Dyson School of Design Engineering

Imperial College London

DE2 Electronics 2

Lab 6: Beat Detection

(webpage: http://www.ee.ic.ac.uk/pcheung/teaching/DE2_EE/)

Introduction

In this lab, you will experiment with moving average filter and learn how to perform beat detection to live music.

Task 1: Moving Average filter

Task 1 will be conducted using Matlab alone. The goal is for you to explore the lowpass filtering effect of the moving average filter.

Download the music file: **bgs.wav** from the course webpage. This is a short segment of music from "Staying Alive".

Create the following Matlab script as **lab6task1a.m**.

```
clear all
[sig fs] = audioread('bgs.wav');
% Add noise to music
x = sig + 0.2*rand(size(sig));
% Plot the signal
figure(1);
clf;
plot(x);
xlabel('Sample no');
ylabel('Signal (v)');
title('Stay Alive Music');
% Filter music with moving average filter
N = size(x);
for i=4:N
    y(i) = (x(i)+x(i-1)+x(i-2)+x(i-3))/4;
end
y(1)=x(1)/4;
y(2)=(x(2)+x(1))/4;
y(3)=(x(3)+x(2)+x(1))/4;
% Play the original & then the filtered sound
sound(x, fs)
disp('Playing the original - press return when finished')
pause;
sound(y, fs)
disp('Playing the filter music')
```

When you compare the noise-corrupted music with the filtered version, you should notice a slight reduction in the noise.

Modify **lab6task1a.m** to **lab6task1b.m** so that you can use a variable number of taps. Change this to 10, 20 and 50. Comment on how the filtered change the music.

Task 2 – Exploring MICROPHONE Class

Download from the course webpage the file 'audio.py.zip' and unzip this onto the SD Card. This package defines a MICROPHONE class. Its function is to capture 160 samples of audio signal from the microphone using interrupt, put it in a buffer, and at the same time computes the instantaneous energy E.

Study the code in this file carefully and make sure you know how to write a class. Here are some explanations.

```
12  from array import array
13  import micropython
14  micropython.alloc_emergency_exception_buf(100)
15
16  class MICROPHONE(object):
17      def __init__(self, timer, mic, N):
18
19          # initialise variables used for beat detection
20          self.s_buf = array('h', 0 for i in range(N))    # reserve space for samples
21          self.count = 0  # sample buffer index pointer
22          self.E = 0
23          self.sum = 0
24          self.mic = mic
25          self.N = N
26          self.buffer_full = False
27
28          # Specify timer 7 interrupt service routine
29          timer.callback(self.isr_sampling)
```

Line 14: required by uPy when using interrupts

Line 17: This class takes three parameters: a timer object, a microphone object and number of samples

Line 20: Reserve memory for s_buf array which is signed short integer ('h') of length N and initialize it to 0. Always re-allocate storage if you know its size – much faster.

Lines 21 to 26: Define and initialize variable used within this class and its methods.

Line 29: Make the timer generate interrupts and specify isr_sampling is the ISR.

```
31      # Interrupt service routine to fill sample buffer s_buf
32      def isr_sampling(self,dummy):  # timer interrupt at 8kHz
33          MIC_OFFSET = 1523  # ADC reading of microphone for silence
34          s = self.mic.read() - MIC_OFFSET  # read one sample and remove dc offset
35          self.s_buf[self.count] = s
36          self.count = self.count + 1  # increment sample count
37          self.sum = self.sum + s * s
38          if self.count == self.N:  # when reach N
39              self.count = 0
40              self.E = self.sum
41              self.sum = 0
42              self.buffer_full = True
```

This is the interrupt service routine. We need a dummy variable 'dummy' (you can give it any name).

Line 33: MIC_OFFSET is the value when microphone has no signal. It is around 1523.

Line 34 to 37: fill the buffer, increment the pointer 'count' and compute the sum-of-square

Line 38 to 42: When we have 160 sample values, we save a the instantaneous energy in E, reset everything, set the buffer_full flag, and finish the interrupt.

```
44      def inst_energy(self):
45          return self.E
46
47      def buffer_is_filled(self):
48          return self.buffer_full
49
50      def reset_buffer(self):
51          self.buffer_full = False
52
53      def data(self):
54          return self.s_buf
```

The rest of the code define four methods used externally:

inst_energy – return the instantaneous energy value for 160 sample values.

buffer_is_filled – the flag (semaphore) to indicate that the latest instanteneous energy value is available.

reset_buffer – lower the flag for another instantaneous energy value to be read.

data – return the address of the data buffer

Create the file **lab6task2a.py** with following uPy code. It demonstrates how the MICROPHONE class can be used. Modify user.py to execute this script. In the terminal window, type CTRL-D to restart PyBench.

```
from audio import MICROPHONE
import pyb
from pyb import ADC, Pin, Timer

mic = ADC(Pin('Y11'))
sample_timer = Timer(7, freq = 8000)

audio = MICROPHONE(sample_timer, mic, 160)
```

You should see the **REPL >>>** because this code snippet only creates the microphone object '**audio'**. It looks as if nothing is happening, because it hands back control to you with >>>.

Now try typing: **audio.data()**. You will see the data buffer contents printed on the terminal. Use the up-arrow key to recall the last command and type RETURN. You can see that the contents of the data buffer is changed.

Similarly, try **audio.inst_energy()** and change the volume of music you play on your phone to the microphone and see the difference.

LESSON HERE: Once you create the MICROPHONE Class object '**audio**', the timer interrupt will continuously capturing new audio samples at 8kHz rate, filling the buffer and updating the instantaneous energy reading. The main program can do something else, or, in this case NOTHING!

Optional exploration:

lab6task2b.py is provided in the solution folder. This adds a small piece of code to **lab6task2a.py** and plot the contents of the captured data on the OLED display. It serves no other purpose than to reassure you that indeed the timer interrupt is working proper. A new block of 160 data values is being captured continuously and is being displayed. Note that OLED display is very slow. So only some of the data is being displayed.

Task 3 – Basic Beat Detection

In this lab, you are provided with a skeleton program "**beat_detect_0.py**" for detecting beats in real-time running on Pybench (in MicroPython). This program works reasonably well for the music "Staying Alive". Your job is to try to improve this basic program to obtain a better performing one.

CONTEXT

Our goal is to run **real-time code** in MicroPython using **Pybench** to detect when a beat occurs. In this version, the blue LED is flashed whenever a beat is detected. You can substitute flashing the LED with a dancing step (or do both!) in one of the later challenges.

Debugging interrupt driven program is difficult. In the past, I found that some students are struggling to get a basic version of the code running without error on **Pybench**. Given the number of deadlines you have, I decided to provide you with this "basic" version of code from which you can learn. Your challenge is to make my implementation better.

You can download this program from the course webpage.

EXPLANATION

You should be able to work out the code up to line 75. Lines 76 onward is where beat detection occurs.

```
65  # Calculate energy over 50 epochs, each 20ms (i.e. 1 sec)
66  M = 50                      # number of instantaneous energy
67  BEAT_THRESHOLD = 2.0        # threshold for c to indicate a
68  MIN_BEAT_PERIOD = 500   # no beat less than this
69
70  # initialise variables for main program loop
71  e_ptr = 0                   # pointer to energy buffer
72  e_buf = array('L', 0 for i in range(M)) # reserve storage fo
73  sum_energy = 0              # total energy in last 50 epochs
74
75  tic = pyb.millis()          # mark time now in msec
76
77  while True:             # Main program loop
78      if audio.buffer_is_filled():        # semaphore signal f
79
80          # Fetch instantaneous energy
81          E = audio.inst_energy()         # fetch instantenous
82          audio.reset_buffer()            # get ready for next
83
84          # compute moving sum of last 50 energy epochs with c
85          sum_energy = sum_energy - e_buf[e_ptr] + E
86          e_buf[e_ptr] = E            # over-write earliest en
87          e_ptr = (e_ptr + 1) % M     # increment e_ptr with w
88          average_energy = sum_energy/M
89
90          # Compute ratio of instantaneous energy/average ener
91          c = E/average_energy
92
93          if (pyb.millis()-tic > MIN_BEAT_PERIOD):    # if lon
94              if (c>BEAT_THRESHOLD):      # look for a beat
95                  flash()                 # beat found, flash
96                  tic = pyb.millis()      # reset tic
97          buffer_full = False             # reset status flag
```

Line 66: M is the number of instantaneous energy values to average over to obtain the average local energy.

Line 67: **BEAT_THRESHOLD** is the ratio of *instantaneous energy / local average energy* beyond which a beat is detected.

Line 71: **e_ptr** is the index for a buffer storing M instant energy values.

Line 72: **e_buf** is the instant energy buffer of length M. Data format is a regular unsigned integer. '**L**' is normal integer, i.e. 32-bits, uppercase is unsigned.

Lines 77 – 97: Main program loop. This is what all real-time program would look like. It loops around forever.

Line 78: The time it takes to go around the loop once is determined by the **audio.buffer_is_filled()** flag, which is set in the sampling interrupt service routine once the buffer is full. The buffer has N=160 locations, and the sampling period is 1/8000 = 125 μsec. Therefore, the loop goes around once every 20msec.

Line 81: Fetch energy in sample buffer – one epoch. This returns the instantaneous energy E.

Line 85: This is a clever trick! We want to find the average energy of the past M instant energy values. We could do this by summing up what's stored in **e_buf[0]** to **e_buf[M-1]**. That takes M-1 adds. However, we can also keep a running sum of instant energy **sum_energy**, take away the earliest instant energy value, and then add the current **E**. This takes only two adds (or subtract) – much quicker!

Line 86: Overwrite the earliest sample in buffer with this new instantaneous energy **E. e_ptr** is pointing to (i.e. providing the index for) the oldest sample in **e_buf[].**

Line 87: Update **e_ptr** to move to the next oldest sample, soon to be overwritten. The **"% M**" operation is modulo M (divide by M and get the remainder). It is a method to increment the index value, make sure that this value stay within 0 to M-1, and wrap it around whenever it reaches M. In that way, **e_buf[ ]** will always have the past M instant energy values, and this buffer get updated each echo (20msec) period.

Line 91: Calculate the ratio **c**, *instantaneous energy / average energy*. **sum_energy** has the total energy over 50 epochs. **sum_energy/M** is the average.

Line 93: Check that the elapsed time is 500msec or more since detecting the last beat. We know that "Staying Alive" has a beat period of around 570msec from your MATLAB analysis. So we only expect the next beat 500msec or later.

Line 94: Beat is detected only if *c > some threshold*. Change the threshold will affect accuracy of detection.

Line 97: Reset the **buffer_full** status flag, ready for another 20msec period

After understanding the beat_detect_0.py file, you should explore the various values such and music, to see how robust this beat detection algorithm is. Feel free to improve upon this.

WHAT NEXT?

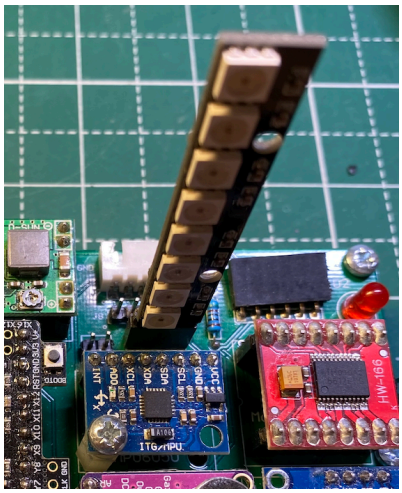
Some of the challenges you may choose to attempt is to make the mini-Segway dance to music, you would need to have created the dance routine in the form of steps encoded in ASCII characters. The dance routine can be created manually or automatically. You can then replace "**flash()"** with the appropriate function to move the mini-Segway.

You will also need to make your own improvised stabilizer for the mini-Segway. With the stabilizer installed, your Segway will be able to dance to the music in real-time.

For a different song, the beat period would be different. You would need to change the program so that it looks for a beat earlier or later than 500msec in the current basic program.

Experiment your beat detection algorithm on different songs.

Task 4 - First Challenge: the Dancing light show



If you have time, try Challenge 1 now. You are provided with the Neopixel strip. Plug this into the 4-way socket as shown below.

The following code snippet demonstrates how you can use the Neopixel Class to control the LEDs:

```
from neopixel import NeoPixel
import pyb
from pyb import Pin

# create neopixel object
np = NeoPixel(Pin("Y12", Pin.OUT), 8)
for i in range(8):  # all LEDs dark
    np[i] = (0, 0, 0)
    np.write()
    pyb.delay(1)

for i in range(8):  # turn LEDs red one at a time
    np[i] = (64, 0, 0)
    np.write()
    pyb.delay(300)

for i in range(8):  # turn LEDs blue one at a time
    np[i] = (0, 0, 64)
    np.write()
    pyb.delay(300)

for i in range(8):  # turn all LEDs off
    np[i] = (0, 0, 0)
    np.write()
    pyb.delay(1)
```

The challenge is for you to program the neopixel strip to “dance” to live music, and synchronized to the beats of the music.