

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).

```
1  # playAM_blocking_fix.py
2  # Play a mono wave file with amplitude modulation.
3  # This implementation reads and plays a block at a time (blocking)
4  # and corrects for block-to-block angle mismatch.
5  # Assignment: modify file so it works for both mono and stereo wave files
6  # (where does this file have an error when wave file is stereo and why? )
7  """
8  Read a signal from a wave file, do amplitude modulation, play to output
9  Original: pyrecplay_modulation.py by Gerald Schuller, October 2013
10 Modified to read a wave file - Ivan Selesnick, September 2015
11 """
12
13 # f0 = 0          # Normal audio
14 f0 = 400          # 'Duck' audio
15
16 BLOCKSIZE = 64    # Number of frames per block
17
18 import pyaudio
19 import struct
20 import wave
21 import math
22
23 # Open wave file (mono)
24 wave_file_name = 'author.wav'
25 # wave_file_name = 'sin01_mono.wav'
26 # wave_file_name = 'sin01_stereo.wav'
27 wf = wave.open( wave_file_name, 'rb')
28 RATE = wf.getframerate()
29 WIDTH = wf.getsampwidth()
30 LEN = wf.getnframes()
31 CHANNELS = wf.getnchannels()
32
33 print 'The sampling rate is {0:d} samples per second'.format(RATE)
34 print 'Each sample is {0:d} bytes'.format(WIDTH)
35 print 'The signal is {0:d} samples long'.format(LEN)
36 print 'The signal has {0:d} channel(s)'.format(CHANNELS)
37
38 # Open audio stream
39 p = pyaudio.PyAudio()
40 stream = p.open(format = p.get_format_from_width(WIDTH),
41                 channels = 1,
42                 rate = RATE,
```

```

43         input = False,
44         output = True)
45
46 # Create block (initialize to zero)
47 output_block = [0 for n in range(0, BLOCKSIZE)]
48
49 # Number of blocks in wave file
50 num_blocks = int(math.floor(LEN/BLOCKSIZE))
51
52 # Initialize angle
53 theta = 0.0
54
55 # Block-to-block angle increment
56 theta_del = (float(BLOCKSIZE*f0)/RATE - math.floor(BLOCKSIZE*f0/RATE)) * 2.0 * math.pi
57
58 print('* Playing...')
59
60 # Go through wave file
61 for i in range(0, num_blocks):
62
63     # Get block of samples from wave file
64     input_string = wf.readframes(BLOCKSIZE)      # BLOCKSIZE = number of frames read
65
66     # Convert binary string to tuple of numbers
67     input_tuple = struct.unpack('h' * BLOCKSIZE, input_string)
68     # (h: two bytes per sample (WIDTH = 2))
69
70     # Go through block
71     for n in range(0, BLOCKSIZE):
72         # Amplitude modulation (f0 Hz cosine)
73         output_block[n] = input_tuple[n] * math.cos(2*math.pi*n*f0/RATE + theta)
74         # output_block[n] = input_tuple[n] * 1.0 # for no processing
75
76     # Set angle for next block
77     theta = theta + theta_del
78
79     # Convert values to binary string
80     output_string = struct.pack('h' * BLOCKSIZE, *output_block)
81
82     # Write binary string to audio output stream
83     stream.write(output_string)
84
85 print('* Done')
86
87 stream.stop_stream()
88 stream.close()
89 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.

```

1  # play_vibrato_simple.py
2  # Reads a specified wave file (mono) and plays it with a vibrato effect.
3  # (Sinusoidal time-varying delay)
4  # This implementation uses a circular buffer with two buffer indices.
5  # No interpolation..
6
7  import pyaudio
8  import wave
9  import struct
10 import math
11 from myfunctions import clip16
12
13 wavfile = 'author.wav'
14 print 'Play the wave file: {0:s}'.format(wavfile)
15
16 # Open wave file
17 wf = wave.open( wavfile, 'rb')
18
19 # Read wave file properties
20 CHANNELS = wf.getnchannels()          # Number of channels
21 RATE = wf.getframerate()              # Sampling rate (frames/second)
22 LEN = wf.getnframes()                 # Signal length
23 WIDTH = wf.getsampwidth()             # Number of bytes per sample
24
25 print('The file has %d channel(s).' % CHANNELS)
26 print('The file has %d frames/second.' % RATE)
27 print('The file has %d frames.' % LEN)
28 print('The file has %d bytes per sample.' % WIDTH)
29
30 # Vibrato parameters
31 f0 = 2
32 W = 0.2
33 # W = 0 # for no effect
34
35 # Create a buffer (delay line) for past values
36 buffer_MAX = 1024                    # Buffer length
37 buffer = [0.0 for i in range(buffer_MAX)] # Initialize to zero
38
39 # Buffer (delay line) indices
40 kr = 0 # read index

```

```

41 | kw = int(0.5 * buffer_MAX) # write index (initialize to middle of buffer)
42 |
43 | # print('The delay of {0:.3f} seconds is {1:d} samples.'.format(delay_sec, delay_samples))
44 | print 'The buffer is {0:d} samples long.'.format(buffer_MAX)
45 |
46 | # Open an output audio stream
47 | p = pyaudio.PyAudio()
48 | stream = p.open(format      = pyaudio.paInt16,
49 |                 channels    = 1,
50 |                 rate        = RATE,
51 |                 input       = False,
52 |                 output      = True )
53 |
54 | print ('* Playing...')
55 |
56 | # Loop through wave file
57 | for n in range(0, LEN):
58 |
59 |     # Get sample from wave file
60 |     input_string = wf.readframes(1)
61 |
62 |     # Convert string to number
63 |     input_value = struct.unpack('h', input_string)[0]
64 |
65 |     # Compute output value - time-varying delay, no direct path
66 |     output_value = buffer[int(kr)] # use int() for integer
67 |
68 |     # Update buffer (pure delay)
69 |     buffer[kw] = input_value
70 |
71 |     # Increment read index
72 |     kr = kr + 1 + W * math.sin( 2 * math.pi * f0 * n / RATE )
73 |     # Note: kr is not integer!
74 |
75 |     # Ensure that 0 <= kr < buffer_MAX
76 |     if kr >= buffer_MAX:
77 |         # End of buffer. Circle back to front.
78 |         kr = 0
79 |
80 |     # Increment write index
81 |     kw = kw + 1
82 |     if kw == buffer_MAX:
83 |         # End of buffer. Circle back to front.
84 |         kw = 0
85 |
86 |     # Clip and convert output value to binary string
87 |     output_string = struct.pack('h', clip16(output_value))
88 |
89 |     # Write output to audio stream
90 |     stream.write(output_string)
91 |
92 | print('* Done')
93 |
94 | stream.stop_stream()
95 | stream.close()
96 | 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.

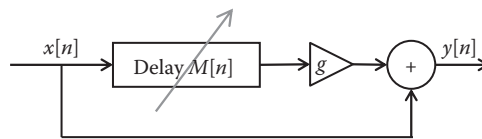


Figure 2.11
Block diagram of a basic flanger without feedback. The delay length $M[n]$ changes over time.

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.