# ecc lab

Coding the Matrix, Summer 2013

Please fill out the stencil file named "`ecc_lab.py`". While we encourage you to complete the Ungraded Problems, they do not require any entry into your stencil file.

## 1 *Lab: Error-correcting codes*

> The purpose of computing is insight, not numbers.
>
> Richard Hamming

In this lab, we work with vectors and matrices over $GF(2)$. So when you see 1's and 0's in this description, remember that each 1 is really the value `one` from the module `GF2`.

### 1 *The check matrix*

As we have seen, in a linear binary code, the set $\mathcal{C}$ of codewords is a vector space over $GF(2)$. In such a code, there is a matrix $H$, called the *check matrix*, such that $\mathcal{C}$ is the null space of $H$. When the Receiver receives the vector $\tilde{\boldsymbol{c}}$, she can check whether the received vector is a codeword by multiplying it by $H$ and checking whether the resulting vector (called the *error syndrome*) is the zero vector.

### 2 *The generator matrix*

We have characterized the vector space $\mathcal{C}$ as the null space of the check matrix $H$. There is another way to specify a vector space: in terms of generators. The *generator matrix* for a linear code is a matrix $G$ whose columns are generators for the set $\mathcal{C}$ of codewords.[1]

By the linear-combinations definition of matrix-vector multiplication, every matrix-vector product $G * \boldsymbol{p}$ is a linear combination of the columns of $G$, and is therefore a codeword.

### 3 *Hamming's code*

Hamming discovered a code in which a four-bit message is represented by a seven-bit *codeword*. The generator matrix is

$$G = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

A four-bit message is represented by a 4-vector $\boldsymbol{p}$ over $GF(2)$. The encoding of $\boldsymbol{p}$ is the 7-vector resulting from the matrix-vector product $G * \boldsymbol{p}$.

[1] It is traditional to define the generator matrix so that its *rows* are generators for $\mathcal{C}$. We diverge from this tradition for the sake of simplicity of presentation.

Let $f_G$ be the encoding function, the function defined by $f_G(\boldsymbol{x}) = G * \boldsymbol{p}$. The image of $f_G$, the set of all codewords, is the row space of $G$.

**Task 1:** Create an instance of `Mat` representing the generator matrix $G$. You can use the procedure `listlist2mat` in the `matutil` module. Since we are working over $GF(2)$, you should use the value `one` from the `GF2` module to represent 1. What is the encoding of the message $[1, 0, 0, 1]$?

## 4 *Decoding*

Note that four of the rows of $G$ are the standard basis vectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3, \boldsymbol{e}_4$ of $GF(2)^4$. What does that imply about the relation between words and codewords? Can you easily decode the codeword $[0, 1, 1, 1, 1, 0, 0]$ without using a computer?

**Task 2:** Think about the manual decoding process you just did. Construct a $4 \times 7$ matrix $R$ such that, for any codeword $\boldsymbol{c}$, the matrix-vector product $R * \boldsymbol{c}$ equals the 4-vector whose encoding is $\boldsymbol{c}$. What should the matrix-matrix product $RG$ be? Compute the matrix and check it against your prediction.

## 5 *Error syndrome*

Suppose Alice sends the codeword $\boldsymbol{c}$ across the noisy channel. Let $\tilde{\boldsymbol{c}}$ be the vector received by Bob. To reflect the fact that $\tilde{\boldsymbol{c}}$ might differ from $\boldsymbol{c}$, we write

$$\tilde{\boldsymbol{c}} = \boldsymbol{c} + \boldsymbol{e}$$

where $\boldsymbol{e}$ is the error vector, the vector with ones in the corrupted positions.

If Bob can figure out the error vector $\boldsymbol{e}$, he can recover the codeword $\boldsymbol{c}$ and therefore the original message. To figure out the error vector $\boldsymbol{e}$, Bob uses the check matrix, which for the Hamming code is

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

As a first step towards figuring out the error vector, Bob computes the *error syndrome*, the vector $H * \tilde{\boldsymbol{c}}$, which equals $H * \boldsymbol{e}$.

Examine the matrix $H$ carefully. What is special about the order of its columns?

Define the function $f_H$ by $f_H(\boldsymbol{y}) = H * \boldsymbol{y}$. The image under $f_H$ of any codeword is the zero vector. Now consider the function $f_H \circ f_G$ that is the composition of $f_H$ with $f_G$. For any vector $\boldsymbol{p}$, $f_G(\boldsymbol{p})$ is a codeword $\boldsymbol{c}$, and for any codeword $\boldsymbol{c}$, $f_H(\boldsymbol{c}) = \boldsymbol{0}$. This implies that, for any vector $\boldsymbol{p}$, $(f_H \circ f_G)(\boldsymbol{p}) = \boldsymbol{0}$.

The matrix $HG$ corresponds to the function $f_H \circ f_G$. Based on this fact, predict the entries of the matrix $HG$.

**Task 3:** Create an instance of `Mat` representing the check matrix $H$. Calculate the matrix-matrix product $HG$. Is the result consistent with your prediction?

## 6 *Finding the error*

Bob assumes that at most one bit of the codeword is corrupted, so at most one bit of $\boldsymbol{e}$ is nonzero, say the bit in position $i \in \{1, 2, \ldots, 7\}$. In this case, what is the value of $H * \boldsymbol{e}$? (Hint: this uses the special property of the order of $H$'s rows.)

**Task 4:** Write a procedure `find_error` that takes an error syndrome and returns the corresponding error vector $\boldsymbol{e}$.

Imagine that you are Bob, and you have received the *non*-codeword $\tilde{\boldsymbol{c}} = [1, 0, 1, 1, 0, 1, 1]$. Your goal is to derive the original 4-bit message that Alice intended to send. To do this, use `find_error` to figure out the corresponding error vector $\boldsymbol{e}$, and then add $\boldsymbol{e}$ to $\tilde{\boldsymbol{c}}$ to obtain the correct codeword. Finally, use the matrix $R$ from Task 2 to derive the original 4-vector.

**Task 5:** Write a one-line procedure `find_error_matrix` with the following spec:

- *input:* a matrix `S` whose columns are error syndromes
- *output:* a matrix whose $c^{th}$ column is the error corresponding to the $c^{th}$ column of `S`.

This procedure consists of a comprehension that uses the procedure `find_error` together with some procedures from the `matutil` module.

Test your procedure on a matrix whose columns are $[1, 1, 1]$ and $[0, 0, 1]$.

## 7 *Putting it all together*

We will now encode an entire string and will try to protect it against errors. We first have to learn a little about representing a text as a matrix of bits. Characters are represented using a variable-length encoding scheme called *UTF-8*. Each character is represented by some number of bytes. You can find the value of a character c using `ord(c)`. What are the numeric values of of the characters 'a', 'A' and space?

You can obtain the character from a numerical value using `chr(i)`. To see the string of characters numbered 0 through 255, you can use the following:

```
>>> s = ''.join([chr(i) for i in range(256)])
>>> print(s)
```

We have provided a module `bitutil` that defines some procedures for converting between lists of $GF(2)$ values, matrices over $GF(2)$, and strings. Two such procedures are `str2bits(str)` and `bits2str(L)`:

The procedure `str2bits(str)` has the following spec:

- *input:* a string
- *output:* a list of $GF(2)$ values (0 and one) representing the string

The procedure `bits2str(L)` is the inverse procedure:

- *input:* a list of $GF(2)$ values
- *output:* the corresponding string

**Ungraded Task:** Try out `str2bits(str)` on the string `s` defined above, and verify that `bits2str(L)` gets you back the original string.

The Hamming code operates on four bits at a time. A four-bit sequence is called a *nibble* (sometimes *nybble*). To encode a list of bits (such as that produced by `str2bits`), we break the list into nibbles and encode each nibble separately.

To transform each nibble, we interpret the nibble as a 4-vector and we multiply it by the generating matrix $G$. One strategy is to convert the list of bits into a list of 4-vectors, and then use, say, a comprehension to

multiply each vector in that list by $G$. In keeping with our current interest in matrices, we will instead convert the list of bits into a matrix $B$ each column of which is a 4-vector representing a nibble. Thus a sequence of $4n$ bits is represented by a $4 \times n$ matrix $P$. The module `bitutil` defines a procedure `bits2mat(bits)` that transforms a list of bits into such a matrix, and a procedure `mat2bits(A)` that transforms such a matrix $A$ back into a list of bits.

**Ungraded Task:** Try converting a string to a list of bits to a matrix $P$ and back to a string, and verify that you get the string you started with.

**Task 6:** Putting these procedures together, compute the matrix $P$ which represents the string "I'm trying to free your mind, Neo. But I can only show you the door. You're the one that has to walk through it."

Imagine that you are transmitting the above message over a noisy communication channel. This channel transmits bits, but occasionally sends the wrong bit, so one becomes 0 and vice versa.

The module `bitutil` provides a procedure `noise(A, s)` that, given a matrix $A$ and a probability parameter $s$, returns a matrix with the same row- and column-labels as $A$ but with entries chosen from $GF(2)$ according the probability distribution `{one:s, 0:1-s}`. For example, each entry of `noise(A, 0.02)` will be `one` with probability 0.02 and zero with probability 0.98

**Ungraded Task:** To simulate the effects of the noisy channel when transmitting your matrix $P$, use `noise(P, 0.02)` to create a random matrix $E$. The matrix $E + P$ will introduce some errors. To see the effect of the noise, convert the perturbed matrix back to text.

Looks pretty bad, huh? Let's try to use the Hamming code to fix that. Recall that to encode a word represented by the row vector $\boldsymbol{p}$, we compute $G * \boldsymbol{p}$.

**Task 7:** Encode the words represented by the columns of the matrix $P$, obtaining a matrix $C$. You should not use any loops or comprehensions to compute $C$ from $P$. How many bits represented the text before the encoding? How many after?

**Ungraded Task:** Imagine that you send the encoded data over the noisy channel. Use `noise` to construct a noise matrix of the appropriate dimensions with error probability 0.02, and add it to $C$ to obtain a perturbed matrix `CTILDE`. Without correcting the errors, decode `CTILDE` and convert it to text to see how garbled the received information is.

**Task 8:** In this task, you are to write a one-line procedure `correct(A)` with the following spec:

- *input:* a matrix $A$ each column of which differs from a codeword in at most one bit
- *output:* a matrix whose columns are the corresponding valid codewords.

The procedure should contain no loops or comprehensions. Just use matrix-matrix multiplications and matrix-matrix additions together with a procedure you have written in this lab.

**Ungraded Task:** Apply your procedure `correct(A)` to `CTILDE` to get a matrix of codewords. Decode this matrix of codewords using the matrix $R$ from Task 2, obtaining a matrix whose columns are 4-vectors. Then derive the string corresponding to these 4-vectors.

Did the Hamming code succeed in fixing all of the corrupted characters? If not, can you explain why?

**Ungraded Task:** Repeat this process with different error probabilities to see how well the Hamming code does under different circumstances.