Milestone 1

Due Date : 18th April 11:59 pm.

Make sure that you have read through the setup instructions and the audiocom demo works on your laptop.

You should find the following code in the audiocom demo folder :

- source.pyc, sink.pyc, common_srcsink.pyc, transmitter.pyc, receiver.pyc, common_txrx.pyc
  - All these files are compiled versions of the actual source code and provided only for testing purposes. Throughout the quarter, you will be writing code to replace these compiled files.
- audio_channel.py, bypass_channel.py, sendrecv.py, graphs.py, config.py
  - These source files are provided in their human-readable format. Some of them (especially sendrecv.py, audio_channel.py and bypass_channel.py) are critical to the audiocom setup and we expect you to understand their functionality.

STARTER CODE

Download the starter code for the first milestone here and unzip it. Notice that the starter code is the same as the audiocom demo code, except the following changes :

- source.pyc, sink.pyc, common_srcsink.pyc are deleted.
- Bare-bones code is provided in source.py, sink.py and common_srcsink.py instead.

GOAL

- The goal for this milestone is to complete the code for Source and Sink blocks (i.e. complete source.py, sink.py and common_srcsink.py).
- Once you are done, your code should work just like the audiocom demo. That is, it should be able to perform :
  - Sending monotone using:

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

  - Sending a text/image using:

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

sendrecv.py

This file contains the main function which calls all the individual blocks one-by-one. You do not need to modify it for this milestone. However, reading through it and understanding the order in which each function is called will be useful for your understanding. The following abstract is especially critical :

```
    #instantiate and run the source block
    src = Source(opt.monotone, opt.fname)
    src_payload, databits = src.process()

    # instantiate and run the transmitter block
    xmitter = Transmitter(fc, opt.samplerate, opt.one, opt.spb, opt.silence)
    databits_with_preamble = xmitter.add_preamble(databits)
    samples = xmitter.bits_to_samples(databits_with_preamble)
    mod_samples = xmitter.modulate(samples)

###################################
    # create channel instance
    if opt.bypass:
        h = [float(x) for x in opt.h.split(' ')]
        channel = bch.BypassChannel(opt.noise, opt.lag, h)
    else:
                channel  =  ach.AudioChannel(opt.samplerate,  opt.chunksize,
opt.prefill)

    # transmit the samples, and retrieve samples back from the channel
    try:
        samples_rx = channel.xmit_and_recv(mod_samples)
    except ZeroDivisionError:
        # should only happen for audio channel
        print "I didn't get any samples; is your microphone or speaker OFF?"
        sys.exit(1)
################################

    # process the received samples
    # make receiver
    r = Receiver(fc, opt.samplerate, opt.spb)
    demod_samples = r.demodulate(samples_rx)
    one, zero, thresh = r.detect_threshold(demod_samples)
    barker_start = r.detect_preamble(demod_samples, thresh, one)
    rcdbits = r.demap_and_check(demod_samples, barker_start)

    # push into sink
```

```
sink = Sink()
rcd_payload = sink.process(rcdbits)
```

source.py

This file defines a source object which is responsible for implementing the functionality required from the Source block. That is,

- Given a source file (image/text), convert it into data bits (header + payload). Here, the payload contains the source file converted into bits and the header contains all the information that needs to be communicated to the sink (file type and payload length).
- If no source file is mentioned, this block should transmit a monotone. This could be useful for testing purposes. To transmit a monotone,
  - Header format could stay the same (though make sure that you use a different file type for monotone - or else the sink will assume that it is receiving a file).
  - Payload is an array of all 1s ([1 1 1 .... 1]), the length of this array should be equal to the "monotone" parameter passed into the source block.

To complete this code, you will need to fill up the following functions :

- process : Overarching function which should return the data bits. Read through sendrecv.py and see where this function gets called.
  - Input : NULL
  - Output : payload bits, data bits (header bits + payload bits)
    - The payload bits will be used eventually, to compare with the received payload to evaluate the quality of our reception.
- text2bits : Converting text to bits (this function should be called by the process function, if it needs to transmit a text file)
  - Input : filename (text file)
  - Output : payload bits (corresponding to the text)
- bits_from_image : Converting image to bits (this function should be called by the process function, if it needs to transmit an image).
  - Input : filename (image file)
  - Output : payload bits (corresponding to the image)
  - Please look through the Python Image Module which has a lot of the basic image manipulation functions implemented
  - You can assume that the image provided will be a **black and white 32*32 pixel** image in **.png** format. The image files present in milestone1_starter/testfiles satisfy these requirements and can be used to test your code.
- get_header : Forming the header bits given the source type (image/text/monotone) and the payload length (in bits). This function should be called by the process function to form the header.
  - Input : payload length, source type
  - Output : header bits

sink.py

This file defines a sink object which is responsible for implementing the functionality required for the Sink block. That is,

- Given an array of received bits passed on from the receiver, estimate the file that was transmitted.
- If the transmitted file is an image, save it as "rcd-image.png" and if its a text, just print out the text.

To complete this code, you will need to fill up the following functions :

- process: Overarching function which is called with the received bits as input. Read through sendrecv.py and see where this function gets called.
  - Input : received bits (array of bits)
  - Assume that the input array of received bits start with the header (that is, preamble has been detected and deleted).
  - However, the length of this array could be arbitrary. Hence, you need to truncate this array according to the payload length mentioned in the header to capture the payload precisely.
  - Output :
    - If the transmitted file is an image, save it as "rcd-image.png" .
    - If the transmitted file is text, print out the text.
    - Irrespective of the source type (image/text/monotone), this function should always return the received payload (after deleting the header and truncating according to the length mentioned in the header). This received payload will be compared with the transmitted payload to evaluate the quality of our reception (Bit Error Rate).
- read_header : Analyze the header bits to figure out the payload length and the source type (image/text/monotone). This function should be called by the process function.
  - Input: Header bits
  - Output: Source Type, Payload Length
- bits2text: Convert the payload bits (corresponding to a text) back into the text. This function should be called by the process function, whenever it observes that the incoming transmission corresponds to a text.
  - Input: Payload bits (corresponding to a text)
  - Output: Text (as a string)
- image_from_bits: Convert the payload bits (corresponding to an image) back into the image and save it as "rcd-image.png". This function should be called by the process function, whenever it observes that the incoming transmission corresponds to an image.
  - Input : Payload bits (corresponding to an image)
  - Output: Save the image as "rcd-image.png" (no return value required from this function)

common_src_sink.py

common_srcsink.py

Finally, this file consists of those functions which could be useful for both the source and sink objects. We expect you to complete the “hamming” function in this file. However, you are free to add more functions here.

To complete this code, you will need to fill up:

- Hamming: This function should compare the sent payload with the received payload and evaluate the quality of reception. That is, we are looking for an answer to : How well were we able to reconstruct the transmitted data (image/text/monotone)?
  - Input : Two arrays of bits (s1, s2) which correspond to the received payload and the sent payload respectively.
  - Compute the Hamming Distance between the two arrays.
    - The Hamming distance between two (equal-length) vectors is the number of positions where the vectors vary. For eg., the hamming distance between [1 1 1 0 0 0] and [0 0 0 0 0 0] would be 3.
    - Note that, in our case, the lengths of the two arrays could be different. It would for example, happen when the received header is corrupted leading to an incorrect estimation of the payload length by the Sink block. To counteract this issue, you can truncate the longer of the two arrays so as to equalize their lengths.
  - Compute the Bit Error Rate.
    - The Bit Error Rate is the ratio : (number of incorrect bits) / (number of total bits).
    - Note that the numerator in the above ratio is the Hamming Distance and the denominator is the (equalized) length of the arrays.
  - Output : Hamming Distance, Bit Error Rate
  - Note that this output will be eventually printed out to the terminal (in sendrecv.py).

SUBMISSION INSTRUCTIONS

Will be up soon!