Milestone 3 (Channel Coding, Compression)

Due Date : 5th June 11:59 pm.

Version 2: Source and Channel Coding

UPDATES

- [Caution!] The numbers recommended in the source coding part are for codes which have followed the recommended `ENCODE` function in milestone1 and for files A.txt and 32pix.png. If you,
  1. used a different number of bits to represent a character in text or a pixel in image for milestone1,
  2. define m-bits chunk as a symbol where m is not 4,
  3. try your code with a different file,

  the appearance count for each symbol will be different. Thus you will need to check the appearance counts first and assign enough space for each of the appearance counts by yourself.

- Input argument of `huffman_decode` has been simplified for the sake of symmetry with `huffman_encode`. Preprocessing (checking srctype, grabbing bits according to bit length, etc.) can be done at `process`. It is fine if you already done preprocessing inside `huffman_decode`. As long as the entire logic of sink module is same, there will be no difference in the result.

STARTER CODE

Download the `sendrecv_coding.py` and `hamming_db.py` for this milestone here

- `sendrecv_coding.py` is a modified version of `sendrecv.py` to take '-H' argument that specifies the codeword length of Hamming code (values from 3, 7, 15, 31). Also the `transmitter` and `receiver` part is changed to incorporate the new channel coding functions (read carefully to understand how to write the channel coding related functions) You are going to execute this code in the terminal instead of `sendrecv.py`.
- `hamming_db.py` contains a lookup table to return matrices and parameter for the choice of Hamming code (from '-H'). You will need to modify the functions `gen_lookup` and `parity_lookup`.

You will also work on the files that you have implemented until now:

- `source.py` (Source encoding)
- `sink.py` (Source decoding)
- `transmitter.py` (Channel encoding)
- `receiver.py` (Channel decoding)

If you need to write functions needed in both transmitting side and the receiving side, implement them in the following 2 files (most likely this will be to get the header lengths)

- `common_srcsink.py` (Source and Sink related)
- `common_txrx.py` (Transmitter and Receiver related)

GOAL

- The goal for this milestone is to build the code for source coding (compression) and channel coding (error correction).
- For source coding, you will print out (1) the source-coded data length (2) compression rate (= (1)/[original source length]) on the terminal. (original source length is already printed out from the previous milestone)
  - Print out along with the information from the `source`. (refer to the result of the original demo code)
  - Use the title 'compressed payload length: ', 'compression rate: ' for (1), (2) respectively so that we can easily identify.
- For channel coding, you will test your code on the synthetic bypass channel and ensure performance (in terms of bit error rate) for different level of noise. Use -s 128 (128 samples/bit)
  - Specifically **you must submit a `RESULTS.txt` file** that lists the BER values for 2 different channel coding rates (n=31 and n = 7) at two different noise levels (0.7 and 0.9). You must run the experiment 5 times for each noise and rate combination and publish the average BER. We will expect you to achieve an average BER value similar to the sample `RESULTS.txt` file ±.01
  - Additionally, please print the following string for grading purposes: 'channel coding rate: {rate}' with {rate} being the k/n in the `transmitter` and 'errors corrected: {num_errors}' in the `receiver`.
- Because we are compressing data and using channel coding for error correction, we want to test our system using larger files. *Please test using the two images we gave you as well as the three new testfiles* `Time.txt` *(600 bytes)* `Who.txt` *(1 kbyte),* `Shakes.txt` *(5 kbytes)*

Simple but *Important* Changes before Implementation

Change of Preamble

To test the performance of channel coding, you need to test the system in a very noisy channel. But if the channel gets messier, it gets harder to detect the preamble. If the preamble is not detected, you cannot even decode the bits at all. Thus we recommend a longer preamble that will be improve the preamble detection. Use the following preamble

```
preamble = [1,1,1,1,1,0,1,1,1,1,0,0,1,1,1,0,1,0,1,1,0,0,0,0,1,0,1,1,1,0,0,0,
            1,1,0,1,1,0,1,0,0,1,0,0,0,1,0,0,1,1,0,0,1,0,1,0,1,0,0,0,0,0,0]
```

Remove preamble check

In demap_and_check function of receiver.py, you need to remove the preamble check part. Previously, we were checking if the guessed preamble is exactly the same with the original preamble. But the noisy environment will significantly reduce the chance of a perfect match. So instead of doing a very cautious verification, we just use a long preamble to try to pinpoint the exact start of the message, even though the recovered preamble might have some errors. Still, there is a possibility of finding a wrong starting position and this will fail the entire transmission since the bit sequence is shifted; no matter how your good your channel coding is. (So when you test the system, check whether the recovered preamble is similar with the original preamble and whether the contents in the headers are not compromised)

Encapsulation

In this milestone, we perform (1) source coding to compress the source data and (2) channel coding to add redundancy in the communication data for reliability. Since the subject of these two codings are different, they can be performed independently in different layers. But these codings require additional information for decoding. So we convey such information in the headers. Because source coding and channel coding are two independent tasks performed by two different layers, we use the ideal of ***Encapsulation*** to design the frames and the two-layered headers. The basic structure of the frame is represented in Figure 1.

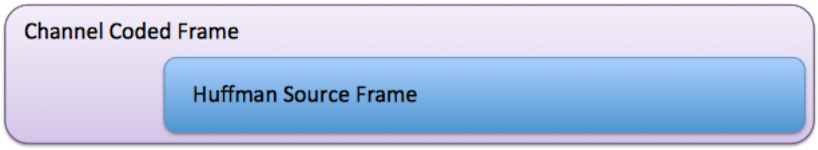


Figure 1. Each frame has header and data portions, where the data portion for the Channel Coded frame is the Huffman Source Frame.

The main reason why we design the frame in this manner is to create simple abstractions. For each layer, we only need the information that is relevant at that particular layer. For example, you do not need any information about the source type or length when we are doing channel coding in the `transmitter`, `receiver` modules. If you can treat header information and coded data as simply a payload of bits, you can more easily apply channel coding to those bits. Similarly, you do not need any information about channel coding in the `source`, `sink` modules. So the channel coding information is kept out of the `source`, `sink` modules.

If you use a single header that contains all the information that is needed in every layer, you will have a hard time manipulating the header to make it suitable to pass it through different layers. Encapsulation also allows us to apply a more cautious channel encoding scheme for the Channel Coding header while maintaining a higher data rate for the data portion (which includes the Source Coding header). If you were to merge the two headers, you would get an unnecessarily long header length because you must still maintain the cautious channel coding rate to preserve the channel coding information.

In our implementation we will have two frames as shown in Figure 1. This allows us to compress the data bits in the `source` and then encode the date for error correction in the `transmitter`. The Source Coding frame is shown below in Figure 2.

Figure 2. Source Coding Frame (for Source and Sink)

This Source Coding frame is then encapsulated inside the Channel Coding frame in the `transmitter` shown in Figure 3. The Source frame is encoded using a Hamming code and treated as the data portion of the Channel Coding frame.

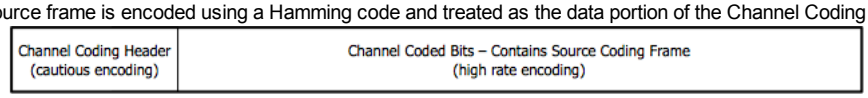


Figure 3. Channel Coding Frame (for `transmitter` and `receiver`)

Putting it all together the final output sent across the channel will appear as Figure 4. Note that the original data is compressed using the Huffman algorithm, then concatenated with the Huffman Source Header to create the Huffman Source Frame. This frame and the Channel Coding Header are encoded using Hamming codes and concatenated to create the final frame sent across the channel.

Figure 4. Frame structure with multiple headers.

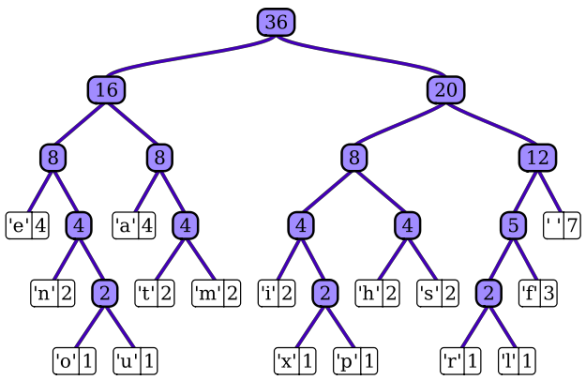
Source Coding

Primer

The purpose of source coding is to compress the source bits. This will allow us to represent the same data using less bits. Based on the statistics of each possible chunk of data, the coding scheme will assign ***codewords*** of optimal length for each chunk of data.* This results in an efficient use of bit resources. Once the designer selects a suitable code for the system, the source and the sink have prior knowledge of what code they are going to use. If the source uses additional information that the sink does not have, it must pass the information to the sink during communication (usually in a header). In milestone 3, you will be using a ***Huffman code*** which is a straightforward scheme for lossless compression.
(*)There are a family of codes which does not need the statistics of the data, called universal codes; but we do not cover them in this milestone

Data can be represented as a stream of the smallest irreducible units called ***symbols***. For example, let's focus on english text. Every stream of text consists of symbols (a, b, c, …) from the set known as the ***alphabet***. (In coding theory terminology, we use the same word to define the set of symbols for any kind of data) When we are using ascii code to represent text data, the alphabet also contains numbers, special characters, etc. In the language of bits, 8-bit ascii code is picking 8-bit symbols that comes from the alphabet = {0000/0000, …, 1111/1111}. Huffman code will take a look at the statistics of appearance of every symbol in the alphabet and make a more efficient representation. Take a look at the text "Ahhhhhh!". It would be a waste of bits if we assign 8 bits to both 'A', '!' and 'h' when there is a bunch of 'h's. It gets worse if you realize that we are assigning 8 bits in order to distinguish 'A', '!', 'h' with the other characters that does not even come up! Huffman code will cope with this inefficiency.

To deal with the above issue, Huffman code generates a ***Huffman tree*** for the subject data. First, you count the appearance of every symbol from the alphabet which actually appears in the data. Then, you merge the two least frequent symbols and associate the sum of the appearance of the two symbols to the new node. You continue the merging process and the result will be the Huffman tree. Below is an example when the text is "this is an example of a huffman tree"



Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

Figure 5. Example of a Huffman tree and the corresponding code

Now, we assign 0/1 for each edge coming out of a parent node. Then, the codeword for a symbol can be read along the path from the top to the symbol. The resulting codeword table for the example is presented next to the Huffman tree in Figure 5. The codewords will depend on which way you assign 0/1 to each pair of edges but the codes will be essentially equivalent. You can see that the more frequent the symbol is, the later it merges to the tree, thus the shorter the codeword is. This is why Huffman code is efficient. (Note that the same rule also appears in natural language; we use the word 'I' much more than 'stranger')

As you can see, Huffman coding is data-dependent. The encoder (`source`) will execute the above procedure each time there is new data that needs to be encoded. Once the code look-up table is made, the encoder will convert the 8-bit ascii codes into the Huffman codewords. Because of the data-dependent property, the encoder must send the statistics of each symbols along with the compressed data. The statistics can be contained in the header. The decoder (`sink`) will reconstruct the Huffman tree using the statistics and will now have knowledge of the code. Because the codewords are generated by each path from the root of the tree, one codeword cannot be a prefix of another codeword. This allows the sink to decode each codeword by just matching with the lookup table from the the start of the bit sequence to the end, even though the codeword length is not fixed. As you progress along the bit sequences and notice a codeword, you decode with the symbol that matches in the lookup table.

The reason why we do not include symbols with no appearance in the Huffman tree is because we do not want to make the tree unnecessarily large and thus increase the overall codeword length. For example, what if you have three

symbols ‘a’, ‘b’, ‘c’, but ‘c’ does not appear. If we include ‘c’ in the Huffman tree, the respective codewords for the three symbols will be 0, 10, 11. This is not optimal since we can just assign codewords 0, 1 only for ‘a’ and ‘b’ which is shorter overall.

Implementation

source.py

Huffman coding will be performed after the `ENCODE` function of the original `SOURCE` module. The Source Coding header will be added after the source bits from the `ENCODE` function are compressed by Huffman coding. You will also need to change the contents of the Milestone 1 source header.

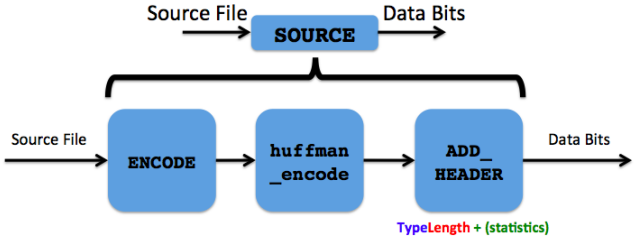


Figure 6. New `SOURCE` module

To complete this code, you will need to define (or modify) the following functions :

- process : Overarching function which should return the data bits. You need to modify the function to execute `huffman_encode` function after the `ENCODE` function when the source type is text or image.
  - Input: NULL
  - Output: source bits, data bits (header + huffman coded source bits)
    - Note that source bits are not same as data bits without header.
- huffman_encode : Performs the Huffman encoding given the source bits. Read through the above primer for the detail of the algorithm.
  - Input: source bits
  - Outputs: data statistics, Huffman encoded bits
    - In the primer, we broke the source bits into blocks of 8-bit symbols to show you a comfortable example. However, using 8-bit symbols will produce a large amount of overhead when storing the statistics due to the wide range of symbols. Thus, regardless of the data type, **we recommend you to use blocks of 4-bit symbols**. However if you find that 8-bit or even 16-bit blocks work better, you can write about it in a README.txt file for extra credit! (in this case, make sure you don’t forget to handle padding)
- get_header : Original function for Milestone 1. Needs to be modified to contain the information for Huffman decoding.
  - Input: source type, Huffman encoded source bits length, data statistics
  - Output: header bits. (recommend 18 bits for monotone, 18+160 bits for text and image)
    - You specify the source type using 2 bits as previous milestone. Also, specify the Huffman encoded bits length using 16 bits.
    - If the source type is text or image, you need to assign an additional 10 bits to store the appearance count for each symbol. If you are using 4 bits for each symbol, you need a total of 10*2^4 = 160 bits to store the entire statistics. (This seems to be an excessively long header. That’s true for small data. But the Huffman coding gets more powerful as the size of the data grows. So to optimize, we could consider data compression – header overhead trade-off but this is not done in this milestone.)

sink.py

Header analysis is done in a 2-step process, since the header format is different for monotone and text/image. After the header is read and the data statistics are retrieved from the `read_type_size`, `read_stat` functions (extended version of `read_header`), Huffman decoding will be performed on the encoded data bits using the data statistics in the header. The resulted source bits will be fed to the `DECODE` function.

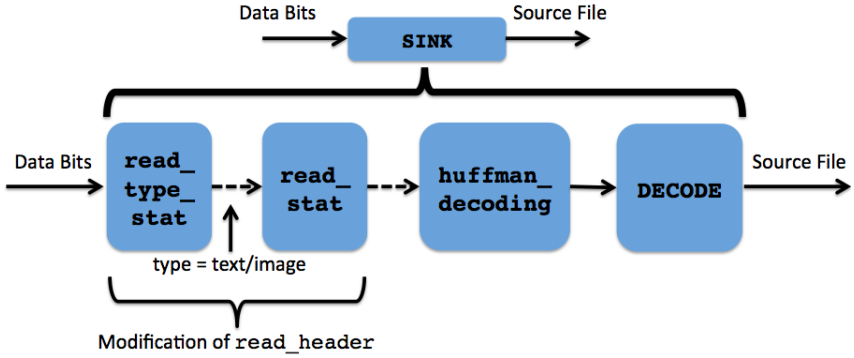


Figure 7. New `SINK` module

To complete this code, you will need to define (or modify) the following functions :

- process : Overarching function which is called with the received bits as input. You need to modify the function to execute `huffman_decode` function before the `DECODE` function only when the source type is text or image. The input of `huffman_decode` should be the huffman source bits of length matched to the result of read_type_size. Also replace `read_header` with 2-step process of `read_type_size` and `read_stat`. `read_stat` is only executed when the source type is text or image.
  - Input : received bits (source header + huffman coded source bits)
  - Output: same as original
- read_type_size : Figure out source type and Huffman coded source bits length by reading first portion of header bits. (Same as original `read_header`)

- Input : first part of header of received bits (which is allocated for source type and length)
- Output : source type, Huffman coded source bits length

- read_stat : Figure out appearance counts of all 4-bit symbols from the extended header bits
  - Input : extended header bits for (source type = text or image)
  - Output: data statistics
- huffman_decode : Performs hamming decoding given the received huffman coded source bits. Read through the above primer for the detail of the algorithm.
  - Input : Data statistics, received Huffman coded source bits
    - Make sure the received Huffman coded source bits are tailored in `process`. (refer to process) There might be additional zeros that was padded when doing channel coding.
  - Output : source bits

Channel Coding

Primer

Communication systems are only useful if they can reliably deliver a message. However, as you may have noticed in the first two milestones, errors during audio transmission will cause the message to be garbled or sometimes unintelligible. In order to improve our communication system so that these errors can be corrected, we will implement an error correcting coding scheme that will undo the errors caused by noise.

There are many schemes designed to check and correct for errors; however for this milestone your job is to implement Hamming encoding. Recall our previous discussion of Hamming distance, D, defined as the number of bit positions where two vectors vary (e.g. for vectors [000] and [111], D = 3). You will see that by using Hamming codes we can correct (D - 1) / 2 bit errors.

Lets start with a simple example so you can understand how Hamming codes can correct bit errors.

Suppose we encode the value “heads” as [000] and “tails” as [111]. If want to transmit “tails” to the receiver, we would send 111. Now, if noise caused a bit flip at the middle position, resulting in the receiver seeing 101, we would correctly guess that the message was originally “tails.” This follows because the Hamming distance between [111] and [101] is less than the hamming distance between [000] and [101]. This can be seen visually in the figure below:

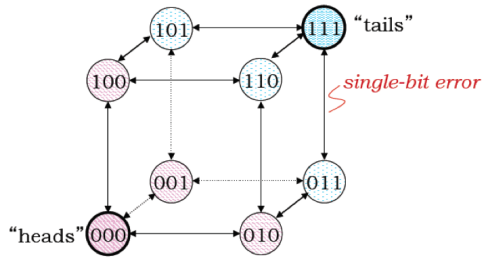


Figure 7. Pictorial representation of simple Hamming Code

One thing to be aware of is that if two bits flip in the same coded block (in our example the coding block was [111]) the receiver will be unable to recover the original message!

One of the characteristics that make Hamming codes so useful is that you can adjust the coding rate depending on the level of noise you expect to encounter. in our simple example above the coding rate was 33% because every bit was encoded using 3 bits. We can increase this rate to 57% by encoding 4 data bits in a 7 bit coded block. While this will result in a higher data rate, it will only be able to correct up to 1 bit error for every 7 transmitted bits (in our previous example we could correct for 1 bit in every 3 transmitted bits). For this milestone you will be required to set different coding rates (with the -H option) for different bypass noise levels (set with the -z option).

The other characteristic that makes hamming codes indispensable is that they can be implemented on both the receiver and transmitter using matrix multiplication. For a Hamming code that takes blocks of k bits and outputs n bit coded blocks (in our 2nd example k = 4 and n = 7) we can write the data bits as a vector D with bits D1,D2,...Dk. Similarly we can write coded bits as C with bits C1,C2,...Cn. To convert D to C we simply multiply by a (k x n) **generation matrix**, G:

$$D \cdot G = C$$

To decode linear block codes, we use ***syndrome decoding***. For a given generation matrix G, there is a corresponding ***parity check matrix***, H that satisfy the following.

$$HG^T = 0$$

The name comes from the fact that if we multiply the codeword with H, we are checking the sum of some portion of data bits and the corresponding parity bit. For example, the following is the parity check matrix for (7,4)-Hamming code.

$$\mathbf{H} := \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}_{3,7}.$$

If we multiply a column vector [D1; D2; D3; D4; P1; P2; P3] representing the codeword where Di is the data

bit and Pi is the parity bit, the multiplication is checking whether the follow statements holds.

$$P1 = D1 + D2 + D4$$
$$P2 = D1 + D3 + D4$$
$$P3 = D2 + D3 + D4$$

So if we multiply c^T(column vector of the codeword) with H, the result should be 0.

$$Hc^T = H(dG)^T = (HG^T)d = 0$$

But what if there is a single bit error at the i-th bit? Let us model the error pattern as $e_i = (0,0,\cdots,1,\cdots,0)$ where 1 is only at the i-th entry. The received bit will be $r = c + e_i$. Then the result of the multiplication with the parity check matrix will be $H(c+e_i)^T = Hc^T + He_i^T = h_i$, which is the i-th column of h. In practice we don't know e_i. So by comparing the result of the multiplication with the columns of the parity check matrix, we can figure out what is the corresponding e_i, which eventually tells us which bit is corrupted. This is how Hamming codes can correct 1 bit error. For more details, refer to the course reader section 5.7, 6.1, 6.3

Implementation

The main goal for channel encoding is to correct any errors caused by the channel. After getting your implementation working, you will need to test whether you achieve Average BER values similar to those in the sample RESULTS.txt. You will be editing the files `transmitter.py`, `receiver.py`, `common_txrx.py` and `hamming_db.py`. We have modified sendrecv.py to call the functions `encode` and `decode` in the `transmitter` and `receiver`. These functions are responsible for calling the various helper functions to `encode` and `decode` the data bits.

To complete this code, you will need to define (or modify) the following functions :

`hamming_db.py` *We recognize that this code could be improved by making it more object oriented. If you are an industrious student who would like to take this on, you are welcome to improve it!*

- generating_matrices: This is a global array of all the G matrices associated with different Hamming Codes. We give you all the codes you need, but if you want to add more you can generate them online or use matlab (help `hammgen`).
- parameters: This is a global array of the n,k parameters corresponding to each G matrix in generating_matrices.
- gen_lookup: Given a channel coding length (cc_len passed in with parameter -H) value the function will determine which G matrix has an n value closest to this length. It will reshape the numpy array into a numpy array of numpy arrays for encoding.
  - Inputs: cc_len
  - Outputs: n, k, index, G
    - You will need to have some way of selecting an appropriate G matrix from generating_matrices based on which n value (stored in parameters) is closest to cc_len.
    - For reshaping purposes check out the numpy reshape and concatenate functions. **Important:** Because numpy arrays are expressed as row vectors, G will be k x n so that we can perform the dot product with a 1 x k row vector. G should be returned in the form [identity matrix(k x k) | A(k x (n-k))] where A corresponds to which bit positions the parity bits are associate with (in the primer example P3 = D2+D3+D4). ***NOTE:*** *We do NOT give you the generating matrices in this form!!!* Please read through Section 6.1.1 Examples in the course reader if you are confused.
    - Output "index" corresponds to the index into your generating_matrices array. This value will be transmitted in your channel coding header as parameter specifying the coding rate.
- parity_lookup: Given the index extracted from the channel coding header, this function looks up the generating matrix G and transforms it into the parity matrix H.
  - Inputs: index
  - Outputs n, k, H
    - After reshaping the G array as you did in gen_lookup, you will need to extract the A submatrix. H is simply the transpose of A(k x (n-k)) concatenated with the identity matrix((n-k) x (n-k)).

`transmitter.py`

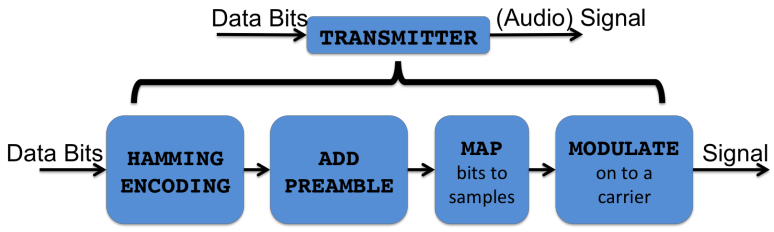


Figure 8. New `Transmitter` module

- encode: Overarching function that calls hamming_encoding on data bits to create coded bits. The function also creates the channel coding header and encodes it as well and returns the entire coded frame.
  - Inputs: databits
  - Outputs: coded_bits (coded header + coded data)
    - The header contains the length of the coded frame (including the coded header) and a parameter that specifies which coding rate you used for encoding the data portion (this is the index into generating_matrices returned by hamming_encoding). Since the

`transmitter` varies the coding rate of the data based on the user specified channel coding length, the `receiver` must rely on the parameter specified in the header for decoding. The header also needs to be encoded to protect against bit errors. Since the coding rate is inside the header, the `receiver` will not be able to decode the header unless we pre-specify its coding rate. For this reason, ***we must always encode the header using a fixed conservative rate*** (Hamming (3,1), which is the bit duplication example in the primer).

- hamming_encoding: This function is used to encode both the data and channel header of the frame. It breaks the input bits into k sized blocks (padding with 0's if the input is not a multiple of k) and encodes them using the G matrix corresponding to the channel coding length (cc_len) specified by the user with the -H flag. These coded bits are extended to an array and returned.
  - Inputs: databits, is_header
  - Outputs: index, coded_bits
    - If the is_header input is True you will call the gen_lookup function in hamming_db with a value of 3 to produce the most conservative rate G matrix. Otherwise you will lookup the matrix based on self.cc_len.
    - Please use numpy.dot for multiplication of databits and G and make sure your output array is ONLY ones and zeros.

`receiver.py`

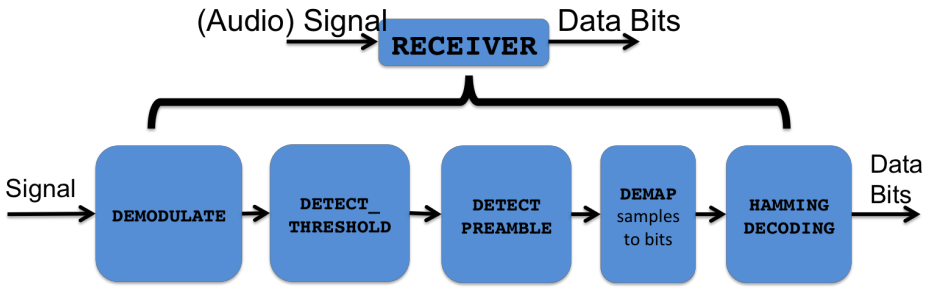


Figure 9. New `Receiver` module

- decode: Overarching function that calls hamming_decoding on coded bits to extract the original data bits. The function first decodes the header to get the coding rate (index into generating_matrices) and then decodes the payload.
  - Inputs: rcd_bits
  - Outputs: decoded_data
    - Using shared knowledge of the channel header length and conservative coding rate we can extract the payload length and index fields from channel header.Then using the index we can decode the data.
    - Please print the following string for grading purposes: "channel coding rate: {rate}" with {rate} being the k/n.
- hamming_decoding: This function decodes received bits based on the rate specified by index. It calls the parity_lookup function to get n, k and H, which is uses to decode coded blocks of size n into data blocks of size k.
  - Inputs: coded_bits, index
  - Outputs: decoded_bits
    - After looking up the H matrix and splitting the coded bits into n blocks you will need to calculate the syndrome. If the syndrome is all 0s, you have no errors and can simply output the data bits in the coded block. Otherwise you need to decode. If there is an error, the syndrome will exactly match a column in H. Therefore, we compare the columns of H to the syndrome and the column that matches will correspond to the bit index that needs to be corrected. Note that we only need to compare the first k columns because we don't need to correct erroneous parity bits. Note, this can all be done in a single for loop using linear algebra and array manipulation.
    - Please calculate the number of errors corrected and print the following string for grading purposes: "errors corrected: {num_errors}"

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`, `source.py`, `sink.py`, `common_srcsink`, `hamming_db.py`, `RESULTS.txt`. Do NOT rename the files, do NOT create sub-folders (upload these files into your main drop box folder for the course)
5. *If you are implementing anything special or will also submit Milestone 3 - Modulation and Demodulation, place a*