

# MATH 6340 Project

## - Audio Trimmer using Timestamps -

Marcos Saucedo  
*School of Mathematics and Statistics*  
*University of Texas Rio Grande Valley*  
*Edinburg, TX*  
*Email: marcos.saucedo01@utrgv.edu*

**Abstract**—Listening to music is one of the simplest forms of entertainment one can experience in life. Chances are you have a favorite album (digital or not). Or maybe you have a huge collection of music, but for some people it can be a lot of work collecting audio in its desired form. We all know those YouTube videos that play your favorite song but have that pesky ten seconds of silence at the beginning or the end of the video. So what do we do? We trim.

Let's take this a step further, you want to trim more than two songs from your album. Finding the spots where the song begins and ends can be annoying right, so you scroll down into the comments. Luckily, a person, possibly on the other side of the world knows this struggle, and posted the timestamps. Great! Now you can trim with some precision and save some minutes! Time passes and those two songs took you about 10 minutes to trim.

I have created a program that can trim an ALBUM for you in under 5 minutes! All I need is the song and the timestamps.

## 1. Introduction

In this paper we will use C++ and a C++ library named AudioFile by Adam Stark to read and write audio. My focuses will be on:

- 1) Text file extraction
- 2) Read/Writing Audio using the AudioFile library
- 3) Additional Functionalities

### 1.1. Text Extraction Code

I ran my code on CoCalc for the majority of the project using the C++ language. Below is a snippet of code for extracting and converting a timestamp into seconds. I will explain it.

The first five lines of code declare variables needed to start reading a text file along with the declaration of a vector.

- a `std::string` line that will be the container for every line in the txt file.
- a `fstream` object `fs`
- a vector of integers named `starting_points`
- using `open()` member function on `file_name`, which is the txt file.

Assuming the txt file did get opened as a line is being read these things will happen:

- print to the screen the line of text
- create an int variable named `where_to_start`
- count how many colons are in the line of text
- The number of colons will equal one if the timestamps are in M:SS format.
- The code finds where the colon is, converts the substrings 2 characters before and after the colon into seconds.
- The strings converted to seconds get summed and assigned to `where_to_start` and appended to the vector `starting_points`.

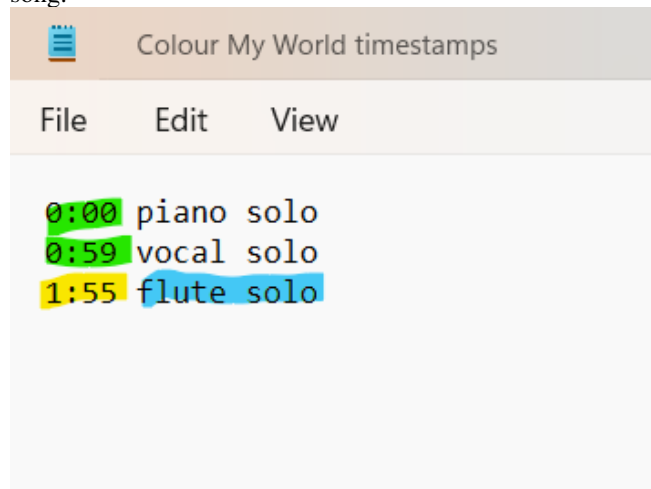
```
std::vector<int> txt_to_time(const std::string& file_name){

std::string line;
std::fstream fs;
std::vector<int> starting_points {};
fs.open(file_name);
if(fs.is_open()){
while( getline(fs, line) ){
    std::cout<<line<<'\n';
    int where_to_start {};
// have to count the number of colons in a line and perform the appropriate action
    size_t colon_count = std::count_if(line.begin(), line.end(), \
    [](const char& n){return n == ':';});
```

```
// have to locate the colon and parse
if(colon_count == 1){
    auto colon_index = line.find(':');
    where_to_start = std::stoi(line.substr(0,colon_index))*60 +\
    std::stoi(line.substr(colon_index+1,2));
    starting_points.push_back(where_to_start);
}
```

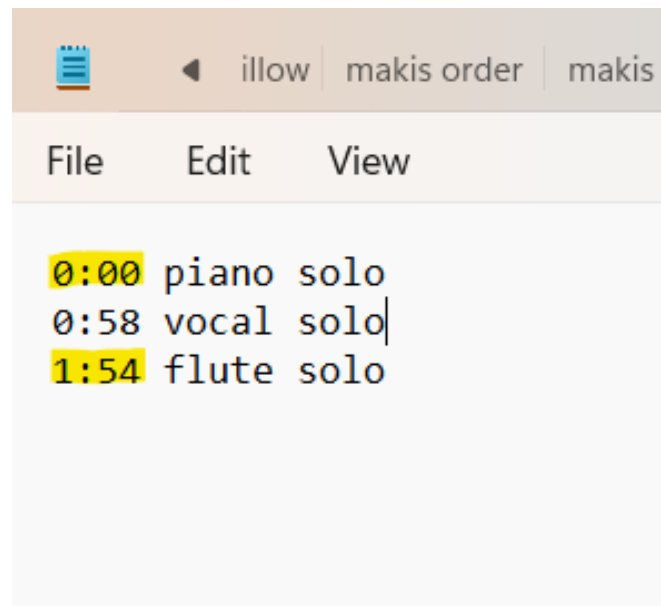
## 1.2. Text File Extraction

Let's talk about the song "Colour My World" by Chicago. If you've heard the song before you'd know it has three solos. Each approximately a minute long. This is a screenshot of a txt file containing the timestamps for the song.

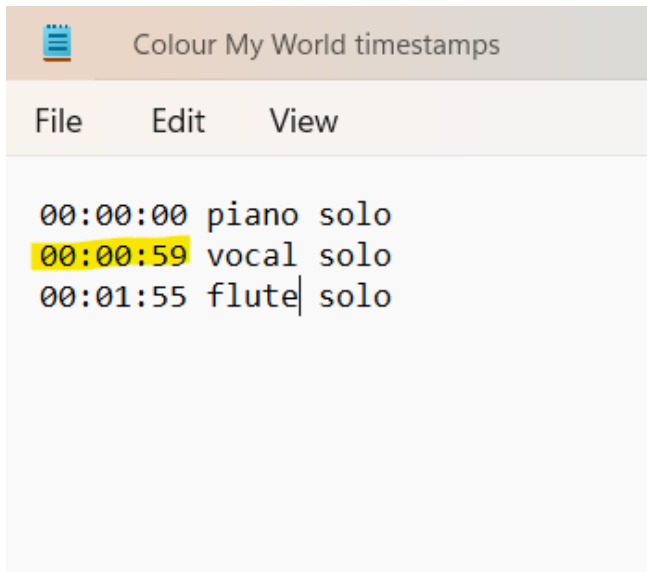


It doesn't look like much but actually you can get a lot of information from these three lines of text. This is the list of things to extract:

- Highlighted in green: Subtract these timestamps for the duration of piano solo.
- Highlighted in yellow is the timestamp for the beginning of the flute solo.
- Highlighted in blue is the name of the song that'll be created by the trimmer.



The highlighted timestamps are in minute:second format (M:SS). It's a preference whether the person who made the timestamps uses MM:SS format or M:SS format for the first ten minutes of timestamps. This example has format M:SS, like 0:00 instead of 00:00 and this is the most common way you'll see it. The program can handle both cases.



Using HH:MM:SS format for the timestamps is also supported.

### 1.3. Using the AudioFile library

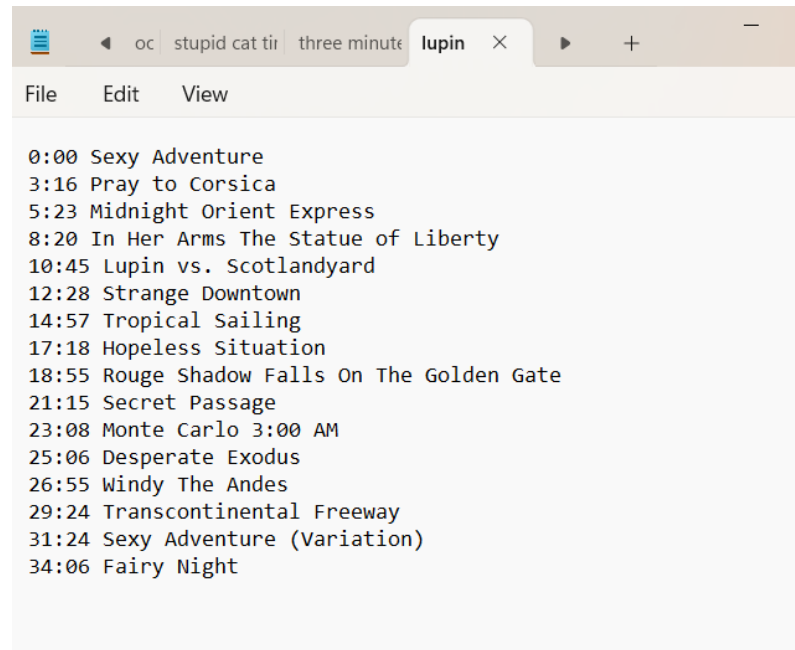
I used a C++ library created by Adam Stark named [AudioFile](#). It allows the user to read/write to audio files provided it is a WAV or AIFF. Here is a snippet of the sample code I manipulated to read my first audio file in C++. This is the final product.

```
if (sample_load == 1){

    // getting some stats about the wav file , wav is uncompressed audio
    song_to_sample.printSummary();
    // hoping to record audio up to a certain point and explore , achieved 11/4/23
    int channel = 0;
    int numSamples = song_to_sample.getNumSamplesPerChannel();
    float sampleRate = 44100.f; //sample rate default , dvd quality audio
    float frequency = 440.f;
    double length_in_seconds = song_to_sample.getLengthInSeconds();
    int numChannels = song_to_sample.getNumChannels();
    int numSamplesPerChannel = song_to_sample.getNumSamplesPerChannel() ...
        * ((seconds)/length_in_seconds); //magic formula for X seconds of playback

    int start = song_to_sample.getNumSamplesPerChannel() ...
        * ((starting_timestamp)/length_in_seconds);
    AudioFile<double>::AudioBuffer buffer;

    // Set number of samples per channel
    buffer.resize (2); // stereo audio
    buffer[0].resize (numSamplesPerChannel); // two channels to fill each has size of
        // 'track' in terms of samples
    buffer[1].resize (numSamplesPerChannel); // resize the buffer to the duration
        //you want to record
    //duration of recording == numSamplesPerChannel
    for (int i =0; i < numSamplesPerChannel; i++){
```



This is an album written down as timestamps. Yes, the program can give you all the songs.

```

        for (int channel = 0; channel < numChannels; channel++){
            song_to_sample.samples[channel][i] = song_to_sample.samples[channel][i+start];
        }
    }

    for (int i = 0; i < numSamplesPerChannel; i++){
        float sample = song_to_sample.samples[channel][i+start];

        for (int channel = 0; channel < numChannels; channel++){
            buffer[channel][i] = sample;
        }
    }

    // put into the AudioFile object
    // song_to_sample.setAudioBuffer (buffer);

    // Resize the audio buffer
    // Set both the number of channels and number of samples per channel
    song_to_sample.setAudioBufferSize (numChannels, numSamplesPerChannel);
    // Set the number of samples per channel
    song_to_sample.setNumSamplesPerChannel (numSamplesPerChannel);
    // Set the number of channels
    song_to_sample.setNumChannels (numChannels);
    // Set bit depth and sample rate
    song_to_sample.setBitDepth (16);
    song_to_sample.setSampleRate (44100); // halves the length of the audio and highers pitch

    // Save the audio file to disk
    song_to_sample.save (track_name + ".wav", AudioFileFormat::Wave);
}
else {std::cout << "File did not load." << '\n';}

```

## 1.4. trim\_audio function

The code above belongs to the function trim\_audio(). The function accepts arguments:

- int seconds
- int starting\_timestamps
- const std::string& track\_name
- AudioFile<double> song\_to\_sample
- bool sample\_load

This code is the major step of my program. This function is responsible for creating one buffer of audio using information you've extracted from the txt file! I will talk about the five arguments and how they are used.

The variable song\_to\_sample is the AudioFile object needed to perform actions on the WAV file (like loading and accessing samples). It's used very often for these purposes and more. The AudioFile object has options for storing the samples of the WAV file as doubles, ints, floats, etc... This variable stores the samples as doubles.

The variable sample\_load is a bool that is created before the call of trim\_audio(). It was created as a solution to loading the song only once.

The variable track\_name does exactly what you think it does. It tells trim\_audio() what to name the output WAV file.

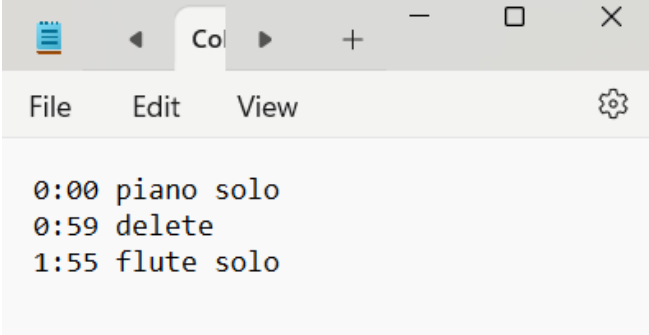
The variable named starting\_timestamp tells trim\_audio() where to start the buffer. Starting\_timestamps should be an integer greater than zero and represents the location in the audio file in terms of seconds. In the code it gets converted to an integer named start. The variable start is obtained by multiplying a fraction of seconds and the number of samples the song contains. I'm glad this worked, because I was expecting hexadecimal math or something!

The variable named seconds follows the same procedure as starting\_timestamps but instead of using it as a starting point, it tells trim\_audio() how long the playback of the WAV file is.

You will get stereo audio (left ear and right ear), and the audio quality will be CD quality (1980's CD quality, but still gold)

## 1.5. Additional Functionalities

My program can also omit but really ignore audio you don't want to preserve from the WAV file. Here's an example of txt file using the delete key word.



```
0:00 piano solo
0:59 delete
1:55 flute solo
```

This txt file tells my program to only create piano solo.wav and flute solo.wav. Everything from 0:59-1:55 will not be created and skipped.

## 1.6. Limitations and Conclusion

This ends my small talk about my audio trimmer. I've shown the simplest use (trimming one song). Its true power lies in separating ALBUMS, big WAV files, and giving you freedom to preserve what audio you want. Here is a list of limitations I've found through stress testing:

- 1) Only WAV and AIFF files are supported.
- 2) There is a limit to file size. It is somewhere between 1.39GB and 4.3GB.
- 3) My program cannot merge audio, only separate.
- 4) There is no 'universal' style of timestamps. You will get a warning message if the structure differs from my examples.

## 1.7. Additional Resources (for mp3 users)

In case you want to make quick conversions from WAV to mp3 and vice versa, I will leave some links in the references section. [FFmpeg guide](#) will open a tutorial to install ffmpeg for Windows. [VLC guide](#) will take you to a small guide on how to use VLC, which I found useful for converting multiple files. Use these so you never run out of credits on those free conversion websites.

## 1.8. References

- <https://github.com/adamstark/AudioFile>
- [txt file reading](#)
- [VLC guide](#)
- [VLC](#)
- [FFmpeg](#)
- [FFmpeg guide](#)
- [my github](#)