ECE420 Final Project Report
A Pitch corrector with integrated Voice/BGM seperation

Ethan Zhou, Eric Tang

{yz69, leweit2} @ illinois.edu

Contents

I. Introductions/Literature Review

Auto-tune technology is used to correct or manipulate pitch in vocal and instrumental music recordings and performances. The primary function of Auto-Tune is to make pitch correction automatic and imperceptible. The role of pitch correction in Auto-Tune involves automatically detecting the pitch of a vocal performance and altering any out-of-tune notes to the target pitch. This is done by shifting the pitch of the sound without affecting other aspects of its audio profile, which can greatly enhance performance by making it sound more accurate and harmonically correct.

Our design goal is to build an Android app to perform the Pitch-correction function for user inputs to target music. The music can be recorded in real-time or pre-processed. This app is designed to perform voice/background separation for the music, spectro-analysis for the music, and pitch correction for the user input. The algorithms used will be implemented from the ground up.

While there are novel and high-performance algorithms and approaches that utilize neural network [1], similarly we have high-performance background separation algorithms and approaches that can detect specific voices [2]. For our purpose, latency, computational resources, and implementation complexity are prioritized while delivering results of sufficient quality for non-professional standards. Therefore, we utilized algorithms described in Rafii's paper [3] for background/voice separation, which identifies a repeating segment of a song via spectro-analysis and creates a soft mask to separate the voice segment and background segment. We then designed a custom algorithm for pitch correction that utilizes low-latency algorithms like TD-PSOLA for pitch synthesis and Auto-correlation for pitch analysis.

Due to the complexity of our program, a multi-screen approach for app design is selected. Data passage between Java activities has proven to be extremely memory inefficient considering the length for typical audio data, thus a storage-based data management system was implemented. We utilized a template android studio project (a blank multi-screen app) [4] to speed up development.

II. Algorithm Overview

A. REpeating Pattern Extraction Technique REPET (Paper)

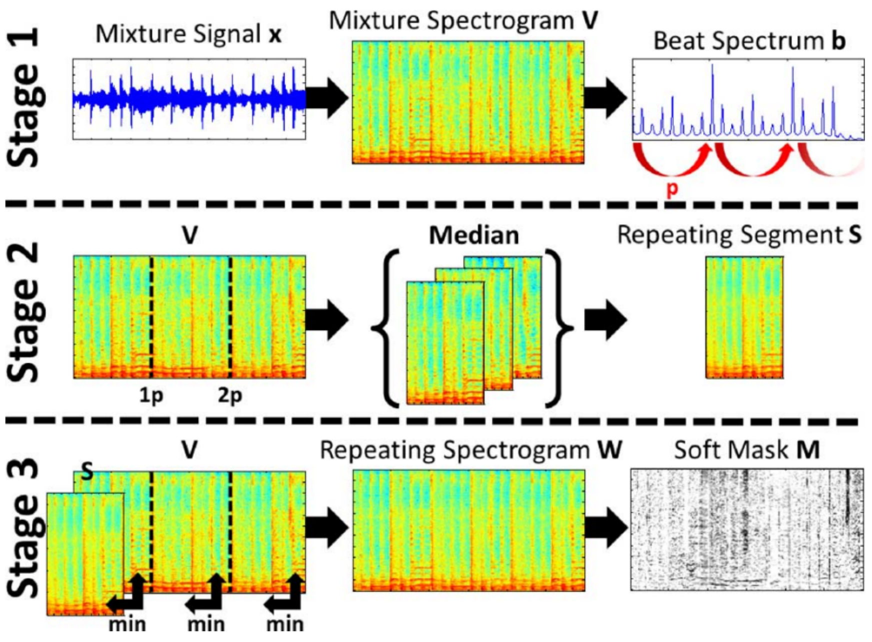

Fig. 1. Algorithm diagram

We derived the algorithm from the paper "Repeating pattern extraction technique (REPET): A simple method for music/voice separation" [3], for Voice/Background Separation. Where it identifies and extracts repeating structures within a song, which are typically characteristic of the instrumental background.

The overview of the algorithm is as follows:

1) Perform the short-time Fourier transform (STFT) on the audio signal, to convert time-domian signal into frequency domain, allowing the algorithm to analyze the audio in terms of its frequency components.

2) Analyze the frequency-domain representation to identify repeating segments. The beat spectrum, b, is first calculated, representing the rhythmic elements of the audio repeat over time. Then, we dynamically adjust the search range, j(all possible intervals), by adjusting delta1 and delta2. Within each interval, the algorithm searches for peaks in the beat spectrum within he specified ranges, with the highest peak values stored in h1 and h2, and the sum of the spectrum values over these ranges. If h1 and h2 are equal, it indicates a strong and consistent rhythmic

pattern at that specific interval, the algorithm then enhances the value I by emphasizing how much the peak stands out compared to the average in the surroundings. Finally, each value in J is normalized on the number of intervals checked. The populated J provides a metric of rhythmic strength across different intervals, providing information to find the repeating segments' period p.

```
Algorithm to calculate the repeating period $p$:
$l$ = length of b after discarding longest ¼ lag
$b$ = beat spectrum of the original signal

 1: Initialize $j$ with size $l|3$ and all values set to 0
 2: for $j$ from 0 to $l|3$ do
 3:     if $j \geq 2$ then
 4:         delta1 = 2
 5:     else
 6:         delta1 = $j$
 7:     end if
 8:     delta2 = $floor$(3 * j / 4)
 9:     for $i$ from $j$ to $l$ step $j$ do
10:         h1 = index of maximum b[k] for k from ($i$ - delta1) to ($i$ + delta1)
11:         h2 = index of maximum b[k] for k from ($i$ - delta2) to ($i$ + delta2)
12:         sum = sum of b[k] for k from ($i$ - delta2) to ($i$ + delta2)
13:         if h1 = h2 then
14:             I = I + (b[h1] - (sum / ($i$ + delta2 - ($i$ - delta2) + 1)))
15:         end if
16:     end for
17:     J[j - 1] = I / $floor$ (1 / j)
18: end for
```

3) Once the repeating segments are identified, the next task is to create a model of the background music using these segments, this model represents the typical or median background sound throughout the track. To do this, we first construct the repeating segment model S. The matrix S is initialized with the same dimensions as the absolute value of the STFT of the input signal. This matrix will be storing in the median values of the repeating segments for each frequency bin. We then iterate over each frequency bin i, within each repeating period l. For each combination, it collects values from the STFT matrix "V" across all repetitions 'r' of that period. For each frequency bin and repeating period, it computes the median of the collected values and stores it in S.

```
Algorithm to calculate the repeating segment model S:
$V$ = absolute value of STFT of input signal($x$)
$r$ = number of repetitions when repeating period is $p$
$N$ = FrameSize(we picked 1024)
$n = N/2 + 1$
$m$ = number of segments of times

 1: Initialize matrix S with the same shape as V, filled with zeros.
 2: for i from 0 to n-1 do
 3:     for l from 0 to p-1 do
```

```
4:         Create an empty list valuesAtL
5:         **for** k from 0 to r-2 **do**
6:             idx = l + k * p
7:             Append V[i, idx] to valuesAtL
8:         **end for**
9:         S[i,l] = median of valuesAtL
10:     **end for**
11: **end for**
```

4) After repeating segment model S is created, the next step towards creating the soft mask would be constructing the repeating spectrum model W. W is initialized as array of zeros with dimensions mirroring S. The loops are set up similarly to the previous part, but now the algorithm uses the model "S" to generate "W". For each frequency and time index, it checks if the index is valid, and then sets W[i,idx] as the minimum between model S[i,l] and the original value V[i, idx]. This step ensures that the model does not exaggerate the amplitude of the background beyond what is actually present in the original signal.

```
Algorithm to calculate repeating spectrum model $W$:
$V$ = absolute value of STFT of input signal($x$)
$r$ = number of repetitions when repeating period is $p$
$N$ = FrameSize(we picked 1024)
$n = N/2 + 1$
$m$ = number of segments of times

1: Initialize matrix W with the same shape as V, filled with zeros.
2: **for** i from 0 to n-1 **do**
3:     **for** l from 0 to p-1 **do**
4:         **for** k from 0 to r - 1 **do**
5:             idx = l + k * p
6:             **if** idx within the bounds of V's column **then**:
7:                 W[i, idx] = $Min$(S[i, l], V[i, idx])
8:             **end if**
9:         **end for**
10:     **end for**
11: **end for**
```

5) After V is generated, a soft mask M could be created.The algorithm calculates the ratio of the values in W to those in V for each element. This ratio, which ranges from 0 to 1, serves as the soft mask. It represents how much of each element in V is considered background. A value close to 1 means it is almost entirely background, while values closer to 0 indicate foreground dominance.

```
Algorithm to calculate the soft Mask $M$:
$V$ = absolute value of STFT of input signal($x$)
$r$ = number of repetitions when repeating period is $p$
$N$ = FrameSize(we picked 1024)
$n = N/2 + 1$
$m$ = number of segments of times

1: Initialize matrix M with dimensions n by m, filled with zeros.
```

```
2: **for** i from 0 to n - 1 **do**
3:     **for** j from 0 to m - 1 **do**
4:         **if** V[i, j] $\neq$ 0 **then**
5:             M[i, j] = W[i, j] / V[i, j]
6:         **else**
7:             M[i, j] = 0
8:         **end if**
9:     **end for**
10: **end for**
```

6) Finally, we apply the soft mask to STFT of the input signal, then apply inverse STFT to obtain time domain signals of the background and voice audio track.

B. Pitch Correction Algorithm

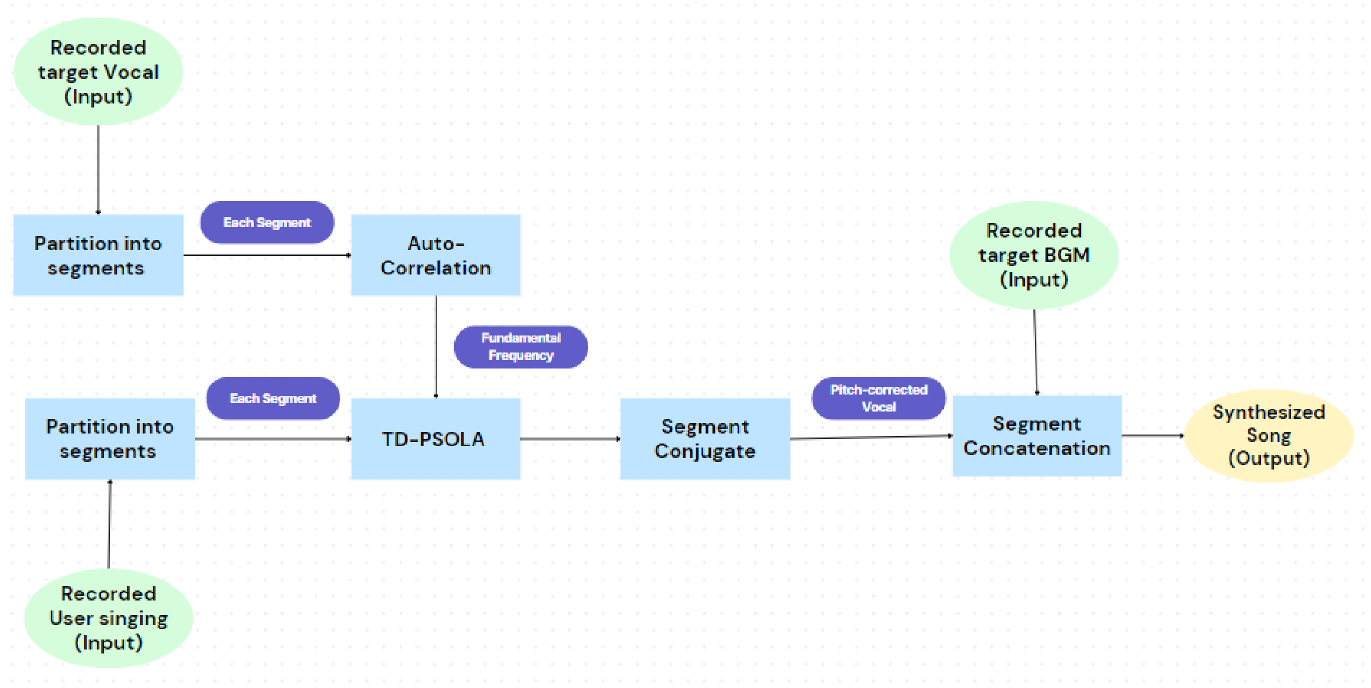

Fig. 2. Algorithm Flow Chart

To perform Pitch correction, the algorithm utilizes TD-PSOLA for pitch synthesis and Auto-correlation for frequency analysis. The algorithm makes a few assumptions about both input signals. First is the User singing audio is rhythmically correct (temporal correction needed is minimal). Second, the upper bound of the tempo for the music is 130 beats per minute. (roughly 2 Hz). Third, if the second assumption holds, we assume within each 23ms window, the fundamental frequency of the target voice does not change.

1) Both input signals (vocal component from REPET and User singing) will be partitioned into smaller segments (segment width = 1024 samples/ 23ms). the remainder of both signals will be discarded, this way both signals entering the next stage will be exactly the same length.

2) We then iterate through each segment of the signals. For each segment, we perform frequency analysis on the target signal's segment via autocorrelation to obtain fundamental/target frequency f. We then apply TD-PSOLA to the respective user signal's segment to target frequency f.

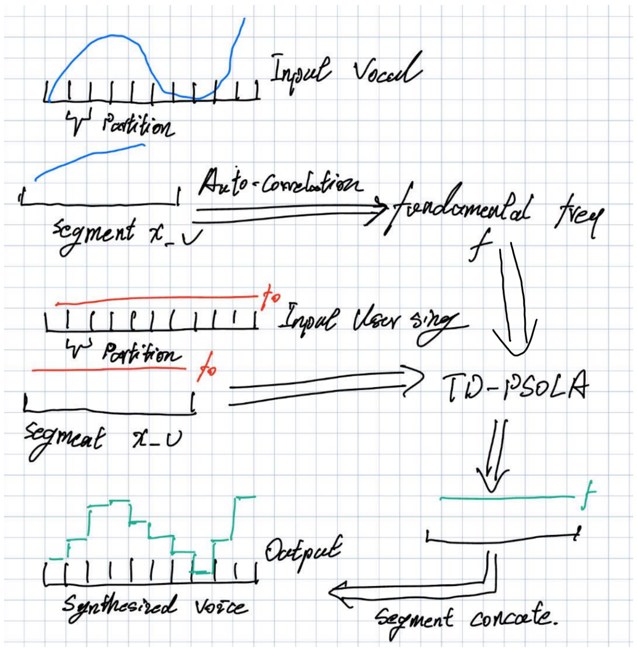

Fig. 3. Algorithm illustration

3)After every segment is processed, we now concatenate the synthesized/pitch-corrected segments back into a full-length audio track. We then overlay this audio track over the music component of the target song, length mismatch (the synthesized audio track is always shorter) is dealt with by zero-padding.

III. Technical Description

A. Prototyping

We prototyped the algorithm in Python, with library usage limited to Numpy and Scipy. This approach created some complexity that is not expected in Android in the data source (Stereo audio source, different sampling frequency, different data types). Making testing difficult. However, during testing, aside from a significant 40 Hz artifact, everything functioned as expected. We tried to combat this with a high-pass filter with limited success. Equipped with that knowledge that knowledge, in our Android (C++) implementation, we switched to an overlay + hanging window method to smooth out the window transition. While using Scipy is convenient for Python prototyping, this did present a significant challenge during Android implementation due to missing STFT/ISTFT implementation/verification.

B. Android Implementation

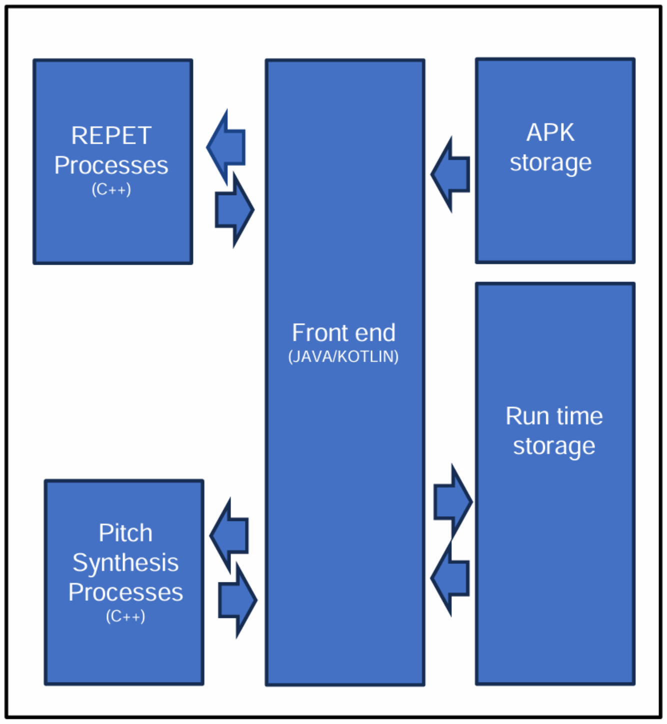

Fig. 4. System architecture overview

1) Overview: The Application mainly consists of three parts, storage, front end, and back end. Due to the inherent size of audio data, and the need for data storage for future use, we implemented a storage-based data management system (as opposed to memory-based), where each input, intermediate, and final results will be encoded and stored into the run-time storage (a volatile app specific storage space). This not only allowed us asynchronous access to data, but it also significantly simplified memory management and expanded the capability of the app (by allowing it to handle much longer audio data). The app consists of three activities (screens), due to our unique data management system, no data passage between activities is needed. Instead, each activity reads data from storage when commanded, and releases all memory when excited. As the system architecture diagram shows, each front-end activities would read data from storage and pass in the back-end Cpp modules if needed.

2) UI: The application separates its two main functionalities, Voice/BGM separation and Pitch correction, into two separate screens, and incorporates a home screen as access. In the target song screen, the user records the target song and hits "GO!" for processing/encoding. 2 playback button is included for their respective post-process result (BGM and Voice). The voice synthesis screen is where the user records his own singing and hits "GO!" for pitch-correction and processing. A dropdown menu is included for target source selection.

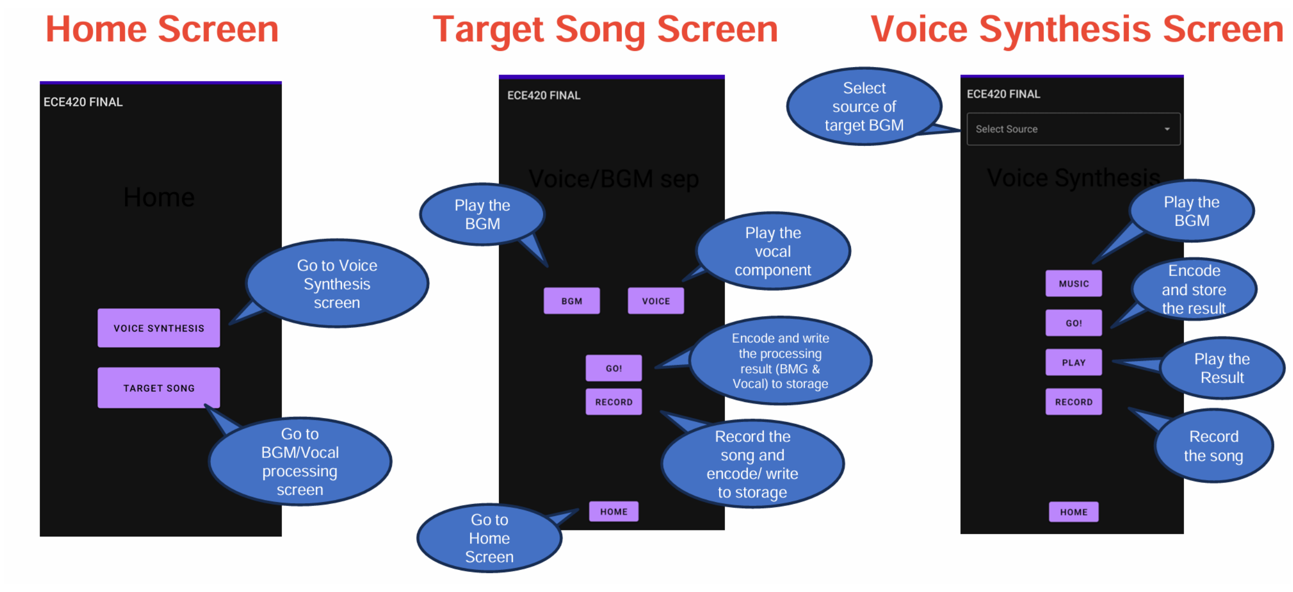

Fig. 5. System architecture overview

3) Front-end: For simplicity, I will refer to everything that is implemented and executed in Java/Kotlin as front-end, and everything implemented and executed in C++ as back-end. The front-end is implemented based on a multi-screen application template by "Code with Manik" [4], where an empty multi-screen application was provided with buttons to traverse between the activities. This is the reason for implementing the Android project partially in Kotlin (the template is in Kotlin). We then modified the UI (.xml) files to construct our own design, since no data is passed through intent (memory-based data handling), this was relatively easy. The Main activity (Mainactivity.kt) remains largely unchanged with buttons to access the second and third activities.

The Second activity (SecondActivity.kt) is implemented in Kotlin, to stay aligned with the project template. In this activity, the user records the target song and enters the recording in to repet.cpp for processing, stores and playback the separated BGM and voice. A recorded and write-to-file system is implemented in this activity, as well as Cpp (REPET.cpp) function call, file read as int16 array, file write by int16 array. This activity handles the target song's BGM/voice separation. It does so by allowing the User to record any soundtrack, encode (by Android native "AAC" codec and "MP4" wrapper), and store it as " "Targetrecording.wav" (fun fact, android native audio handling system handles audio files regardless of the suffix, in other words, wrapper and codec are engineer specified, and it will work regardless of file naming, this is why naming .wav works even the bitstream is actually that of an mp4, And .wav came from a mistake I made during initial implementation of the recording function, and for unified file handling, every other audio file is named .wav in this project as well). A function is then developed to read the audio files. To handle the AAC codec, we imported ffmpeg library, which is used to decode AAC-encoded file into a temporary raw audio file(.PCM). The PCM file will provide us with an uncompressed bit stream in little-endian. We handle the little-endian to int16 conversion in Kotlin, at the end of the process, the temporary PCM file is deleted. An int-16 to .wav file encoder is implemented in a very similar way. The audio player is handled by Android Native Media player thus I won't go into details here. A comprehensive function description could be found in the table below:

File Name	**Function Name**	**Input**	**Output**	**Notes**
SecondActivity.kt	encodeSamplesToFile	samples/intarray, outputFilePath/string	void	Encodes and write the intarray into a file specified by the file path in runtime storage. Audio data is encoded in AAC format and wrapped by MP4 headers.
SecondActivity.kt	decodeAudioToSamples	inputFilePath/string	intarray	Decodes the audio file specified in File path and convert the data into an int array(JAVA)
SecondActivity.kt	playAudio	datapth/string	void	invoke the native android media player to play the audio file specified by datapath
SecondActivity.kt	pausePlaying	void	void	destructor for the native android media player
SecondActivity.kt	startRecording	void	void	Record an audio and encode by AAC codec, wrap in mp4 headers and store to default location
SecondActivity.kt	pauseRecording	void	void	Native android audio recorder destructor
SecondActivity.kt	repet	see cpp part	see cpp part	function call to invoke repet.cpp

TABLE I
SECOND ACTIVITY FUNCTION LIST

The Third activity (ThirdActivity.java) is implemented in java. This activity allows user to record his own singing, enters the recording, along with decoded results form second activity for processing, encode and playback the result. The majority of the functions here (audio decoding and encoding) is implemented in the same way as in kotlin (second activities) with the high light being an drop-down many for target song source selection. However due to the complexity of reading files in "res" folder (apk storage) this function is not implemented fully. A comprehensive function list can be found in the table below.

File Name	**Function Name**	**Input**	**Output**	**Notes**
ThirdActivity.java	encodeSamplesToFile	samples/intarray, outputFilePath/string	void	Encodes and write the intarray into a file specified by the file path in runtime storage. Audio data is encoded in AAC format and wrapped by MP4 headers.
ThirdActivity.java	decodeAudioToSamples	inputFilePath/string	intarray	Decodes the audio file specified in File path and convert the data into an int array(JAVA)
ThirdActivity.java	playAudio	datapth/string	void	invoke the native android media player to play the audio file specified by datapath
ThirdActivity.java	pausePlaying	void	void	destructor for the native android media player
ThirdActivity.java	startRecording	void	void	Record an audio and encode by AAC codec, wrap in mp4 headers and store to default location
ThirdActivity.java	pauseRecording	void	void	Native android audio recorder destructor
ThirdActivity.java	tune	see cpp part	see cpp part	function call to invoke ece420$_{m}ain.cpp$

TABLE II
THIRD ACTIVITY FUNCTION LIST

4) Back-end: The two main algorithms are implemented in C++. Both C++ file is implemented by pass by reference, with memory allocation handled by Java/Kotlin.

Repet.cpp: we imported kissfft to handle fft and ifft, with that we could implement STFT and ISTFT functions needed for REPET function. Aside from that, we mostly followed python prototype. The implementation for remaining function is trivial. For pitch-correction, we implemented the algorithm entirely in ece420-main.cpp, with it mostly following the python prototype, and the implementation is mostly trivial. A comprehensive list could be found in the table below:

File Name	Function Name	Input	Output	Notes
Repet.cpp	processAudioe	input1/int* output1/int* output2/int*	void	Serves as the Main function for this file, and is where most of the REPET algorithm resides.
Repet.cpp	autocorrelation	input/vector<float>	output/vecotr<float>	performers autocorrelation and returns the result
Repet.cpp	inverseSTFT	input/vector<vector<complex<float»> windowlength/int numberofoverlap/int signallength/int	vector<float>	performs STFT to frequency domain data and output the result as vector of float
Repet.cpp	stft	intputsignal/vector<float> lengthofsegment/int numberofoverlap/int	outpu/ vector<vector<complex<float»>	performs STFT to an input signal and output the result as vector of vector of complex of float
Repet.cpp	applyhanningwindow	signal/vector<float> start/int size/int	output/vector<float>	calculate the hanging window coefficient and apply it to the input signal, output the signal as vector of float.

TABLE III
REPET.CPP FUNCTION LIST

File Name	Function Name	Input	Output	Notes
ece420-main.cpp	Tune-Main	input1/int* output1/int* output2/int* length/int	void	Serves as the Main function of the algorithm, it handles the partitioning of the audio data and passes the correct fragment of data to different functions.
ece420-main.cpp	ProcessFrame	dataBuf/int* , dataOut/int* , bufferIn/float*, bufferOut/float*	void	This function handles the overlaying between buffers to achieve the concatenation of the segments to form the full length audio-track
ece420-main.cpp	detectBufferFrequency	buffer/float* , bufferIn/float*	int	detect the fundamental frequency of the input buffer and output that frequency as int.
ece420-main.cpp	detectBufferPeriod	buffer/float* , bufferIn/float*	int	detect the fundamental period of the input buffer and output that period as int.
ece420-main.cpp	findEpochLocations	epochLoaction/vector<int> buffer/float* periodlen/int bufferIn/float*	void	finds the peak of each epoch and stores the index of the peaks with in the buffer and store it in a vector of int
ece420-main.cpp	overlapAddArray	dest/float* , src/float*, startIdx/int, len/int	void	over lay the source array into the destination array, of the index specified by startIdx and len
ece420-main.cpp	PitchShift	bufferIn/ float* , bufferOut/flaot*	bool	performs TD-PSOLA

TABLE IV
ECE420MAIN.CPP FUNCTION LIST

IV. Results

We generally achieved our primary goal. Our primary goal is to achieve voice/BGM separation of sufficient quality and pitch correction with sufficient quality. Given the algorithm selected (REPET), our implementation can separate voice very clearly with almost no discernable background music remaining. Except high-energy drum beats, which is a limitation of the algorithm. And our pitch correction and synthesis algorithm produced noticeable pitch correction, even when given a flat tone as user input, and did not produce significant artifacts and noise, at least not any more than pure TD-PSOLA does.

Since Auto-tune in itself caters to human perception, it is very difficult to quantify our results. However, we, along with a decent amount of peers agree that, given the algorithms, out application performed adequately. Thus overall, it is safe to say, that we achieved the primary design goals of our project.

V. Suggestions

A. *Pre-Processed Audios*

As mentioned in our proposal and in the final presentation, our original vision for this app involves pre-processed audio tracks stored in APK storage (res/raw). However, the reading of APK resources with non-native methods proves to be too difficult in the scope of our project. A future upgrade would be figuring out this function and subsequently implement the remaining portion of the drop-down menu in ThridActivity.java.

B. *Temporal correction via word recognition*

Another major shortcoming of our application is its inability to perform temporal correction to the user singing, making user input being rhythmically corrected essential. We have tried multiple approaches to resolve this, but they are all unsatisfactory. Thus a word recognition system would provide us word-wise temporal correction for more accurate auto-tune.

VI. Contribution Results

Here is a summary of major work involved in this project:

Contents	**Number**
task1	Write the Skeleton of Android App, including UI, file management, library inclusion and Cpp integration.
task2	Implemented REPET algorithm in C++ with associated STFT and ISTFT
task3	Implemented the remaining algorithm in C++ with all associated helper functions
task4	Implementation of Python prototype
task5	Setup GitHub for version control, code clean ups, and project timeline management

TABLE V
Tasks before Final Project Demo

- Task numbers are as in table V
- Ethan Zhou: task 1, 2, 3, 4, 5
- Bill Yang: task 2, 4

References

[1] S. W. G. T. C. i Wang; Minje Kim, "Deep autotuner: A pitch correcting network for singing performances," *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020.

[2] K. Sato, "Extracting specific voice from mixed audio source," *IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, 2019.

[3] Z. Rafii and B. Pardo, "Repeating pattern extraction technique (repet): A simple method for music/voice separation," *IEEE transactions on audio, speech, and language processing*, vol. 21, no. 1, pp. 73–84, 2012.

[4] C. W. Manik, "How to create multi screen app in android," 2023.