Digital Signal Processing Report: Final project

Interacting with songs

Group 18

Department of Information and Communication Technology, USTH

Digital Signal Processing

Dr Tung Tran Hoang

April 16, 2022

**Table of contents**

[Digital Signal Processing Report: Final project 1](#_Toc101005740)

[Abstract 3](#_Toc101005741)

[Digital Signal Processing Report: Final project 4](#_Toc101005742)

[Introduction 4](#_Toc101005743)

[Project analysis 5](#_Toc101005744)

[Import dependencies 5](#_Toc101005745)

[Get raw data and interact with it 5](#_Toc101005746)

[Get raw data 5](#_Toc101005747)

[Interact with raw data 6](#_Toc101005748)

[FIR methods 7](#_Toc101005749)

[High pass filter and low pass filter 7](#_Toc101005750)

[Band-pass and band-reject filter 9](#_Toc101005751)

[Another method 12](#_Toc101005752)

[Mix channel using the frame buffer 12](#_Toc101005753)

[*Librosa* framework 13](#_Toc101005754)

[Subpart: compressed and resize using FFmpeg framework 13](#_Toc101005755)

[Outro 14](#_Toc101005756)

# Abstract

Digital signal processing is an important part of the music. A song consists of vocals and instrumental sounds. Separating vocals from songs is useful for many purposes, such as using the melody of an instrument in it as a karaoke soundtrack. We will draw different methods for this project and compare them. In particular, the first method is about creating simple FIR filters to filter out frequency bands that contain vocals. The second method is through computation between framebuffers thus overwriting the original file. The third method used is through the use of pre-written Python libraries for interacting with music data. The musical data in this project includes a variety of sources, including using studio music or live recordings to provide the most intuitive results for comparison and interpretation.

Digital Signal Processing Report: Final project

Interacting with songs

# Introduction

Humans can recognize sounds and classify them through the ear, and more specifically, each type of sound can be recognized separately. For example, a Jazz song, with it consisting of a variety of separate sources such as guitar, saxophone, piano and most songs also include vocals.

Separating vocals from songs has many uses, either with vocals or with instrumentals. Instrument sounds can be made sounds for karaoke. Vocals are the most important element in a song. The analysis of a singer with low, mid or high tones can be discerned through digital signal processing. There are many applications that we may not be able to cover, but it is one of the most useful examples of digital signal processing in real life.

# Breakdown and analysis

The audio signal that the microphone receives is in analog form, which is also a format that can be heard by humans. Every computer processor has a built-in analog to digital converter and vice versa. With a computer, to process the sound, it is necessary to convert the song into digital form. The process is that after the computer converts from analog data to digital, we process that digital data and return the result with analog data after converting it back. The goal of this project is to process that data source to be able to separate the voice from the audio source or the audio source that contains instruments sound.

We use Python for this project. The reason we chose Python is that Python comes with libraries that can handle audio data sources and is quite clear in the way the command lines are arranged. Furthermore, we are also currently using Python in our courses in the curriculum. In the processing stage, we will use three methods. The first is based on what our team has learned from DSP to create simple audio filters, and this is also the method we will go into very closely. The second is based on phase cancellation, a method that we have learned through the process of learning the technique of separating voice from audio files. The third technique we use is Python's rather powerful audio framework for audio extraction.

# Project Approach

## Import dependencies

To make the project more convenient, we use the following Python libraries and will explain why we use them.

**Figure 1**

*Python dependencies*

*Text

Description automatically generated*

*Note.* The *numpy*, *scipy*, *matplotlib*, *librosa*, *Ipython* (of *Jupyter*) and *contextlib* framework must be added to the machine's Python environment before it can be used in a Python project based on that environment.

The *numpy* framework is used in creating arrays for buffers, calculating averages, transposing array matrices, .. mainly related to data flow tuning.

The *scipy* framework is used to overwrite ‘\**.wav’* files

The *wave* framework is used to read and get data like sample width, number of channels, number of frames, sample rate, ... from the ‘*.wav*’ file.

The *contectlib* framework is used to return a context manager that closes everything after block completion.

The *IPython* framework is used to display the audio file in the *jupyter notebook* editor.

The *librosa* framework is an advanced framework for interacting with the audio file.

## Get raw data and interact with it

### Get raw data

The main purpose of this section is to get the ‘*.wav*’ file data from an audio file into a digital file that can be read by the computer. *Getting raw data* has three main stages, that is getting data from audio files through a wave framework, getting interactive time using the frame of the audio file and finally getting data from buffers of the wave files and the corresponding array for later use.

The first job is to use the wave library to open the file to be read. Before that, we used the contextlib framework to get the final data of the wave file before processing. The data we get from the wave file using this library are sample rate, sample width, number of channels and the total number of frames.

Given the number of frames and the sampling rate, we combine it with a time metric to determine the frame length at the beginning and the end, as well as the number of frames between those two intervals. This determination is very simple, through learning about the interaction between the sample rate in a frame and the number of frames in a time, we know that after multiplying these two figures together, it will produce the sample rate as the sum of that frame. It should also be noted that the sample rate here can be roughly understood as the smallest unit in digital audio, equivalent to a quantum.

Once we get the necessary information about the beginning and end frames, we use that to get each frame's data for that period. Then comes the third part, which is to put the data into an array for interaction, we have also defined the required data types as 'uint8' and 'int16', corresponding to signed and unsigned data. Now, all that needs to be done is to read the data logically.

### Interact with raw data

For the data to be reasonable to use, since the amount of knowledge we gained could not be applied to multidimensional audio processing, we converted the required data into a one-dimensional form. Thereby making it more convenient for later data use

Now comes one of the important parts that our team has worked on, which is sound recognition. The data we collect will be presented in the form of a graph. This is part of the final exam this time around, but our team has gone further than that. This identification of audio data is mainly in the form of spectrograms of audio frequencies and amplitudes.

The use is very simple, by obtaining from one-way audio processing combined with the original sample rate collected, we have determined the spectrogram by applying the matplotlib library. The results obtained are very positive. In the next section, there will be specific examples of what we have done in this project.

## FIR methods

For this part, we use the knowledge that we have learned from the DSP course. We will create a low pass filter and a high pass filter, as well as walk through the use of bandpass and band-rejected filters.

### High pass filter and low pass filter

The first thing we did was to learn about the working mechanism of the low pass filter. A low pass filter is used for filtering high frequencies and allowing lower frequencies to pass through. The ideal low pass filter is a sinc filter.

The *sinc function* (after normalizing) is defined as:

Then we have the impulse response of a sinc filter:

*with is the cutoff frequency, specified as a fraction of the sampling rate*

because the sinc filter has an infinite length, which means that the delay of the filter will also be infinite, making this filter unrealizable. The solution is to combine it with a window, in this project we would like to use Hamming window, i.e.

The reason we choose this window is that it has a good tradeoff between frequency and amplitude accuracy, and reduced spectral leakage.

After combining with the sinc filter, we got a windowed-sinc filter

*where N is the filter length, it must be odd*

For the FIR high pass filter, we just implement a spectral inversion, i.e.

1. Change the sign of each value in ℎ[𝑛].
2. Add one to the value in the centre.

How does a spectral inversion work? It is based on the following idea. A low-pass filter generates a signal with the high frequencies removed. Hence, if you subtract this signal from the original one, you have exactly the high frequencies. This means that you can implement a high-pass filter in two steps. First, you compute:

*where is the original signal, is the low pass filter and is the low-pass-filtered signal. This is a convolution, represent by the asterisk.*

Second, you compute:

*where is the high pass filtered signal*

The alternative is to adapt the filter through spectral inversion. To show that spectral inversion has the same result, first note that, where is the *impulse response*. Now we have:

This means the high pass filter is

In our project, we are using the live version of a jazz song, the extraordinary “What A Wonderful World” by Louis Armstrong. The result after combining our simple filter function and the raw data gets a great result, which we present below.

**Figure 2.**

*First two minutes spectrogram*

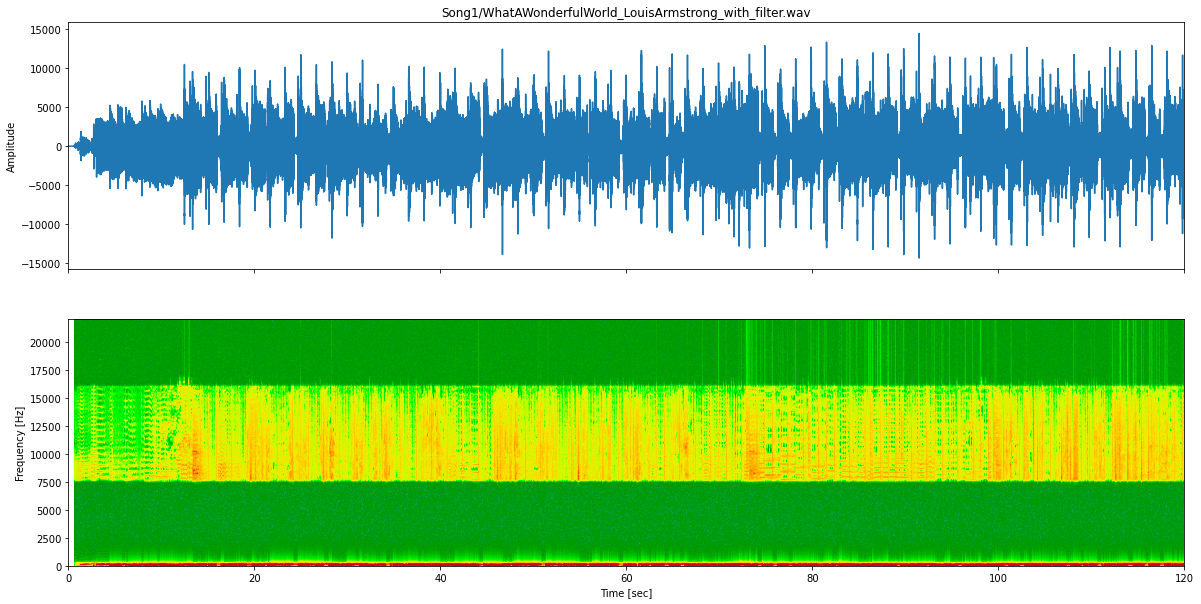
*A picture containing background pattern

Description automatically generated*

*Note.* Our version of this song is a wave file converted from the original ‘*.mp3*’ file. So the frequency above 16kHz does not appear because ‘*.mp3*’ is a compressed version of the wave file, some details may be missed from this file after being converted to a wave file.

**Figure 3.**

***Low pass filter and high pass filter combined***

******

*Note.* This is the result after we filled up the low pass with the low-frequency 199Hz and high pass with the high-frequency 7600Hz, both with filter length N = 461 and after two passes. The result is really impressive given that the vocals are gone, but the sound is not quite clear.

### Band-pass and band-reject filter

A band-pass filter passes frequencies between the lower limit and the higher limit , and rejects other frequencies. If you don’t create a specific filter for this, you can get this result in two steps. In the first step, you apply a low-pass filter with cutoff frequency ,

*where is the original signal,  is the low-pass filter with cutoff frequency , and  is the low-pass-filtered signal?*

The asterisk represents *convolution*. The result is a signal in which the rejection of frequencies larger than  has been taken care of. You can then filter that signal again, with a high-pass filter with cutoff frequency,

,

where  is the high-pass filter with cutoff frequency , and is the required band-pass-filtered signal.

However, you can do better and combine both of these filters into a single one. How does that work? You can write

where the last step follows from the associative property of convolution. This means that the required band-pass filter is

Hence, a band-pass filter can be created from a low-pass and a high-pass filter with appropriate cutoff frequencies by *convolving* the two filters.

A band-reject filter *rejects* frequencies between the lower limit  and the higher limit , and *passes* other frequencies. As for the band-pass filter, you can get this result in two steps. In the first step, you apply a low-pass filter with cutoff frequency ,

*where  is the original signal,  is the low-pass filter with cutoff frequency , and  is the low-pass-filtered signal.*

The result is a signal in which the frequencies in the rejection interval have been eliminated, but in which the frequencies higher than are also gone. This can be corrected by filtering the original signal again, with a high-pass filter with cutoff frequency , and adding the result to the first signal,

*where is the high-pass filter with cutoff frequency , and is the required band-reject-filtered signal.*

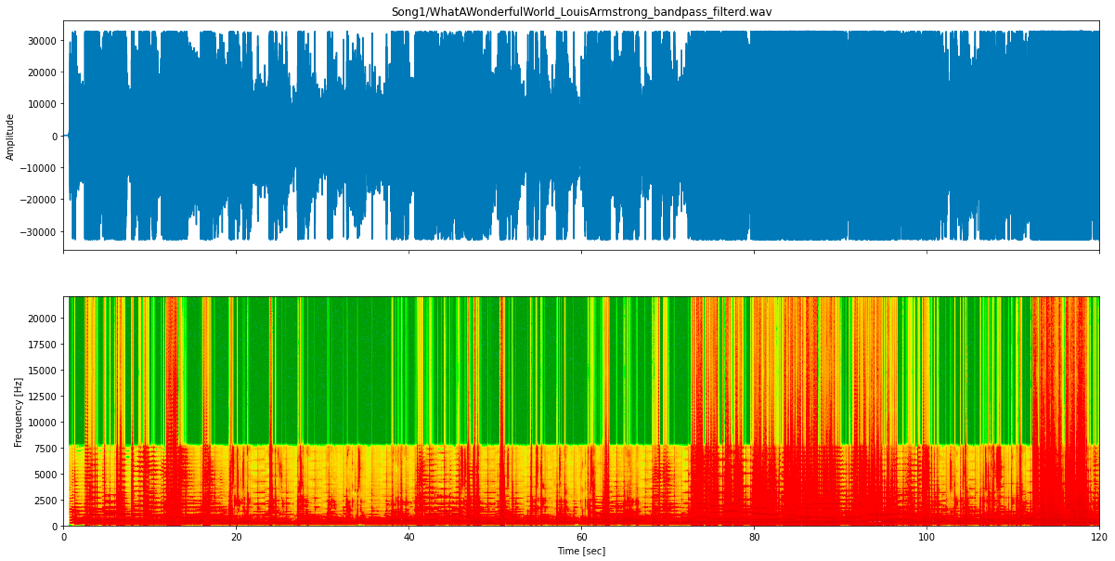
You can again to better and combine both operations into a single filter. You can write:

where the last step follows from the distributive property of convolution. This means that the required band-reject filter is

Our implementation of Python for this part ran well, but we still have to find a good frequency or filter length. The audio quality is really bad and cannot be used in other industries.

**Figure 4.**

*Band-pass filter*



*Note.* The frequency we used for this part is still the same as in the last part. i.e. 199Hz for low and 7600 for high frequencies, and the filtered length is still 461 for both.

**Figure 5.**

*Band-rejected filter*

*Timeline

Description automatically generated*

*Note.* The frequency and filter length are the same with the last part

## Another method

### Mix channel using the frame buffer

The above method didn’t give us a good result. So we have looked for another method. This one comes from a solution based on a feature of audio processing software.it’s inverting audio sample of one channel and mixed with the other one.

Before diving in, let’s talk about channels in the ‘*.wav*’ file. A wave file may many numbers of channels. This is very useful when it comes to gaming or cinema but typically in this case, they usually have 1 ( mono ) or 2 (stereo [left, right] ) channels

For the method, the process in audio software can be described as follows. Inverting is flipping the audio samples upside down, and reversing their polarity. The positive samples are moved below the zero line (so it’s becoming the negative) and the negative samples are made positive Invert does not usually affect the sound of the audio at all, but it can be used for audio cancellation. If Invert is applied to one track and that track is [mixed](https://manual.audacityteam.org/man/mixing.html) with another uninverted track that has identical audio, the identical audio is cancelled out (silenced). For the preparation of the song, if the song is mono you can try using software to convert it into stereo. We choose the method because we think the audio is the same in mono on both sides while music has a different thing on the left and the right also this method only lose bass, not treble which is appropriate for jazz.

We mimic this method in python by just taking the subtraction of two channels. The result after implementing this method is really good for a live recording, but we have tried another song, eg. “Dream a little dream of me” by Louis Armstrong. The result from a song recorded in the studio is not impressive with this method, because it can’t manage to separate vocals from the audio file. From the result, our comment is when your live recording is suitable for this method because of the stereo setup which leads to music being different in both channels

**Figure 6.**

*Mix channel in a live song*

*Chart

Description automatically generated with medium confidence*

We also implement a version that combined both the FIR filter and the frame buffer, the result is still good with the live song and not great for a studio record.

**Figure 7.**

*Mixing method filter for a live song*

A picture containing text

Description automatically generated

### *Librosa* framework

For a better result, we are using the *librosa* framework. It has a great result in separating vocals from audio files. It uses the matrix to separate between spectrum and is somehow too advanced to understand all. Please find the result in our attached code for a better view of these results.

### Subpart: compressed and resize using FFmpeg framework

We compressed and resized our wave file to the ‘*.mp3*’ file. For context, the ‘.mp3’ file is the compressed file of audio and has much less size than the original wave file. FFmpeg is the framework that can transfer files between formats, and it works extremely well.

# Outro

This is the end of our report. We have collected a numerous knowledge after doing this project, and although it took a lot of time to manage to get the code run, the results have been improved quite a lot. Thank you for your lecture. And me. the report writer, have a thank our team to manage their effort and time to give a good project