Lab 4

Lab 4 - Pitch Detection

Summary

In this lab, you will learn how to detect the pitch of a signal in real time via autocorrelation. An example of the final solution can be found here. Next lab will utilize this pitch detector in order to do pitch synthesis a la Auto-Tune.

Downloads

- test_vector.wav
- Android project source code

Python

As in Lab 3, your task for the Python portion of this lab will be to prototype your Android system. For every frame of audio data, you will determine whether the frame contains background noise or a voiced signal. You will then compute the autocorrelation using a fast frequency-based method. Finally, you will determine the pitch from the autocorrelation output. These blocks are described in further detail below.

> **Note**
> For the sake of completeness, the notes from the prelab describing voiced/unvoiced detection and autocorrelation are reproduced below.

Part 1 - Voiced/Unvoiced Detector

Voiced/unvoiced signal classification is an incredibly well-studied field with a number of vetted solutions such as Rabiner's pattern recognition approach or Bachu's zero-crossing rate approach. Pitch shifting (next lab) does not require highly-accurate voiced/unvoiced detection however, so we will use a much simpler technique.

The energy of a signal can be a useful surrogate for voiced/unvoiced classification. Put simply, if a signal has enough energy, we assume it is voiced and continue our pitch analysis. The energy of a discrete-time signal is given as follows:

$$E_s = \sum_{n=-\infty}^{\infty} |x(n)|^2 \tag{1}$$

In this block, you will have to determine a useful threshold for E_s and classify frames as voiced or unvoiced depending on E_s.

Part 2 - Autocorrelation

Autocorrelation is the process of circularly convolving a signal with itself. That is, for a real signal, the discrete autocorrelation is given as:

$$R_{xx}[l] = x[n] \circledast \tilde{x}[-n], \tag{2}$$

where $\tilde{x}[-n]$ is the complex conjugate of the time reversal of $x[n]$. The output $R_{xx}[l]$ measures how self-similar a signal is if shifted by some lag l. If normalized to 1 at zero lag, this can be written equivalently as:

$$R_{xx}[l] = \frac{\sum_{n=0}^{N-1} x[n]x[n-l]}{\sum_{n=0}^{N-1} x[n]^2} \tag{3}$$

For a periodic signal, the lag l that maximizes $R_{xx}[l]$ indicates the frequency of the signal. In other words, the signal takes l samples before repeating itself. This algorithm, combined with some additional modifications to prevent harmonics from being detected, comprises the most well-known frequency estimator for speech and music.

Big O Notation

Unfortunately, computing R_{xx} in the time domain is an extremely slow operation. The complexity in Big O Notation is $O(N^2)$, meaning that if a signal has N samples, computing the autocorrelation will require N^2 operations. The real-world ramifications of this is that computing the autocorrelation will be too slow to fit in our timing window.

Fortunately, there is a way around this. Recall that convolution in time is the same as multiplication in frequency. Convolution is generally an $O(N^2)$ operation, but computing the FFT of a signal is only $O(N \log N)$. Converting to the frequency domain then, we have the following equation:

$$R_{xx}[l] = \mathcal{F}^{-1}\left\{\mathcal{F}\{x\}\,\tilde{\mathcal{F}}\{x\}\right\}, \tag{4}$$

or in pseudocode, `autoc = ifft(fft(x) * conj(fft(x))`. This is an enormously powerful result. Computing the FFT or inverse FFT is only $O(N \log N)$, and element-wise multiplication only $O(N)$. We can reuse the result of one of the FFTs to only require one FFT, one inverse FFT, and one full multiplication. Our total computational complexity is then O(2 N
log N + N). Big O notation typically drops all but the dominant term, so our final cost is considered $O(N \log N)$.

> **Note**
> For visualization, the table below demonstrates the growth in N of the two Big O costs.

N	$O(N^2)$number of operations	$O(N \log N)$number of operations	Ratio of $O(N \log N)$ to $O(N^2)$
10	100	10	0.1
100	10000	200	0.02
1000	1000000	3000	0.003
10000	100000000	40000	0.0004
100000	10000000000	500000	0.00005
1000000	1000000000000	6000000	0.000006

For our frame length of N = 2048 , we get a speedup of around 600x (approximately) using the frequency domain method.

Part 3 - System Requirements

The Python test code has been given to mimic the final Android system. You will get an input frame of size 2048, corresponding to about 40 ms of data, and you will determine the pitch (if it is voiced). If the pitch is unvoiced, you should return -1. Think about the following questions.

Question

- The autocorrelation for speech signals will be periodic with many candidate peaks. How do you decide which peak to use?
- The autocorrelation for any signal will be maximal in the neighborhood surrounding zero lag. How do you decide what to ignore?
- Why did we choose 40 ms frames?

Assignment 1
Implement the Python system described above, and plot the output frequency over time. Also compute

> the output frequency for `N` equals 512, 1024, 2048, 4096, and 8192. What is the resolution and detectable frequency range (minimum and maximum pitch) for a given `N`? These plots and discussion on resolution/range will count for 2 demo points.
>
> Do not rely on built-in functions such as `np.correlate()`, and don't worry about frame boundaries.

COPY

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import read, write
from numpy.fft import fft, ifft

FRAME_SIZE = 2048

################## YOUR CODE HERE ######################
def ece420ProcessFrame(frame, Fs):
    freq = -1

    return freq


################# GIVEN CODE BELOW ####################

Fs, data = read('test_vector.wav')

numFrames = int(len(data) / FRAME_SIZE)
frequencies = np.zeros(numFrames)

for i in range(numFrames):
    frame = data[i * FRAME_SIZE : (i + 1) * FRAME_SIZE]
    frequencies[i] = ece420ProcessFrame(frame.astype(float), Fs)

plt.figure()
plt.plot(frequencies)
plt.axis('tight')
plt.xlabel('Frame idx')
plt.ylabel('Hz')
plt.title('Detected Frequencies in Hz')
plt.show()
```

Android

Part 4 - Pitch detection in Android

If you've implemented your Python system without relying on many built-ins, it should translate nicely to Android. Your code will reside in `cpp/ece420_main.cpp:ece420ProcessFrame()` as before. The library `ece420_lib.cpp` contains some functions that you may find useful. Namely, `findMaxArrayIdx()` will return the index of the maximum value in a `float` array, given some minimum index (inclusive) and some maximum index (exclusive). For example:

```
float arr[10] = {8, 2, 3, 4, 5, 4, 3, 2, 1, 10};

// Finds the index of the maximum value only considering indices 3, 4, 5, 6
// Remember, C++ and Python are both zero-indexed
int maxIdx = findMaxArrayIdx(arr, 3, 7);

// maxIdx == 5
```

COPY

> **Assignment 2**
> Implement the block diagram described in the Python section in `cpp/ece420_main.cpp:ece420ProcessFrame()`. At the end of each processing block, write your detected frequency to `lastFreqDetected`. If the buffer was unvoiced, write `-1` to `lastFreqDetected`.
>
> The Android implementation will count for two demo points. The following will be considered when grading:
>
> - Is the voiced/unvoiced detection reasonable?
> - Are you accurately detecting the fundamental frequency and not the harmonics?

Grading

Lab 4 will be graded as follows:

- Prelab: 2 points
- Quiz: 2 points
- Lab: 4 points
  - Python
    - ***Assignment 1*** *–> 2 points*
      - *Voiced/Unvoiced detection (0.5 point)*
      - *Autocorrelation implementation (1 point)*
      - *Short answer question (0.5 point)*
  - Android
    - ***Assignment 2*** *–> 2 points*
      - *Voiced/Unvoiced detection (0.5 point)*
      - *Autocorrelation implementation (1 point)*
      - *Full functionality and app demo (0.5 point)*
