Final Project: Implementing a Channel Vocoder

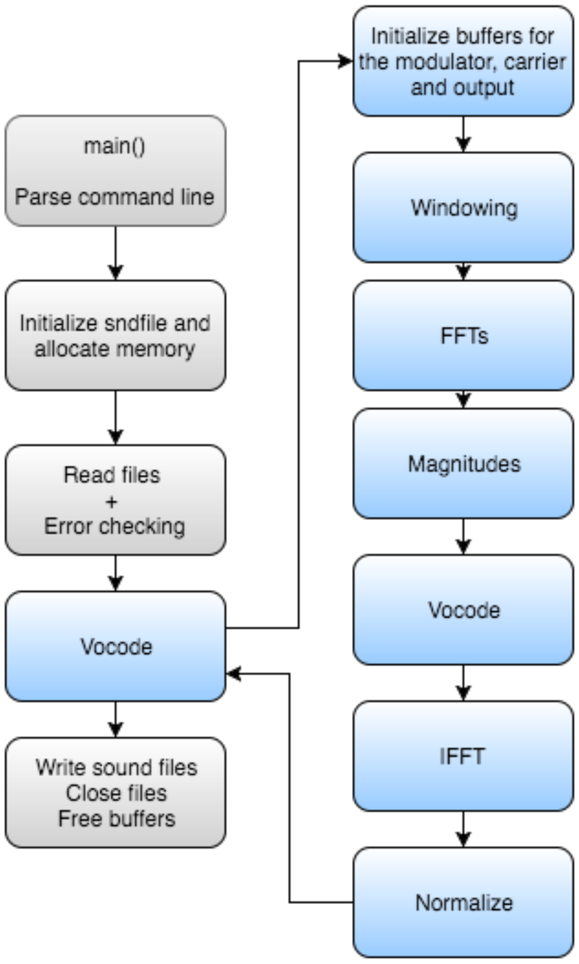
Xiao Lu
2618 C Programming
May 5, 2016

Description

The project is an implementation of a classic channel vocoder instrument. A vocoder is a tool to, in some way, combine two kinds of sounds together by using signal processing techniques. The most common way of processing vocals with a vocoder is to use a synthesizer part as the carrier signal and the voice as a modulator; this creates the sort of filtered sound you hear in the vocals of Herbie Hancock's 'I Thought It Was You' and more recently, the bridge of Daft Punk's 'Get Lucky'.

Specifically, the program reads in two WAV files, a carrier and a modulator, and then writes the resulting signal into a new WAV file as the output. Although the program is used in a non-realtime way, it aims to simulate a real-time process. Therefore, the overall framework is generally designed as a real-time program in which I employed an input buffer and an output buffer with a size of N (e.g. 512 or 1024) frames. Also there are parameters defined in the code that can significantly improve the effect, such as number of bands and hop size (related to overlapping).

Program Flow


main()
Parse command line
Initialize sndfile and allocate memory
Read files + Error checking
Vocode
Write sound files
Close files
Free buffers
Initialize buffers for the modulator, carrier and output
Windowing
FFTs
Magnitudes
Vocode
IFFT
Normalize

I wrote all of the code in files test.c and vocode.c, with corresponded header files. The code of fft.c was mainly borrowed from *convolve.c* that was used in the previous assignment; what I did was re-organizing the code and writing a .h file for it.

In file main.c:

- The main() function parses the command line; sets up and open the audio object WAV file; opens the modulator and carrier WAV files, reads them into buffers in memory and does necessary error checking; calls the vocoder function where the core algorithm is performed; after that, it writes the output buffer into a new WAV file; and free used buffers.
- A number of important parameters are defined in the header file including NUM_BAND which is the number of bands to use, BUFFER_SIZE that is the size of a buffer and a window, and HOP_SIZE which means how many frames the buffer will hop.

In file fft.c:

- Two datatypes are defined in fft.c, real and complex, which are in essence floating points data.
- The fft and ifft function was originally written by the author of *convolve.c,* which require the input buffer, the length, and a scratch buffer in order to complete a recursive process.

In file vocode.c:

- In this file, a set of buffers for window, modulator, carrier and output are initialized and allocated with memory; windowing and FFTs are performed on both the modulator and carrier; the magnitudes of the two resulting arrays from FFTs are calculated by the magnitude function I wrote; then, for each filterbank, a ratio of x_mod_mag to x_car_mag is obtained as a gain value that is then used to calculate the output values; after getting the whole spectrum of the output, an IFFT is performed to re-construct the time-domain signal; finally a normalization process is applied by calculating RMS values of input and output and then scaling the output.
- In addition, in the outer loop, the increment of each count is set to the hop size; the filterbanks are linearly arranged across half of the spectrum (only the content below the Nyquist frequency is useful); the window type is flexible but usually a Hanning or Hamming window is a good choice.

I developed and ran my project on Mac OS X, but it will also compile and run on Windows/Cygwin.

My Code and Libraries

I wrote the code described above, and used the following libraries:

Sndfile

I obtained the libraries by using "brew"

```
brew install sndfile
```

Computational Complexity

The critical part contributing to the complexity of the program is in the vocode function, where a number of buffers are accessed, a series of mathematics are involved, and there are also FFTs being called.

For FFTs, the Big-O complexity is Log-Linear Time (N*log2(N)). For most of the computations in the code, the Big-O complexity is Linear Time.

The non-realtime program works fairly fast and stable, so it is highly possible to work in real-time (which is also my future work) as well.

Comilation and Linking

The makefile shows how to compile and link my program for the OS X platform.

Command Line

The generic command line to run the program is:

```
   ./vocoder modulator.wav carrier.wav [output.wav]
```

A specific command line is

```
   ./vocoder mod_testing.wav car_synth.wav output.wav
```