

Audio Overlay Filter

Name : Mayukhmali Das

Degree: Bachelor of Engineering

Discipline: Electronics and Telecommunications

Year of College: 4th(Final)

College Name: Jadavpur University

Country: India

Website: <https://smartmayukh.github.io/Personal-site/>

Github: <https://github.com/Smartmayukh>

Quick Navigation :

1. [User options for C implementation of Overlay Filter](#)
2. [Flowchart for C implementation of Overlay Filter](#)
3. [Complete C implementation using FFMPEG and system function](#)

User Options

I plan to introduce the following options for the users

1. Position
2. Loop_Time
3. Gain_of_Base
4. Gain_of_Overlay
5. Silent

Auxiliary features

6. Fade in Fade out options for the Overlay
7. Crossfade
8. Time stretch
9. Best Sync Point

Explanation of the features :

Main features

Position : This will be a number in milliseconds where the user wants to insert the overlay file.

Loop_Time : Number of times the overlay will be used. If the net length of overlay exceeds the base audio length, the overlay will be trimmed to match the base audio length.

Gain_of_Base : Gain of the base audio.





Gain_of_Overlay: Gain of the overlay audio.

Silent : Make base audio silent while overlaying.

Auxiliary features that can be added :

Fade in Fade out:

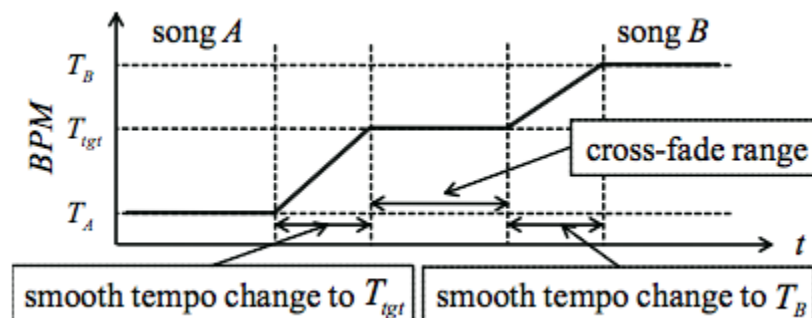
FFmpeg's afade filter is capable of applying fade-in and fade-out effects to input audio streams. Additionally, it offers several options to modify the volume of the audio during the fade-in effect as shown below. We can incorporate this afade filter to give a fading in/out effect to our overlay filter.

afade argument	Meaning	Curve
tri	triangular, linear slope (default)	
qsin	quarter of sine wave	
hsin	half of sine wave	
esin	exponential sine wave	

Crossfade :

If we are using the silent option to mute the base audio while overlaying we can add a crossfade for a smoother transition from the base audio to the overlaying audio. FFmpeg has a acrossfade filter. We can also create our own filter.

We can use a technique called Optimal Tempo Adjustment which automatically generates a smooth Song to Song transition



This algorithm of a Mixing System with Optimal Tempo Adjustment is given in the following paper

<https://ismir2009.ismir.net/proceedings/PS1-14.pdf>

Also, when two songs are played back-to-back and the strong beats are adjusted to match the weaker beat of the other song, it can create a jarring listening experience. To avoid this problem, a cross-fade technique is proposed in the above paper, which involves computing the cross-correlation of the beats of the songs within the crossfade range. This helps to seamlessly blend the two songs together, without causing any discomfort to the listener.

Time stretching:

Time stretching refers to the technique of altering the speed or length of an audio signal, while maintaining its original pitch. We can incorporate FFmpeg atempo filter for this purpose.

Best Sync Point:

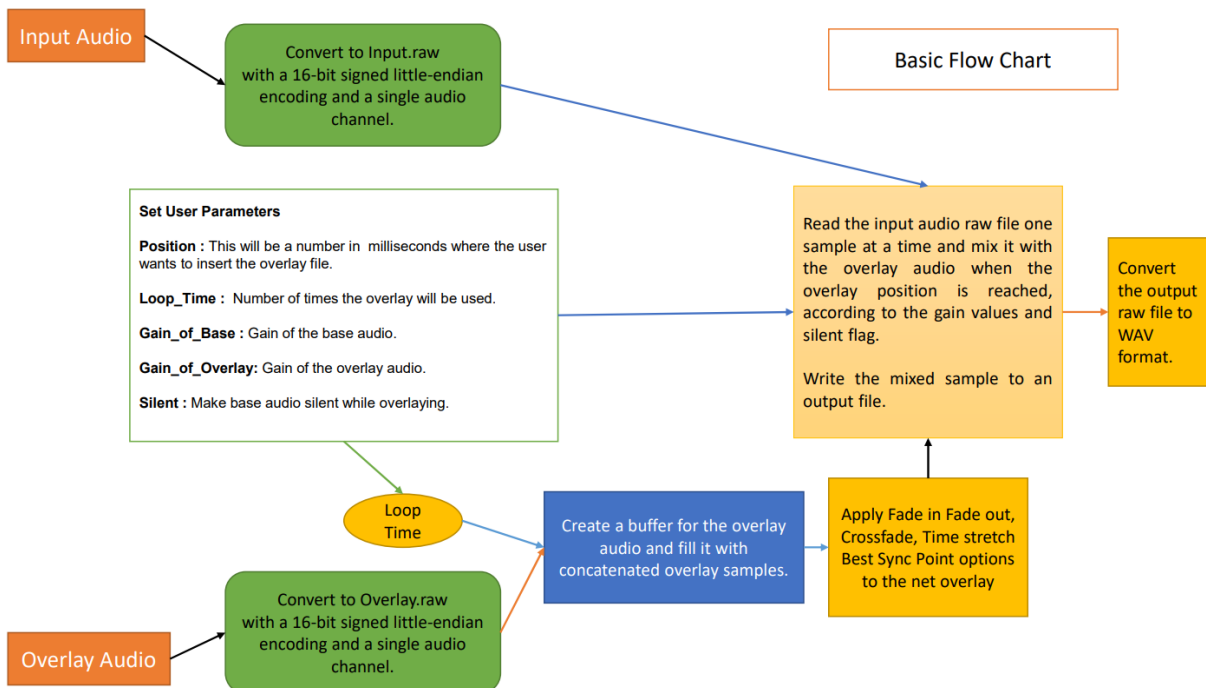
We can mix the two audio files, the base and the overlay, based on their beats. To achieve this, we can utilize a function that determines the **optimal synchronization point** between the two audio files before mixing them. This function will take the beat samples of both the top and bottom files as well as the maximum sample to be mixed.

To calculate the best synchronization point, the function will slide the **beat samples of the overlay file** over the **beat samples of the base file**, while measuring the number of matches between the two files within a given offset.

The algorithm will be like this. First we will initialize a variable called "best_sync_point" variable to zero, which represents the most favorable synchronization point found so far. We will then iterate through each possible synchronization point in the received signal, calculating the correlation coefficient between the reference signal and the received signal at that point. The correlation coefficient determines the similarity between the two signals at a particular time offset.

In summary, this algorithm functions by comparing the correlation coefficient between the reference signal and the received signal at every possible synchronization point. It then selects the synchronization point that maximizes the correlation coefficient as the optimal synchronization point.

Flowchart for the C implementation of the overlay Filter



C implementation using FFMPEG

Let us step by step understand the C code that I have written

Including all necessary libraries :

```
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include <stdlib.h>
```

Convert input files to raw format using FFmpeg and system function :

```
int main() {
    // Convert input files to raw format using FFmpeg
    int ret = system("ffmpeg -y -i base_audio.wav -f s16le -ac 1 input.raw");
    if (ret != 0) {
        printf("Error converting input file.\n");
        return 1;
    }
    ret = system("ffmpeg -y -i overlay_audio.wav -f s16le -ac 1 overlay.raw");
    if (ret != 0) {
        printf("Error converting overlay file.\n");
        return 1;
    }
}
```

The `system()` function is a standard library function in C that allows us to execute a command as if it were entered on the command line. In case of the base audio, the command being executed is `"ffmpeg -y -i base_audio.wav -f s16le -ac 1 input.raw"`. This command tells FFmpeg to read in the file `"base_audio.wav"`, and then convert it to a raw audio file format with a 16-bit signed little-endian encoding and a single audio channel. Same is with the overlay audio.

Opening the Input and Output raw files created in the previous step :

```
// Open input and output files
FILE *input_file = fopen("input.raw", "rb");
if (!input_file) {
    printf("Error opening input file.\n");
    return 1;
}
FILE *overlay_file = fopen("overlay.raw", "rb");
if (!overlay_file) {
    printf("Error opening overlay file.\n");
    return 1;
}
FILE *output_file = fopen("output.raw", "wb");
if (!output_file) {
    printf("Error opening output file.\n");
    return 1;
}
```

Additionally an output.raw file is also created and opened in “write binary” mode to write down the modified output.

Setting all the parameters for overlaying :

```
// Set position and loop time

int position = 5130; // position in milliseconds
int loop_time = 5; // number of times to loop the overlay


// Set gain values
double gain_of_base = 0.2; // gain of base audio
double gain_of_overlay = 0.6; // gain of overlay audio
```

```
// Set silent flag  
int silent = 0; // make base audio silent while overlaying
```

Position : This will be a number in milliseconds where the user wants to insert the overlay file. Here it is 5130 milliseconds.

Loop_Time : Number of times the overlay will be used. Here it is looped five times.

Gain_of_Base : Gain of the base audio. Set to 0.2 to lower its volume.

Gain_of_Overlay: Gain of the overlay audio. Set to 0.6.

Silent : Make base audio silent while overlaying. Set to 0 to show that mixing is actually happening between the base and the overlay audio files.

Converting the time position to position in samples for overlaying :

```
// Calculate position in samples  
int sample_rate = 44100; // assume sample rate of 44100 Hz  
int position_samples = (int)(position / 1000.0 * sample_rate);
```

To convert this time position to a position in samples, we multiply the time position by the sample rate and divide by 1000.

Creation of buffer using malloc to store the net overlay audio after looping :

```
// Create a buffer for the net overlay audio
fseek(overlay_file, 0, SEEK_END);
long overlay_size = ftell(overlay_file);
fseek(overlay_file, 0, SEEK_SET);
int16_t *overlay_buffer = malloc(overlay_size * loop_time);
```

The first line of the code uses the `fseek()` function to move the file position indicator to the end of the `overlay_file`. This is done so that we can determine the size of the file by calling the `ftell()` function, which returns the current position of the file pointer.

The second line of the code sets the file position indicator back to the beginning of the file using the `fseek()` function with the `SEEK_SET` argument, so that the file can be read from the beginning.

The third line of the code allocates memory for the overlay buffer using the `malloc()` function, which dynamically allocates a block of memory of size `overlay_size * loop_time` bytes. The `int16_t` type is used to ensure that the buffer contains 16-bit signed integers, which is the most common format for audio data.

Overlay Samples are then concatenated to fill the buffer :

```
// Fill the buffer with concatenated overlay samples
for (int i = 0; i < loop_time; i++) {
    fread(overlay_buffer + i * (overlay_size / sizeof(int16_t)),
sizeof(int16_t), overlay_size / sizeof(int16_t), overlay_file);
    fseek(overlay_file, 0, SEEK_SET);
}
```

Mixing of the base and overlay audio files :

```
int16_t input_sample, output_sample;
int count;
int n = 0; // sample counter
int overlay_index = 0; // index for overlay buffer

while(1) {
    count = fread(&input_sample, 2, 1, input_file); // read one 2-byte sample
    from input file
    if (count != 1) break;

    if (n >= position_samples && overlay_index < overlay_size * loop_time /
    sizeof(int16_t)) { // if position is reached and overlay buffer is not exhausted
        if (silent) { // make base audio silent while overlaying
            output_sample = (int16_t)(overlay_buffer[overlay_index] *
            gain_of_overlay);
        } else { // mix the two samples together with gain applied
            output_sample = (int16_t)(input_sample * gain_of_base +
            overlay_buffer[overlay_index] * gain_of_overlay) / 2;
        }
        overlay_index++; // increment overlay buffer index
    } else { // use input sample only
        output_sample = (int16_t)(input_sample);
    }

    fwrite(&output_sample, 2, 1, output_file);
    n++; // increment sample counter
}
```

The code enters into a loop that reads one sample at a time from the input file, modifies it with the overlay signal if the sample position is reached, and writes the resulting sample to the output file. The loop continues until there are no more samples to read from the input file.

In each iteration of the loop:

```
while(1) {  
    count = fread(&input_sample, 2, 1, input_file); // read one 2-byte sample  
    from input file  
    if (count != 1) break;
```

1. We start by reading one 2-byte sample from the input file using the `fread()` function and store it in the variable `input_sample`. The code reads one 2-byte sample because the audio samples are stored as 16-bit signed integers, which take up 2 bytes of memory. The `count` variable is set to the number of elements successfully read by `fread()`, which should be 1 if a sample was successfully read. If `count` is not equal to 1, then the `break` statement is executed, which terminates the loop. This is necessary because if there are no more samples to read from the input file, the program should exit the loop and stop processing audio.

```
> if (n >= position_samples && overlay_index < overlay_size * loop_time / sizeof(int16_t)) {  
    } else { // use input sample only  
        output_sample = (int16_t)(input_sample);  
    }
```

2. The code then checks if the sample position is reached and if the overlay buffer is not exhausted. If both conditions are met, the code modifies the sample by overlaying the corresponding sample from the `overlay_buffer` onto the input sample with the specified gains. The code then writes the resulting sample to the `output_file` using the `fwrite()` function. If this condition is not satisfied the input sample is written to the output sample.

```

if (n >= position_samples && overlay_index < overlay_size * loop_time /
sizeof(int16_t)) { // if position is reached and overlay buffer is not exhausted
    if (silent) { // make base audio silent while overlaying
        output_sample = (int16_t)(overlay_buffer[overlay_index] *
gain_of_overlay);
    } else { // mix the two samples together with gain applied
        output_sample = (int16_t)(input_sample * gain_of_base +
overlay_buffer[overlay_index] * gain_of_overlay) / 2;
    }
    overlay_index++; // increment overlay buffer index
} else { // use input sample only
    output_sample = (int16_t)(input_sample);
}
}

```

3. The if statement checks whether the current sample position (n) is greater than or equal to the position in samples to overlay (position_samples), and whether the overlay buffer still has data available (overlay_index < overlay_size * loop_time / sizeof(int16_t)).

If both conditions are true, the audio data at the current position of the input file is mixed with the audio data from the overlay buffer, with the gain of each audio source applied (gain_of_base and gain_of_overlay, respectively).

If the silent flag is set, the output sample is simply equal to the audio data from the overlay buffer multiplied by the gain_of_overlay.

If the silent flag is not set, the audio data from both sources is mixed and divided by 2 to avoid clipping, and then cast to an int16_t data type and stored in the output_sample variable.

The overlay_index variable is incremented to move to the next audio sample in the overlay buffer.

If the conditions of the if statement are not met, the output sample is simply set to the input sample, with no modification applied.

```
fwrite(&output_sample, 2, 1, output_file);  
    n++; // increment sample counter
```

4. `fwrite(&output_sample, 2, 1, output_file)` writes one 2-byte sample to the output file. `n++` increments the current sample counter, which keeps track of how many samples have been processed.

Closing the I/O files and freeing the overlay buffer

```
// Close input and output files  
fclose(input_file);  
fclose(overlay_file);  
fclose(output_file);  
  
// Free the overlay buffer  
free(overlay_buffer);
```

Finally converting the output raw file to out.wav file

```
// Convert output file to WAV format using FFmpeg  
ret = system("ffmpeg -y -f s16le -ar 44100 -ac 1 -i output.raw out.wav");  
if (ret != 0) {  
    printf("Error converting output file.\n");  
    return 1;  
}  
  
return 0;
```

Here we convert the output file from raw PCM format to the WAV format. The conversion is done using the `system()` function, which executes the FFmpeg command specified in the argument string. The command includes various parameters, such as the output file format (s16le for 16-bit signed PCM), sample rate (44100 Hz), and number of audio channels (1 for mono).

If the conversion is successful, the `system()` function will return 0, indicating that the conversion was completed without errors. If there is an error, such as if FFmpeg is not installed or if the output file cannot be created, the function will return a non-zero value. In this case, if the return value is non-zero, the code prints an error message and exits the program with a return value of 1, indicating that an error occurred during the conversion. Otherwise, the function returns 0, indicating that the conversion was successful and the output file has been created in WAV format.

Some example outputs :

Audio Censoring :

```
// Set position and loop time

int position = 5130; // position in milliseconds
int loop_time = 5; // number of times to loop the overlay

// Set gain values
double gain_of_base = 0.2; // gain of base audio
double gain_of_overlay = 0.6; // gain of overlay audio

// Set silent flag
int silent = 0; // make base audio silent while overlaying
```

Base Audio

https://drive.google.com/file/d/1A_V6ycl4DMyGEyzNZmv60ZxUZObwIUil/view?usp=sharing

Overlay Audio

https://drive.google.com/file/d/1bIOtMAF08ZCaXR0dIJiqWXnVMI_yipFm/view?usp=share_link

Overlaid Audio (loop the overlay file 5 times starting at 5130 ms)

https://drive.google.com/file/d/1_UL01T-zlBFpRIwxDI6NLVC9RP-9GDeR/view?usp=sharing

Audio Mashup/ Remix :

```
// Set position and loop time
int position = 61700; // position in milliseconds
int loop_time = 1; // number of times to loop the overlay

// Set gain values
double gain_of_base = 0.2; // gain of base audio
double gain_of_overlay = 1; // gain of overlay audio

// Set silent flag
int silent = 1; // make base audio silent while overlaying
```

For mashup the base audio is made silent while overlaying. I made a mash up of a English song “ Let Me Down Slowly” by Alec Benjamin and an Indian Hindi Song “ Jiske Aane Se Mukammal - Arijit Singh “

Base Audio

<https://drive.google.com/file/d/1O9YpCH7OhEhAc1Pwi230FLS6gliHTpCJ/view?usp=sharing>

Overlay Audio

https://drive.google.com/file/d/1LoxOImZ8lDAzNzbiV-_5b4LqNV1SaqFT/view?usp=sharing

Overlaid Audio (loop the overlay file 1 time keeping the base silent starting at 61700 ms)

<https://drive.google.com/file/d/1DVUYQqHjpfqB-L0x9itsYl3O66FkpyNH/view?usp=sharing>

Audio Mixing :

```
int ret = system("ffmpeg -y -i base_audio.wav -f s16le -ar 44100 -ac 1
input.raw");
    if (ret != 0) {
        printf("Error converting input file.\n");
        return 1;
    }
    ret = system("ffmpeg -y -i overlay_audio.wav -f s16le -ar 44100 -ac 1
overlay.raw");
    if (ret != 0) {
        printf("Error converting overlay file.\n");
        return 1;
    }
}
```

The audio files had different sampling rates so I first converted both to 44100 Hz.

```
// Set position and loop time
int position = 0; // position in milliseconds
int loop_time = 1; // number of times to loop the overlay

// Set gain values
double gain_of_base = 0.15; // gain of base audio
double gain_of_overlay = 1; // gain of overlay audio
```

Also the sound of the mixed audio was coming out to be less so I removed the clipping prevention (dividing by 2)

```
// mix the two samples together with gain applied
        output_sample = (int16_t)(input_sample * gain_of_base +
overlay_buffer[overlay_index] * gain_of_overlay)/2 ;
```

With

```
// mix the two samples together with gain applied
        output_sample = (int16_t)(input_sample * gain_of_base +
overlay_buffer[overlay_index] * gain_of_overlay) ;
```

For mixing, I used “ What a Wonderful World” by Louis Armstrong and “ Mozart – Sonata No. 13 In B Flat Major, K.333 – II. Andante Cantabile”

Base Audio

https://drive.google.com/file/d/1t5HUQJNhYNh6XasatrPOZsWC4IU774mG/view?usp=share_link

Overlay Audio

https://drive.google.com/file/d/1TiBwH_jiLxneLCd045LMmsxR7gFVCiqK/view?usp=share_link

Overlaid Audio (loop the overlay file 1 time keeping the base audible starting at 0 ms)

https://drive.google.com/file/d/1d2zF4Fyj3LG0Xm_2pEUb60DJopcZCpmP/view?usp=sharing

Complete code :

```
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include <stdlib.h>

int main() {
    // Convert input files to raw format using FFmpeg
    int ret = system("ffmpeg -y -i base_audio.wav -f s16le -ac 1 input.raw");
    if (ret != 0) {
        printf("Error converting input file.\n");
        return 1;
    }
    ret = system("ffmpeg -y -i overlay_audio.wav -f s16le -ac 1 overlay.raw");
    if (ret != 0) {
        printf("Error converting overlay file.\n");
        return 1;
    }

    // Open input and output files
    FILE *input_file = fopen("input.raw", "rb");
    if (!input_file) {
        printf("Error opening input file.\n");
        return 1;
    }
    FILE *overlay_file = fopen("overlay.raw", "rb");
    if (!overlay_file) {
        printf("Error opening overlay file.\n");
        return 1;
    }
    FILE *output_file = fopen("output.raw", "wb");
    if (!output_file) {
        printf("Error opening output file.\n");
        return 1;
    }
}
```

```

// Set position and loop time
int position = 5130; // position in milliseconds
int loop_time = 5; // number of times to loop the overlay

// Set gain values
double gain_of_base = 0.2; // gain of base audio
double gain_of_overlay = 0.6; // gain of overlay audio

// Set silent flag
int silent = 0; // make base audio silent while overlaying

// Calculate position in samples
int sample_rate = 44100; // assume sample rate of 44100 Hz
int position_samples = (int)(position / 1000.0 * sample_rate);

// Create a buffer for the net overlay audio
fseek(overlay_file, 0, SEEK_END);
long overlay_size = ftell(overlay_file);
fseek(overlay_file, 0, SEEK_SET);
int16_t *overlay_buffer = malloc(overlay_size * loop_time);

// Fill the buffer with concatenated overlay samples
for (int i = 0; i < loop_time; i++) {
    fread(overlay_buffer + i * (overlay_size / sizeof(int16_t)),
sizeof(int16_t), overlay_size / sizeof(int16_t), overlay_file);
    fseek(overlay_file, 0, SEEK_SET);
}

```

```

int16_t input_sample, output_sample;
int count;
int n = 0; // sample counter
int overlay_index = 0; // index for overlay buffer

while(1) {
    count = fread(&input_sample, 2, 1, input_file); // read one 2-byte sample
from input file
    if (count != 1) break;

    if (n >= position_samples && overlay_index < overlay_size * loop_time /
sizeof(int16_t)) { // if position is reached and overlay buffer is not exhausted
        if (silent) { // make base audio silent while overlaying
            output_sample = (int16_t)(overlay_buffer[overlay_index] *
gain_of_overlay);
        } else { // mix the two samples together with gain applied
            output_sample = (int16_t)(input_sample * gain_of_base +
overlay_buffer[overlay_index] * gain_of_overlay) / 2;
        }
        overlay_index++; // increment overlay buffer index
    } else { // use input sample only
        output_sample = (int16_t)(input_sample);
    }

    fwrite(&output_sample, 2, 1, output_file);
    n++; // increment sample counter
}

// Close input and output files
fclose(input_file);
fclose(overlay_file);
fclose(output_file);

```

```
// Free the overlay buffer
free(overlay_buffer);

// Convert output file to WAV format using FFmpeg
ret = system("ffmpeg -y -f s16le -ar 44100 -ac 1 -i output.raw out.wav");
if (ret != 0) {
    printf("Error converting output file.\n");
    return 1;
}

return 0;
}
```

Note : Using the overlay buffer has a downside in space complexity but it also gives us the opportunity to apply fade-in/fade-out, crossfade, etc. much more easily.