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 11$^{\text{th}}$ 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

- Scaling and resizing algorithms will be significantly reworked in the future.

6 Listings

6.1 imageToWav.py

```python
"""

imageToWav.py
Encodes image data as an audio signal.
One channel of audio per channel of image data.

Author: Nicolas Avrutin, nicolasavru@gmail.com
        Jordan Perr-Sauer, jordan@jperr.com

Revision history:
    See http://github.com/nicolasavru/DSP-Experiments

Bugs:

Todo:
    - Tweak scaling/encoding algorithm to lessen artifacts
    - Variable "slice" width for more space efficient encoding

"""


##### IMPORTS #####

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


##### DEFS AND ARGS #####

SAMPLE_RATE = 44100

YRES = 400
T_PER_COL = 0.03

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, nSamples, data):
    wave = open(filename, 'wb')
    dataSize = nSamples *  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)
    # 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)
xres = im.size[0]
yres = im.size[1]
# resize to 500px height for convenience
im = im.resize((int((float(xres)/yres)*YRES), YRES), Image.BICUBIC)
d = list(im.getdata())

# either the column width or the song length must be fixed width
# and if song length is fixed, we have to limit the frequency
# spectrum we use to maintain aspect ratio
xres = im.size[0]
yres = im.size[1]
yscale = 22000 / float(yres)
# 1/100 of a second of audio for every column in image
sampsPerCol = int(SAMPLE_RATE*T_PER_COL)


#because this is easier than finding the flag to disable broadcasting
```

```
out = [np.zeros(0), np.zeros(0), np.zeros(0)] #more mehh
elfMagic = (float(sampsPerCol)/SAMPLE_RATE)
for x in xrange(xres):
    t = np.linspace(x*elfMagic, (x+1)*elfMagic, num=sampsPerCol)
    tones = [np.zeros(sampsPerCol), np.zeros(sampsPerCol), np.zeros(sampsPerCol)] # 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)
        out[c] = np.append(out[c],tones[c])


if ARG_COLOR:
    out = np.array(out)
    out = out.flatten('F')
else:
    out = out[0]

writewav(ARG_OUTFILE, CHANNELS, SAMPLE_RATE, 16, int(xres*sampsPerCol), out)
```

6.2 wavToImage.py

```python
"""

wavToImage.py
Creates a spectrogram image from a wav file.
Auto-detects color or grayscale.

Author: Jordan Perr-Sauer, jordan@jperr.com
        Nicolas Avrutin, nicolasavru@gmail.com

Revision history:
    See http://github.com/nicolasavru/DSP-Experiments

Bugs:
    - Scaling leaves nasty artifacts

Todo:
    - Tweak scaling/encoding algorithm to lessen artifacts
    - Variable resolution spectrogram

"""


##### IMPORTS #####

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


##### DEFS AND ARGS #####

YRES = 400
T_PER_COL = 0.03

ARG_WAVFILE = sys.argv[1]
ARG_IMGFILE = sys.argv[2]


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

print "Loading WAV..."

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

# Separate loaded frames by channel
data = np.zeros((nchannels, nframes), np.int16)
for f in xrange(nframes*nchannels):
```

```
    data[f%nchannels][f/nchannels] = out[f] # integer division used intentionally

# Compute the dimensions of the encoded image
# Setup some constants for decoding
sampsPerCol = framerate*T_PER_COL
yres = YRES
xres = int(math.ceil((nframes)/sampsPerCol))

# Create a PIL Image
if nchannels == 3:
    im = Image.new("RGB", (xres+1, yres+1))
else:
    im = Image.new("L", (xres+1, yres+1))


print "Generating Spectrogram..."

# Loop over all "slices" in WAV file.
for x in xrange(xres):
    fft = list()
    # Perform an FFT on the sound contained in each slice
    for c in range(nchannels):
        foo = np.fft.rfft(data[c][x*sampsPerCol:(x+1)*sampsPerCol])
        fft.append([z.real for z in foo])
    print "{0}%".format(round(100.0 * x / xres, 2))
    # Compute pixel colors from fft result
    for y in xrange(len(fft[0])):
        if nchannels == 3:
            c = (int(math.log(abs(fft[0][y])+.001)*10),
                 int(math.log(abs(fft[1][y])+.001)*10),
                 int(math.log(abs(fft[2][y])+.001)*10))
        else:
            c = int(math.log(abs(fft[0][y])+.001)*10)
        # Plot pixels, scaling to YRES
        im.putpixel((int(x), int(yres-int(((float(y)/len(fft[0]))*YRES)))), c)

print "Encoding Image File..."

# Save image file using PIL
im.save(ARG_IMGFILE, ARG_IMGFILE.split(".")[-1].upper())

print "Done."
```