Milestone 2

Due Date : 10th May 11:59 pm.

UPDATES

- In demap_and_check function under receiver.py : After demapping the demodulated samples, make sure that the samples corresponding to the preamble get demapped to the actual preamble bits. If not, declare that the preamble was not detected.
- common_txrx_mil3.pyc and receiver_mil3.pyc (in starter code) updated. Make sure you use the latest version of these files.
- Due date updated to 10th May.
- Slides on the Milestone 2 Discussion held in class can be downloaded here : [pdf]

STARTER CODE

Download the starter code for the second milestone here and unzip it. The directory will contain these 5 files.

- `transmitter.py`
- `receiver.py`
- `common_txrx.py`
- `receiver_mil3.pyc`
- `common_txrx_mil3.pyc`

From your milestone 1 directory, delete 3 files (`transmitter.pyc`, `receiver.pyc`, `common_txrx.pyc`) and replace them with these 5 files.

GOAL

- The goal for this milestone is to build the code for `TRANSMITTER` and `RECEIVER` blocks (i.e. to complete `transmitter.py, receiver.py` and `common_txrx.py`).
- Once you are done, your code should work just like the audiocom demo. That is, you should be able to:
  - Send monotone using:

```
python sendrecv.py -m 100 -c 1000 -s 256 -q 200
```

  - Send a text/image using:

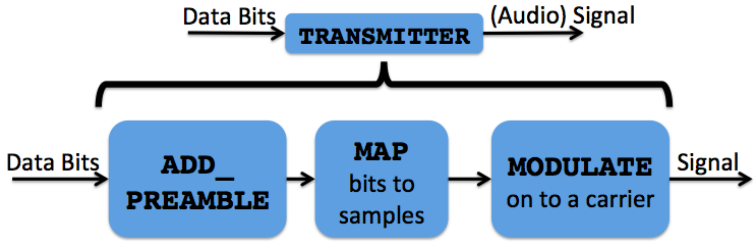
```
python sendrecv.py -f testfiles/<filename> -c 1000 -s 256 -q 200
```

- Also, if you feed the transmitted samples directly to the receiver (i.e., without introducing the channel) in `sendrecv.py`, you should see no bit errors.
  - You can do this by modifying the line "`demod_samples= r.demodulate(samples_rx)`" to "`demod_samples = r.demodulate(mod_samples)`"

`transmitter.py`

This file defines a transmitter object which is responsible for implementing the functionality required from the `TRANSMITTER` block. That is,

- Given a stream of data bits from the `SOURCE`, convert it to audio samples that will be played on the speaker. This is done by the following 3 steps.
  - Attach the preamble so that the receiver can figure out where the signal starts
  - Convert bit sequences into samples
  - ~~Modulate the samples onto a carrier~~ (You do not need to worry about this for Milestone 2; we'll provide this to you)



The constructor (`__init__`) and the function `modulate` is given :

- `__init__` : The transmitter object takes the following parameters
  - *carrier_freq* : carrier frequency (in Hz)
  - *sample_rate* : sampling rate (in Hz) for the audio signal samples. Number of samples transmitted

by the speaker per second (default = 48000 samples/second).
    - *one* : sample value corresponding to bit '1'
    - *spb* : number of samples per bit
    - *silence* : number of '0' bits before the actual preamble
- `modulate` : calls the modulate function in `common_txrx.py`
    - Input : raw samples (before modulation)
    - Output : audio samples (after modulating onto a carrier)

To complete `transmitter.py` for Milestone 2, you will need to fill up the following functions :

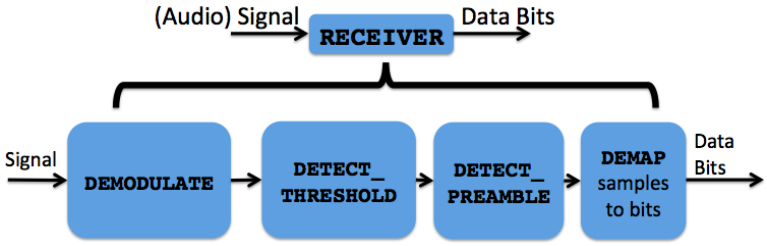
- `add_preamble` : adds preamble bits to data bits
    - Input : data bits (output of `SOURCE`)
    - Output : data bits with preamble (silence bits + preamble bits + data bits)
    - Before the actual preamble bits, you will need to add some *silence bits*, a bunch of bit '0's. This is because the microphone may not start capturing samples as soon as you start playing them on the speaker, so you add a period of silence at the beginning to make sure that the microphone doesn't miss any useful sample.
        - `self.silence` tells you the number of silence bits to add.
    - We recommend that you use the following sequence as the preamble :
    [1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1]

- `bits_to_samples` : maps each bit in the input sequence (silence + preamble + data bits) to its corresponding samples
    - Input : data bits with preamble
    - Output : samples (as a numpy array)
    - Expand bits to samples by replacing each bit with `spb` samples. Note that the sample value for bit '1' is `one`, not 1 (refer to `__init__`).

`receiver.py`

This file defines a receiver object which is responsible for implementing the functionality required from the `RECEIVER` block. That is,

- Given a stream of audio samples, determine the data bits that were sent. This is done by the following steps.
    - ~~Demodulate the audio signal samples~~ (You do not need to worry about this for Milestone 2, we'll provide this to you)
    - ~~Detect threshold for demapping~~ (You do not need to worry about this for Milestone 2, we'll provide this to you)
    - Detect the preamble to find out the start of the signal
    - Demap the samples into data bits that can be fed as input to `SINK`



Before jumping into the details of the code, lets walk through the functions that the receiver needs to perform.

PRIMER

Let's skip the details of the demodulator for now and assume that we have a set of demodulated samples.

Threshold detection

First, the receiver needs to make sense of the demodulated samples. The original audio samples that are transmitted by the speaker get attenuated and corrupted as they travel over-the-air and before they are captured by the microphone. For example, if the transmitter used 1.0 volt samples to signal a bit "1" and 0.0 volt samples to signal bit "0", the corresponding demodulated samples could be significantly different from "1" volt and "0" volt (see Figure 1). Hence, the receiver cannot detect the value of the bit by just looking at a sample.

However, the receiver knows that the transmitter is transmitting only two different sample values (corresponding to bit values “0” and “1”). The receiver can take advantage of this prior information and cluster the received samples into 2 different clusters. That is, the receiver can split all received samples into 2 sets of similarly valued samples. For eg., in figure 1, it is easy to conclude that most sample values lie in either a cluster around 0.66 or in a cluster around 0.1. Hence, the receiver (just by analyzing all its received samples) can conclude that sample values around 0.66 correspond to bit “1” and sample values around 0.1 correspond to bit “0”. In other words, the receiver can choose the mid-point of the average sample values of these two clusters (i.e. mid-point of 0.66 and 0.1 = 0.38) as a rough estimate of the *threshold*, and decide that a sample corresponds to bit “1” if its value is greater than the threshold or bit “0” otherwise.

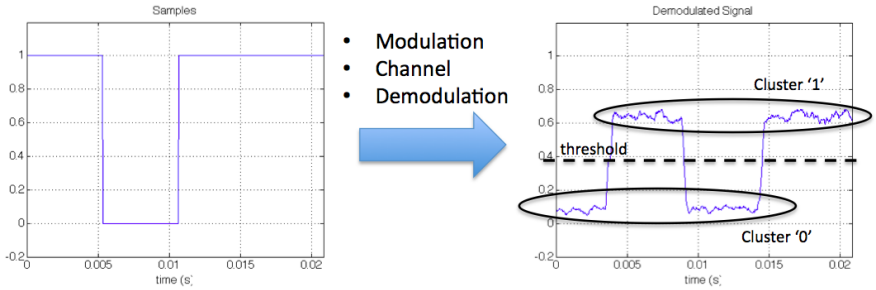


Figure 1: Detecting the Threshold

Detecting start of the packet

With this rough estimate of the threshold, the receiver can make sense of the received samples. However, it still does not know where the packet exactly starts, since the first audio sample captured by the microphone does not always correspond to the start of the packet. This is due to a variety of reasons - silence bits, speaker-microphone asynchrony, propagation delay etc. That is, `sample[0:spb]` does not always represent the first bit and the receiver does not know where to start looking for the data! In other words, there is an `offset` between the switching-on of the microphone and the start of the packet, and the receiver needs to figure out this offset before it can start extracting the data out of the demodulated samples.

After ignoring the first “offset - 1” samples, we can start treating each set of `spb` samples as corresponding to a meaningful bit. That is, `sample[offset:offset+spb]` correspond to the first bit and so on. The receiver needs to figure out this “offset” value.

It is for this exact purpose that the transmitter added a “preamble” sequence before the data bits. The preamble sequence is something that the transmitter and receiver agree upon prior to the communication. Hence, the receiver knows the “preamble” that is transmitted at the start of the packet. Also, this preamble sequence has a specific pattern of bits (and hence a specific pattern of samples) which is very unlikely to be found elsewhere in the signal. The receiver takes advantage of this scenario and searches for the “preamble” sample-pattern in the demodulated samples. This “search” of the preamble pattern is done using **correlation**.

We define the “cross-correlation” of two real-valued signals f[n] and g[n] with an offset k as:

$$(f \star g)[k] = \sum_{m=-\infty}^{\infty} f[m]g[m+k]$$

The output of this operation is a measure of the “similarity” between the two sequences, i.e. how much the signal f[n] “looks like” the signal g[n] when g is offset from f by k samples.

Recall that the receiver needs to “search” for the preamble sample-pattern in the demodulated samples. It can perform this “search” by first forming the preamble samples (by mapping each preamble bit to `spb` samples, like in the transmitter) and then evaluating the cross-correlation between the preamble samples and the demodulated samples at different time-shifts.

Let us work out a numeric example to see what is happening.
Assuming that the preamble bit sequence is [1 1 0 1], the preamble samples will be `raw_samples = [1 1 1 1 1 1 0 0 0 1 1 1]`, where `one` = 1 and `spb` = 3. After channel distortion and a random delay, the demodulated samples would be something like :

```
demod_samples =  [0.0470 0.0011 0.0572 0.0279 0.0119 0.0279 0.0056 0.0147
 0.0479 0.0381 0.3827 0.4786 0.5315 0.5268 0.5239 0.4765 0.0813 0.0001 0.0027
0.3979 0.5326 0.4851 0.0404 0.0330 0.0342 0.0457 0.0240 0.0259]
```

The raw preamble samples and the demodulated samples are shown in Figure 3. Note that these sample values are generated using a channel response of $h[n] = 0.4*\delta[n] + 0.1*\delta[n-1]$, adding some noise and

an additional delay of 10 samples.

Also shown in figure 3 is the cross-correlation between the preamble samples and the demodulated samples at different shifts. For example, the correlation value at offset = 2 is 1.68. This value is obtained by carrying out the following computation :

$$Correlation[2] = \sum_{m=-\infty}^{\infty} rawSamples[m]*demodSamples[m+2]$$

$$= \sum_{m=0}^{11} rawSamples[m]*demodSamples[m+2]$$

```
= 1*0.0572 + 1*0.0279 + 1*0.0119 + 1*0.0279 + 1*0.0056 + 1*0.0147 + 0*0.0479
+ 0*0.0381 + 0*0.3827 + 1*0.4786 + 1*0.5315 + 1*0.5268
= 1.62
```

Repeating a similar calculation for all values of shifts gives a cross-correlation graph similar to the one shown in Figure 3. We can see that the cross-correlation is highest when the offset is 10, which is exactly the delay in the demodulated samples.

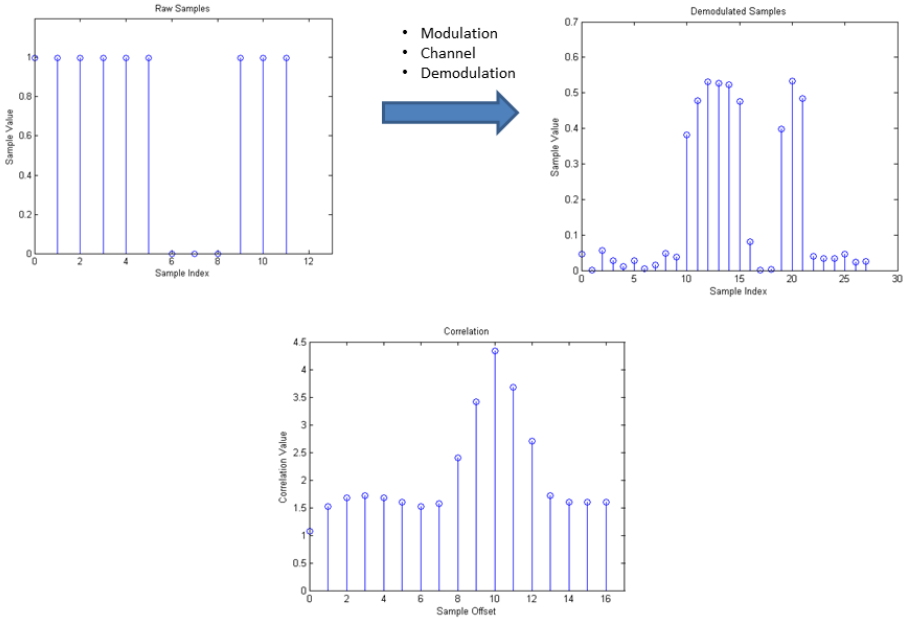


Figure 2: Correlation

Demapping symbols to bits

Once the receiver figures out the start of the packet (i.e. the sample offset after which the packet begins), it needs to convert the samples into bits. Note that the threshold has been computed to distinguish between samples corresponding to bit 0 from those corresponding to bit 1.
So, for example, `samples[offset: offset+spb]` correspond to the first bit. One naive method of obtaining the first bit would be :

- randomly pick one sample (let's call it `rand_samp`) from `samples[offset: offset+spb]`.
- Compare `rand_samp` with `threshold`.
- If `rand_samp > threshold`, then declare the first bit to be 1, else declare it to be 0.

Think about the problems this naive approach could lead to!
First, since all samples are noisy, making a decision based on just one sample seems risky. Also, note from Figure 1, that the sharp transition of sample values at bit boundaries gets smoothed out in the demodulated samples. That is, whenever for example, the bit value changes from 0 to 1, the sample values take some time to catch up. So there are intermediate sample values (close to the threshold) at the transition which are unreliable; unlike the original samples which were just clean `one`s or 0s.

To ensure higher reliability of demapping, we propose the following demapping algorithm (you are welcome to try out other algorithms and document the performance) :

- From the samples corresponding to each bit, ignore the first `spb/4` and the last `spb/4` samples.
- Compute the mean of the middle `spb/2` samples

- Compute the mean of the middle `spb/2` samples.
- Compare this mean with `threshold`. If the mean exceeds the threshold, declare the bit as 1, else declare it as 0.

For example, for the first bit , compute the mean of `samples[offset+spb/4:offset+3spb/4]` and compare it with the threshold.

That concludes our primer on receiver algorithms for now. Let's get back to the functions that you have to deal with in the implementation.

The constructor (`__init__`) and the function `demodulate` is given :

- `__init__` : The receiver object takes the following parameters
  - *carrier_freq* : carrier frequency (in Hz)
  - *sample_rate* : sampling rate (in Hz) for the audio signal samples, which is the number of samples received by the microphone per second (default = 48000 samples/second).
  - *spb* : number of samples per bit
- `demodulate` : calls the `demodulate` function in `common_txrx.py`
  - Input : audio signal samples (captured by the microphone)
  - Output : demodulated samples
- `detect_threshold` : calls the `detect_threshold` function in `receiver_mil3.pyc`
  - Input : demodulated samples
  - Output : threshold and the respective means of clusters corresponding to bit '1', '0'
  - This function implements the clustering as mentioned in the Primer above.

To complete the `RECEIVER` block for Milestone 2, you will need to fill in the following functions :

- `detect_preamble` :
  - Input : demodulated samples, threshold, mean_one (mean of the cluster corresponding to bit "1")
  - Output : starting index of preamble (in sample domain)
  - The basic approach (as described in the primer), would be to perform a cross-correlation of the preamble samples with the demodulated samples (for all possible shifts). However, this is extremely computation intensive. Instead, we can use the following hand-waiving procedure.
  - Zero energy is a good sign that there is no incoming transmitted signal. Thus, try to detect the offset from the start of the received samples from which a set of `spb` consecutive samples has enough energy to correspond to a bit "1". Let's call this value `energy_offset`.
    - For each value of `energy_offset` (starting from 0 and continuing till the end of demodulated samples), we want to see whether the samples, `samples[energy_offset:energy_offset+spb]` have energy high enough to represent a bit '1'. So, as described in the primer, take the the central `spb/2` samples of this interval and average it. Compare this value with `(mean_one + threshold)/2` and if it's higher, terminate the loop. If not, increment `energy_offset` and keep running the loop. Visually, this procedure can be seen as averaging over `spb/2` samples in a sliding window of length `spb` samples (refer to Figure 3).
    - `(mean_one + threshold)/2` can be seen as the lower bound of a sample value to be decoded as bit '1'. Remember, there is a "forbidden region" near the threshold that we can't decode with confidence (Refer Fig. 4).

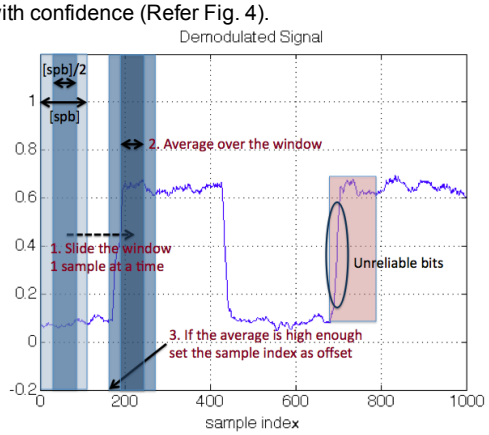
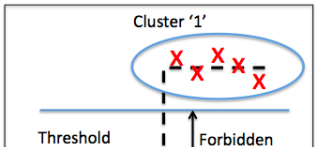


Figure 3:Moving average method to find the energy_offset



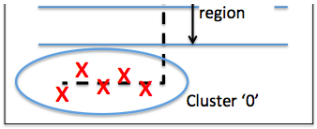
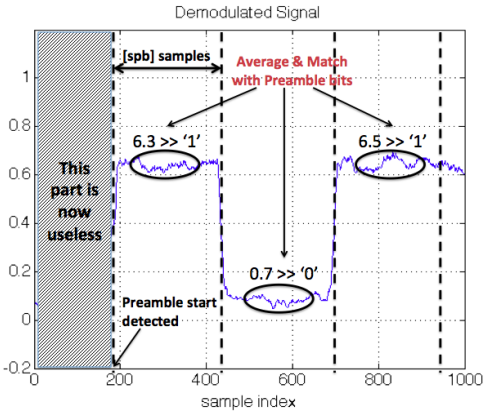


Figure 4. Forbidden region near the threshold

- Note that the above offset measurement was based purely on energy. This was done to reduce our workload in the correlation function. Now, since we have a rough estimate of the start point of the preamble, we can now use the correlation function to find the precise starting point of the preamble.
  However, instead of cross-correlating the preamble samples with the entire set of demodulated samples (for all possible time shifts), we cut down our search space! We claim that the earlier energy detection was good enough to roughly reach the start of the preamble and the actual `offset` is very close to `energy_offset`. Hence, we cross-correlate the preamble samples with `samples[energy_offset:energy_offset+3*length(preamble_samples)]`. This leads to reduced workload in the correlate function while ensuring that the preamble is not missed. The offset can now be calculated as the `energy_offset + preamble_offset` (shift for which the cross-correlation peaks).

- `demap_and_check` :
  - Input : demodulated samples, starting index of preamble (in sample domain - output of detect_preamble function)
  - Output : received data bits
  - Convert the demodulated samples into bits. Now you know the starting index of the preamble and the number of samples per bit (`spb`). We can use the same demapping algorithm as described in the primer.
  - However, remember that the threshold we were using was based on every sample in the demodulated samples (even the samples before the actual signal)! Now the receiver knows:
    - When the actual signal starts (start of preamble)
    - When each transition occurs (bit boundaries)
    - Bit values corresponding to the preamble samples (receiver knows the preamble bits)

    The receiver, with all this new information, can do a better job of calculating the threshold. Hence, in this function, we recalculate the threshold; using only the preamble samples and excluding the samples before the actual signal starts and the samples near transitions. (Preamble bits has enough '0' and '1' to analyze and to get a representation)
  - For example, let's take a look at Figure 5. First we divide the sample array into `spb` chunks from the starting index. Let's say the middle `spb/2` sample values of the first chunk are [6.12 6.47 6.43 ...] and it turns out that their average is 6.3. We map this value to bit '1', because the corresponding preamble bit at that point is '1'. We repeat this for all the following preamble bits , 0.7 gets mapped to '0', 6.5 gets mapped to '1'. Now we have a bunch of (average sample value, bit value) pairs. We make a table out of this pairs like Figure 5 and calculate the average of each column (like 0.6 and 6.2 in Figure 5). These are now the representative value of the sample corresponding to '0' and '1' respectively. Finally, we take the midpoint of the two representative values as the new threshold. In our example the threshold will be (0.6+6.2)/2 = 3.4



Bit	'0'	'1'
Average values	0.7	6.3
	...	6.5
	...	...
	...	...
Mean	0.6	6.2

Figure 5. Recalculating the threshold

- After you calculate the new threshold, you do the same average-and-demap procedure based on the new threshold. (You can try another decision logic since this is only a recommendation. As long as you can retrieve the data correctly, it's fine.)
- After demapping the demodulated samples, make sure that the samples corresponding to the

preamble get demapped to the actual preamble bits. If not, declare that the preamble was not detected.

`common_txrx.py`

This file consists of those functions which are needed by both `TRANSMITTER` and `RECEIVER`. You are free to add more functions here.

In this file, you will be provided with the following functions :

- `modulate`: modulate the samples onto a carrier signal of a given carrier frequency; the resulting modulated samples can be played on the speaker
  - Input : carrier frequency, sample rate (number of samples transmitted by the speaker per second), raw samples (before modulation)
  - Output : modulated samples for the speaker
- `demodulate` : demodulate the audio signal samples with the given carrier frequency
  - Input: carrier frequency, sample rate (number of samples captured by the microphone per second), received samples from the microphone
  - Output: demodulated samples

SUBMISSION INSTRUCTIONS

1. Login to coursework.stanford.edu using your SUNet ID.
2. Click on *Sp13-ENGR-40N-01* in the top menu to enter the ENGR 40N website on Coursework.
3. Click on *Drop Box* in the left menu to access your online drop box for this course.
4. Upload your files: `transmitter.py, receiver.py, common_txrx.py`. Do NOT rename the files, do NOT create sub-folders (upload the three files into your main drop box folder for the course)

As always, if you encounter any problem, post on Piazza!