Signals and Systems Final Project: Channel Vocoder

Deniz Celik and Jacob Riedel

May 6, 2015

Abstract

During the course of this project, we were successful in emulating a channel vocoder using Python. Our project takes in an audio clip, either provided or recorded, and passes the clip through a filtering process. Once completed, the processed signal is mixed with a chosen modifier signal to produce an output. The output wave sounds like the original sound, but has a fundamental frequency and harmonics of the chosen modifier signal. Our vocoder is accessed through an easy to use interface, which shows plots of the different signals for both teaching and understanding purposes.

Introduction

The first vocoders were developed in 1928 in Bell Labs by the scientist, Homer Dudley. He was granted a patent in 1939 and in the same year, the device was introduced at the New York World's Fair. The vocoder was originally created for compression and security purposes to transmit speech across copper phone lines. During World War II, the vocoder was used to transmit encrypted transatlantic messages between Roosevelt and Churchill. In the 1950s, a German physicist, Werner Meyer-Eppler, saw the potential uses for the vocoder in musical applications. From there, its musical history blossomed. The vocoder went through many different re-design phases, and was used in the production of soundtracks for movies, such as Clockwork Orange, and by artist such as ELO, Pink Floyd, Eurythmics, Tangerine Dream, Telex, David Bowie, Kate Bush, and many more.

Theory

A channel vocoder operates by taking in a carrier wave, like a voice, breaking the input into time segments. Those segments are then Fourier Transformed to the frequency domain and the same signal is passed through several band-pass filters to separate out specific ranges. A modifier signal, like a 440Hz sawtooth signal, is passed into the vocoder. It undergoes the same process to have matching frequency bands as the carrier. The amplitudes of the frequencies are multiplied together and the final frequency spectrum is rebuilt using the phase of the modifier and the multiplied amplitudes. This spectrum is then passed through a Inverse Fourier Transform and each small time segment is added back together to produce the output signal.

Channel Vocoder

In addition to creating a working channel vocoder, we also wanted it to be easily accessible and enhance understanding of the process. This was accomplished via a two graphs for each signal, one in the time domain and the other in the frequency domain. This not only allows the user to check that their inputs are correct but also allows them to use it in order to learn about how vocoders function. The code used to create the project can be separated into two categories, the front-end GUI and the vocoder back-end.

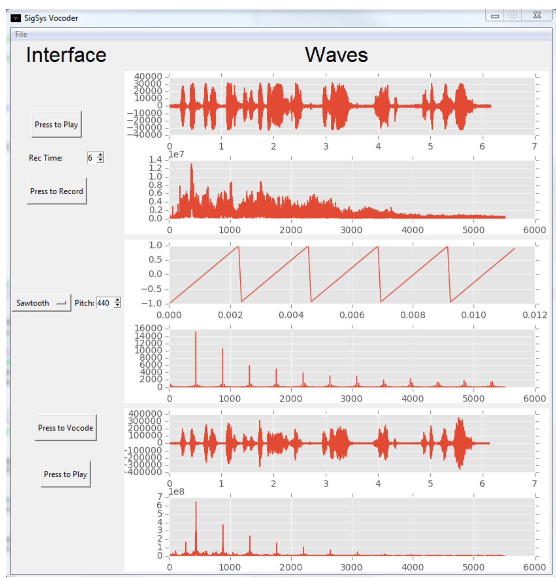

Figure 1: Entire user interface

GUI

For our GUI we decided to use TkInter, a python wrapper for tk/tcl that allows for creation of basic interfaces and has support for the matlab python plotting library. It separates the window into two pieces, one that contains all of the generate plots and the other with the interface options for controlling the vocoder. We supported 3 signals, an input, a channel modulator, and the output.

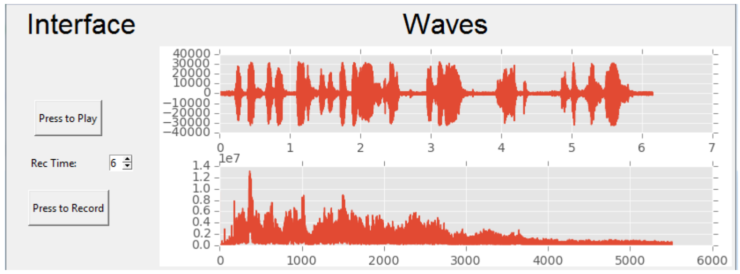

Figure 2: Interface and Plots for Input Signal

Input

The input consisted of a default wave made up of a famous quote from Monty Python which could be recorded over using the record button. We also included a variable time limit for how long the program would record input, rather than detecting using a volume threshold. The included play button is self-explanatory and only plays the current input. The plots are re-generate for each new recording and display the wave and it's spectrum.

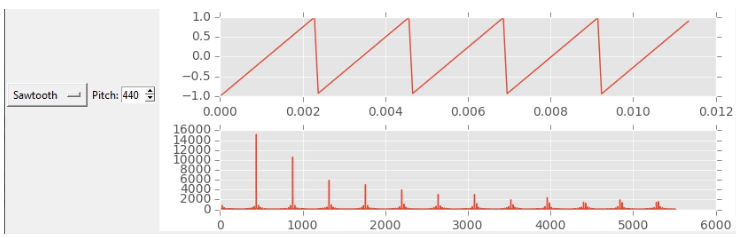

Figure 3: Interface and Plots for Channel Modulation Signal

Modulator

The channel modulator consists of a default 440Hz sawtooth wave that can be changed to be a number of different wave types, including Sin, Cos, and Square. It also allows for any pitch between 100-1000Hz for any of the selected waves. When modified, the plots will automatically recalculate to reflect the change in either pitch or signal type.

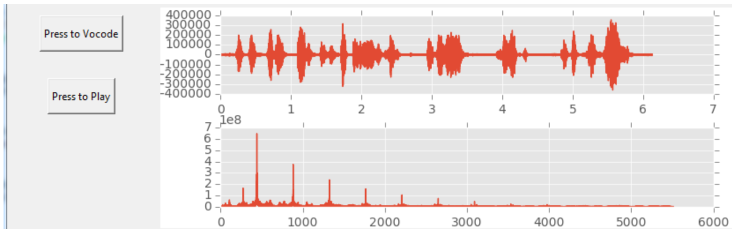

Figure 4: Interface and Plots for Output Signal

Output

The output interface consists of the vocoded input and modulator that can be played using the aptly named play button. The output interface also contains the vocode button, that when pressed will vocode the current input and modulator and re-plot all of the graphs. This is the only way to recreate the output and uses function calls to the vocoder back-end in order to do so.

Back-end

The back-end consists of a single class called vocoder which provides methods for recording audio, creating modulation waves and vocoding an input with a modulator. It works by creating a single instance within the GUI and then setting input, output and other variables within the vocoder instance to what is being shown or modified on the GUI. As such, the method calls to the back-end run on the variables within the vocoder rather than acting as utility methods which allows for a distinct separation between the user interface and the background code. This also allows the vocoder to be used in nearly any program as it functions as a stand alone import.

Conclusion and Improvements

In conclusion, it is possible to produce a channel vocoder using python. Channel vocoders also happen to be enjoyable to play with and an interesting learning opportunity, especially when using our GUI. Having the wave and spectrum plots for each signal is a helpful tool for visualizing the process that a channel vocoder uses. For future implementations of this project, we would like to add a "upload" feature where the user specifies an audio file for the input to the vocoder. Adding noise to the vocoder will provide better fricative reproduction to produce more realistic sound. Auto-time segmentation for the files will provide more normalized audio outputs and better results from the vocoder. Moving away from the ThinkDSP package and utilizing more low-level numpy commands will create a faster vocoding process. The final improvement to this project would be packaging it as an installer and have it as a standalone application to run on anybody's computer. By doing this, anyone has the ability to learn about vocoding and the joy it brings.

Code

Code can be found on our GitHub Repository

Gui Code

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 22 22:52:10 2015

@author: Deniz Celik and Jacob Riedel
"""

import Tkinter as tk
#import matplotlib
from matplotlib import pyplot as plt
from matplotlib import  style
import Vocoder as vc
#import thinkdsp
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg#, NavigationToolbar2TkAgg
import winsound


HEADER_FONT = ("Helvetica",24)
#MIN_HEIGHT = 600
#MIN_WIDTH  = 800
style.use('ggplot')

def callback(val):
    print val

def play_audio(wave):
    wave.write('temp.wav')
    winsound.PlaySound('temp.wav', winsound.SND_FILENAME)
    return 'temp.wav'

class gui(tk.Tk):
    def __init__(self, *args, **kwargs):
        tk.Tk.__init__(self, *args, **kwargs)
        #Don't allow gui to be resized
        self.resizable(False,False)
        self.title('SigSys Vocoder')
        self.iconbitmap(self, default='Vocoder_logo.ico')
        self.vocoder = vc.vocoder()#filename = 'temp.wav')


        #Create frame for main window
        container = tk.Frame(self)
        container.pack(side="top", fill="both", expand="True")

```

```
        #Configure grid weights
        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0, weight=1)

        #Create interface frame
        self.interface= Interface(container,self)
        self.interface.grid(row=0,column=0,sticky="NEWS",padx = 0)

        #Create waves frame
        self.waves = Waves(container,self)
        self.waves.grid(row=0,column=1,sticky="NEWS",padx = 0)

        #Create the MenuBar
        menu = tk.Menu(container)
        tk.Tk.config(self, menu=menu)

        #Add file dropdown menu
        filemenu = tk.Menu(menu,tearoff=0)
        menu.add_cascade(label="File", menu=filemenu)

        #Add options to file menu
        filemenu.add_command(label="About", command=quit)
        filemenu.add_separator()
        filemenu.add_command(label="Exit", command=quit)


class Interface(tk.Frame):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self,parent, highlightthickness=0)
        label = tk.Label(self, text="Interface",font=HEADER_FONT)
        label.grid(row=0)
        self.gui = controller

        #INPUT STUFF
        inputs = tk.Frame(self, height = 150, width = 150)
        inputs.grid(row=1,pady=50)

        input_play = tk.BooleanVar()
        input_play.set(False)

        play_input = tk.Button(inputs,height = 2, text="Press to Play",
                               command = lambda: play_audio(self.gui.vocoder.input))
        play_input.grid(row=0, pady = 15)

        record_input = tk.Button(inputs,height = 2, text="Press to Record",
                                 command = lambda: self.record_audio(rec_time.get()))
        record_input.grid(row=2, pady = 15)

        rec_time = tk.IntVar()
        rec_time.set(6)

        input_label = tk.Label(inputs, text="Rec Time:")
```

```
        input_label.grid(row=1, column= 0, pady=5, sticky = "w")
        rec_timer = tk.Spinbox(inputs,width = 2,wrap = True,
                               from_ = 1,to=10,
                               textvariable = rec_time,
                               command = lambda: callback(rec_time.get()))
        rec_timer.delete(0,tk.END)
        rec_timer.insert(0,6)
        rec_timer.grid(row=1,column = 1, pady=5, sticky = "w")

        #OUTPUT STUFF
        outputs = tk.Canvas(self, height = 150, width = 150)
        outputs.grid(row=3,pady=70)

        output_play = tk.BooleanVar()
        output_play.set(False)

        volume = tk.IntVar()
        volume.set(5)

        play_output = tk.Button(outputs,height = 2, text="Press to Play",
                                command = lambda: play_audio(self.gui.vocoder.output))
        play_output.grid(row=1, column=0, pady = 15)

        vocode = tk.Button(outputs,height = 2, text="Press to Vocode",
                                command = lambda: self.vocodestuff())
        vocode.grid(row=0, column=0, pady = 15)

#        volume_output = tk.Scale(outputs,label = "Happiness Factor",variable = volume,
#                                  from_ = 10,to=0,
#                                  command = lambda volume: callback(volume))
#        volume_output.set(5)
#        volume_output.grid(row=0, column=0, pady=15)

        #Channel stuff
        channels = tk.Frame(self, height = 450, width = 150)
        channels.grid(row=2, pady=0)
        channel_options = ('Sawtooth', 'Sin', 'Cos',
                           'Square', 'Triangle', 'Parabolic')
        #Channel 1 Stuff
        ch1 = tk.Canvas(channels,height = 150, width = 150)
        ch1.grid(row=1, pady=55)
#
#        ch1_var = tk.BooleanVar()
#        ch1_var.set(False)

        ch1_freq = tk.IntVar()
        ch1_freq.set(440)

        ch1_wave = tk.StringVar()
        ch1_wave.set(channel_options[0])
```

```

        ch1_dropdown = tk.OptionMenu(ch1,ch1_wave,*channel_options,
                                     command = lambda ch1_wave: self.update_modulator(ch1_wave, ch1_freq.get()))
        ch1_dropdown.grid(row=0, columnspan = 1, sticky = "w")

        ch1_label = tk.Label(ch1, text="Pitch:")
        ch1_label.grid(row=0, column= 1, pady=20, sticky = "w")
        ch1_pitch = tk.Spinbox(ch1,width = 4,wrap = True,
                               from_ = 100,to=1000,
                               textvariable = ch1_freq,
                               command = lambda: self.update_modulator(ch1_wave.get(), ch1_freq.get()))
        ch1_pitch.delete(0,tk.END)
        ch1_pitch.insert(0,440)
        ch1_pitch.grid(row=0, column =2, pady=5, sticky = "w")

#         ch1_toggle = tk.Checkbutton(ch1,
#                                     text="Toggle Channel 1 Modulation",
#                                     variable=ch1_var,
#                                     command = lambda: callback(ch1_var.get()))
#         ch1_toggle.grid(row=1, pady = 5,columnspan = 3, sticky = "w")

    def record_audio(self, time):
        self.gui.vocoder.record_input(recordtime = time)
        self.gui.vocoder.update("record")
        self.gui.waves.update()

    def update_modulator(self, sig, pitch):
        self.gui.vocoder.set_channel(sig,pitch)
        self.gui.vocoder.update("update")
        self.gui.waves.update()

    def vocodestuff(self):
        self.gui.vocoder.update("v")
        self.gui.waves.update()

class Waves(tk.Frame):

    def __init__(self, parent, controller):
        tk.Frame.__init__(self,parent, highlightthickness=0)
        label = tk.Label(self, text="Waves",font=HEADER_FONT)
        label.grid(row=0)
        self.gui = controller

        #INPUT STUFF
        input_fig = plt.figure(figsize=(6.5,2.5), dpi=100)
        input_fig.subplots_adjust(left=0.11, right=0.96,
                                  top=0.95, bottom=0.11,
                                  wspace = 0.2, hspace = 0.43)

        self.input_wave = input_fig.add_subplot(211)
```

```python
        self.input_spec = input_fig.add_subplot(212)

        self.input_plot = FigureCanvasTkAgg(input_fig, master=self)
        self.input_plot._tkcanvas.config(highlightthickness=0)
        self.input_plot.show()
        self.input_plot.get_tk_widget().grid(row=1)

        #CHANNEL STUFF
        channel_fig = plt.figure(figsize=(6.5,2.5), dpi=100)
        channel_fig.subplots_adjust(left=0.11, right=0.96,
                                    top=0.95, bottom=0.11,
                                    wspace = 0.2, hspace = 0.43)

        self.channel_wave = channel_fig.add_subplot(211)
        self.channel_spec = channel_fig.add_subplot(212)

        self.channel_plot = FigureCanvasTkAgg(channel_fig, master=self)
        self.channel_plot._tkcanvas.config(highlightthickness=0)
        self.channel_plot.show()
        self.channel_plot.get_tk_widget().grid(row=2)

        #OUTPUT STUFF
        output_fig = plt.figure(figsize=(6.5,2.5), dpi=100)
        output_fig.subplots_adjust(left=0.11, right=0.96,
                                    top=0.95, bottom=0.11,
                                    wspace = 0.2, hspace = 0.43)

        self.output_wave = output_fig.add_subplot(211)
        self.output_spec = output_fig.add_subplot(212)

        self.output_plot = FigureCanvasTkAgg(output_fig, master=self)
        self.output_plot._tkcanvas.config(highlightthickness=0)
        self.output_plot.show()
        self.output_plot.get_tk_widget().grid(row=3)

        self.update()
#         toolbar = NavigationToolbar2TkAgg( input_plot, self )
#         toolbar.update()
#         toolbar.grid(row=5,sticky='W')

    def update(self):
        self.input_wave.clear()
        self.input_spec.clear()
        self.input_wave.plot(self.gui.vocoder.input.ts,self.gui.vocoder.input.ys)
        self.input_spec.plot(self.gui.vocoder.input_spec.fs,self.gui.vocoder.input_spec.amps)
        self.input_plot.show()

        self.channel_wave.clear()
        self.channel_spec.clear()
        channel_wave_seg = self.gui.vocoder.channel.segment(duration=(1.0/self.gui.vocoder.pitch)*5)
        self.channel_wave.plot(channel_wave_seg.ts,channel_wave_seg.ys)
```

```
        self.channel_spec.plot(self.gui.vocoder.channel_spec.fs, self.gui.vocoder.channel_spec.amps)
        self.channel_plot.show()

        self.output_wave.clear()
        self.output_spec.clear()
        self.output_wave.plot(self.gui.vocoder.output.ts,self.gui.vocoder.output.ys)
        self.output_spec.plot(self.gui.vocoder.output_spec.fs,self.gui.vocoder.output_spec.amps)
        self.output_plot.show()

if __name__ == "__main__":
    #mod = C.Model()
    #con = C.Controller(mod)
    app = gui()

    app.mainloop()
```

Vocoder Code

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 22 22:52:10 2015

@author: Jacob Riedel and Deniz Celik
"""

import numpy as np
import thinkdsp
import pyaudio
from array import array
import time


class vocoder():
    def __init__(self,filename = "flesh_wound.wav", signal_type = "Sawtooth", pitch = 440, num_channel = 1024, num_band = 32):

        self.num_bands = num_band
        self.num_channels = num_channel
        self.input = thinkdsp.read_wave(filename)
        self.input_spec = self.spectrum_gen(self.input)
        self.framerate = self.input.framerate
        self.duration = self.input.duration

        self.signal_type = signal_type
        self.pitch = pitch
        self.channel = self.Sig_generate()
        self.channel_spec = self.spectrum_gen(self.channel)

        input_seg = self.segmentor(self.input)
        channel_seg = self.segmentor(self.channel)

        voded_wave = self.vocode(input_seg,channel_seg)
        self.output = voded_wave
        self.output_spec = self.spectrum_gen(self.output)
```

```python

    def set_input(self, new_file):
        self.input = thinkdsp.read_wave(new_file)
        self.framerate = self.input.framerate
        self.duration = self.input.duration

    def set_channel(self, new_type, new_pitch):
        self.signal_type = new_type
        self.pitch = new_pitch

    def set_num_channel(self, new_num):
        self.num_channels = new_num

    def record_input(self,recordtime = 6):
        chunk = 1024
        self.framerate = 41000
        self.pya = pyaudio.PyAudio() # initialize pyaudio
        self.stream = self.pya.open(format = pyaudio.paInt16,
           channels = 1,
           rate = self.framerate,
           input = True,
           output = True,
           frames_per_buffer = chunk)
        data = self.get_samples_from_mic(self.framerate,300,chunk,recordtime)
        self.input = thinkdsp.Wave(data,self.framerate)
        self.duration = self.input.duration
        self.stream.close()
        self.pya.terminate()

    def segmentor(self, wave):
        'Turns the input wave into segmented parts to vocode properlly'
        Seg = []
        for i in np.arange(0, wave.duration, wave.duration/self.num_channels):
            Seg.append(wave.segment(start=i, duration=wave.duration/self.num_channels))
        return Seg

    def spectrum_gen(self, wave):
        spectrum = wave.make_spectrum()
        return spectrum

    def Sig_generate(self):
        'Chooses what generated signal to use, and at what pitch.'
        #print self.signal_type
        if self.signal_type == 'Sawtooth':
            sig = thinkdsp.SawtoothSignal(freq=self.pitch, amp=1, offset=0)
        elif self.signal_type == 'Sin':
            sig = thinkdsp.SinSignal(freq=self.pitch, amp=1, offset=0)
        elif self.signal_type == 'Cos':
            sig = thinkdsp.CosSignal(freq=self.pitch, amp=1, offset=0)
        elif self.signal_type == 'Triangle':
```

```
            sig = thinkdsp.TriangleSignal(freq=self.pitch, amp=1, offset=0)
        elif self.signal_type == 'Square':
            sig = thinkdsp.SquareSignal(freq=self.pitch, amp=1, offset=0)
        elif self.signal_type == 'Parabolic':
            sig = thinkdsp.ParabolicSignal(freq=self.pitch, amp=1, offset=0)
        wav=sig.make_wave(framerate = self.framerate, duration = self.duration)
        return wav

    def get_wave(self):
        return self.vocoded_wave

    def make_file(self, wave):
        wave.normalize
        wave.write('temp.wav')

    def vocode(self, segment_voice, segment_gen):
        """This is the vocoder.  It multiplies the amplitudes of two seperate signals
        to produce a singular response"""
        temp_final = []
        for j in range(self.num_channels):
            saw_spec = segment_gen[j].make_spectrum()
            input_spec = segment_voice[j].make_spectrum()

            input_hs = input_spec.hs
            saw_hs = saw_spec.hs

            saw_bands = np.array_split(saw_hs, self.num_bands)
            input_bands = np.array_split(input_hs, self.num_bands)

            final_bands = np.empty_like(saw_bands)
            for i in range(self.num_bands):
                amp_multi = np.abs(saw_bands[i])*np.abs(input_bands[i])
                phase_multi = np.angle(saw_bands[i])
                final_bands[i] = amp_multi*(np.cos(phase_multi)+(np.sin(phase_multi)*1j))

            temp_final.append(np.ma.concatenate(final_bands).data)
        final_wave = []
        for i in range(len(temp_final)):
            final_wave.append(thinkdsp.Spectrum(hs=temp_final[i], framerate = self.framerate).make_wave())
        output = final_wave[0]
        for i in range(1,len(final_wave)):
            output |= final_wave[i]
        return output


    def update(self,ident):
        self.channel = self.Sig_generate()
        self.channel_spec = self.spectrum_gen(self.channel)
        self.input_spec = self.spectrum_gen(self.input)
```

```

        if(ident=="v"):
            input_seg = self.segmentor(self.input)
            channel_seg = self.segmentor(self.channel)

            voded_wave = self.vocode(input_seg,channel_seg)
            self.output = voded_wave
            self.output_spec = self.spectrum_gen(self.output)

    def get_samples_from_mic(self, sample_rate = 8000, threshold = 1000, chunk_size = 1000, recordtime = 6):
        # initialize pyaudio object
        p = pyaudio.PyAudio()
        stream = p.open(format=pyaudio.paInt16, channels=1, rate=sample_rate,
                input=True, output=True,
                frames_per_buffer=chunk_size)
        def is_silent(snd_data, th):
        #"Returns 'True' if below the 'silent' threshold"
            return max(snd_data) < th
        # initialize an array to store the data
        data_vec = array('h')
        # wait until we hear something
        while 1:
            # read a chunk of samples from the mic
            data = stream.read(chunk_size)
            # convert the data into an array of int16s
            snd_data = array('h', data)
            # if no longer silent break out of detection loop
            if not is_silent(snd_data, threshold):
                break
       # append the sound data from the previous chunk into the array
        data_vec.extend(snd_data)
        # collect samples until we get a silent block
        start = time.time()
        while 1:
            # read a chunk of data
            data = stream.read(chunk_size)
            snd_data = array('h', data)
            # stick the chunk of samples at the end of the vector that stores
            # the samples
            data_vec.extend(snd_data)
            # if silent break out of loop
            end = time.time()
            if end-start>recordtime:#is_silent(snd_data, threshold):
                break
        # convert to a numpy array
        # we don't use a numpy array directly because it's slower than
        # array
        x = np.frombuffer(data_vec, dtype= np.dtype('int16'))
        # close the pyaudio stream
        stream.stop_stream()
        stream.close()
```

```
        p.terminate()
        # return the data_vec samples
        return x
```