ASSIGNMENT 0: DSP-Experiments

Final Project

Avrutin, Perr-Sauer

CS102
Nicolas Avrutin,
2014 EE
nicolasavru@gmail.com

CS102
Jordan Perr-Sauer
2014 EE
jordan@jperr.com

Wednesday 11th August, 2010

Wednesday 11th August, 2010

Jordan Perr-Sauer

Class of 2014

Electrical Engineering

jordan@jperr.com

Nicolas Avrutin

Class of 2014

Electrical Engineering

nicolasavru@gmail.com

Stack Overflow

http://stackoverflow.com/questions/2063284/what-is-the-easiest-way-to-read-wav-files-us

Scott Wilson, Stanford University

https://ccrma.stanford.edu/courses/422/projects/WaveFormat/

1 Description

DSP-Experiments consists of python scripts that can encode images into sound files such that the image can be seen in the spectrogram of the produced sound. The code also provides a method for viewing the spectrogram of the generated audio, decoding the sound back into a (lossy) version of the original image.

In our spectrograms, the x-axis represents time (in the audio file) and the y-axis represents frequency. The brightness of a given pixel represents amplitude (loudness). To encode an image such that it can be viewed in a spectrogram, we first separate the image by it's columns of pixels. Each column of pixels will be represented by a "tone" in the output audio stream

that lasts for a fixed amount of time. These tones consist of sin waves of varying frequencies, at varying amplitudes, depending on the intensity and position of each pixel in the column, respectively. Each column in the input image is converted into such a tone, and these tones are concatenated and output to a wave file.

2 Compilation

You do not need to compile any part of this program. There are some dependencies, however, which you must install prior to execution.

2.1 Dependencies

- Python 2.6 or higher (NumPy and PIL are not yet compatible with Python 3)
- NumPy (http://numpy.scipy.org/)
- Python Imaging Library (http://www.pythonware.com/products/pil/)

3 Execution

3.1 Manual Page

To produce sound (as a wav file) from an image:

```
python imageToWav.py g|c pathToImage pathToWav
```

The g and c options stand for "Grayscale" or "Color". Using the c option will result in a 3 channel wav file, where each channel of audio represents one channel of color (red, green, and blue). Using the g option will result in a single channel (mono) wav file and all color data will be lost.

To generate the spectrogram of a produced (or other) wav file:

```
python wavToImage.py pathToWav pathToImage
```

The format of the image will be derived from the extension you give pathToImage. You can use any format supported by PIL.

3.2 Sample Inputs

Sample images are provided in the test_images dir:

```
python imageToWav.py c test_images/test.png out.wav
```

Sample wav files generate with our script are located in outputs:

```
python wavToImage.py outputs/cpu_color.wav out.png
```

4 Features

We support color images by using 3-channel wav files. You can specify grayscale or color using command line arguments at execution. This is discussed in "Execution."

5 Notes

- Proper functionality is not guaranteed for very large files.
- Scaling and resizing algorithms will be significantly reworked in the future.

6 Listings

6.1 imageToWav.py

```
import numpy as np
import Image, struct, math, sys

"""

Usage:
    ./imageToWav.py [c|b] image_path wav_path

"""


##### DEFINES AND ARGS #####

SAMPLE_RATE = 44100

ARG_IMAGE = sys.argv[2]
ARG_OUTFILE = sys.argv[3]
ARG_COLOR = False
if sys.argv[1] == "c":
    ARG_COLOR = True

CHANNELS = 1
if ARG_COLOR:
    CHANNELS = 3

#rgb aliases
R=0
G=1
B=2


##### FUNCTIONS #####

def oscillator(x, freq=1, amp=1, base=0, phase=0):
    return base + amp * np.sin(2 * np.pi * freq * x + phase)

def writewav(filename, numChannels, sampleRate, bitsPerSample, time, data):
    wave = open(filename, 'wb')
    dataSize = time * sampleRate *  numChannels * bitsPerSample / 8
    #https://ccrma.stanford.edu/courses/422/projects/WaveFormat/
    ChunkID = 'RIFF'
    ChunkSize = struct.pack('<I', dataSize + 36)
    Format = 'WAVE'
    Subchunk1ID = 'fmt '
    Subchunk1Size = struct.pack('<I', 16)
    AudioFormat = struct.pack('<H', 1)
    NumChannels = struct.pack('<H', numChannels)
    SampleRate = struct.pack('<I', sampleRate)
    ByteRate = struct.pack('<I', sampleRate * numChannels * bitsPerSample / 8)
```

```
    BlockAlign = struct.pack('<H', numChannels * bitsPerSample / 8)
    BitsPerSample = struct.pack('<H', bitsPerSample)
    Subchunk2ID = 'data'
    Subchunk2Size = struct.pack('<I', dataSize)
    header = ChunkID + ChunkSize + Format + Subchunk1ID + Subchunk1Size +\
             AudioFormat + NumChannels + SampleRate + ByteRate + BlockAlign +\
             BitsPerSample + Subchunk2ID + Subchunk2Size
    wave.write(header)
    # wav header: 30 s at 44100 Hz, 1 channel of 16 bit signed samples
    # wave.write('RIFF\x14'(\x00WAVEfmt \x10\x00\x00\x00\x01\x00\x01\x00D'
    #            '\xac\x00\x00\x88X\x01\x00\x02\x00\x10\x00data\xf0_(\x00')
    # little endian
    # write float64 data as signed int16
    # amplitude/volume, max value is 32768
    # higher amplitude causes noise (vertical bars)
    print "Packing WAV..."
    (1000 * data).astype(np.int16).tofile(wave)
    wave.close()


##### MAIN ROUTINE #####

# Open image and extract pixel data
im = Image.open(ARG_IMAGE)
size = im.size
d = list(im.getdata())

xres = size[0]
yres = size[1]
yscale = 22000 / float(yres)
time = int(round(22.0 * xres / yres))
xlen = time / float(size[0])


#because this is easier than finding the flag to disable broadcasting
out = [np.zeros(0), np.zeros(0), np.zeros(0)] #more mehh
for x in xrange(xres):
    t = np.arange(x*xlen, x*xlen + xlen, 1./SAMPLE_RATE)
    tones = [np.zeros(t.size), np.zeros(t.size), np.zeros(t.size)] # mehh
    print "{0}: {1}%".format("Color" if ARG_COLOR else "Grayscale",
                             round(100.0 * x / xres, 2))

    for y in xrange(yres):
        p = d[x+xres*y]
        for c in range(CHANNELS):
            if p[c] > 10 or p[R] > 10 or p[G] > 10 or p[B] > 10:
                if ARG_COLOR:
                    amplitude = 10**(1-5.25+4.25*(p[c])/(255))
                else:
                    amplitude = 10**(1-5.25+4.25*(p[R]+p[G]+p[B])/(255*3))
                tones[c] += oscillator(t, amp=amplitude, freq=yscale * (yres - y))
    for c in range(CHANNELS):
        tones[c] = tones[c] + 1
        tones[c] = tones[c] / math.log(128)
```

```
104             out[c] = np.append(out[c],tones[c])
105
106
107 if ARG_COLOR:
108     out = np.array(out)
109     out = out.flatten('F')
110 else:
111     out = out[0]
112
113 #pad with silence at end if necessary
114 if out.size < SAMPLE_RATE * time * CHANNELS:
115     out = np.append(out, np.zeros(SAMPLE_RATE * time *CHANNELS - out.size))
116
117 writewav(ARG_OUTFILE, CHANNELS, SAMPLE_RATE, 16, time, out)
```

6.2 wavToImage.py

```
import wave
import sys
import struct
import math

import render

import numpy as np

xres = 1000
yres = 1000

fname = sys.argv[1]

# http://stackoverflow.com/questions/2063284/what-is-the-easiest-way-to-read-wav-files-usi
wav = wave.open (fname, "r")
nchannels, sampwidth, framerate, nframes, comptype, compname = wav.getparams()
frames = wav.readframes(nframes * nchannels)
out = struct.unpack_from("%dh" % nframes * nchannels, frames)
wav.close()

print nchannels, nframes
data = np.zeros((nchannels, nframes), np.int16)
for f in xrange(nframes*nchannels):
    data[f%nchannels][f/nchannels] = out[f] #integer division used intentionally

# set this to the number of the channel you want to use
channel = 0
#loadWav(fname)
time = float(nframes) / framerate
print time, framerate
yres = int(22 * xres / time)
print yres
#yscale = float(yres) / 22000


interval = nframes / xres
print interval, nframes

screen = render.createScreen(xres, yres)

def generateColor(val):
    v = math.log(abs(val)+.001)*10

for x in range(xres):
    fft0 = np.fft.rfft(data[0][x*interval:(x+1)*interval+10])
    fft1 = np.fft.rfft(data[1][x*interval:(x+1)*interval+10])
    fft2 = np.fft.rfft(data[2][x*interval:(x+1)*interval+10])
    fft0 = [z.real for z in fft0]
    fft1 = [z.real for z in fft1]
    fft2 = [z.real for z in fft2]
#    print len(fft)
```

```
53  #    print fft
54      print "{0}%".format(round(100.0 * x / xres, 2))
55      for y in range(interval / 2):
56          c = [math.log(abs(fft0[y])+.001)*10,
57              math.log(abs(fft1[y])+.001)*10,
58              math.log(abs(fft2[y])+.001)*10]
59  #         print x, y, c
60          render.plot(x, yres-y, c, screen)
61
62  #render.display(screen)
63  render.saveExtension(screen, sys.argv[2])
```