Phase Vocoder

Tobias Kim, Sunny Ng

***Abstract*—This report describes the implementation of a Phase Vocoder on an H7 board that utilizes Digital Signal Processing techniques, such as PSOLA and WSOLA, to timescale audio without changing the pitch. The main motivation for this project is to see how well audio could be successfully modified on smaller microcontrollers. Implementation involves using MATLAB and CubeIDE to create the Phase Vocoder software and running them on an H7 board along with a USB used as memory storage. As a result, the Phase Vocoder implemented on the H7 Board is fully functional, capable of transforming any kind of input audio into longer or shorter durations of itself all while maintaining its original pitch.**

***Index Terms*—FFT, Phase Vocoder, PSOLA, WSOLA**

I. INTRODUCTION

A Phase Vocoder is an algorithm that is used to timescale audio without changing the pitch. The goal of this project was to successfully implement a fully functional Phase Vocoder, that can timescale any given audio input, on a STMicroelectronics H7 Nucleo board based on an online MATLAB implementation.

A. History

In the past, audio modification was done via analog circuits, but with today's technology, audio modification can be done via digital algorithms. A few of these algorithms involve PSOLA and WSOLA, which utilize pitch synchronicity and waveform similarity respectively. [3][2] In this paper, we'll be using a different algorithm known as the phase vocoder to perform our audio modification.

B. Global constraints

For this project, we only had about 10 weeks to develop the phase vocoder, presenting a sizable time pressure on development, which limits the number of optimizations and code improvements that could be done afterwards.

Additionally, our team didn't fully understand how the phase vocoder worked initially which meant we had to spend a few weeks researching how the algorithm worked. File size was another constraint because sound recordings are usually MB's large, especially those with high sample rates.

II. MOTIVATION

In some systems like PSOLA and WSOLA, those systems do not take phase into account, which can lead to issues. The phase vocoder utilizes the phase component in its calculations when doing the overlap add operations. [5] For our team, we chose to implement this particular algorithm because we were aware that PSOLA had already been implemented on the H7 board in recent years. Additionally, research and code already exist for these applications, which makes the development much easier.

TABLE I
INTENTION VS. WHAT HAPPENED

Create pseudocode from scratch and get it working on Matlab	Found an existing Matlab implementation of the code
Turn the Matlab code into working code on the H7 board	Translated Matlab code into C code and got it working on a C compiler
Have it fully working on the H7 with USB	Used USB as an additional memory storage alongside getting H7 to work
Improve quality only if there's extra time	Got the implementation working much earlier than predicted so used time to further improve implementation to make it more

III. APPROACH

C. Team Organization

Our team worked together on many aspects of the project, such as debugging, feature optimizations, etc., but the main responsibilities assigned to each team member are as follows:

Tobias was in charge of picking apart the MATLAB code implementation and writing the code for the time-frequency processing part of the roadmap.

Sunny was in charge of writing code for the windowing/FFT part of the code and dealing with the USB integration.

D. Plan

Refer to Table I.

E. Standard

In our code, we do not utilize many widely used standards other than the FFT algorithm. In particular, we use the CSMIS-DSP FFT functions. This library provided optimized FFT code made for ARM architectures, especially for those with floating point capabilities like the H7 board. [4]

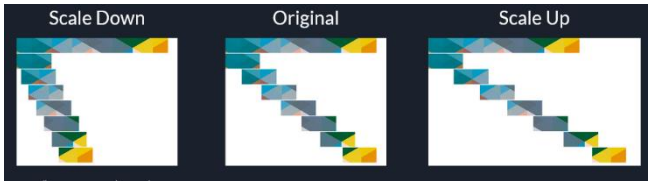

Fig. 1. Graphic demonstrating the process of overlap adding (Courtesy of http://www.baltana.com/abstract/geometric-shape-background-wallpapers-23075.html)

F. Theory

At its core, the phase vocoder retrieves overlapping snippets of an audio recording before converging or diverging those snippets closer or farther away from each other as demonstrated by Fig. 1. Those convergence of the frames correspond to the speeding up of an audio recording while a divergence of the frames correspond to the slowing down of an audio recording.

However, when performing these time shifts on the frames, a corresponding phase shift must also be added as shown in (1). This Fourier property expresses that when we have a time shift, a corresponding phase shift must occur.

$$x(t - t_0) \overset{F}{\Leftrightarrow} e^{-j\omega t_0} X(j\omega) \tag{1}$$

For the phase vocoder, we do not compute the phase change for all frequencies for multiple reasons. One, the processing time would be incredibly long if we were to calculate the phase change for every single frequency, and two, some frequencies are not actually relevant to the actual characteristics of the sound. As such, in our model, we only focus on the magnitude peaks of the Fourier spectrum.

While it's easy to find these magnitude peaks and use those to calculate the phase change, it's important to remember that we're using the discrete Fourier transform (DFT). Because the discrete version does a "sampling" of the frequency domain, we lose lots of information as demonstrated in Fig. 2.

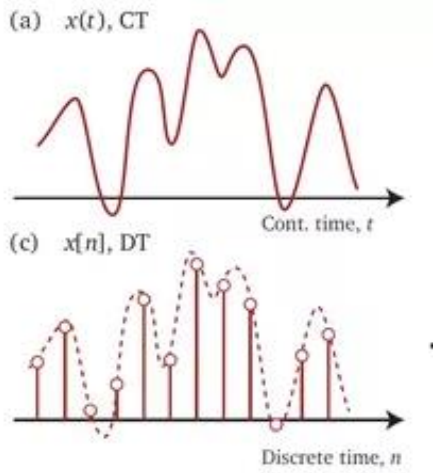

Fig. 2. Going from Continuous time to Discrete time
(Courtesy of https://www.quora.com/What-is-the-exact-difference-between-continuous-Fourier-transform-discrete-time-Fourier-transform-DTFT-discrete-Fourier-transform-DFT-Fourier-series-and-discrete-Fourier-series-DFS-In-which-cases-which-one-is-used)

In particular, the last peak in the continuous Fourier transform (CFT) when sampled by the DFT is not represented correctly. If we were to assume that the peak shown in the DFT was the correct one, we'd be off by a sizable amount. For example, if we were sampling at 44 kHz with an FFT size of 4096, our frequency fidelity would be about 22 Hz. In Fig. 2, we'd be off by about 11 Hz roughly estimating from looking at the spectrum.

Thus, in order to estimate the actual frequency of the observed peak, we use frequency estimation in the form of phase unwrapping. The idea behind phase unwrapping is that as time moves forward, the phase of a sinusoid changes at the rate of the corresponding frequency.

By using (2), we can find the actual frequency by iteratively changing the value of n and seeing which one is closest to the frequency. By knowing the phase of the current and previous frame and the time difference between those frames, we can estimate the actual frequency as opposed to just using the peak.

$$f_n = \frac{(\theta_2 - \theta_1 + 2\pi n)}{2\pi(t_2 - t_1)} \tag{2}$$

Once we find those frequencies, we can calculate the phase change of the frame shifted forwards or backwards, apply that phase change, and then compute the IFFT of that frame before overlap adding it with the previously processed frames.

G. Software

We first used MATLAB to develop the Phase Vocoder implementation and also to format our audio files so that they are able to be inputted into through the Phase Vocoder to be processed. After the audio has been processed, it then gets reconverted from a text file back into an audio file to be played.

CodeLite was the C compiler used to test and run our Phase Vocoder implementation. This step was used to make sure the implementation worked in C before porting it over to STM32CubeIDE, which was what we needed to use to actually get our Phase Vocoder running on the H7 board.

The final piece of software we used was STM32CubeIDE. Without it, we could not run our code on the H7 board.

To see the most important code snippets for our Phase Vocoder, refer to Appendix.

H. Hardware

The only hardware used in this project was an 8 GB USB, an USB cable adapter, a computer, and the STMicroelectronics Nucleo H743ZI board itself. For complete the entire setup, See Fig. 3.

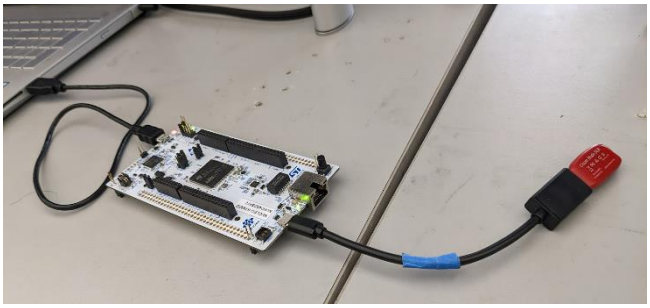

Fig. 3. Our entire hardware setup for the Phase Vocoder

I. Operation

To replicate making our project, these are the following steps:

1) Study the MATLAB implementation and provided code given in [1]
2) Translate the MATLAB code into C
3) Run that C code on STM32CubeIDE
4) Write and include code that integrates the USB to be able to read and write files

To actually use our Phase Vocoder, one would have to first use MATLAB to transform any audio file into a bin file. Next, you setup the hardware and run the, hopefully working code on the H7 board. After the H7 is done running, obtain the fully processed bin file from the USB and use MATLAB to transform it back to an audio file. Now, that audio is ready to be played.

IV.RESULTS

J. Description

For our final results, we had three aspects of our project to focus on: Memory, Sound quality, and Speed

With the introduction of the USB, memory constraints are not an issue anymore. We're able to store and process songs that contain MBs of data.

The sound quality, while not terrible, had multiple artifacts that lowered the quality: One of the artifacts was warbling, similar to how one would take a thin sheet of metal and wave it around except with much higher pitch, another artifact was echoing, and the last artifact that could be discerned was slurring.

The speed of the program on the H7 board was noticeably slower than the speed in MATLAB, to the point we were getting only about 20 frames per second max.

K. Discussion

With the introduction of the USB, we no longer have memory issues like we did previously, although we did have to do a lot of adjustments to the USB code for this work, like for an example we read the direct binary representation of the floats instead of storing values through text to simplify the reading and writing. We can now process audio recordings of large sizes while leaving us lots of memory to work with when doing the actual processing. However, this implementation is a bit slow as described in the speed section.

Warbling may occur due to phase discontinuities that still appear despite having frequency estimations. In addition, the overlap adding might not be adding the frames on top of each other correctly, which could also lead to the warbling effect. Better frequency estimation and other metrics can be used to correct this. Echoing may occur due to the fact that because we do overlap adding, some of the shifted signals are being phase shifted too far ahead or behind. Similar to the warbling problem, better frequency estimation could also be used to fix this. Slurring, or smearing, usually occurred when the FFT size was too "large" in comparison to the sampling rate. While we are not too sure why this is so, we speculate that it is due to some highly localized maxima that are being detected and considered peaks when they should not be considered peaks. This can be remedied by changing the criteria of peaks to not only be greater than their adjacent neighbors but also neighbors further out.

There are a few factors that affected the overall processing speed. The USB, while convenient and large in storage, is slow due to being only able to send information via serial connection. Considering that we're taking in kB's of information, this transference of information is painfully slow. In our implementation, we made the program go a little faster by only writing in what we need to read and write. Another bottleneck for the speed is the use of the FFT. The best optimization we can do to counter that is use the R-FFT. This algorithm takes advantage of the fact real signals have symmetries in their Fourier spectrums and cuts the processing time in half. Lastly, for hardware operations, we can use tightly-coupled memory (TCM) to reduce the number of cache flush-reloads. RAM can only be accessed indirectly by the CPU via the cache. However, TCM is always readily available to the CPU. Any information known to be frequency accessed such as frame spectrums can be stored here instead of RAM, which should improve the speed overall.

APPENDIX

Listed down below are the most important code in our Phase Vocoder

```
  void process_spectrum (float * mag, float * phs) {

      static float freq[NH_SIZE + 1];
      static int f_init = 0;
      if(!f_init) { initialize_frequencies(freq); f_init =
1; }
      static float ph_hld[NH_SIZE + 1] = {0};
      static float ph_adv[NH_SIZE + 1] = {0};

      int* peak_inds;
      int ci;
      int num_peaks;
      float * best_frequency;
      float t;

      peak_inds = peaksort(mag, &num_peaks);

      best_frequency = malloc(sizeof(float)*num_peaks);
      for (int pk = 0; pk < num_peaks; pk++) {
              ci = peak_inds[pk];
          t = DT_IN * freq[ci] + (ph_hld[ci]- phs[ci])
/(2*M_PI);
          t = round(t);
          t = (phs[ci] - ph_hld[ci]) + 2 * M_PI * t;
          t = t / (2 * M_PI * DT_IN);
          best_frequency[pk] = t;
      }

      for (int i = 0; i < NH_SIZE + 1; i++) {
          ph_hld[i] = phs[i];
      }

      for (int pk = 0; pk < num_peaks; pk++) {
          ci = peak_inds[pk];
           t = best_frequency[pk];
          ph_adv[ci] = fmod( ph_adv[ci]+ 2*M_PI*t*DT_OUT,
2*M_PI);
          phs[ci] = ph_adv[ci];
      }

      free(peak_inds);
      free(best_frequency);

  }

  int* peaksort(float* spectrum, int* num_peaks) {

        int* peak_inds = find_peaks(spectrum, num_peaks);

        float max_val = find_max_peak_val(spectrum,
peak_inds, *num_peaks);

        float threshold = max_val * EPS;
        peak_inds = clip_peaks(spectrum, peak_inds,
threshold, num_peaks);


        return peak_inds;
  }

  void write_file (TCHAR filename[], UINT * arr,
unsigned int write_num) {
```

```
      FIL fp;
      unsigned int byteswritten;

      if (Appli_state == APPLICATION_READY) {

          if(f_mount(&USBH_fatfs, USBHPath, 1) !=
FR_OK) {
              Error_Handler();
          }

          if (f_open(&fp, filename, FA_OPEN_APPEND |
FA_WRITE) != FR_OK ) {
              Error_Handler();
          }
          else {
              if (f_write(&fp, arr, write_num,
&byteswritten) != FR_OK) {
                  Error_Handler();
              }
              f_close(&fp);
          }

          if (f_mount(NULL, USBHPath, 0) != FR_OK){
                Error_Handler();
          }
      }
  }
```

```
  void read_file(TCHAR file_name[], UINT *read_buff,
unsigned int read_num, unsigned int * bytesread, int
offset) {

        FIL MyFile;

        if (Appli_state == APPLICATION_READY) {

          if(f_mount(&USBH_fatfs, USBHPath, 0) !=
FR_OK){
               Error_Handler();
          }

          if (f_open(&MyFile, file_name, FA_READ) !=
FR_OK ) {
               Error_Handler();
          }

          else
          {
               f_lseek(&MyFile, offset);
               f_read(&MyFile, read_buff, read_num,
bytesread);

               if (f_close(&MyFile) != FR_OK) {
                      Error_Handler();
               }

               if(f_mount(NULL, USBHPath, 0) !=
FR_OK){
                      Error_Handler();
               }
           }
      }
  }
```

REFERENCES

[1] W. A. Sethares, “A Phase Vocoder in Matlab,” [Online] Available: https://sethares.engr.wisc.edu/vocoders/phasevocoder.html

[2] J. *Driedger, and M. Müller, “*A Review of Time-Scale Modification of Music Signals*” Applied Sciences, 6(2), 57*, doi:10.3390/app6020057

[3] M. Krakower “The PSOLA Algorithm,” [Online] Available: https://engprojects.tcnj.edu/autotuner16/2016/04/11/the-psola-algorithm/

[4] "Complex FFT Functions," *CMSIS.*, [Online]. Available: https://www.keil.com/pack/doc/CMSIS/DSP/html/group__ComplexFFT.html

[5] T. Royer, “Pitch shifting algorithm design and applications in music,” Kth Royal Institute, Stockholm, Sweden, 2019, pp. 18–21 [Online] Available: http://kth.diva-portal.org/smash/get/diva2:1381398/FULLTEXT01.pdf