Problem Set 4

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**.

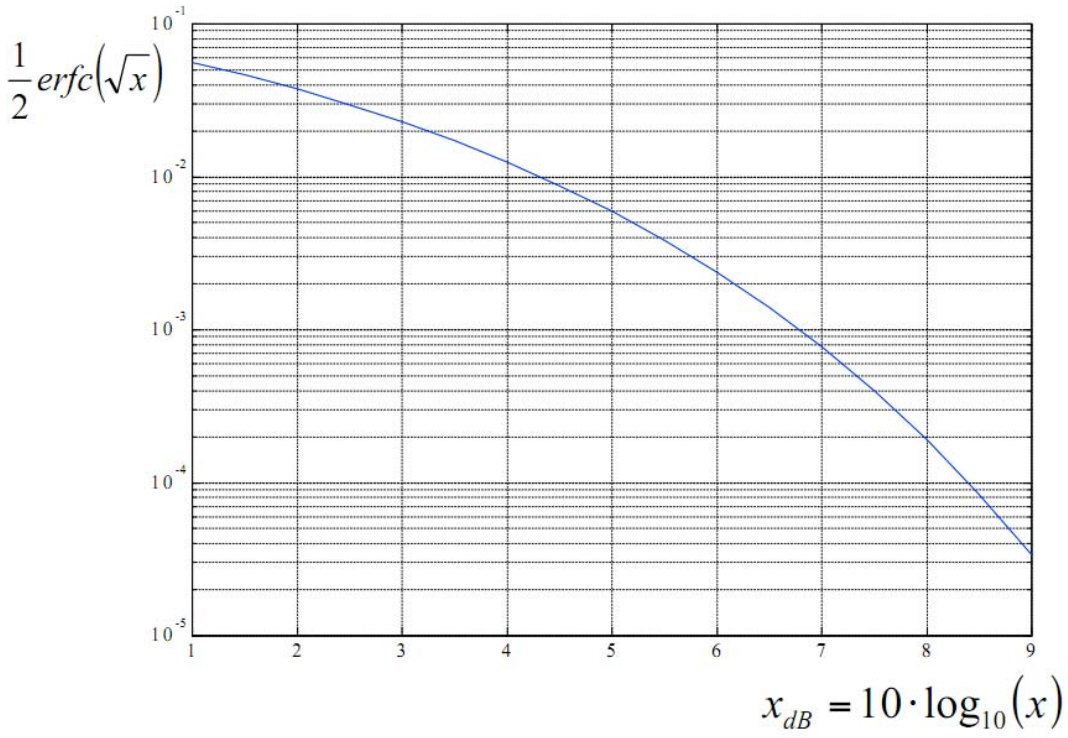
Before doing this PSet, please read Chapters 9 and 10 of the readings. Also attempt the noise and LTI practice problems on this material.

Problem 1. What's the SNR?

Ben Bitdiddle is running a communication link between a sender and receiver, transmitting a bit stream of equally likely "1"s and "0"s. His scheme uses a voltage of +sqrt(Es) to send binary digit "1", and a voltage of either 0 or -sqrt(Es) to send binary digit "0", depending on whether he uses on-off or bipolar signaling, respectively. Assume these voltage levels are preserved at the receiver in the absence of noise. The receiver takes a single sample in each bit slot, and uses a threshold halfway between the two nominal voltage levels for deciding whether a received sample is "1" or "0". Assume the samples are perturbed by additive white Gaussian noise (AWGN), with N_0 denoting twice the noise variance.

In this problem you will help Ben evaluate the two schemes.

You may use the picture shown below, which plots the function 0.5*erfc(sqrt(x)) as a function of x on the dB scale. You ought to be able to eyeball the result from the picture (a very useful skill for an engineer; it's the fastest way to answer this question), and then confirm that your answer is correct by typing in numerical formulas into WolframAlpha.



Courtesy of Roberto Gaudino. Used with permission.

(Pic from http://www.tlc.polito.it/~gaudino/foundations/didactical_material/erfc_curves/)

We may not have been as clear as we should have been about the definition of SNR in this problem. It turns out that there are many reasonable interpretations/definitions of SNR in different settings.

On p.118 of the notes, in connection with bipolar communication, we refer to E_s/N_0 as the SNR. The square root of this ratio is what appears in the erfc expression in Eq. (9.9) for the bit error rate, BER.

What we intended in this problem was for you to figure out how the BER expression changed in going from bipolar to on-off signaling. You should find that BER = 0.5 erfc(sqrt{something}) for on-off signaling. That "something" is what we were thinking of as the effective SNR for on-off signaling.

A. What happens to the SNR when Ben goes from on-off to bipolar signaling?

(points: 1)

B. Suppose we were operating at 2dB SNR with an on-off scheme. Read the graph to get an approximate BER.

(points: 0.5)

C. Use the graph again to figure out what the BER is for bipolar signaling under the same noise conditions as in part (B).

(points: 0.5)

D. Suppose in the on-off scheme considered in (B) we were to average 4 samples in each bit slot before comparing to a threshold. The AWGN assumption guarantees that noise samples are independent on each sample. What is the BER now?

(points: 2)

Problem 2. Attenuation

The cable television signal in your home is poor. The receiver in your home is connected to the distribution point outside your home using two coaxial cables in series, as shown in the picture below. The power of the cable signal at the distribution point is P. The power of the signal at the receiver is R.

The first cable attenuates (i.e., reduces) the signal power by 7 dB. The second cable attenuates the signal power by an additional 13 dB. Calculate P/R as a numeric ratio.

(points: 1)

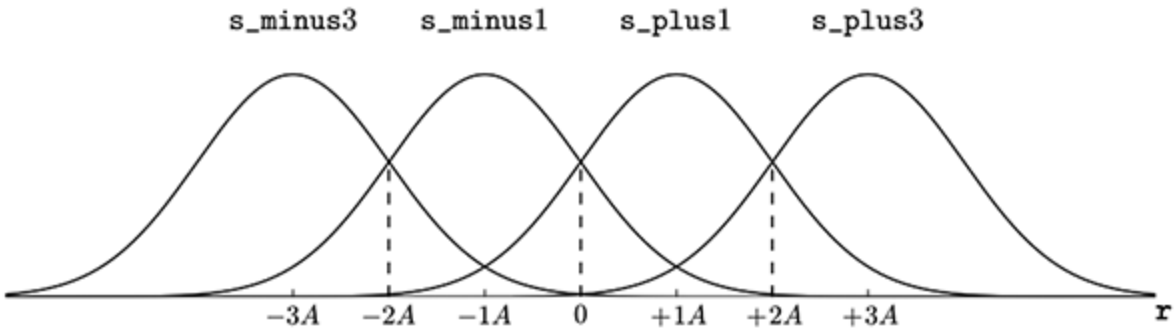
Problem 3. Mininmum-error threshold detection for 4-level signaling

Ben Bitdiddle studies the bipolar signaling scheme from 6.02 and decides to extend it to a **4-level signaling scheme**, which he calls Ben's Aggressive Signaling Scheme, or **BASS™**. In BASS, the transmitter can send four possible signal levels, or voltages: (-3A, -A, +A, +3A) where A is some positive value. To transmit bits, the sender's mapper maps consecutive pairs of bits to a fixed voltage level that is held for some fixed interval of time, creating a **symbol**. For example, we might map bits "00" to -3A, "01" to -A, "10" to +A, and "11" to +3A. Each distinct pair of bits corresponds to a unique symbol. Call these symbols `s_minus3`, `s_minus1`, `s_plus1`, and `s_plus3`. Each symbol has the same prior probability of being transmitted.

The symbols are transmitted over a channel that has no distortion but does have additive noise, and are sampled at the receiver in the usual way. Assume the samples at the receiver are perturbed from their ideal noise-free values by a zero-mean additive white Gaussian noise (AWGN) process with noise intensity $N_0 = 2\sigma^2$, where $2\sigma^2$ the variance of the Gaussian noise on each sample. In the time slot associated with each symbol, the BASS receiver digitizes a selected voltage sample, **r**, and returns an estimate, **s**, of the transmitted symbol in that slot, using the following intuitive digitizing rule (written in Python syntax):

```
def digitize(r):
  if r < -2*A: s = s_minus3
  elif r < 0: s = s_minus1
  elif r < 2*A: s = s_plus1
  else: s = s_plus3
  return s
```

The figure below illustrates the detection strategy:



Conditional noise distributions P(`s` | `r`)

Ben wants to calculate the **symbol error rate** for BASS, i.e., the probability that the symbol chosen by the receiver was different from the symbol transmitted. Note: we are not interested in the bit error rate here. Help Ben calculate the symbol error rate by answering the following questions.

A. Suppose the sender transmits symbol `s_plus3`. What is the conditional symbol error rate given this information; i.e., what is P(symbol error | `s_plus3` sent)? Express your answer in terms of A, N_0, and the erfc function defined as

$$\text{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-x^2}\, dx.$$

(points: 1.5)

B. Now suppose the sender transmits symbol `s_plus1`. What is the conditional symbol error rate given this information, in terms of A, N_0, and the erfc function? The conditional symbol error rates for the other two symbols don't need to be calculated separately.

(points: 1.5)

C. The symbol error rate when the sender transmits symbol `s_minus3` is the same as the symbol error rate of which of these symbols?

i. `s_minus1`
ii. `s_plus1`
iii. `s_plus3`

(points: 0.5)

D. The symbol error rate when the sender transmits symbol `s_minus1` is the same as the symbol error rate of which of these symbols?
   i. `s_minus3`
   ii. `s_plus1`
   iii. `s_plus3`

(points: 0.5)

E. Combining your answers to the previous parts, what is the symbol error rate in terms of A, N_0, and the erfc function? Recall that all symbols are equally likely to be transmitted.

(points: 2)

Problem 4. Averaging filter

In this problem set we will build a communication system using your computer's speakers and microphone as transmitter and receiver, respectively. We will multiply the message sequence with a high-frequency sine wave sequence (the carrier) at the transmitter in order to match the transmitted wavelength with the characteristic length of your antenna (your speakers) and thereby radiate power efficiently. The receiver will recover the message sequence by running the received sequence through an averaging filter represented by the following difference equation:

$$y[n] = \frac{1}{m+1}\left(x[n-k] + x[n-k-1] + \cdots + x[n-k-m]\right)$$

A. Show that the system is linear and time-invariant.

(points: 1)

B. Find the unit-sample response of this system for k=2, m=5.

(points: 0.5)

C. Compute the output y[n] when k=2, m=5 and the input sample sequence is

$$x[n] = 2u[n-3] - 2u[n-7].$$

u[n] is the unit-step function and it's defined as

$$u[n] = \begin{cases} 1, & n \geq 0 \\ 0, & n < 0 \end{cases}$$

(points: 0.5)

D. Compute the output y[n] when k=2, m=5 and the input sequence is

$$x[n] = 12 + \sin\left(\frac{\pi n}{3}\right).$$

(points: 1)

Programming Tasks

PSets 4-6 will use the Audiocom communication system, which uses your computer's speakers to transmit signals, and your computer's microphone to receive signals. In this lab, we will be exploring the effect of noise in communication channels, and working with a

particular demodulation scheme known as envelope detection.

Task 1: Audiocom

The first task is simply to familiarize yourself with the Audiocom system, and to use it to send a text file from your speakers to your microphone.

Download Audiocom and unzip it. For PS4, there are two .pyc files: preamble.pyc and demodulate_audiocom.pyc that you need to set to the appropriate pyc file depending on whether you have Python 2.6 or 2.7. For example, if you have Python 2.7, symbolically link `preamble_27.pyc` to `preamble.pyc` (or rename the former to the latter). Similarly for `demodulate_audiocom.pyc`

The documentation for Audiocom is available at http://audiocom602.blogspot.com. This document should be your starting point; it contains some useful tips for getting started and simple debugging.

Once you have successfully sent some bits using the process described in section 3.1 of the documentation, it might be fun to send a text file. We have included several test files (conveniently located in the `testfiles` directory). Replacing the `-S 1` and `-n 100` options with `-f testfiles/A`, for example, will attempt to send the text "Mens et manus" across your audio channel. (Note that if you're using IDLE, command-line options won't work and you have to specify the program's run-time parameters in config.py as explained in the documentation.) If all goes well, you should see that text appear shortly after transmission.

Task 2: Bypass Channel

We will first implement part of our Bypass Channel, which we will expand in PSets 5 and 6. This synthetic model of the communication channel will prove to be useful and informative moving forward, primarily as a means of debugging various demodulation schemes. At the very least, it will save us from having to listen to lots of long beeps during our initial debugging stages.

As mentioned in lecture, we will model environmental noise as a zero-mean Gaussian random variable. In the next section, we will try to convince ourselves that this is a decent model, but for now, we will simply implement it.

The file `bypass_channel.py` provides a template class `BypassChannel`. Notice that its `__init__` method takes three arguments. For this lab, we will only need to consider the first argument, `noise`; we will get to `lag` and `h` in future labs.

We will implement our synthetic channel by filling in the method `xmit_and_recv`, which takes a list of samples we wish to transmit, and returns a list of samples modeling those we would expect to receive, were we actually transmitting over the audio channel. For now, it will suffice to add a sample from a zero-mean Gaussian distribution to each element in `tx_samples`. The variable `self.noise` contains the desired **variance** of the noise.

Take each input sample in the numpy array, `tx_samples` and return a numpy array in which each sample is replaced with the original value plus a Gaussian random variable with zero mean and variance given by `self.noise`. You may find `numpy.random` useful (especially the

`normal` function in that module).

Upload your bypass_channel.py: Browse...

(points: 4)

Task 3: Measuring Noise

In this section, we will look at the effect of noise on our communcation channel, and will also try to convince ourselves that the Gaussian model of environmental noise is, in fact, reasonable.

Throughout this task, you will be making use of our **envelope demodulator** (which you will implement in the next task). This demodulator begins by "rectifying" the signal; that is, by taking the absolute value of the received samples so that every value is positive. Then, we replace each sample by a value representing the mean of the next few samples (specifically, the mean over the next half-period of the carrier waveform). This **averaging filter** was discussed in lecture. Given a carrier frequency and sampling rate, you can figure out easily how many samples are in one period of the waveform, and divide that number by 2 (more on this averaging in the next task).

We will start by taking a few measurements. First, let's try sending a bunch of 1's across the channel, and recording the voltages we receive back. Let's start by running `sendrecv.py` with the options `-S 1` (send only 1's), `-H` (send a header), `-g` (display graphs), `-n 500` (send 500 bits), `-c 1000` (use a 1000 Hz carrier), and `-o 1.0` (use a voltage of 1.0 to represent a 1). Some other options (such as `-q`, `-i` and `-p`) may be necessary for some machines. You should have found acceptable values for these options in task 1; make sure to use those values again here!

The bottom plot displayed by the `-g` option is a histogram of the demodulated samples. Assuming that environmental noise is Gaussian, we should expect to see a roughly Gaussian-shaped distribution over received samples.

Save and upload a graph that results from running this test:

1's plot Browse...

(points: 0)

What does your plot look like? In some cases, it might not look Gaussian, but look like two or more Gaussians with different means added together. Why does that happen (hint: think about what might happen if too few samples are being averaged together).

(points: 1.5)

We also made an assumption that environmental noise is not only Gaussian, but is *additive* as well; that is, the effect noise has on a signal does not depend on the value being sent. Let's test this by sending 1's at a lower voltage. Try all of the same options as above, but change `-o 1.0` to `-o 0.7`. (You may also achieve this goal by reducing the transmission volume.)

Save and upload a graph that results from running this test:

1's plot: 0.7v Browse...

(points: 0)

Try running this a few times. Compare and contrast the plots you get from this test, versus the plots you generated with `-o 1.0`:

(points: 1)

Let's now try sending 0's instead of 1's. Change `-S 1` to `-S 0` and run the test again.

Save and upload a graph that results from running this test:

0's plot Browse...

(points: 0)

Try running this a few times. Likely, you find that this graph does not look quite like a Gaussian centered at 0, but rather looks more like a "*half-Gaussian*" whose left edge runs up against 0. Explain why this is happening; what is different about the all-0's case, versus the all-1's case? (Hint: think about what the rectification step in the demodulation does.)

(points: 1.5)

Task 4: Demodulation

In this task, we will implement the envelope detector demodulation scheme described above. Changes for this section should be made in the template file `demodulate.py`. **To run your demodulator instead of the one we provided, change line 13 in `receiver.py` to:** **`import demodulate`** First, define a method **`avgfilter(samples_in, window)`**. This method should return a new **numpy array** `x`, each of whose elements `x[i]` is the average of all values `samples_in[i]` to `samples_in[i+window-1]`, inclusive. It makes sense to separate this averaging step (which is a form of "low pass" filtering) out of our envelope demodulator, because we will use it in other demodulators as well (in later labs).

`x` should have the same length as `samples_in`. Note that this means that values near the end of x will not be the average of `window` samples (we would run off the end of the list if we tried to do that); in this case, `x[i]` should be the average of all values from i to the end of `samples_in`.

Test your `avgfilter` function on multiple inputs. Here is the output of one test:

```
>>> avgfilter([1,1,2,3,5,8,13,21], 3)
array([  1.33333333,   2.        ,   3.33333333,   5.33333333,
        8.66666667,  14.        ,  17.        ,  21.        ])
```

Once your `avgfilter` method is working, move on to implementing the actual demodulator, which should now be a straightforward application of absolute value and the averaging filter you just wrote.

One question remains, though: what window size should we use during demodulation? Try playing around with a few different values, and see how this affects the shape of the "demodulated samples" plot.

For "low-enough" carrier frequencies, a good value to use for the window size is the number of samples in **one-half of the carrier waveform's period**. Change your demodulator to use this window size. You can calculate this value easily from the sampling rate and the carrier frequency of the waveform.

Submit your code for the demodulator below.

Upload your demodulate.py: Browse...

(points: 4)

As an extra challenge, try to think for yourself why this window size might begin to fail as we move to higher and higher carrier frequencies, and how we might go about improving its performance at these higher frequencies.

6.02 Introduction to EECS II: Digital Communication Systems

Fall 2012