GSMST

Applications of Linear Algebra in Programming

Error Correcting Lab

Submitted By:
Anish Goyal
4th Period

Submitted To:
Mrs. Denise Stiffler
Educator

May 8, 2023

Table of contents

4.14.3 Hamming's code

Hamming discovered a code in which a four-bit message is represented by a seven-bit *code-word*. 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 $G * \boldsymbol{p}$.

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 4.14.1

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.

```
# Import all necessary libraries
from mat import Mat
from matutil import listlist2mat
from GF2 import one

G = listlist2mat([
  [one, 0, one, one],
  [one, one, 0, one],
  [0, 0, 0, one],
  [one, one, one, 0],
  [0, 0, one, 0],
  [0, one, 0, 0],
  [one, 0, 0, 0]
])
```

Task 4.14.2

What is the encoding of the message [1, 0, 0, 1]?

```
from vec import Vec
from vecutil import list2vec

p=list2vec([one, 0, 0, one])
encoded = G * p
print(encoded)
```

```
 0 1   2   3 4 5   6
--------------------
 0 0 one one 0 0 one
```

4.14.4 Decoding

Note that four of the rows of G are the standard basis vectors e_1, e_2, e_3, 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?

The codeword vector will always match the i^{th} positions of the word vector corresponding to the standard basis vectors of the generator matrix. This means that the codeword [0, 1, 1, 1, 1, 0, 0] would correspond to the word [0, 0, 1, 1] since rows 7, 6, 5, and 3 are the standard basis vectors of G.

Task 4.14.3

Think about the manual decoding process you just did. Construct a 4×7 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.

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

Performing the operation RG gives us the identity matrix:

```
R = listlist2mat([
[0, 0, 0, 0, 0, 0, one],
[0, 0, 0, 0, 0, one, 0],
[0, 0, 0, 0, one, 0, 0],
[0, 0, one, 0, 0, 0, 0]
])

print(R*G)
```

```
         0   1   2   3
      ----------------
 0   |  one   0   0   0
 1   |    0 one   0   0
 2   |    0   0 one   0
 3   |    0   0   0 one
```

4.14.5 Eror 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.

I believe the matrix HG will look like $\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ because f_H of any codeword is the zero vector for any entry $\boldsymbol{p}$.

Task 4.14.4

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

```
H = listlist2mat([
  [0, 0, 0, one, one, one, one],
  [0, one, one, 0, 0, one, one],
  [one, 0, one, 0, one, 0, one]
])
print(H * G)
```

```
       0 1 2 3
     ---------
 0  |  0 0 0 0
 1  |  0 0 0 0
 2  |  0 0 0 0
```

Just as expected, HG is equal to the 4×4 zero matrix.

4.14.6 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 $H * \boldsymbol{e}$?

$H * \boldsymbol{e}$ should be the value of the i^{th} column of H.

Task 4.14.5

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

```
def find_error(error_syndrome):
  i = sum([2**(2-x) for x in error_syndrome.f.keys() if
↪   error_syndrome.f[x]==one])
  if i==0:
    return Vec({0, 1, 2, 3, 4, 5, 6}, {})
  else:
    return Vec({0, 1, 2, 3, 4, 5, 6}, {i-1:one})
```

Task 4.14.6

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_eror` to figure out the corresponding vector $\boldsymbol{e}$, and then add $\boldsymbol{e}$ to $\tilde{\boldsymbol{c}}$ to obtain the correct codeword. Finally, use the matrix R from `Task 4.14.3` to derive the original 4-vector.

```
non_c = Vec({0, 1, 2, 3, 4, 5, 6}, {0: one, 1:0, 2:one, 3:one, 4:0,
↪   5:one, 6:one})
error = find_error(H*non_c)
c = non_c+error
original = R*c
print(original)
```

```
 0   1 2   3
------------
 0 one 0 one
```

Task 4.14.7

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].

```
from matutil import coldict2mat, mat2coldict

def find_error_matrix(S):
  return coldict2mat({key: find_error(mat2coldict(S)[key]) for key in mat2coldict(S).keys()})

S = listlist2mat([[one, 0],[one, 0],[one, one]])
print(find_error_matrix(S))
```

```
          0   1
     ---------
 0   |     0 one
 1   |     0   0
 2   |     0   0
 3   |     0   0
 4   |     0   0
 5   |     0   0
 6   |   one   0
```

4.14.7 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 the characters 'a', 'A', and space?

```
print(ord('a'), ord('A'), ord(' '))
```

```
97 65 32
```

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(0, 256) if chr(i).isprintable()]) # Some characters are excluded due to the fact that they are not UTF-8 encodable.
print(s)
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
   abcdefghijklmnopqrstuvwxyz{|}~
   ÂąÂćÂčÂďÂěÂęÂğÂĺÂľÂłÂńÂňÂőÂŕÂřÂśÂšÂşÂťÂţÂűÂůÂÿÂźÂžÂżÂ Â¡Â¿Â£ÃÃÃĄÃĆÃČÃĎÃĚÃĘÃĞÃĹÃĽÃŁÃŃÃŇÃĐÃ
```

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

Task 4.14.8

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

```python
from bitutil import *
print(str2bits(s))
print(bits2str(str2bits(s)))
print(bits2str(str2bits(s)) == s)
```

```
[0, 0, 0, 0, 0, one, 0, 0, one, 0, 0, 0, 0, one, 0, 0, 0, one, 0, 0, 0, one,
0, 0, one, one, 0, 0, 0, one, 0, 0, 0, 0, one, 0, 0, one, 0, 0, one, 0, one,
0, 0, one, 0, 0, 0, one, one, 0, 0, one, 0, 0, one, one, one, 0, 0, one, 0,
0, 0, 0, 0, one, 0, one, 0, 0, one, 0, 0, one, 0, one, 0, 0, 0, one, 0, one,
0, one, 0, 0, one, one, 0, one, 0, one, 0, 0, 0, 0, one, one, 0, one, 0, 0,
one, 0, one, one, 0, one, 0, 0, 0, one, one, one, 0, one, 0, 0, one, one,
one, one, 0, one, 0, 0, 0, 0, 0, 0, one, one, 0, 0, one, 0, 0, 0, one, one,
0, 0, 0, one, 0, 0, one, one, 0, 0, one, one, 0, 0, one, one, 0, 0, 0, 0,
one, 0, one, one, 0, 0, one, 0, one, 0, one, one, 0, 0, 0, one, one, 0, one,
one, 0, 0, one, one, one, 0, one, one, 0, 0, 0, 0, 0, one, one, one, 0, 0,
one, 0, 0, one, one, one, 0, 0, 0, one, 0, one, one, one, 0, 0, one, one, 0,
one, one, one, 0, 0, 0, 0, one, one, one, one, 0, 0, one, 0, one, one, one,
one, 0, 0, 0, one, one, one, one, one, 0, 0, one, one, one, one, one, one,
0, 0, 0, 0, 0, 0, 0, 0, one, 0, one, 0, 0, 0, 0, 0, one, 0, 0, one, 0, 0, 0,
0, one, 0, one, one, 0, 0, 0, 0, one, 0, 0, 0, one, 0, 0, 0, one, 0, one, 0,
one, 0, 0, 0, one, 0, 0, one, one, 0, 0, 0, one, 0, one, one, one, 0, 0, 0,
one, 0, 0, 0, 0, one, 0, 0, one, 0, one, 0, 0, one, 0, 0, one, 0, 0, one, 0,
one, 0, 0, one, 0, one, one, 0, one, 0, 0, one, 0, 0, 0, one, one, 0, 0, one,
0, one, 0, one, one, 0, 0, one, 0, 0, one, one, one, 0, 0, one, 0, one, one,
one, one, 0, 0, one, 0, 0, 0, 0, 0, one, 0, one, 0, one, 0, 0, 0, one, 0,
one, 0, 0, one, 0, 0, one, 0, one, 0, one, one, 0, 0, one, 0, one, 0, 0, 0,
one, 0, one, 0, one, 0, one, 0, one, 0, one, 0, one, 0, 0, one, one, 0, one,
0, one, 0, one, one, one, 0, one, 0, one, 0, 0, 0, 0, one, one, 0, one, 0,
one, 0, 0, one, one, 0, one, 0, 0, one, 0, one, one, 0, one, 0, one, one, 0,
one, one, 0, one, 0, 0, 0, one, one, one, 0, one, 0, one, 0, one, one, one,
0, one, 0, 0, one, one, one, one, 0, one, 0, one, one, one, one, one, 0, one,
0, 0, 0, 0, 0, 0, one, one, 0, one, 0, 0, 0, 0, one, one, 0, 0, one, 0, 0,
0, one, one, 0, one, one, 0, 0, 0, one, one, 0, 0, 0, one, 0, 0, one, one,
0, one, 0, one, 0, 0, one, one, 0, 0, one, one, 0, 0, one, one, 0, one, one,
one, 0, 0, one, one, 0, 0, 0, 0, one, 0, one, one, 0, one, 0, 0, one, 0, one,
one, 0, 0, one, 0, one, 0, one, one, 0, one, one, 0, one, 0, one, one, 0, 0,
0, one, one, 0, one, one, 0, one, 0, one, one, 0, one, one, 0, 0, one, one,
```

one, 0, one, one, 0, one, one, one, one, 0, one, one, 0, 0, 0, 0, 0, one,
one, one, 0, one, 0, 0, 0, one, one, one, 0, 0, one, 0, 0, one, one, one,
0, one, one, 0, 0, one, one, one, 0, 0, 0, one, 0, one, one, one, 0, one, 0,
one, 0, one, one, one, 0, 0, one, one, 0, one, one, one, 0, one, one, one, 0,
one, one, one, 0, 0, 0, 0, one, one, one, one, 0, one, 0, 0, one, one, one,
one, 0, 0, one, 0, one, one, one, one, 0, one, one, 0, one, one, one, one,
0, 0, 0, one, one, one, one, one, 0, one, 0, one, one, one, one, one, 0, 0,
one, one, one, one, one, one, 0, one, 0, 0, 0, 0, one, 0, one, 0, one, 0, 0,
0, one, 0, one, one, one, 0, 0, 0, one, 0, one, 0, 0, one, 0, 0, one, 0, one,
one, 0, one, 0, 0, one, 0, one, 0, one, one, 0, 0, one, 0, one, one, one,
one, 0, 0, one, 0, one, 0, 0, 0, one, 0, one, 0, one, one, 0, 0, one, 0, one,
0, one, 0, one, 0, one, 0, one, 0, one, one, one, 0, one, 0, one, 0, one, 0,
0, one, one, 0, one, 0, one, 0, one, one, one, 0, one, 0, one, one, one, one,
one, 0, one, 0, one, 0, 0, 0, 0, one, one, 0, one, one, 0, 0, 0, one, one,
0, one, 0, one, 0, 0, one, one, 0, one, one, one, 0, 0, one, one, 0, one, 0,
0, one, 0, one, one, 0, one, one, 0, one, 0, one, one, 0, one, 0, one, one,
0, one, one, 0, one, one, one, one, 0, one, one, 0, one, 0, 0, 0, one, one,
one, 0, one, one, 0, 0, one, one, one, 0, one, 0, one, 0, one, one, one, 0,
one, one, one, 0, one, one, one, 0, one, 0, 0, one, one, one, one, 0, one,
one, 0, one, one, one, one, 0, one, 0, one, one, one, one, one, 0, one, one,
one, one, one, one, one, 0, one, 0, 0, 0, 0, 0, 0, one, one, one, 0, 0, 0, 0,
0, one, one, 0, one, 0, 0, 0, 0, one, one, one, one, 0, 0, 0, 0, one, one, 0,
0, one, 0, 0, 0, one, one, one, 0, one, 0, 0, 0, one, one, 0, one, one, 0, 0,
0, one, one, one, one, one, 0, 0, 0, one, one, 0, 0, 0, one, 0, 0, one, one,
one, 0, 0, one, 0, 0, one, one, 0, one, 0, one, 0, 0, one, one, one, one, 0,
one, 0, 0, one, one, 0, 0, one, one, 0, 0, one, one, one, 0, one, one, 0, 0,
one, one, 0, one, one, one, 0, 0, one, one, one, one, one, one, 0, 0, one,
one, 0, 0, 0, 0, one, 0, one, one, one, 0, 0, 0, one, 0, one, one, 0, one, 0,
0, one, 0, one, one, one, one, 0, 0, one, 0, one, one, 0, 0, one, 0, one, 0,
one, one, one, 0, one, 0, one, 0, one, one, 0, one, one, 0, one, 0, one, one,
one, one, one, 0, one, 0, one, one, 0, 0, 0, one, one, 0, one, one, one, 0,
0, one, one, 0, one, one, 0, one, 0, one, one, 0, one, one, one, one, 0, one,
one, 0, one, one, 0, 0, one, one, one, 0, one, one, one, 0, one, one, one,
0, one, one, 0, one, one, one, one, 0, one, one, one, one, one, one, one, 0,
one, one, 0, 0, 0, 0, 0, one, one, one, one, 0, 0, 0, 0, one, one, one, 0,
one, 0, 0, 0, one, one, one, one, one, 0, 0, 0, one, one, one, 0, 0, one, 0,
0, one, one, one, one, 0, one, 0, 0, one, one, one, 0, one, one, 0, 0, one,
one, one, one, one, one, 0, 0, one, one, one, 0, 0, 0, one, 0, one, one, one,
one, 0, 0, one, 0, one, one, one, 0, one, 0, one, 0, one, one, one, one, one,
0, one, 0, one, one, one, 0, 0, one, one, 0, one, one, one, one, 0, one, one,
0, one, one, one, 0, one, one, one, 0, one, one, one, one, one, one, one, 0,
one, one, one, 0, 0, 0, 0, one, one, one, one, one, 0, 0, 0, one, one, one,
one, 0, one, 0, 0, one, one, one, one, one, one, 0, 0, one, one, one, one,

```
0, 0, one, 0, one, one, one, one, one, 0, one, 0, one, one, one, one, 0, one,
one, 0, one, one, one, one, one, one, one, 0, one, one, one, one, 0, 0, 0,
one, one, one, one, one, one, 0, 0, one, one, one, one, one, 0, one, 0, one,
one, one, one, one, one, one, 0, one, one, one, one, one, 0, 0, one, one,
one, one, one, one, one, 0, one, one, one, one, one, one, 0, one, one, one,
one, one, one, one, one, one, one, one, one, one, one, one]
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
    abcdefghijklmnopqrstuvwxyz{|}~
    ÂąÂćÂčÂďÂěÂęÂğÂĺÂľÂłÂńÂňÂőÂŕÂřÂśÂšÂşÂťÂţÂűÂůÂÿÂźÂžÂżÂ Â¡Â¿Â£ÃÃÃĄÃĆÃČÃĎÃĚÃĘÃĞÃĹÃĽÃŁÃŃÃŇÃĐÃ
```

```
True
```

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 a matrix, and a procedure `mat2bits(A)` that transforms a matrix A back into a list of bits.

Task 4.14.9

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.

```
s = "Hello world!"
print(s == bits2str(mat2bits(bits2mat(str2bits(s)))))
```

```
True
```

Task 4.14.10

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."

```
s = "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."
P = bits2mat(str2bits(s))
print(P)
```

```
          0    1   10  100  101  102  103  104  105  106  107  108  109   11  110  111  112  113
             114  115  116  117  118  119   12  120  121  122  123  124  125  126  127  128
            129   13  130  131  132  133  134  135  136  137  138  139   14  140  141  142
            143  144  145  146  147  148  149   15  150  151  152  153  154  155  156  157
            158  159   16  160  161  162  163  164  165  166  167  168  169   17  170  171
            172  173  174  175  176  177  178  179   18  180  181  182  183  184  185  186
            187  188  189   19  190  191  192  193  194  195  196  197  198  199    2  20  200
             201  202  203  204  205  206  207  208  209   21  210  211  212  213  214  215
            216  217  218  219   22  220  221  222  223   23   24   25  26   27   28   29    3
              30   31   32   33   34   35  36   37   38   39    4   40   41   42   43   44
            45  46   47   48   49    5   50   51   52   53   54   55   56   57  58   59  6   60
              61   62   63   64   65   66   67  68   69    7   70   71   72   73   74   75  76
              77   78   79    8  80   81   82   83   84   85   86   87  88   89    9   90   91
              92   93   94   95   96   97  98   99
    ------------------------------------------------------------------------------------------

0   |  one   0   0 one one   0   0 one   0 one one   0   0 one one one one   0
      one one   0   0   0 one one   0   0 one   0   0   0   0   0 one   0 one
     one   0   0 one   0   0   0   0 one one one one   0 one one one   0   0
     one one   0   0   0   0   0 one   0   0 one   0   0   0   0 one   0   0
     0 one   0   0   0   0 one   0   0   0 one   0   0 one   0   0   0   0 one
     one   0 one one   0   0   0 one one   0   0   0   0 one one one   0   0
     0 one   0 one  0   0   0   0 one   0   0   0 one one   0   0 one one one
       0   0   0   0   0 one   0   0   0 one   0   0 one one   0  0   0   0   0
        0   0 one one   0 one   0  0   0 one one one one   0 one one   0 one  0
        0 one   0   0 one   0   0   0   0   0   0   0  0   0 0   0   0 one   0
     one   0   0   0  0   0   0   0   0 one one   0 one  0   0 one   0   0  0
       0 one   0 one   0   0   0  0   0 one one   0   0   0   0   0 one one  0
       0
1   |    0   0 one one one   0 one one one one one   0 one one   0 one one one
        0 one   0 one   0 one   0   0 one   0 one   0 one   0 one one one one
     one one one one one one   0 one   0   0   0 one one   0 one one one one
     one   0 one one   0 one   0 one   0 one   0 one   0 one one one one one
     one   0 one   0 one   0 one one   0 one   0 one   0 one   0 one   0 one
     one   0 one one one   0 one   0 one one one one   0 one one one   0 one
     0 one one one one   0   0 one   0 one   0 one one one one one one   0 one
     one one   0 one   0 one   0 one   0   0 one one one one one one  0 one one
      one one one one   0 one   0 one  0 one   0 one   0 one one   0 one one
     one  0 one   0 one one   0 one one one   0 one   0 one  0 one 0 one   0
     0 one one one one one  0 one one one   0   0 one   0 one  0 one   0   0
     0  0 one one one   0 one one one  0 one one one one one one   0 one   0
     one  0 one
2   |    0 one   0   0 one   0 one one one one one   0   0 one   0 one one one
      one one   0   0 one one   0   0 one one one   0   0 one one one one one
     one one   0 one one   0   0   0   0 one   0 one one one one one   0   0
```

```
    one one one one   0   0 one one   0 one one one   0   0 one one one one
    one one one   0   0 one one one   0 one   0 one one one   0   0   0 one
    one   0 one   0 one   0   0 one one one one one   0   0 one one   0 one
    one one   0 one one  0   0   0 one one   0 one   0 one one one   0 one one
     one one   0 one   0   0   0 one one one one one   0 one one one  0   0
    one one   0   0 one one one one one  0   0   0 one one one one one one   0
     one  0   0 one one one   0 one one one one one one   0  0   0 0 one one
    one one one one one   0  0   0   0   0 one one one one one  0   0   0 one
    one  0   0   0 one   0 one one one  0   0 one one one one one one one   0
    one  0   0
3  |  one   0   0   0   0 one   0 one   0   0   0   0   0   0 one   0 one   0
       0   0   0   0   0   0 one one   0   0   0   0   0   0   0 one   0   0
    one   0   0   0 one   0   0   0 one   0 one one   0   0   0   0   0   0
    0   0   0   0   0   0   0   0 one   0   0   0   0   0 one one   0 one   0
      0   0   0   0   0   0   0 one   0   0   0   0   0   0   0 one   0   0
    0   0   0   0   0   0   0   0 one   0   0   0   0   0   0   0   0 one   0
    one   0   0  0   0   0   0   0 one   0   0   0 one   0   0   0   0   0   0
     one   0   0   0 one   0   0   0   0 one   0   0 one   0  0   0   0   0
    0   0   0   0   0   0   0  0   0 one   0 one one   0   0   0   0   0
    0 one   0   0 one   0 one   0   0   0 one   0  0   0 0 one   0   0   0 one
       0 one   0  0   0   0   0   0   0   0   0   0  0   0 one   0   0  0   0
      0   0   0   0 one   0  0   0   0 one   0 one   0 one   0 one   0  0   0
```

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 to 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.

Task 4.14.11

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.

```
E = noise(P, 0.02)
errorP = P + E
print(bits2str(mat2bits(errorP)))
```

```
I'm$pv}ingÂăto free y/ur0mind, Nto.`But K can onhY sh-w xou thÃě door. You'
    pM$the one Ãťhat hÃąsÂăto walo thr0ugh0yt.
```

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 4.14.12

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?

```python
C = G * P
unencoded_size = len(mat2bits(P))
print(unencoded_size)
```

```
896
```

```python
encoded_size = len(mat2bits(C))
print(encoded_size)
```

```
1568
```

Task 4.14.13

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 information is.

```python
CTILDE = noise(C, 0.02) + C
print(bits2str(mat2bits(R*CTILDE)))
```

```
I'm`tryinf to fr%e your minl, NeoÂő But IÃčaÃő only shgw iou the donr. You're
    the one that Has to walk throughÂăit*
```

Task 4.14.14

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.

```
def correct(A):
  return A+find_error_matrix(H*A)
```

Task 4.14.15

Apply your procedure `correct(A)` to `CTILDE` to get a matrix of codewords. Decode this matrix of codewords using the matrix R from `Task 4.14.3`, 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?

```
print(bits2str(mat2bits(R * correct(CTILDE))))
```

```
I'm`trying to fr5e your mind, Neo. But I can only shgw you the door. You're
    the one that has to walk through it.
```

The Hamming code did not succeed in fixing *all* of the characters. This is because the `correct(A)` procedure assumed at most one bit error for each nibble in $\tilde{\boldsymbol{c}}$. If there was more than a single bit error for a particular nibble, the error syndrome would map to the incorrect index.

Task 4.14.16

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

```
for error in range(0, 21, 1):
  noiseMat = noise(C, error*0.01)
  print("Probability of error: " + str(error) + "%")
  print("Decoded string without any corrections: " + bits2str(mat2bits(R
    * (C + noiseMat))))
  print("Decoded string with attempted corrections: " +
    bits2str(mat2bits(R * correct(C + noiseMat))))
  print("--------------------\n")
```

```
Probability of error: 0%
Decoded string without any corrections: 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.

Decoded string with attempted corrections: 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.
--------------------

Probability of error: 1%
Decoded string without any corrections: I'm trying to free your mind, Neo.
    ButI cqn only show you the door, YoU're the one that has to wa,k througj it
    .

Decoded string with attempted corrections: 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.
--------------------

Probability of error: 2%
Decoded string without any corrections: I#m tzyhng to$Ãęree you2 m+nd, Neo.(
    But i cao!Onlh wnow you the!dooz. YouÂğre the onu that has to wamk through
    iU.

Decoded string with attempted corrections: I'm trying to free your mind, Neo.
    But I can only sfow you the door. You're the one that has to walk through
    it.
--------------------

Probability of error: 3%
Decoded string without any corrections: I'm tryhNg to free yntr miÃőd, Neo.
    ÃĆut M$Can only sxW$yow the door. You'rÃğ!the one$uhat hap to valk through
    yt

Decoded string with attempted corrections: I'm tryiNg to free your mind, Neo.
    But I can only show you the door. You're the one that hap to walk through
    it.
--------------------

Probability of error: 4%
Decoded string without any corrections: I'm trymng DO freey/ur mind, Neo. RuV
    Iban mnmy!slow you txa`tor.You'rg the onE that hÃąs To walc ÃťhrougH ÃĽt>

Decoded string with attempted corrections: I'm 4rying Do free your mind, Neo.
    But I can only show you the door. You're the one that has To walk through
    it.
--------------------

Probability of error: 5%
Decoded string without any corrections: I'}"trymng to free {ur Ãŋind, %o.Bu|$I
    "san only s)gw }ou$4He Ãďoor.O[ot&Re)the g~ethat has tm walk thr*ugi it&
```

```
Decoded string with attempted corrections: I'm trying to free your miÃőd, Neo.
     Bu| I"can only show you 4he door. You're)the one that has to walk thr:ugh
    it.
--------------------

Probability of error: 6%
Decoded string without any corrections: Ie$lriiÃğg toÂÃf2eg"Ãźo5r0mind<0NEo.`
    Bu4I(can ov|y"sÃÍG7xou the dmnr. You'R the`Ãŕne that ha{(tm ual tjrough m$.

Decoded string with attempted corrections: Ie`tryigg toÂÃf"ee Ãźour mind, Neo.
     Bu< I can on<y sÃĹow you thg dmor. You'rÂĚ the`one that has to wal through
     l.
--------------------

Probability of error: 7%
Decoded string without any corrections: M&m$trxifg$tm fRee hotr`m)^d<N`on Bu4
    I sqn oney show"you t( $xÃőnr.(Yku/re the1ooa u(aT has Po alm t  souwh it.

Decoded string with attempted corrections: I$m%trying to free your`mi^d| N`on
    But I can oney show yo} th`%joor, You're vhe1one that haq po waLl tÂĹrouUh
    it.
--------------------

Probability of error: 8%
Decoded string without any corrections: I'm"Trymn&"tofree y}wRh0ijf,"Nen.2
    CuT$Y cyn Zol9shou y1!4ze$doÃŋrn!YoU%2g0thepknebt`Ad&iÃąs to"7xÂňi
    Phrnugi0i4n

Decoded string with attempted corrections: I'-#Trying to free your Nind, Neo.
    But I cÃśn Xnl) show yo5 the$door. You're thePone that.hÃąs to 7hÂÑi
    through it.
--------------------

Probability of error: 9%
Decoded string without any corrections: H#M vryÃĽmÃğ`Ã¿ÃŕÂÍnreaykur -ind,0\Ãěo
    . hud M Aan0mnl9 s`gw$yow 6`edor.`Yot're(thE Ãőe Ãťhit ,a{(4o wÂąlÃł"tLrg}
    gÃĹÂĆht.

Decoded string with attempted corrections: I'm tryÃĽmg zo freayour mind, NÃěo.
     Iu$ I canÂřonly show$you 4he eoor. You're the Ãőe Ãťhat mas $o wÂĄlÃń
    througÂĹÃĆit.
--------------------

Probability of error: 10%
Decoded string without any corrections: K'm vrzingÂćTofrd  yurmiÃőd, Fe, B}U
    I0cef oolyÃŚhÃŋ? yoUxhadonÃšn&_ou'rew`u0nbÃůpHet his(uÃŕwc~; tlroqg( iv.

Decoded string with attempted corrections: I'm trzing"tobre` your mind, Geo. B
    }T I cag only ÃČho yoUzhedoor.&Wou're$whUpo`Ãţ that has tÃŕ wal; throug8 it
    .
--------------------

Probability of error: 11%
```

```
Decoded string without any corrections: h/m nrymnGÃăÃn$^pmw qourMind, NdBÃŢt(
    H"bEf`wn.]ÃşXkwy?u p(e`dgor*!Yo5'pÃď$txi%onev-atzeC un0Ua`k 0lz/t7h"su.

Decoded string with attempted corrections: i'm zrÃźmnÃĞÃÃto$Vree youvÂÃmind,
    $Neo.ÂřBÂŢt I"jQnponlY sowy?u the`door.#You're$thi%onet=atÂřhaÃČ tl Wabk
    tlzou7h Ãůq.
--------------------

Probability of error: 12%
Decoded string without any corrections: IÂęo tx9inÂďÂąz fÃűuÃą {+4r$eiv$,Geo
 cut iÂąbqn"g.lqÃăsho#yÃŕÃţehgÃă$o?j.YÃŕw&($ 4hÃĕ"on% 4hatHas0uo w@Ãŋk thvoueh
      Iu'

Decoded string with attempted corrections: I'm t|yin`ÃĽzÃ£ fr5e ykur ei6d, Geo
    *But i jan"onlyÃřshow#youÂďE(gÃÃdg?r.PYown)e 4hÂĕ#one 4hat has to wdlK
    through Iw'
--------------------

Probability of error: 13%
Decoded string without any corrections: I#d6r)ngtk fseEqnÂůs }on@l ÃŁugÂő4But
    i(Bin"o.lÃź,rxÃ£!9lubÃťhm Âĕkcr.0Ãů'r`tÃű`a$o;%Ãăt`UtÂĺ`Ãăs Âť0 r@miÂŘp*
    rmÃĕg( y4Âő

Decoded string with attempted corrections: I'dÂř'rwingtk fqeE youw mgndl Keo.
    But I can"onhy.shÃ£w )lubtHm ÂĎoar. ]ou%r`pt`e$o;eÃÃthQt hds tW
    paiiÂŘtjrmÃĕGh Ãźt.
--------------------

Probability of error: 14%
Decoded string without any corrections: I M`Tra@ngÂąÃťM"nzee!yo}r min@*Neo~(
    Ut0I#af n6dy shÂŕÃů$Ãą0u vÃ e"Dogz>`QoÃţ'ÃĕxoÂąMrg 4x!t0jk} toÂĕqlo!t`
    z0UnÂňyÃĕ

Decoded string with attempted corrections: I ]ptrqAng to"oree your min`"N%o~(
    Up0I kan oÂűly shÂŔw$ÂľouvÃ e Togr~ Qou'ÂŠe ÂŤxk ÃŔ`e thatho} to%alk
    ThroEÂĞoÂőiÂď
--------------------

Probability of error: 15%
Decoded string without any corrections: Ãľ&er]IÃŕg$^0!vR5u!1naÂš2hÃľf@$*nek/"5
    tÃÃI ÃčqjhÂőÃň|yqx}g+x_5(pÃŇ( lGor   iu'pm"vde onmyÃţ(@tÂćÃÿ Âż Ã oi~hhk Ãţ
    `p0Ãťfh(cq.
Decoded string with attempted corrections: Lm.$ryYng%^`dÃŠ55 )luÂŠhÃĽfP,.~eo+
    Âć%tÃÃI cÂśkpnlly sÃÿ?u+y_u tÂĹ( moor.Âř    Ãľu're"wfe$onm{ÂďAtÂćÃÿ!{ ÃňoÃľ
    >lhk qhq_Âďgh)gq.
--------------------

Probability of error: 16%

Decoded string without any corrections: Q+M"Ãăs19~Ãğ!t.(fvee0y'ur iigaÃň@'o.
    JÃŤt`"caN(o*lS!ÃćjÂ£u0uoÃśÂďwndfoMz&owÂğPm)T Ãĕ ÃőJD`xit yikpoÂÃwÃąlK pH[
    oefhqÃř.
```

```
Decoded string with attempted corrections: I)m Ãďwy9~g tl(free 9our
    migaÃÑPFÂĕo.(JÂ t J can o.lw ÃęhÃ£w toÃŢ%wLd foOr.&\Ãŕu'ql)t1e Ãłjetxit
    ÃÿaspoÃÃwalK though9t.
--------------------

Probability of error: 17%
Decoded string without any corrections: gi!uriÃĽ"d"TÃİ f2gk0uÃńÃţÂżmÃčbl'uÃŋ*(
    bÃŋtdX can`/~,yÃşÃãm`+n}Ãĕl <kÃŔÃš& Y{u/2uAFLu&nu!e8dÃŸ)acwb5o"waÃĘk
    tÃ Rcugx aÃűC

Decoded string with attempted corrections: !gh!tryi`d tÃŹ f2ek wÃŕuÂŹMÃğ`l'Geo
    . bÃŋt I can`o~ly Ãčhmw )oyÃąd_ 4ÃŕÂŃr. Y?u/25PD\])fne u8dÃŹ+!as $o ÃůaÂĆk
    tÃ RbugX itÃČ
--------------------

Probability of error: 18%
Decoded string without any corrections: IÂĞ(Ãřw69ÂĽn tm eÃĕÃĕ#9gÂĕ*(m|zÃę-Âć\
    fo 0FqI(cAj4/~li sÂĺgÃą+Âŕu$ÃęjÂĔÂĕfo~r:p/uoRcm|ÃÿqoÃ£%(t9id!#aÃş(eÃő
    ÃğlkVhjowGÃÿHu

Decoded string with attempted corrections: IÂĞ(ÃřwryÂźng tl ÂĘEÃţe#9oÂĕ*(
    mmjÃťÂĽÂčNfo 0BuwI)ganp/nl) sÂŇgÃĂ)ÂŔu%ÃďhÂĔÂğfolr*P/uoBc-TÃĺSoÃ£% t=it%cas
    `$l wglkTh"muGÃÿ ItÃř
--------------------

Probability of error: 19%
Decoded string without any corrections: y&}60Cp)n'Ãűg gxegÂŰ>ÃĕuÃŠ =+>L(4
Ãŕ(BÃątYÃĆg*!mÃ¿hx$Ãş`owYO1Ãţ(gdÃńkrÃć(qnuÃĘÃşmdÂŁÃİ Ãğ>ut(eÂď pc3ÂšÂűJ0wq}
    zDhÃřÃŕ]sx(at.

Decoded string with attempted corrections: Ãź'm~$Aping Ãťo fxegÂđ>auÃŠ )>L,4
Ãŕ)BÃĕ4ÃŹ ÂČgj oÃži| sdowyKuÂřtheÂŘdokrÃă(qouÃŐrmÂĺtÂĹÃ oe thaÂď `a3ÂřkpwlÃ£{
     hÂšÂŕuWX(it.
--------------------

Probability of error: 20%
Decoded string without any corrections: ÃŸ3 Ãć2xÃź*ÃĚ!vo1fzekÃÿÃŕwÂŠÂĽ=}nÃğf ^
    eo,`RtÃ ÂćM@`Qn$wm97}j/ÃůygÃśÃ Ãď(M 5Ggr.2ÃĺMuÃťÃą sW`OnÃą phc~hÃńkÂčÃęÃĞ`
    eÂ£mkÂćÃďÃ£6Âŕ1cÃĺ ÃŻw.

Decoded string with attempted corrections: ÃŸ"ÂĐ ÃćrxÃź*ÂĞwo f{ÃĕkyovÂŠÃĽ=m.g
    &"Neo-`|tI@ An$_nl)&}jowÃźguÃŐd(e4eor.pÃĂLuÂĕ|`$Â§cU olÃĕ phazhÃń+#ÃďÂŔ`
    wÂ£lk#ÃĎÃżrÂŕ5ch ÃŹw.
--------------------
```