(pitch, pos)]


            print("".join(line))

    def save_parsons():

        """
        Saves Parson's code to a txt file so webscrape.py can
           access it
        """

        #Writes code to a txt file
        with open("parson.txt", 'w') as f:
            f.write(parsons)

print(ComputeParsons.midi_to_parsons(path_name_of_mid_file))
ComputeParsons.contour(parsons)
ComputeParsons.save_parsons()
```

Listing 3: webscrape.py

```
#Import all dependencies to webscrape
from selenium import webdriver
from selenium.webdriver.common.by import By

#Scrape stuff
class WebScrape():

    def __init__(self):
        pass

    def read_parsons():

        """
        Reads in Parson's code written by compute.py to use as
           variable
        """

```

```
        #So that variable can be accessed elsewhere in the file
        global parsons_code

        with open("parson.txt", "r") as parsons_file:
            parsons_code = parsons_file.read()

    def scrape_musipedia():

        """
        - Uses Selenium & ChromeDriver to extract information
           about top 3 most likely songs
        - Requires scraping html
        - Relies on Musipedia not being updated
        """

        #Path details
        path_of_driver = "/Users/adityatatwawadi/Downloads/
           chromedriver"
        driver = webdriver.Chrome(executable_path =
           path_of_driver)

        #Access musipedia's website
        website_url = "https://www.musipedia.org/melodic_contour.
           html"
        driver.get(website_url)
        #print(driver.title)

        #Enters in the Parson's code to search engine
        driver.find_element(By.NAME, 'tx_mpsearch_pi1[pc]').
           send_keys(parsons_code)

        #Submits & accesses database
        click_button = driver.find_element(By.NAME, '
           tx_mpsearch_pi1[submit_button]')
        click_button.click()

        #Scrolls down to top 3 identified songs
        driver.execute_script("window.scrollBy(0,400)")

        #Reframes & takes a screenshot of top3 recommendtaions
        S = lambda X: driver.execute_script('return document.body
           .parentNode.scroll' +X)
        driver.set_window_size(S('Width'), S('Height'))
        driver.find_element_by_tag_name('body').screenshot("
           musipedia_recommendations.png")

        #Exits automated chrome
        driver.quit()

WebScrape.read_parsons()
WebScrape.scrape_musipedia()
```

Listing 4: Fourier Transform Demonstration

```
"""
Aditya Tatwawadi - Westminster School
Computes a Fast Fourier Transform using Numpy - Turns Time domain signal into Frequency domain signal
Plots the frequency spectrum using matplotlib
"""

#Import dependencies
import numpy as np
import matplotlib.pyplot as plt

def compute():
    #Construct a time signal from sine wave:

    #Sampling Frequency (Hz):
    frequency_sample = 2000
    #Sample time interval:
    tstep = 1 / frequency_sample
    #Signal Frequency:
    signal_frequency = 100

    #Number of samples:
    N = int(10 * frequency_sample/ signal_frequency)

    #Time steps:
    t = np.linspace(0, (N-1)*tstep, N)
    #Frequency interval:
    fstep = frequency_sample / N
    #Frequency steps:
    f = np.linspace(0,(N-1)*fstep, N)

    #Create a chaotic wave - new one added is 2 times the original input frequency and has a magnitude of 4
    y = 1 * np.sin(2 * np.pi * signal_frequency * t) + 3 * np.sin(2 * np.pi * 2 * signal_frequency * t)

    #Perform FFT:
    X = np.fft.fft(y)
    X_mag = np.abs(X) / N

    f_plot = f[0:int(N/2+1)]
    X_mag_plot = 2 * X_mag[0:int(N/2+1)]
    X_mag_plot[0] = X_mag_plot[0] / 2

    #Plot
    fig, [ax1, ax2] = plt.subplots(nrows=2, ncols=1)
    ax1.plot(t, y, ".-")
    ax2.plot(f_plot, X_mag_plot, ".-")
    ax1.set_xlabel("Time (s)")
    ax2.set_xlabel("Frequency (Hz)")
```

```
48      ax1.grid()
49      ax2.grid()
50
51      ax1.set_xlim(0, t[-1])
52      ax2.set_xlim(0, f_plot[-1])
53      plt.tight_layout()
54      plt.show()
55  compute()
```

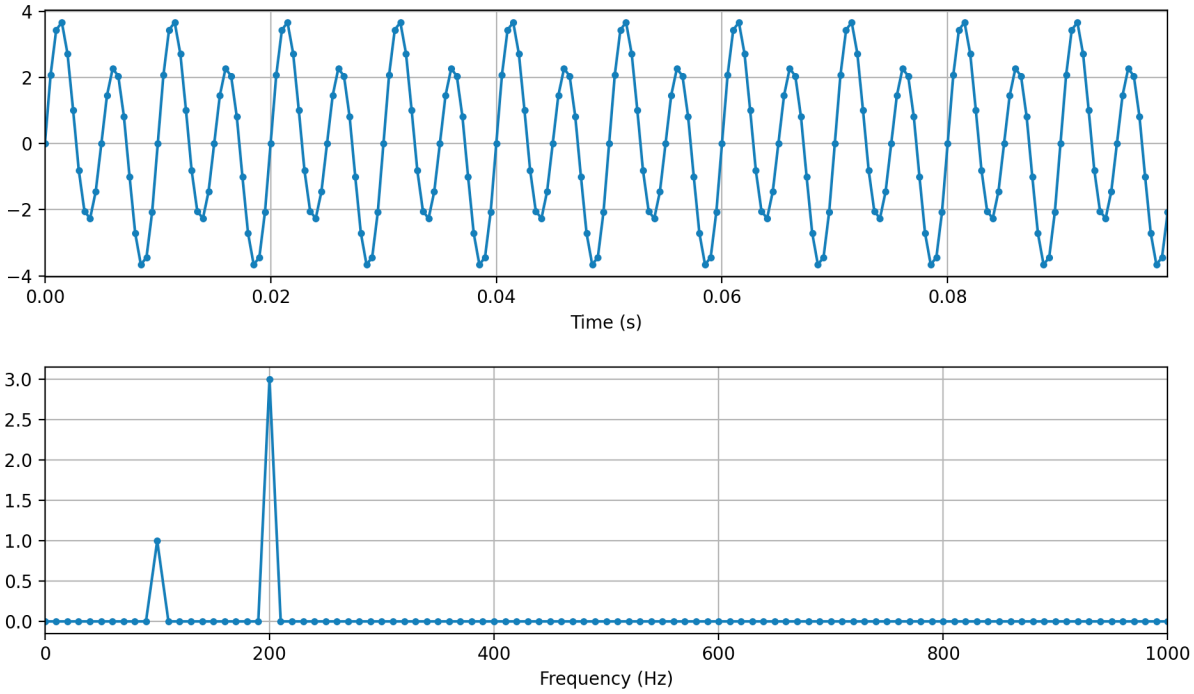

Figure 7: Matplotlib graph of FT taking Time to Frequency domain [1]

5 Postface

This project was incredibly useful in exposing me to the mathematics behind many first year EEE/ EIE degrees. Along the way I gained a lot of practical techincal information in python, such as Object-Oriented Programming, file-handling and proficiency with multiple libraries. This let me perform a wide range of objectives from data analysis, web-scraping to using logic to come to a solution. With more time, I would have looked to either Tkinter or web-development to be able to host my application on a more User-Friendly level (and hope to come back to this later). I also learnt how to host software on GitHub, intricacies on VSCode IDE, and also LaTeX. Audio Signal Processing is a much broader topic than what hasa been covered, and I look forward to returning to it.

6 Bibliography

[1] Aditya Tatwawadi. *John Locke - Audio Signal Processing GitHub Repo, 2021.* https://github.com/aditya-tatwawadi/Audio-Signal-Processing.

[2] BBC News. *Apple to buy Shazam 2017.* https://www.bbc.co.uk/news/business-42299207. Accessed August 2021.

[3] Jovan Jovanovic. *How Shazam Works.* https://www.toptal.com/algorithms/shazam-it-music-processing-fingerprinting-and-recognition. Accessed August 2021.

[4] Udo Zölzer. "Digital Audio Signal Processing". In: (1997). ISBN 0-471-97226-6.

[5] S Brown Denys Parsons. *The Directory of Tunes and Musical Themes.* ISBN 0-904747-00-X. 1975.

[6] Jonathan Allday Steven Adams. *General Classical Physics Theories. Ideas adapted from Advanced Physics.* ISBN 0-199-14680-2. 2000.

[7] Allen B. Downey. *ThinkDSP.* 2014.

[8] Real Engineering Youtube. *How Shazam Works.* https://www.youtube.com/watch?v=kMNSAhsyiDg.

[9] 3Blue1Brown. *But what is the Fourier Transform? A visual introduction.* https://www.youtube.com/watch?v=spUNpyF58BY.

[10] John W. Tukey James W. Cooley. "An Algorithm for the Machine Calculation of Complex Fourier Series." In: (1965).

[11] F. Sullivan J. Dongarra. "Computing in Science and Engineering." In: (2000).

[12] Avery Li-Chun Wang. "An Industrial-Strength Audio Search Algorithm ." In: (2003).