Lab 4: Blocking, Amplitude Modulation, Vibrato, Flanging

EE 4163 / EL 6183 : Digital Signal Processing Lab

Fall 2015

1 Blocking

The usual practice is to read and write blocks of samples to and from audio devices rather than one sample at a time. The following demo program processes a block of samples at a time (to do amplitude modulation).

```
# playAM_blocking_fix.py
# Play a mono wave file with amplitude modulation.
# This implementation reads and plays a block at a time (blocking)
# and corrects for block-to-block angle mismatch.
# Assignment: modify file so it works for both mono and stereo wave files
#  (where does this file have an error when wave file is stereo and why? )
"""
Read a signal from a wave file, do amplitude modulation, play to output
Original: pyrecplay_modulation.py by Gerald Schuller, Octtober 2013
Modified to read a wave file - Ivan Selesnick, September 2015
"""

# f0 = 0      # Normal audio
f0 = 400    # 'Duck' audio

BLOCKSIZE = 64      # Number of frames per block

import pyaudio
import struct
import wave
import math

# Open wave file (mono)
wave_file_name = 'author.wav'
# wave_file_name = 'sin01_mono.wav'
# wave_file_name = 'sin01_stereo.wav'
wf = wave.open( wave_file_name, 'rb')
RATE = wf.getframerate()
WIDTH = wf.getsampwidth()
LEN = wf.getnframes()
CHANNELS = wf.getnchannels()

print 'The sampling rate is {0:d} samples per second'.format(RATE)
print 'Each sample is {0:d} bytes'.format(WIDTH)
print 'The signal is {0:d} samples long'.format(LEN)
print 'The signal has {0:d} channel(s)'.format(CHANNELS)

# Open audio stream
p = pyaudio.PyAudio()
stream = p.open(format = p.get_format_from_width(WIDTH),
                channels = 1,
                rate = RATE,
                 input = False,
                 output = True)

# Create block (initialize to zero)
output_block = [0 for n in range(0, BLOCKSIZE)]

# Number of blocks in wave file
num_blocks = int(math.floor(LEN/BLOCKSIZE))

# Initialize angle
theta = 0.0

# Block-to-block angle increment
theta_del = (float(BLOCKSIZE*f0)/RATE - math.floor(BLOCKSIZE*f0/RATE)) * 2.0 * math.pi

print('* Playing...')

# Go through wave file
for i in range(0, num_blocks):

    # Get block of samples from wave file
    input_string = wf.readframes(BLOCKSIZE)     # BLOCKSIZE = number of frames read

    # Convert binary string to tuple of numbers
    input_tuple = struct.unpack('h' * BLOCKSIZE, input_string)
            # (h: two bytes per sample (WIDTH = 2))

    # Go through block
    for n in range(0, BLOCKSIZE):
        # Amplitude modulation  (f0 Hz cosine)
        output_block[n] = input_tuple[n] * math.cos(2*math.pi*n*f0/RATE + theta)
        # output_block[n] = input_tuple[n] * 1.0  # for no processing

    # Set angle for next block
    theta = theta + theta_del

    # Convert values to binary string
    output_string = struct.pack('h' * BLOCKSIZE, *output_block)

    # Write binary string to audio output stream
    stream.write(output_string)

print('* Done')

stream.stop_stream()
stream.close()
p.terminate()
```

Additional demo programs are available on the course web page, include a program that shows how to read the input signal from the microphone. When reading the input signal from the microphone it is recommended that headphones be used to avoid feedback problems (sound passing from the speaker back into the microphone).

Exercises

1. **Write stereo wave file.** Write a Python program using PyAudio to generate a stereo audio signal with different sounds on the left and right channels. The two sounds can be produced by exciting two second-order systems (one for left channel, another one for right channel). The two second order

systems should generate decaying cosines with different frequencies and different decay-times. Use PyAudio to play the stereo audio signal as it is generated in real time, and in the same PyAudio program write the signal sound to a wave file. Use an audio player (e.g., QuickTime) to play the wave file. Verify that the wave files sound the same as the sound produced in real time using PyAudio.

2. The demo program `playAM_blocking_fix.py` reads a mono wave file, implements amplitude modulation, and plays the output audio signal. Modify this program so that it works for stereo wave files.

3. The demo program `recplayAM_mono.py` reads the input audio signal from the microphone, implements amplitude modulation, and plays the output audio signal. Make a stereo version of this program. Your program should use different modulation frequencies for the left and right channels.

2 Vibrato Effect

The following demo program is a simple implementation of the vibrato effect.

```
# play_vibrato_simple.py
# Reads a specified wave file (mono) and plays it with a vibrato effect.
# (Sinusoidal time-varying delay)
# This implementation uses a circular buffer with two buffer indices.
# No interpoltion..

import pyaudio
import wave
import struct
import math
from myfunctions import clip16

wavfile = 'author.wav'
print 'Play the wave file: {0:s}.'.format(wavfile)

# Open wave file
wf = wave.open( wavfile, 'rb')

# Read wave file properties
CHANNELS = wf.getnchannels()        # Number of channels
RATE = wf.getframerate()            # Sampling rate (frames/second)
LEN  = wf.getnframes()              # Signal length
WIDTH = wf.getsampwidth()           # Number of bytes per sample

print('The file has %d channel(s).'         % CHANNELS)
print('The file has %d frames/second.'      % RATE)
print('The file has %d frames.'             % LEN)
print('The file has %d bytes per sample.'   % WIDTH)

# Vibrato parameters
f0 = 2
W = 0.2
# W = 0 # for no effct

# Create a buffer (delay line) for past values
buffer_MAX =  1024                          # Buffer length
buffer = [0.0 for i in range(buffer_MAX)]   # Initialize to zero

# Buffer (delay line) indices
kr = 0  # read index
```

```
kw = int(0.5 * buffer_MAX)  # write index (initialize to middle of buffer)

# print('The delay of {0:.3f} seconds is {1:d} samples.'.format(delay_sec, delay_samples))
print 'The buffer is {0:d} samples long.'.format(buffer_MAX)

# Open an output audio stream
p = pyaudio.PyAudio()
stream = p.open(format      = pyaudio.paInt16,
                channels    = 1,
                rate        = RATE,
                input       = False,
                output      = True )

print ('* Playing...')

# Loop through wave file
for n in range(0, LEN):

    # Get sample from wave file
    input_string = wf.readframes(1)

    # Convert string to number
    input_value = struct.unpack('h', input_string)[0]

    # Compute output value - time-varying delay, no direct path
    output_value = buffer[int(kr)]  # use int() for integer

    # Update buffer (pure delay)
    buffer[kw] = input_value

    # Increment read index
    kr = kr + 1 + W * math.sin( 2 * math.pi * f0 * n / RATE )
        # Note: kr is not integer!

    # Ensure that 0 <= kr < buffer_MAX
    if kr >= buffer_MAX:
        # End of buffer. Circle back to front.
        kr = 0

    # Increment write index
    kw = kw + 1
    if kw == buffer_MAX:
        # End of buffer. Circle back to front.
        kw = 0

    # Clip and convert output value to binary string
    output_string = struct.pack('h', clip16(output_value))

    # Write output to audio stream
    stream.write(output_string)

print('* Done')

stream.stop_stream()
stream.close()
p.terminate()
```

This implementation is poor because the time-varying fractional delay is implementing by rounding the delay to an integer. For better audio quality interpolation is usually used instead, as illustrated in `play_vibrato_interpolation.py` available on the course web page.

Exercises

1. The demo program `play_vibrato_interpolation.py` does not use blocking (it reads and writes a single frame at a time). Write a version of this program that reads, processes, and writes the audio signal in blocks.

2. Modify the demo program `play_vibrato_interpolation.py` so that it reads and writes the audio signal in blocks (as in previous exercise), and takes the input signal from the microphone instead of a wave file.

3. Modify the demo program `play_vibrato_interpolation.py` so that it reads and writes the audio signal in blocks (as in previous exercise), takes the input signal from the microphone instead of a wave file (as in previous exercise), and produces a stereo output audio signal. The left and right channels of the output signal should have different vibrato parameters (frequency and amplitude).

4. Write a Python program to implement the flanger effect. As described in Chapter 2 of the text book, the flanger effect is like the vibrato effect but it additionally has a direct path. The input signal should be read from a wave file.

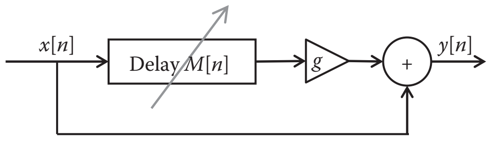

Figure 2.11
Block diagram of a basic flanger without feedback. The delay length $M[n]$ changes over time.

3 To Submit

Submit the following Exercises for this Lab:
Section 1) Exercise 3
Section 2) Exercises 3 and 4

Note: Submit all files as a **single** zip file to NYU Classes under the 'Lab4' assignment. All scripts, function definitions, wave files, and your written report document should be included in the zip file.

Name your main Python scripts:

`Lab4_Sec1_Ex3_NetID.py`

`Lab4_Sec2_Ex3_NetID.py`

`Lab4_Sec2_Ex4_NetID.py`

Please ensure your submitted codes can run independently of the system (Windows, Linux, etc.) and directory. Do not put absolute paths like `C:\\xxxx.wav` or `\usr\local\xxx` in your code.