Introduction

In this assignment, we will be looking at some of the basic 1-D signal transforms we've seen in class. The goal of this assignment is to get a practical feel for what these transforms tell us about the data. We will be focusing on the Short-Time Fourier Transform (STFT) and the Haar transform, and applying these to some real-world signals including your personal music collection, and accellerometer data collected from a smartphone.

Completing the assignment

This assignment can be done individually or in pairs (though we strongly encourage you to work in pairs). Programs should be in Python or Matlab, though Python is preferred — all the supporting code to handle I/O and display that we're providing is in Python.

The example code also uses the SciPy library (`http://www.scipy.org`), which provides Matlab-like arrays with similar functionality. It also has some handy functions for plotting graphs of signals, which the Python standard library lacks. These operations tend to make signal processing algorithms much easier to code. You are not required to use this library, but you may have to re-write some of the support code we're providing. So be warned that you may be making your life unnecessarily difficult.

To install on Ubuntu Linux, simply do

```
sudo apt-get install python-numpy python-scipy python-matplotlib ipython \
    ipython-notebook python-pandas python-sympy python-nose
```

On Windows or Mac, you may have to install a Python distribution containing SciPy, which you can find at `http://www.scipy.org/install.html`.

For submission, package up your code as a zip or tar file. Include your written answers as a pdf or doc file named `writeup.[pdf|doc]`. There's a bunch of graphs we want you to plot, include these in your writeup, and please label them so we know which figures go with which subproblem.

If you have any questions about this assignment, please contact the TA (`afix@cs.cornell.edu`).

1 Haar Wavelet Transform

Recall the Haar transform from class, which computes averages and differences of larger and larger windows of the signal. We want to use the Haar transform to analyze some real-world signals, so your first task is to implement it.

1. Write a function `haar(x)` which takes a signal `x`, given as a sequence of floating-point numbers, and returns the Haar transformed signal. Recall that the Haar transform only works on a signal which has length 2^n. For this part of the assignment, you can assume that `x` has length which is a power of 2. The transformed result `X` should have the same length as `x`.

2. Write a function `inverse_haar(X)` which reverses the Haar transformation. You can also assume that `X` has length which is a power of 2.

2 Accelerometer data

We are going to use the Haar transform to analyze the signal from a smartphone accelerometer as the user performs various actions. For those of you in Deborah's Mobile Systems course, this is the data you'll be collecting in that course's Lab 1, and you're encouraged to use the results from that for this experiment. For

those not taking that course (and also so that you can start working on this assignment before finishing the other) some example outputs are provided on the course github.

The example code in `accel_example.py` loads an accelerometer reading from a file, and graphs the signal from one direction of motion. You should use this code as a starting point.

2.1 Haar transformed data

First, some basic applications of the Haar transform to this data.

1. Use the function you wrote in Problem 1 to transform the data. Recall that `haar` requires input which has length 2^n, so you should first trim the input to the largest 2^n-sized subsequence.[1]

2. Graph the result using `graph_accel`.

2.2 Haar transforms and "edges"

One particular reason for using the Haar transform over other wavelet transforms is that the basic operation it uses is taking pairwise differences. Pairwise differences will be large when the signal is changing rapidly, and small when the signal is constant, so we can use the Haar transform to find rapid changes in the signal, or "edges".

First, look at just the second half (i.e., the last 2^{n-1} entries) of the Haar-transformed signal. These correspond to pairwise differences of individual entries of the original signal. These are typically referred to as the first-order differences, so we'll denote this by X^1.

Write a function to find the 5 biggest positive entries of the pairwise differences X^1, and the 5 most negative entries. Briefly (one or two sentences) describe what is happening to the original signal at the locations corresponding to these points.

2.3 Smoothing

Loosely speaking, the second half of the Haar transformed signal captures rapid changes in the original, while the earlier entries capture more slowly varying changes. This suggests that we can get a smoothed version of the input by ignoring the information in the second half of the Haar-transformed signal.

In particular, we already have a way of inverting the Haar-transform, the function `inverse_haar` that you wrote earlier. So, we can just throw away the information corresponding to rapid changes, and take the inverse transform of the result. Let's see what happens when we do this.

1. First, take the original signal `x` and compute its Haar-transform `X`. Call `inverse_haar(X)` and graph the result. Compare the result to the original signal (hint: they should be identical).

2. Start with the Haar-transformed signal `X`, and truncate `X` to the first 2^{n-1} entries. Take the inverse Haar-transform of this truncated `X`, and graph the result.

3. Do the same thing with the first 2^{n-k} entries, for $k = 2$, 3 and 4, graphing each result. Briefly describe (1 to 2 sentences) what happens as k increases.

[1] I.e., if the input is `[1,2,3,4,5,6]` then you should trim the input to `[1,2,3,4]`.

3	5	1	7	2
1	8	10	7	1
5	11	20	17	0
7	18	12	13	3
3	10	8	3	1

Figure 1: A point (t, f) in the transformed signal (shaded) and the 5×5 window around it. The shaded entry is a peak, since it is larger than all other entries in the window.

3 Audio signals

In this part, we'll be looking at spectrograms and audio signals. Recall that spectrograms give the amplitudes of various frequencies in a signal, so are a natural tool for analyzing audio signals.

The main tool for computing a spectrogram is the Short-Time Fourier Transform (STFT). The STFT operates on a signal represented as a list of floating-point numbers. For example, for an audio signal, these numbers give the amplitude of the sound wave at each time point. The result of the STFT is an array `X` where `X[s,f]` is the response of frequency f at the window starting at time t.

We've provided you with code to compute the STFT of a signal, and to render the STFT as a spectrogram. There is also a handy function in the `scipy` library for loading `.wav` files into `scipy` arrays: `scipy.io.wavfile.read`. The usage of these functions is demonstrated in `example.py`, which you should feel free to use as a starting point for your solution.

For this assignment, you can use any song from your personal library. You'll have to convert it from `.mp3` to `.wav` first, but there are several free tools for doing this on the internet. If you can't find one, our favorite track is linked to from the course github, at `example.wav`.

3.1 Music Fingerprinting

Besides being an interesting graphical representation of an audio signal, the spectrogram can also be used for uniquely identifying a recording, or *fingerprinting*. For a concrete application, think of the music-identification app Shazaam. Unfortunately, the full STFT of the signal is much too much data to use as a concise identifier, and also changes significantly when there is noise in the signal (i.e., in the Shazaam example, a cell-phone microphone recording a song playing over a crowded room).

Since the full STFT is too large, we'll try to find some interesting features of the transformed signal, and see if we can use those for identification. In particular, we're going to be looking at the *peaks* of the transformed signal. A peak is a time-frequency pair (t, f) which has bigger value `X[s,f]` than any other (t', f') which is nearby. By nearby, we mean all the other points in a $k \times k$ grid around (t, f). See Figure 1 for an illustration.

1. Write a function which computes all the peaks of a transformed signal X which are the biggest values in a 20×20 surrounding grid.

2. Using the function `plot_peaks` in the example code provided, plot the peaks and spectrogram of a 10 second clip of audio of your choosing.

3. We want the pattern of peaks to be the same, even if there is noise or other distractions in the signal. We've provided you with a recording of our example track mixed in with a recording of a noisy room. Plot the spectrogram, and peaks for both the original `example.wav` and noisy version `noisy.wav`. Write a couple sentences comparing these fingerprints qualitatively. Are the patterns of peaks similar?

3.2 Haar transformed audio?

This section examines why it's important to pick the right transform for your data. We'll try doing some of the same excercises using the Haar transform instead of the STFT.

1. Write a function `short_time_haar`. This function is analogous to the STFT, in that we break up the signal `x` into windows. We'll arbitrarily choose a window size of 4096 and shift of 2048. So, your windows should be

   ```
   x[0:4096], x[2048:2048+4096], x[4096:4096+4096]...
   ```

   On each window, compute the Haar transform to get a 2D array indexed by t and k where the `X[t,k]` is the kth Haar coefficient at window t.

2. For the same audio clip as Problem 2 above, plot this Short-time Haar transform, using `plot_transform`.

3. Explain what's going on. Write a few sentences (2-4) explaining why the STFT might be better for analyzing audio signals than the Haar transform.