Problem Set 6

Your answers will be graded by actual human beings (at least that's what we believe!), so don't limit your answers to machine-gradable responses. Some of the questions specifically ask for explanations; regardless, it's **always a good idea to provide a short explanation for your answer**.

Before doing this problem set, please read chapters 13 and 14 of the lecture notes.

Nota bene: This problem set is longer than usual, but you have two weeks to complete it. Please start early; don't wait until the day before. The programming is more involved than in the previous two PSets (though it's not onerous). The non-programming problems also require more time and attention.

Problem 1 - Properties of the DTFT. (2 points)

Given the discrete-time Fourier transform pair $x[n] \iff X(\Omega)$ find the DTFT (discrete-time Fourier transform) or inverse DTFT (as appropriate):

A. $x[n - n_0]$ (this is called the time-shift property).

(points: 0.5)

B. $X(\Omega - \Omega_0)$ (this is called the frequency-shift property).

(points: 0.5)

C. $(-1)^n x[n]$.

(points: 1)

D. $nx[n]$.

(points: 1)

Problem 2 - Spectral content of DT signals. (1 point)

Calculate the DTFT of

$$x[n] = u[n](-0.5)^n$$

directly from the definition of the transform. Is the signal content concentrated at low, mid-range, or high frequencies?

(points: 1)

Problem 3. (1 point)

Suppose the continuous-time signal $x(t)$ is sampled regularly every T seconds, yielding the discrete-time signal $x[n] = x(nT)$.

A. What is the sampling rate f_s (in samples/sec) expressed in terms of T?

(points: 0.5)

B. If $x(t) = \cos(2\pi f_0 t)$, so f_0 is the frequency of the cosine in Hz, then the sampled sequence $x[n]$ is also a cosine, of some angular frequency Ω_0 radians/sample, and $x[n] = \cos(\Omega_0 n)$. Express Ω_0 in terms of f_s and f_0. Also determine what f_0 is when $\Omega_0 = \pi$.

(points: 0.5)

Problem 4 - Kill the vuvuzela. (5 points)

If you watched any match of the 2010 World Cup in South Africa, you are intimately acquainted with the vuvuzela, a plastic trumpet that produces a low, obnoxious droning right around $f_0 = 235$ Hz. In this problem we'll design a digital audio filter that kills off sinusoids around 235 Hz but allows other sounds to pass through. Next time you watch a recording of a soccer game, you can use this filter to spare yourself a headache.

A. First, we'll need to convert an analog audio signal $x(t)$ into a digital signal $x[n]$ as follows:

$$x[n] = x(nT)$$

where T is the sampling interval, i.e., the interval **between** samples.

Television audio is recorded at a sampling frequency of $f_s = 48$ kHz. What angular frequency Ω (in radians/sample) corresponds to $f_0 = 235$ Hz?

(points: 0.5)

B. Consider an LTI low-pass filter (LPF) with frequency response

$$H(\Omega) = \begin{cases} e^{-30j\Omega}, & |\Omega| < \Omega_c \\ 0, & \text{elsewhere,} \end{cases}$$

where you are able to specify the cutoff frequency Ω_c. Denote the associated unit sample response by $h[n]$.

Calculate $h[n]$ for $\Omega_c = \pi/4$.

(points: 0.5)

C. Determine the magnitude and phase of the frequency response $G(\Omega)$ associated with an LTI filter whose unit sample response is

$$g[n] = \delta[n - 30] - h[n]$$

with $h[n]$ as in part (B), and with Ω_c at some arbitrary value between 0 and π. (It turns out to be simple to obtain such a filter, once one has a filter with the specifications in part (B)). Specify whether this is a low-pass filter, band-pass filter or high-pass filter.

(points: 1)

D. Determine the unit sample response $f[n]$ of a digital band-stop filter that knocks out components corresponding to frequencies between 230 and 240 Hz from continuous-time signals sampled at 48 kHz. You can do this by combining the filters from parts B and C, with appropriately chosen cutoff frequencies for each filter.

(points: 2)

E. Is your filter usable in real time? Explain.

(points: 0.5)

F. Watch a demo of this filter at work. The filtering scheme shown is slightly more complex than what you have worked out for two reasons:

First, it suppresses sounds around both 235 and 470 Hz. The former is termed the fundamental frequency; the latter is termed the second-order harmonic frequency of the instrument. We can think of the vuvzela itself as a filter with strong frequency response at these two frequencies. This is why an effective vuvuzela-stopping filter must eliminate sounds in the neighborhoods of 235 and 470 Hz.

Secondly, the demo you saw used a cascade of **non-ideal** filters with two band-stops at each of the offending frequencies. What is the advantage of chaining two identical band-stop filters in series?

(points: 0.5)

Problem 5. (3 points)

The Boston news radio station WEEI 850AM broadcasts on a carrier frequency of 850 kHz. Suppose its continuous-time (CT) carrier signal is $\sin(2\pi(850 \times 10^3)t)$, where t is measured in seconds. Denote the CT audio signal that's modulated onto this carrier by x(t), so that the CT signal transmitted by the radio station is

$$q(t) = x(t)\sin(2\pi(850 \times 10^3)t)$$

We use the symbols $q[n]$ and $x[n]$ to denote the discrete-time (DT) signals that would have been obtained by respectively sampling $q(t)$ and $x(t)$ in the equation above at f_s samples/second; more specifically, the signals are sampled at the discrete time instants $t = n(1/f_s)$. Thus

$$q[n] = x[n]\sin(\Omega_c n)$$

for an appropriately chosen value of the angular frequency Ω_c. Take $f_s = 4 \times 10^6$ samples/second, corresponding to a sampling frequency of 4 MHz.

A. Determine the value (in radians/sample) of Ω_c in the equation for $q[n]$ above, restricting your answer to a value in the range $[-\pi, \pi]$.

(points: .5)

B. Determine the range of Ω for which the DTFT of x[n] can be nonzero, assuming the Federal Communications Commision (FCC) insists that the transmitted signal be confined to a frequency band of 5 khz on either side of the 850 khz carrier, i.e., to the range from 845 khz to 855 khz. (Explain your reasoning briefly.)

(points: .5)

Assume now that the receiver detects the CT signal

$$r(t) = 10^{-3} q(t - t_0),$$

where $t_0 = 10^{-5}$ seconds, and that it samples this signal at f_s samples/second, thereby obtaining the DT signal

$$r[n] = 10^{-3} q[n - m] = 10^{-3} x[n - m] \sin(\omega_c (n - m)),$$

for an appropriately chosen integer m.

C. Determine the value of m in the equation for $r[n]$ above, explaining your answer.

(points: 0.5)

D. Noting your answer from part (B), determine for precisely which intervals of the Ω frequency axis the DTFT of the signal $q[n - m]$ in the equation for $r[n]$ above is nonzero. (Explain your reasoning briefly.)

(points: 1)

E. The demodulation step to obtain the DT signal $x[n - m]$ from the received signal $r[n]$ in the equation for $r[n]$ above now involves multiplying $r[n]$ by a DT carrier-frequency signal, followed by appropriate low-pass filtering. Which one of the following six carrier-frequency signals would you choose to multiply the received signal by? Give a brief explanation.

   a. $\cos(\Omega_c n)$
   b. $\cos(\Omega_c (n - m))$
   c. $\cos(\Omega_c (n + m))$
   d. $\sin(\Omega_c n)$
   e. $\sin(\Omega_c (n - m))$
   f. $\sin(\Omega_c (n + m))$

(points: 0.5)

Python Programming Tasks and Experiments

***Sharing is Caring*: Frequency-division Multiplexing for Audiocom**

Download Audiocom and unzip it in your working directory.

Reminder: The documentation for Audiocom is available at
http://audiocom602.blogspot.com.

Our goal is to develop modules in the Audiocom system to transmit and correctly receive multiple concurrent transmissions over the audio medium. There are three challenges that we will overcome in the following tasks:

1. **Noise and inter-symbol interference (ISI)**: We have dealt with these issues in the previous two problem sets. We will use eye diagrams to manually tune the system and pick an appropriate value for the number of samples per bit (see Task 0 below).
2. **Delay (lag)**: The use of a preamble to begin each packet transmission and the development of a robust method to detect a preamble in the presence of noise and channel distortions will allow us to solve one of the problems caused by an unknown delay (lag) between transmitter and receiver (Task 1). We will also need to handle the problem of an unknown phase difference caused by the unknown delay during demodulation (Tasks 3 and 4).
3. **Sharing**: To share the communication medium amongst different concurrent transmissions, one approach is to use different carriers. But how do we extract different transmissions sent at different frequencies? We need to develop a solution to this problem (Tasks 2, 3, and 4).

These challenges are tricky to overcome. They are hard enough on nice, well-behaved channels, but harder still if we were to try to get our solutions working over any real-world communication medium (in our case, audio) off the bat. A general approach to combat real-world complexities is to first design under a simplified simulation of real-world conditions. We will do that using the `BypassChannel`, which we introduced in earlier problem sets. By modeling the communication channel in terms of three basic abstractions -- noise, delay lag, LTI -- we can try out our ideas and test them under controlled conditions. After they work here, we will run them over real-world audio channels and see whether they need further tuning or refinement, and evaluate performance. Such a method is a good principle to adopt in general when confronted with the task of getting a real system working under complex conditions.

In the tasks below, we (you!) will implement functions that will yield a complete, working (we hope) frequency-division multiplexing (FDM) system for Audiocom.

Task 0 [from earlier PSets]: Noise and ISI with eye diagrams

To make sure that everything you did earlier to get the audio labs working continues to work, let's run the following two commands over the bypass channel (use your own implementation, or the one we have provided in bypasschannel.pyc -- remembering to link or rename the appropriate Python version of this file to bypasschannel.pyc). Pick a suitable value for the number of samples per bit (say, 256).

```
# python sendrecv.py -b -z .3 -f testfiles/A
```

```
# python sendrecv.py -b -z .3 -u '.9 .6 .3 .1' -f testfiles/A
```

The preamble should decode correctly and you should see the contents of the file `testfiles/a` printed on screen:

```
Mens et manus.
```

Next, run the following over the real audio channel:

```
# python sendrecv.py -f testfiles/A
```

Make sure to **include whatever other parameters you need to get your audio working, such as -q, -p, -i, -s, from your previous experience**.

If the preamble decodes and the message decodes correctly ("Mens et manus." FTW!), then you're in good shape. Make sure to save the relevant parameters someplace, so you can use them in the following tasks.

Task 1: Building your own preamble detector

Note: this task may not be easy to get working, despite the fairly detailed directions; if you get stuck, just move on to the later tasks below, using the preamble.pyc module we have provided. You can come back to using your own preamble.py implementation later.

The problem: The transmitter may start transmitting at any point in time. How does the receiver know when a legitimate transmission has begun? This problem is tricky because even when the transmitter is not sending anything, a receiver listening on the medium will acquire samples. We need a way to determine that a legitimate transmission has started.

Over many communication media, including audio and radio, this task is performed using a **preamble**. A preamble is a known sequence of bits with some nice properties; each bit is converted to samples_per_bit samples to produce the **preamble sample sequence**. The receiver's task is to look for a sample sequence that most closely resembles the known preamble sample sequence, using that to infer the beginning of a packet. The preamble helps synchronize the transmitter and receiver.

When the receiver starts running, in general it does not get the samples immediately because there is an unknown lag (in our lab, largely because of software overheads). Moreover, to ensure that the receiver is ready, we insert some number of "quiet" (zero) samples to ensure that the receiver does not miss the preamble of any packet.

Look at the file `preamble.py` -- it contains the **Preamble class**. This class defines the main part of the preamble, called the *Barker* sequence (in our case, it is the concatenation of two different Barker sequences, but this is a minor and irrelevant technicality). The Barket bit sequence is 24 bits long whose first bit starts with "`1`". The transmitter sends the corresponding sample sequence at the beginning of each transmission, usually after sending some number of 0-valued "quiet" bits (the number of quiet bits does not matter to the receiver, which should focus on looking for the preamble).

Your task is to write the body of `detect` in the Preamble class to determine the **index** in the numpy array of demodulated samples, `dsamples`, corresponding to the first **sample** of the preamble.

```
detect(self, dsamples, receiver, offset):
```

returns an integer index. The `offset` is a hint provided from the caller telling `detect` to start looking from `dsamples[offset]` for the preamble sequence.

There are many ways to design and implement `detect`. Here's one that we have found works well over the audio channel (and the bypass channel).

1. Convert the known Barker bit sequence, accessible using `self.barker()`, to a sample sequence. To do that, use the Mapper's `bits2samples` function in mapper.py. This function takes a sequence of bits and produces the corresponding numpy array of samples by expanding each bit to samples-per-bit samples. You can get to the Mapper object using `receiver.mapper`
2. Produce the **transmitted signal waveform** corresponding to the numpy array of samples built from the known Barker sample sequence. You can do that by multiplying the array of preamble samples with a local carrier at the carrier frequency. You may use the function `sendrecv.local_carrier` for this purpose (see sendrecv.py), passing to it the carrier frequency (`receiver.fc`), the length of the samples array corresponding to the Barker sequence, and the sampling rate of the system (`receiver.samplerate`). You should take a look at `sendrecv.local_carrier` and make sure you understand what it is doing.
3. Then, **demodulate** the result of the above step by calling `receiver.demodulate`, passing the waveform as an argument. What we're doing here is to capture any imperfections in the demodulation induced by things like the receiver filter.
4. Correlate the result of the previous step (the demodulation of the modulation of the preamble samples) with the received `dsamples` array. You may perform this step by writing a separate helper function, `correlate`, which takes two arguments, x and y, both numpy arrays, and returns the index in array y corresponding to the best occurrence of sequence x. You may assume that len(x) <= len(y), and return 0 if either len(x) > len(y) or if len(x) is 0. Compute this "best correlation" by computing the **maximum normalized dot product of x with all subsequences of y whose length is equal to the length of x**. The correlation between two equal-length numpy arrays, x and y, is defined as the **normalized dot product** between x and y, i.e., as (x dot y) / (norm(x)*norm(y)), where "dot" is the vector dot product. To compute the norm of a numpy array, feel free to use a built-in function from numpy's linear algebra library: `numpy.linalg.norm(y)`, where y is a numpy array.

   The function `correlate` should return the index (argmax) of the maximum correlation between u and all possible y's that are subsequences of v. For example:

   ```
   correlate([0.2, 0.4, 0.6, 0.8], [0.1, -0.05, 0.05, 0.08, 0.14,
    0.22, 0.4, 0.8, 0.1, 1.0]
   ```

   should return 2, because the subsequence [0.05, 0.08, 0.14, 0.22] has the highest correlation among all possible choices for x = [0.2, 0.4, 0.6, 0.8].

   Note that the array `dsamples` may be too long, and searching for the preamble samples through the entire array may be too slow. To speed-up this search, we don't need to search through the entire reception because we know that the preamble is somewhere near the beginning of the reception, once the intial "quiet" samples (which may have been corrupted by noise) have been dealt with. To help here, inside the main receive function, we have computed a likely offset where some non-zero energy corresponding to the first "1" might occur. Our recommendation is that you start your preamble search from dsamples[offset] and try to find the preamble by correlating over a number of samples equal to two or three times the length of the preamble's Barker sequence (self.barkerlen()*3*mapper.spb).

5. Finally, `detect` should return a value of `offset` + `index`, where `index` was the value returned by `correlate`

Upload your preamble.py: Browse...

(points: 4)

Task 2: Envelope demodulation with two transmissions?

Recall the modulation step: multiply the signal (after mapping bit "1" and bit "0" to the appropriate voltage levels, +V=1 volt for a "1" and 0 volts for a "0" in on-off keying) with a cosine carrier of a specified carrier frequency. We did this step to **match the signal sent to the characteristics of the communication channel**, which in our case of the audio channel is able to carry sinusoids across some frequency range. With envelope demodulation, all we do is to take the absolute value of the received samples and then run the result through a simple averaging filter. The averaging filter has a positive value whenever we detect a "1" sample, and is close to "0" when we detect a "0" sample. The final step is to take the average of the middle M bits of each bit, where M is one-half the value of the samples per bit. I.e., if the samples per bit is spb, we take the samples spb/4 to 3*spb/4 in each bit period and return the mean value as the bit.

The averaging filter averages a certain number of samples corresponding one-half of the sinusoid cycle. The number of samples depends on the carrier frequency.

Suppose we use this method (from PS4, or from demodulate_audiocom.pyc) for **two** concurrent transmissions. Will it work? To answer this question, run the following (the -G option specifies the frequency gap, in Hz, between the carrier frequencies of the concurrent transmissions, starting from the base specified in the -c option):

```
python sendrecv.py -f testfiles/A -f testfiles/B -c 1000 -G 800
```

What do you get? Did the method work? Explain what happened.

(points: 2)

Task 3: Heterodyned demodulation

We need a strategy to extract the messages modulated on different carriers at the receiver. A clever way to do that is with **heterodyned demodulation**: multiply the received samples by a local version of the same carrier sinusoid that was used by the transmitter. Let's first do that over the `BypassChannel`, and let's first get it to work for a single transmission.

In demodulate.py, fill in the body of **`heterodyne`** by multiplying the received samples by a **cosine** (the same as the carrier) with the specified carrier frequency. You can do this by calling `sendrecv.local_carrier` as explained in Task 1. Also fill in the body of the `avgfilter` function (which you figured out in PS4), averaging over the number of samples in a half-period of the carrier waveform. Note that some of the interfaces to these functions have changed in this PSet from before.

Then run the following:

```
# python sendrecv.py -f testfiles/B -b -z 0.1 -d het
```

You should see the contents of testfiles/B:

```
E pluribus unum.
```

If you aren't able to decode the preamble or reliably see the contents printed on screen, you could try using the `avgfilter` we have provided in demodulate_audiocom.pyc (rename or link to the appropriate Python version of this file). If that still doesn't work, you probably have a bug in the multiplication by the local carrier.

Now try running the heterodyned demodulator over the audio channel:

```
python sendrecv.py -f testfiles/B -d het
```

It probably won't work reliably. If it does, it was probably good luck, and we still need to understand why you got lucky...!

To understand what's going on, let's go back to our `BypassChannel`. Run the following four experiments, which add a delay lag (in number of samples) between transmitter and receiver, where X is the ratio of the sampling rate (default: 48000 per second) to the carrier frequency (default: 1000 Hz):

```
python sendrecv.py -f testfiles/B -b -z 0.1 -l 0 -d het

python sendrecv.py -f testfiles/B -b -z 0.1 -l X -d het

python sendrecv.py -f testfiles/B -b -z 0.1 -l X/2 -d het

python sendrecv.py -f testfiles/B -b -z 0.1 -l X/4 -d het
```

What can you conclude from these experiments about the heterodyned demodulator and why it does not always work (but sometimes does)? (To be sure, you may want to try this command with **our** implementation of BypassChannel, provided in bypasschannel.pyc.)

(points: 2)

Task 4: Quadrature demodulation

By now, you've probably understood that we need **quadrature demodulation** (see Chapter 14), and not a purely heterodyned demodulation, to handle an arbitrary delay lag between transmitter and receiver. Taking advantage of Python's natural support for complex numbers, we will multiply the received waveform by a complex exponential waveform at the carrier frequency.

We will also take advantage of the fact that we're using **on-off keying**, so after passing the multiplied result through a low-pass filter, we can take the absolute values of the complex numbers to produce the demodulated samples (which will therefore be real-valued).

Write the body of the following function:

```
def quadrature(samples, carrier_freq, config)
```

This function should simply multiply samples by the quadrature version of the local carrier and return the corresponding numpy array. The receive function in receiver.py then calls a filter. You can pick a filter using the -t option: `-t avg` is the default (averaging filter), while `-t sinc` (which you'll write later) picks a sinc low-pass filter.

Let's first investigate the averaging filter from before. Use only one transmission because we first have to show that this simple part works. Produce eye diagrams if you wish, to pick a good samples_per_bit. Run the following tests:

```
python sendrecv.py -f testfiles/B -b -z 0.1 -l X -d quad

python sendrecv.py -f testfiles/B -b -z 0.1 -l X/2 -d quad

python sendrecv.py -f testfiles/B -b -z 0.1 -l X/4 -d quad

python sendrecv.py -f testfiles/B -d quad
```

As before, X = sampling rate / carrier frequency.

Does your quadrature demodulator (with the above commands) correctly receive data? Explain why it works (or doesn't work).

(points: 2)

Now add in a second transmission:

```
# python sendrecv.py -f testfiles/A -f testfiles/B -b -z 0.1 -l X/4 -G
600 -d quad
```

Does this quadrature demodulator correctly receive data? What if you added a third transmission, and a fourth (just use testfiles/C, testfiles/D to add more files). Explain your answer. Your explanation should be informed by playing close attention to the **signal spectrum** arriving at the receiver and **after** passing through the quadrature demodulator and the averaging filter. You can produce these spectra by modifying sendrecv.py as follows: first, add the following at the top of the file:

```
import demodulate
```

Then add this code around line 206 (right before the line that reads `# process the received samples`):

```
if opt.demod == 'quad':
    plot_sig_spectrum(numpy.real(samples_rx), 'modulated samples')
    p.show()
    demod = demodulate.quadrature(samples_rx, opt.channel[0], opt)
    if opt.filter == 'sinc':
        plot_sig_spectrum(numpy.real(demod), 'demodulated samples')
    else:
        filtered = demodulate.avgfilter(demod, opt.channel[0], opt)
        plot_sig_spectrum(numpy.real(filtered), 'demodulated samples filtered by averaging')
    p.show()
```

(points: 2)

It is possible a better filter at the receiver will allow it to correctly extract only the part of the received signal intended for it, ignoring everything else. Each receiver doing that may allow for many concurrent receptions.

We have learned about such a filter: the **sinc LPF**. An ideal sinc has an infinite number of non-zero components in its unit-sample response, which is hard to implement on a computer in finite time. So we will build a sinc filter of some pre-defined length, say L=50.

Your job is to implement the following function:

```
    def lpfilter(samples_in, carrier_freq, config)
```

Here, `samples_in` is the numpy array of samples that we would like to apply the LPF to. The carrier frequency is carrier_freq, and config maintains all the other parameters of the experiment. Inside lpfilter, your first task should be to pick a good cutoff frequency for the LPF. You can approach this task with the following steps:

1. How should you pick the cutoff frequency of the sinc LPF? You need to set this to a value that will support multiple concurrent transmissions. Factors that you may want to consider include the frequency spectrum of the input signal (which depends on the samples per bit, `config.spb`), the sample rate (`config.samplerate`), and the frequency gap between successive carriers (`config.changap`). You don't necessarily need to incorporate all these factors in your cutoff frequency setting.
2. Generate the unit-sample response, h, for the sinc filter, of length L=50. Do that by creating a numpy array, h, with h[i] corresponding to the sinc filter of cutoff frequency omega_cut, for -50 ≤ i ≤ 50, including at i &eq; 0
3. Then, convolve `samples_in` with h, and return that result. You may implement the

convolution yourself or use `numpy.convolve`; the returned result should be a numpy array.

Upload your demodulate.py: Browse...

(points: 4)

The final question concerns an experiment where you "put it all together". Run your quadrature demodulator on multiple transmissions and see if you can demodulate properly. How many concurrent xmissions are you able to handle? You may send actual data files (using -f). When you run the following, how many of the concurrent transmissions is your system able to correctly detect and decode?

```
# python sendrecv.py -f testfiles/A -f testfiles/B -f testfiles/C -f
testfiles/D -f testfiles/E -f testfiles/F -G 600 -d quad -t sinc -b -z 0.2
```

And finally, what about:

```
# python sendrecv.py -f testfiles/A -f testfiles/B -f testfiles/C -f
testfiles/D -f testfiles/E -f testfiles/F -d quad -t sinc -G 600
```

Note that you may need to change the frequency gap to a different value in the -G option above, and may also need to increase samples_per_bit to a value larger than the default of 256.