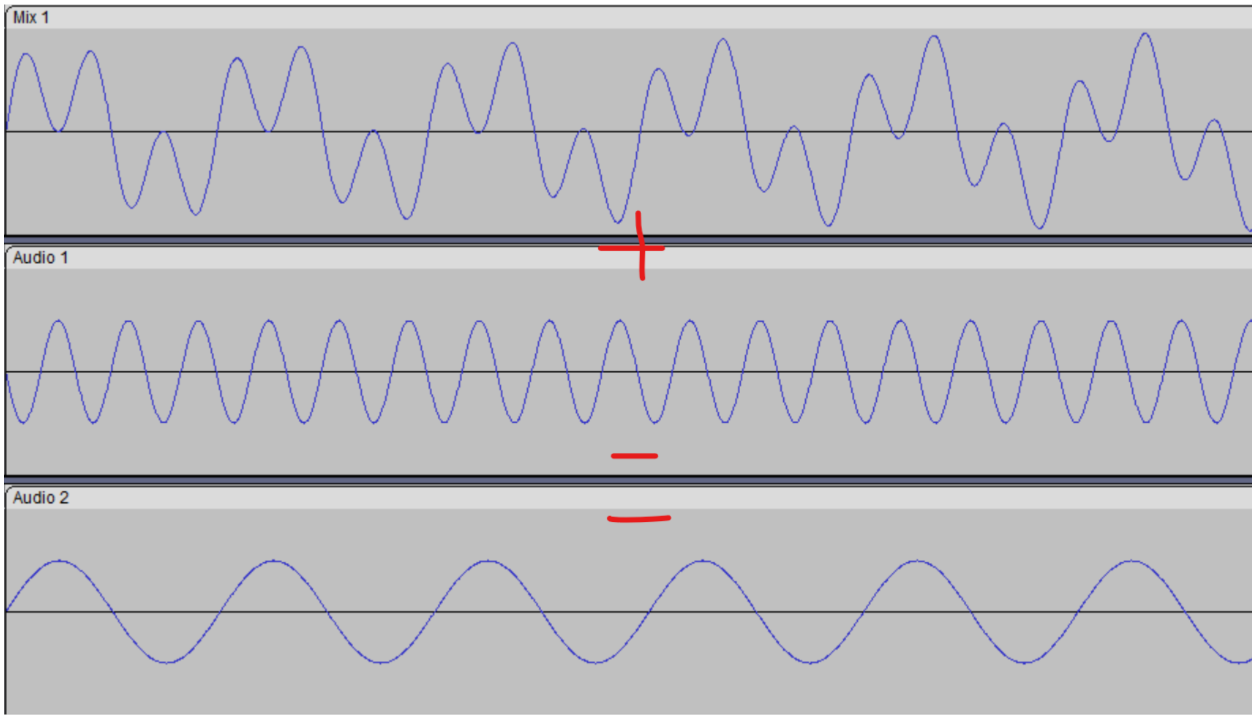
Introduction

Back in high school when I had more free time, I DJ'ed and mashed up songs for fun. A lot of mashups are created by mixing one song's instrumental with another song's acapella (vocals), so I was constantly looking for instrumental/acapella versions of songs. Instrumentals were easy to find (most studios released them for karaoke purposes); acapella versions were not. So I usually created my own vocals.

There is an old producer's trick where if you line up a song file with an instrumental file and invert the instrumental, the instrumental will cancel out and leave only the vocals. This is known as *phase cancellation* or *destructive interference*, and you can see it in action here.



Destructive interference. The top is a combination of a 440Hz and 144Hz sine wave. After adding a reversed 440Hz wave, the 440Hz wave disappears, leaving only the 144Hz wave.

The problem with this method is that you must line the files up by hand, which is tedious and subject to human error. On top of that, the by-hand method will not work if either the song or instrumental file has gone through post-processing such as gain staging, dynamic compression, or limiting. For instance, if the standalone instrumental file is louder than the instrumental in the song, the instrumental will not cancel perfectly.

This program addresses both issues by automatically lining up the files and using a signal processing algorithm to compensate for post-processing effects. This produces better vocals than the by-hand method for less manual work.

Aside from being a personal interest, vocal isolation is also a form of **digital signal processing** (DSP). DSP is used in data science, computer science, and electrical engineering (the field I plan to go into).

Description

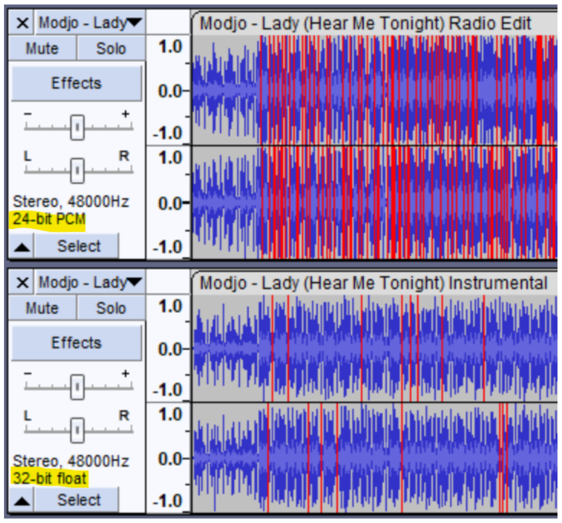
Overview. This program can be broken up into three main parts. Each part is handled by its own user-defined function:

1. File reading and validation (handled by readWavFile) – the user inputs file names, and the program validates those files and converts them to arrays that can be manipulated.
2. Aligning the song and instrumental (handled by alignWavFiles) – the program pads the inputted files with silence until they are aligned.
3. Filtering the instrumental (handled by lmsAdaptiveFilter) – this is where the "signal processing" part comes in. The program will alter ("filter") the instrumental file using a least-mean squares algorithm, then subtract it from the original to leave only the vocals.

The user-defined functions are explained in greater detail in the "User-Defined Functions" section.

Inputs. This program requires two 16-bit WAV files as inputs: one file with the whole song (vocals + instrumental), and another file with only the instrumental. The program will ask the user to enter the names of the WAV files.

This program will only work with .WAV files with 16-bit data types. Most WAV files will fulfil these criteria, but not all. If your files are not in 16-bit .WAV format, you will have to convert them first with another program.

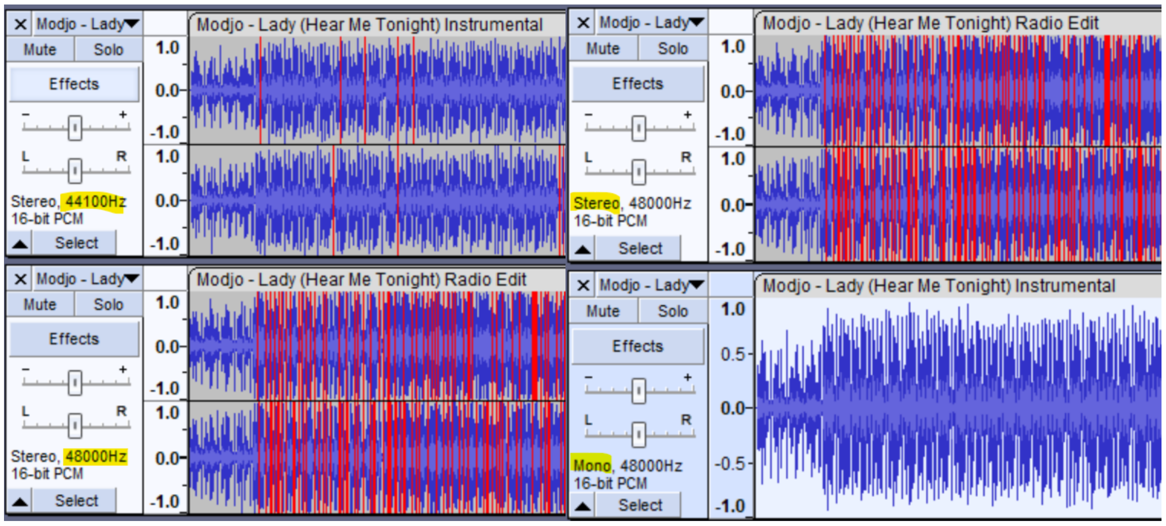


This is an example of two files that would NOT work.

These files are in 24-bit and 32-bit form instead of the required 16-bit form.

24-bit and 32-bit are incompatible data types and will cause the program to throw an error.

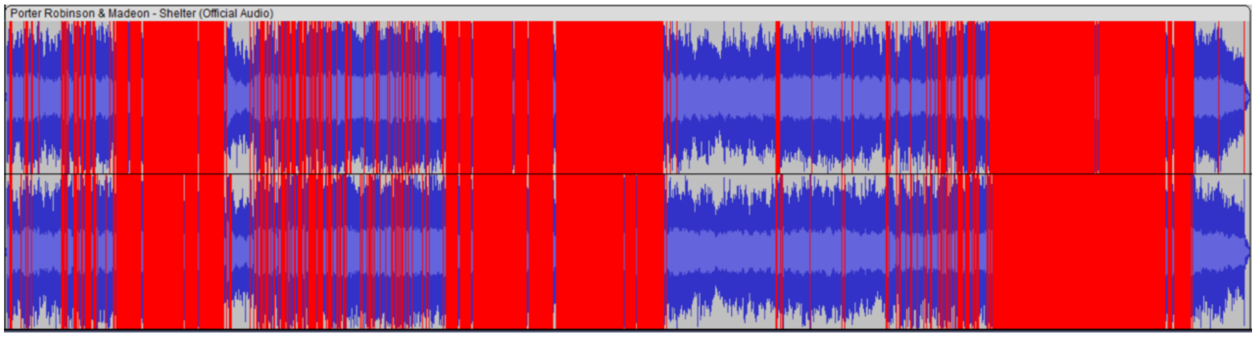
The files must also have the same sample rate. For example, it will work with two 44.1kHz files, or two 48kHz files, but it will NOT work with one 44.1kHz and one 48kHz file. Similarly, the files must also have the same number of channels. For example, it will work with two mono files, or two stereo files, but it will NOT work with one mono and one stereo file.


Modjo - Lady
Mute
Solo
Effects
Stereo, 44100Hz
16-bit PCM
Select
Modjo - Lady (Hear Me Tonight) Instrumental
Modjo - Lady (Hear Me Tonight) Radio Edit
Stereo, 48000Hz
16-bit PCM
Modjo - Lady (Hear Me Tonight) Radio Edit
Stereo, 48000Hz
Mono, 48000Hz
Modjo - Lady (Hear Me Tonight) Instrumental

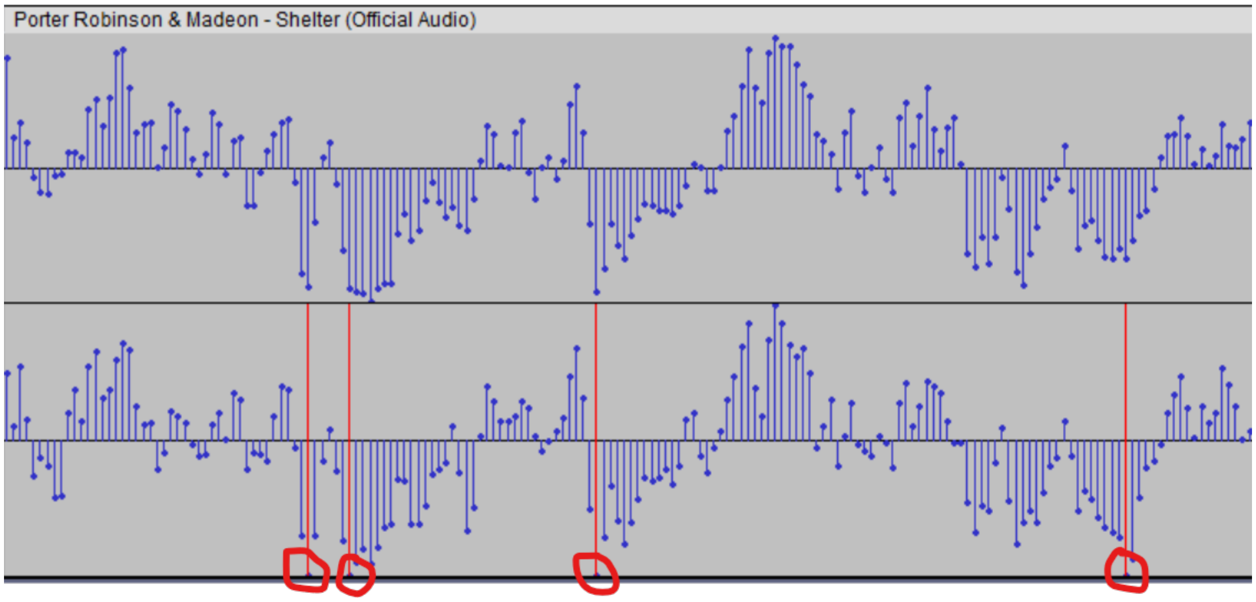
Left: Sample rates are mismatched (44.1kHz vs 48kHz).
Right: Number of channels is mismatched (mono – 1 channel vs stereo – 2 channels).
Both will trigger a program error.

The files must also be from the same recording. It will NOT work if the song and instrumentals were recorded separately, or if one file is remastered and the other is not, or if the instrumental is a recreation. This is because the LMS algorithm only works if the signals come from the same source (i.e., are the same recording).

Similarly, it will NOT work if either file is heavily distorted due to clipping or data compression. Distortion turns the audio files into different signals, which causes the algorithm to fail.



A song with a lot of distortion (shown by the red lines). The program probably won't work well with this.



A close-up of the same song. Notice how parts of the signal are cut off, causing distortion.

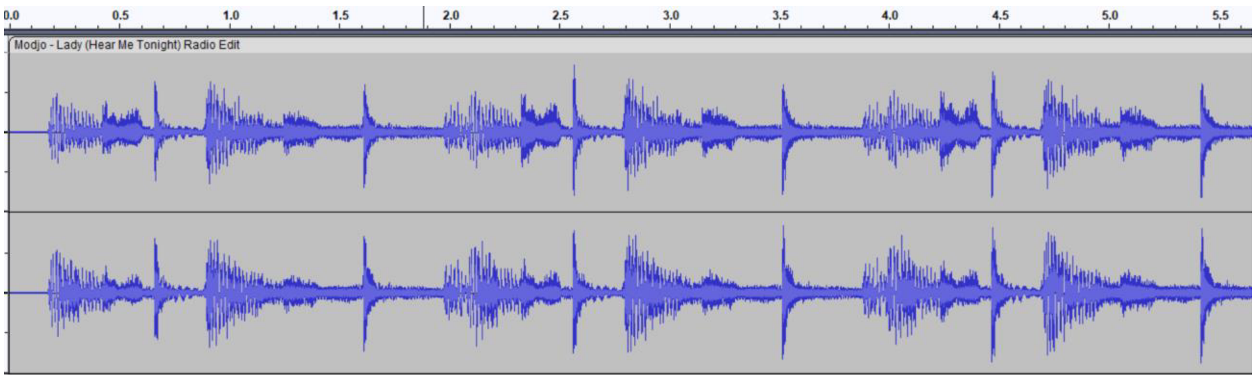
Output. This program will output a single 16-bit WAV file that contains the isolated vocals for the input song. The file will have the same sample rate and number of channels as the input files (e.g., if you input 6-channel 48kHz files, you will get a 6-channel 48kHz acapella in return).

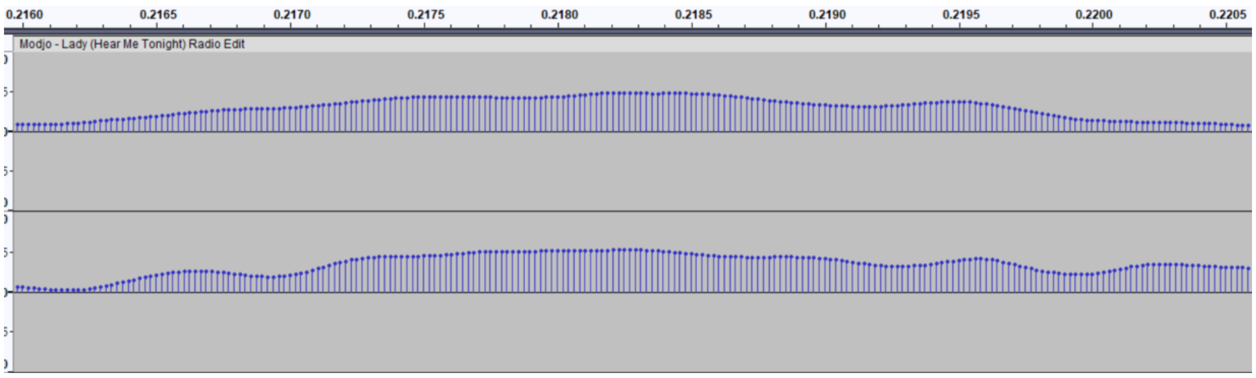
User-Defined Functions

readWavFiles. This function reads a WAV file and returns it along with the sample rate and number of channels.

A WAV file stores an audio waveform as a bunch of samples. Each of these samples is an individual integer. This function reads all the integers and combines them into an array that represents the waveform in number form.

The top image shows the waveform of a song. The bottom image shows a very zoomed in version of that waveform. Each blue dot is a sample and can be represented by an integer. The blue dots are then converted to array form, which the program can further manipulate.



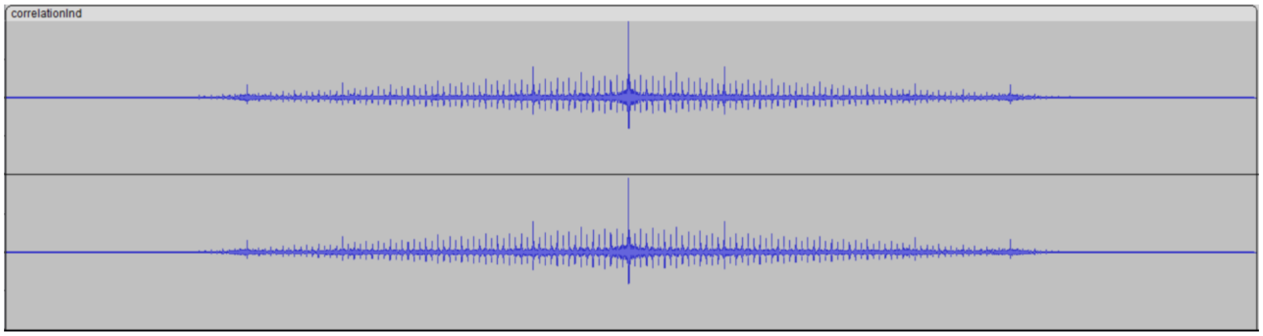


$$array = [x_0 \quad x_1 \quad x_2 \quad x_3 \quad \dots \quad x_{N-1}]$$

The function will also return the sample rate (number of blue dots per second, which in this case is 48000) and number of channels (in this case, 2). It will throw an error if the file does not exist, is not in .WAV format, or does not use 16-bit depth.

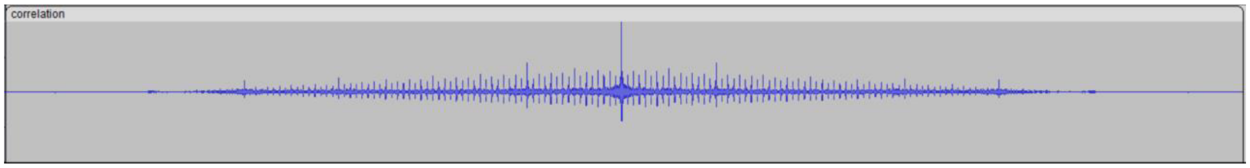
alignWavFiles. Sometimes, songs and instrumental versions aren't synchronized or the same length. This function aligns the song and instrumental waveforms and pads them with silence so that they're the same length.

The lag between the song and the instrumental can be calculated with cross-correlation. The waveforms are lined up when the value in the cross-correlation matrix is highest. The full explanation for this process is out of the scope of this report, but this video illustrates it quite well. The correlation matrix is performed separately for each channel. For example, if the song has 2 channels (left and right), the program will calculate the cross-correlation between the song's left channel and the instrumental's left channel, and then repeat that process for the right channels.



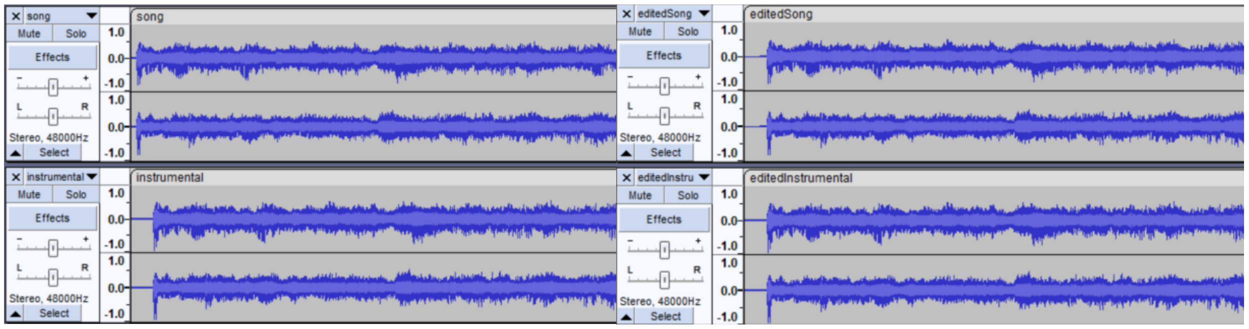
This is the correlation matrix for a 2-channel song and instrumental (the top represents the correlation matrix for the left channel, while the bottom represents the right channel). Usually, the matrixes will look similar, but they're not the same (hence why they're computed individually).

Then, all the correlation matrixes are added up, and that final correlation matrix is used to determine the lag between the song and instrumental waveforms. The program then compensates for this lag to align the waveforms.



Sum of the two correlation matrixes above. The peak indicates where the song and instrumental waveforms are lined up.

Once the waveforms are aligned, the function pads the files so that they're the same length.



Before (left) vs after (right). Note how the files are misaligned on the left, which is fixed by the function on the right.

lmsAdaptiveFilter. In many cases, the amplitude or phase of the song and instrumental may not match. To compensate, we can use a filter to linearly transform the instrumental until the error between the two sound waves is minimized. The filter I'm using is the least-mean squares (LMS) filter and is quite common for applications like this.

In simpler terms, instead of directly subtracting the instrumental from the song (which is prone to issues mentioned above), we are going to modify (filter) the instrumental to better match the original song, and *then* subtract it.

For a very in-depth explanation, here is exactly what the LMS filter is doing. The theory and derivation behind the filter is good to know but not necessary.

The program starts with the song waveform d and instrumental waveform x, both represented by 1D arrays of length N:

$$d = [d_0 \quad d_1 \quad d_2 \quad d_3 \quad \dots \quad d_{N-1}]$$

$$x = [x_0 \quad x_1 \quad x_2 \quad x_3 \quad \dots \quad x_{N-1}]$$

The program will also initialize an array for the filtered instrumental waveform y and vocal waveform e. These are also 1D arrays with length N, though they will be empty at first:

$$y = [y_0 \quad y_1 \quad y_2 \quad y_3 \quad \dots \quad y_{N-1}] = [0 \quad 0 \quad 0 \quad 0 \quad \dots \quad 0]$$

$$e = [e_0 \quad e_1 \quad e_2 \quad e_3 \quad \dots \quad x_{N-1}] = [0 \quad 0 \quad 0 \quad 0 \quad \dots \quad 0]$$

Finally, the program will initialize two vectors of size M: X and W. X is the input vector which contains the samples surrounding sample n (e.g. if $M = 5$, X would contain the two samples before sample n, sample n itself, and the 2 samples after sample n). During initialization when $n = 0$, the samples before sample n don't exist and therefore initialize to 0:

$$X = (x_{-(M-1)/2} \quad \dots \quad x_{n-2} \quad x_{n-1} \quad x_n \quad x_{n+1} \quad x_{n+2} \quad \dots \quad x_{(M-1)/2})^T$$

$$X = (0 \quad \dots \quad 0 \quad 0 \quad x_n \quad x_{n+1} \quad x_{n+2} \quad \dots \quad x_{(M-1)/2})^T$$

W is a weight vector, also with size M. It starts with a center value of 1:

$$W = (w_{-(M-1)/2} \quad \dots \quad w_{n-2} \quad w_{n-1} \quad w_n \quad w_{n+1} \quad w_{n+2} \quad \dots \quad w_{(M-1)/2})^T$$

$$W = (0 \quad \dots \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad \dots \quad 0)^T$$

The program then does the following for each sample n in the input signal x. First, it creates a filtered version of x_n by taking the dot product between the transformed weight vector and the input vector:

$$y_n = W^T{}_n \cdot X_n$$

Then, it subtracts y_n (filtered instrumental) from the composite signal d_n (song) to get the error e_n (vocal):

$$e_n = d_n - y_n$$

Finally, it updates the weights:

$$W_n = W_{n-1} + \mu \cdot e_n \cdot X_n$$

Then, the program moves on to the next sample ($n = n + 1$) and repeats this process. Over time, the weights will approach a value that minimizes the error.

μ is a constant value and must be small enough, or the weights will approach infinity. In theory, it must be smaller than 1/λ, where λ is the max eigenvalue of the autocorrelation matrix of the input signal. But in practice, this is really complicated to find so I just gradually reduce μ until the weights no longer approach infinity.

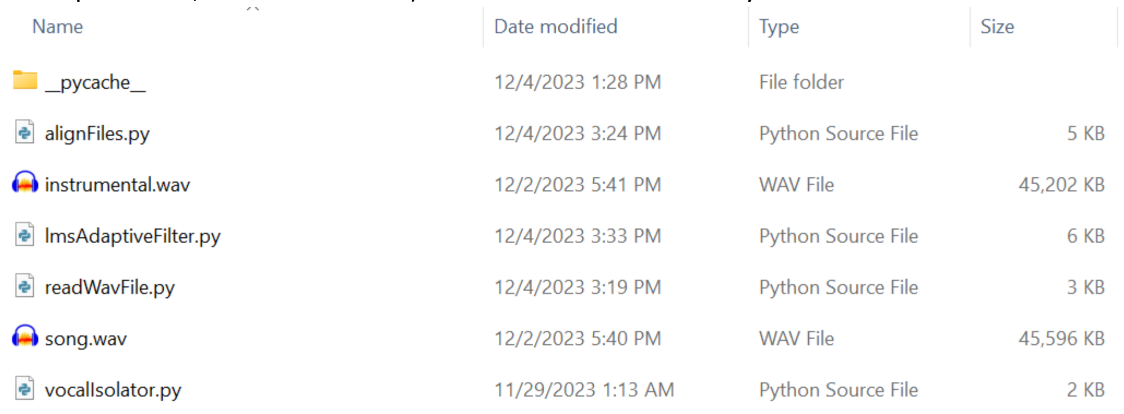
After doing this for every sample in the composite signal (song), the error signal array *e* will be full. It will contain the vocal waveform, which the program will write into a WAV file.

Much like the correlation calculations in alignWavFiles, this algorithm is performed separately on each channel. This is why the program can take some time to run – an average song could easily contain millions of samples, and all these calculations must be performed for *each* sample.

User Manual

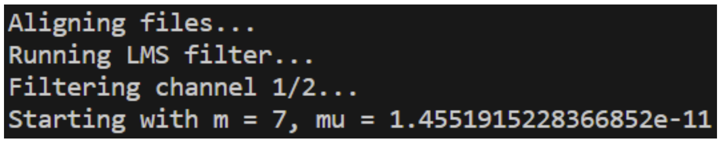
Instructions

1. Ensure that the song file, instrumental file, and all python files (readWavFile, alignFiles, lmsAdaptiveFilter, and vocalIsolator) are all in the same directory.



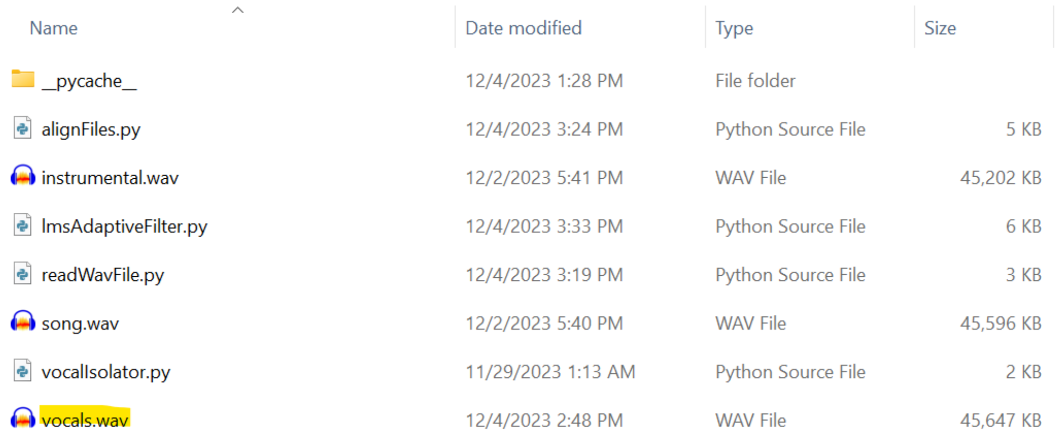
Name	Date modified	Type	Size
__pycache__	12/4/2023 1:28 PM	File folder	
alignFiles.py	12/4/2023 3:24 PM	Python Source File	5 KB
instrumental.wav	12/2/2023 5:41 PM	WAV File	45,202 KB
lmsAdaptiveFilter.py	12/4/2023 3:33 PM	Python Source File	6 KB
readWavFile.py	12/4/2023 3:19 PM	Python Source File	3 KB
song.wav	12/2/2023 5:40 PM	WAV File	45,596 KB
vocalIsolator.py	11/29/2023 1:13 AM	Python Source File	2 KB

2. Ensure that the song and instrumental files are in 16-bit WAV format with the same sample rate and number of channels. (Remember, this program *cannot* convert file formats; you will need 3rd party software for this. I recommend Audacity).
3. Run the main Python file (vocalIsolator).
4. Enter the name of the song file.
5. Enter the name of the instrumental file.
6. Allow the program to run (it may take a few minutes). If you see something like this, it means the program is working:



```
Aligning files...
Running LMS filter...
Filtering channel 1/2...
Starting with m = 7, mu = 1.4551915228366852e-11
```

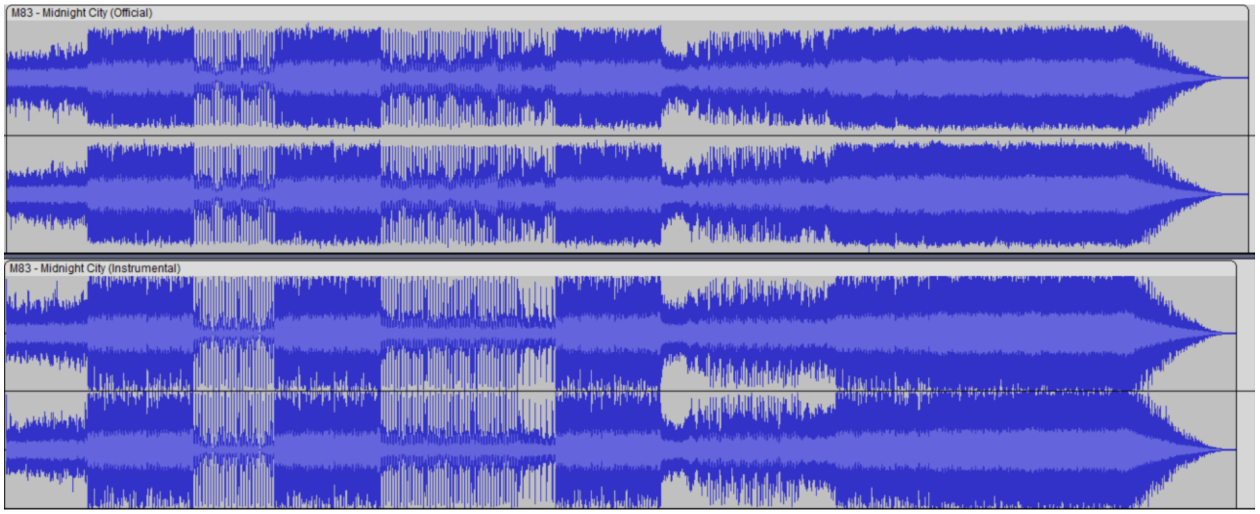
7. The program will print "Finished!" when it is done. The vocal file will be written into a 16-bit WAV file called "vocals.wav" in the same directory as all the other files.



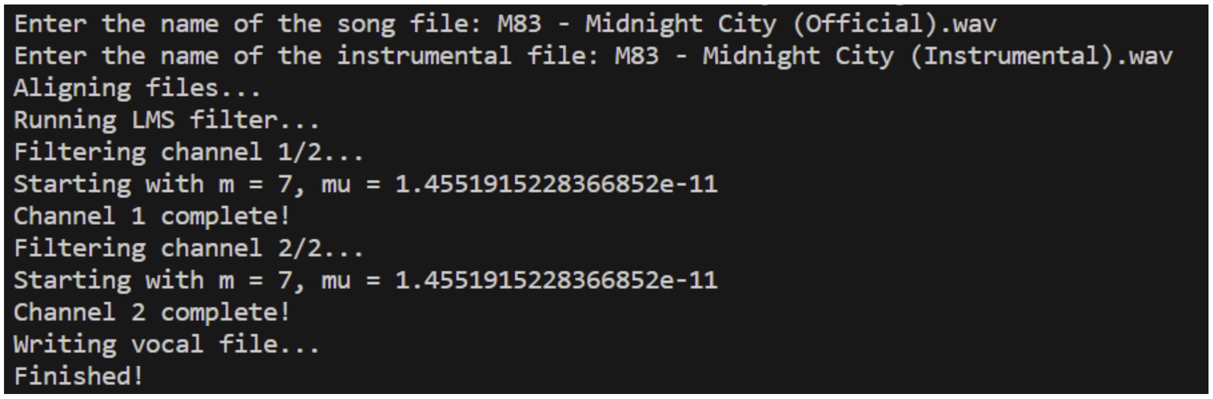
Name	Date modified	Type	Size
__pycache__	12/4/2023 1:28 PM	File folder	
alignFiles.py	12/4/2023 3:24 PM	Python Source File	5 KB
instrumental.wav	12/2/2023 5:41 PM	WAV File	45,202 KB
lmsAdaptiveFilter.py	12/4/2023 3:33 PM	Python Source File	6 KB
readWavFile.py	12/4/2023 3:19 PM	Python Source File	3 KB
song.wav	12/2/2023 5:40 PM	WAV File	45,596 KB
vocalIsolator.py	11/29/2023 1:13 AM	Python Source File	2 KB
vocals.wav	12/4/2023 2:48 PM	WAV File	45,647 KB

Sample Input

Here is a song file (top) with its instrumental file (bottom):

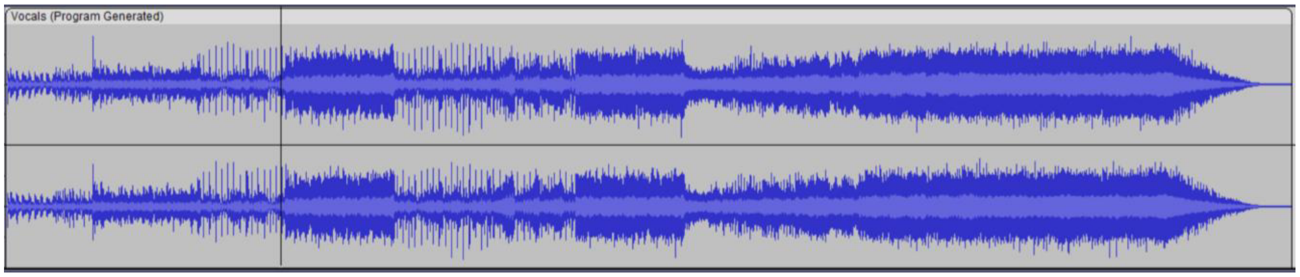

M83 - Midnight City (Official)
M83 - Midnight City (Instrumental)

We can then run the program and enter the names of the files as inputs:

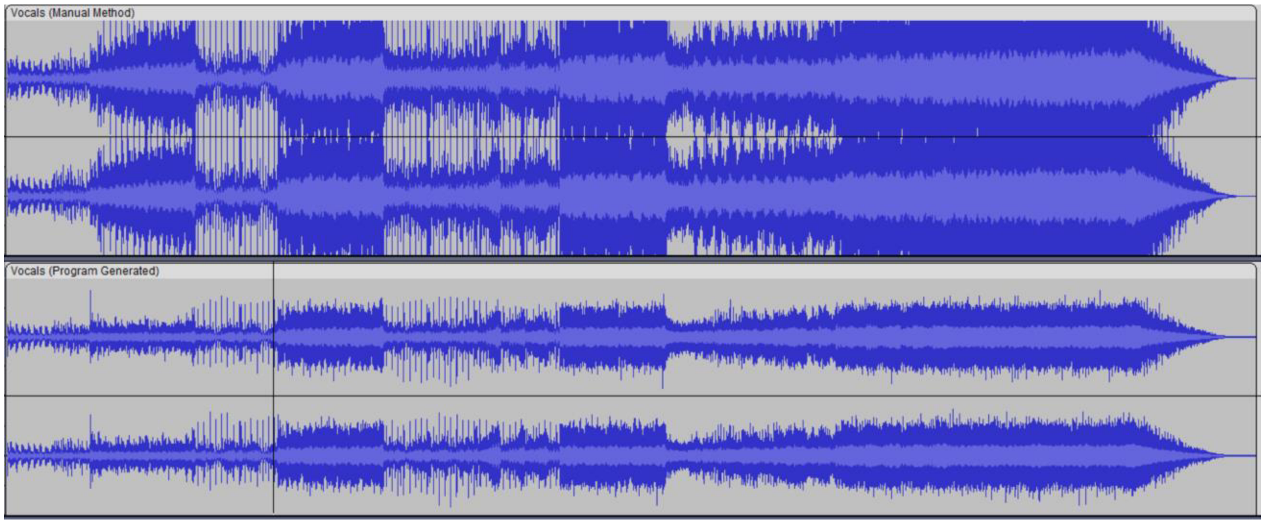


```
Enter the name of the song file: M83 - Midnight City (Official).wav
Enter the name of the instrumental file: M83 - Midnight City (Instrumental).wav
Aligning files...
Running LMS filter...
Filtering channel 1/2...
Starting with m = 7, mu = 1.4551915228366852e-11
Channel 1 complete!
Filtering channel 2/2...
Starting with m = 7, mu = 1.4551915228366852e-11
Channel 2 complete!
Writing vocal file...
Finished!
```

Here are the results:



For comparison, here's a picture of two vocal files. The top one is created by the "manual method," while the bottom one is created by the program. Notice how with the manual method, the vocals get louder and louder. This is because the cancellation starts off decent, but the songs get more misaligned as time goes on, leading to almost no cancellation at the end. In contrast, the program-generated vocals don't get any louder because the LMS filter keeps the instrumental aligned with the song, which results in more cancellation.


Vocals (Manual Method)
Vocals (Program Generated)