
Issued: February 23, 2006

Problem Set 4 Solutions

Due: March 3, 2006

Solution to Problem 1: Nostalgia from 6.050!

Solution to Problem 1, part a.

There are several possible codebooks, and two of them are enumerated in Table 4-1.

Codebook 1					Codebook 2				
0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	1	0	1	1
1	0	0	1	1	1	0	1	1	0
1	1	1	1	0	1	1	1	0	1

Table 4-1: Possible Codebook Implementations

Solution to Problem 1, part b.

There are 32 (2^5) possible bit strings. However, only four of those are allowed at any one time, one for each input.

Solution to Problem 1, part c.

For each code if one bit is in error, that produces up to five different codes. Since we have four legal codes, this gives us twenty codes that we detect as errors and can correct to their correct codes.

Solution to Problem 1, part d.

The number of uncorrectable codes is the total number of codes minus the legal codes minus the correctable codes, or 32 minus 4 minus 20, or eight.

Solution to Problem 2: Divide to encode ..

Solution to Problem 2, part a.

The binary representations of the two message should be divided by the generator:

$$\begin{array}{r}
 10011 \overline{) 1111110} \\
 \underline{1110100010} \\
 10011 \\
 \underline{011100} \\
 10011 \\
 \underline{011110} \\
 10011 \\
 \underline{011010} \\
 10011 \\
 \underline{011010} \\
 10011 \\
 \underline{10011} \\
 10011 \\
 \underline{00000} \\
 00000 \\
 \underline{00000} \\
 00000
 \end{array}
 \qquad
 \begin{array}{r}
 10011 \overline{) 110101} \\
 \underline{1100000000} \\
 10011 \\
 \underline{010110} \\
 10011 \\
 \underline{001010} \\
 00000 \\
 \underline{010100} \\
 10011 \\
 \underline{001110} \\
 00000 \\
 \underline{11100} \\
 10011 \\
 \underline{01111}
 \end{array}$$

We find that 1100000000 has error because of having a non-zero remainder while 1110100010 is valid and considered correct.

Solution to Problem 2, part b.

It is clear that $2^r M - R$ is a valid code. According to our assumption, it is identical to $2^r M + R$ in the binary case. $2^r M$ has r zeros in as its r least significant digits in the binary representation and R has fewer than or equal to r bits, so we can simply append R to the right end of the message M . In summary, we append r zeros (G has $r + 1$ bits) to the right end of the message M , divide it by G and replace the r zeros with the r bits of remainder to find T the encoded message.

Now applying this scheme on M_1 we have 1100000000 whose remainder of dividing by G is found to be 1111 in part (b). For M_2 , the resulting string is 1110100000. Again from part (b), we know 1110100010 is divisible by G so the remainder of M_2 is 0010. Thus, the encoded messages are:

$$T_1 = 1100001111$$

$$T_2 = 1110100010$$

Solution to Problem 2, part c.

We receive $T' = T + E$ and then, divide it by G . As $T = 0 \pmod{G}$, the remainder of T' divided by G is equal to the one of E . Consequently, all strings of error E that are not divisible by G can be detected in this scheme.

Solution to Problem 2, part d.

Solution to Problem 2, part e.

In order to show that all single-bit errors can be detected in this scheme, we have to show that no binary E that has only one non-zero digit can be divisible by a G with two 1s. Note that when we say a number is divisible by another number modulo-2, it means that we can find the dividend by modulo-2 summing some left-shifted versions of the divisor. For instance, in the example pointed out in the statement of the problem, we had $11110 = 110 \times 101$ which means $11110 = 10100 + 01010$ where the one-left-shifted and two-left-shifted versions of 101 add up to 11110.

On the other hand, we know that modulo-2 sum, subtraction and equivalently XOR preserve the total parity of the operands i.e. if the total number of ones in the operands is even (odd), then the number of ones in the result of the operator XOR is even (odd.) Thus, when G has even 1s, it does not divide any E with odd 1s, that is to say, all odd number of errors can be detected.

The fact that all valid codes can be found by adding shifted samples of the generator is so central in this scheme that this class of codes are called *Cyclic Codes*. Cyclic codes are a general class of codes that have interesting properties and are widely used in different applications. A more rigorous treatment of these codes can be done by polynomial representation of the strings and tells us more about the characteristics of these codes (If you are interested in learning more about these codes which historically made first links between coding and pure algebraic concepts, we refer you to the nice textbook *Algebraic Codes for Data Transmission*, by Richard E. Blahut. Simply go to the (<http://print.google.com>) and search for the name of this book and the term “cyclic codes!”)

Solution to Problem 3: A Different Hamming Code

Solution to Problem 3, part a.

```
>>%define the generative matrix g
>>g=[1 0 0 0 1 0 1; 0 1 0 0 1 1 0; 0 0 1 0 1 1 1; 0 0 0 1 0 1 1]
g =
     1     0     0     0     1     0     1
     0     1     0     0     1     1     0
     0     0     1     0     1     1     1
     0     0     0     1     0     1     1
>>%define the Test matrix H
>>H=[1 1 1 0 1 0 0; 0 1 1 1 0 1 0; 1 0 1 1 0 0 1];
H =
     1     1     1     0     1     0     0
     0     1     1     1     0     1     0
     1     0     1     1     0     0     1
>>%generate all the possible 4-bit messages (here we use some matlab
>>%features, dec2bin converts frpom decimal to binary)
>>a=(dec2bin(0:15));
>>messages=reshape(str2num(a(:)),16,4)

messages =
```

```

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1

```

Solution to Problem 3, part b.

```

>>%generate the codebook by multiplying the messages by the matrix g.
>>codebook=mod(messages*g,2)

```

```

codebook =
0 0 0 0 0 0 0
0 0 0 1 0 1 1
0 0 1 0 1 1 1
0 0 1 1 1 0 0
0 1 0 0 1 1 0
0 1 0 1 1 0 1
0 1 1 0 0 0 1
0 1 1 1 0 1 0
1 0 0 0 1 0 1
1 0 0 1 1 1 0
1 0 1 0 0 1 0
1 0 1 1 0 0 1
1 1 0 0 0 1 1
1 1 0 1 0 0 0
1 1 1 0 1 0 0
1 1 1 1 1 1 1

```

Solution to Problem 3, part c.

This code can detect and correct one error per codeword, or detect (only) two errors per codeword (the decoder can be designed to do either the former or the latter but not both). This is a property of all block codes with minimum Hamming distance of three.

Solution to Problem 3, part d.

Since no errors are introduced, the three CHECK variables should be all zero.

1.

```

>> INPUT1 = [0 1 0 0];
>> CODE1 = mod(INPUT1*G, 2)

```

```
CODE1 =
0 1 0 0 1 1 0

>> % No error introduced
>> CHECK1 = mod(CODE1*H', 2)
CHECK1 =
0 0 0

>> OUTPUT1 = CODE1(1:4)
OUTPUT1 =
0 1 0 0
```

```
2. >> INPUT2 = [1 1 0 0];
>> CODE2 = mod(INPUT2*G, 2)
CODE2 =
1 1 0 0 0 1 1

>> % No error introduced
>> CHECK2 = mod(CODE2*H', 2)
CHECK2 =
0 0 0

>> OUTPUT2 = CODE2(1:4)
OUTPUT2 =
1 1 0 0
```

```
3. >> INPUT3 = [1 0 0 1];
>> CODE3 = mod(INPUT3*G, 2)
CODE3 =
1 0 0 1 1 1 0

>> % No error introduced
>> CHECK3 = mod(CODE3*H', 2)
CHECK3 =
0 0 0

>> OUTPUT3 = CODE3(1:4)
OUTPUT3 =
1 0 0 1
```

Solution to Problem 3, part e.

Same input values but now errors are introduced.

```
1. >> % Error in position 3 in CODE1
>> CODE4 = CODE1
CODE4 =
0 1 0 0 1 1 0
>> CODE4(3) = ~CODE4(3)

CODE4 =
```

```
0 1 1 0 1 1 0

>> CHECK4 = mod(CODE4*H', 2)

CHECK4 =
1 1 1

>> >> CODE4(3) = ~CODE4(3)

CODE4 =
0 1 0 0 1 1 0

>> OUTPUT4 = CODE4(1:4)

OUTPUT4 =
0 1 0 0
```

```
2. >> % Error in position 4 in CODE2
>> CODE5 = CODE2
CODE5 =
1 1 0 0 0 1 1

>> CODE5(4) = ~CODE5(4)

CODE5 =
1 1 0 1 0 1 1

>> CHECK5 = mod(CODE5*H', 2)

CHECK5 =
0 1 1

>> % Repair damage by flipping bit
>> CODE5(4) = ~CODE5(4)

CODE5 =
1 1 0 0 0 1 1

>> OUTPUT5 = CODE5(1:4)

OUTPUT5 =
1 1 0 0
```

```
3. >> CODE3
>> % Error in position 5 (a parity bit)
CODE6 =
1 0 0 1 1 1 0

>> CODE6(5) = ~CODE6(5)
```

```
CODE6 =
1 0 0 1 0 1 0

>> CHECK6 = mod(CODE6*H', 2)

CHECK6 =
1 0 0
>> % Correction is optional since data bits are all OK >> CODE6(5) = ~CODE6(5)

CODE6 =
1 0 0 1 0 0 1

>> OUTPUT6 = CODE6(1:4)

OUTPUT6 =
1 0 0 1
```

Solution to Problem 3, part f.

This code cannot correct or even detect double errors. Instead, it interprets the symptoms as a single error and changes some bit that is probably OK. (If the only tool you have is a hammer, everything tends to look like a nail.)

```
>> INPUT7 = [0 1 0 0];
>> CODE7 = mod(INPUT7*G, 2)
CODE7 =
0 1 0 0 1 1 0

>> % Two errors, in positions 3 and 7
>> CODE7(3) = ~CODE7(3);
>> CODE7(7) = ~CODE7(7);
>> CODE7
CODE7 =
0 1 1 0 1 1 1

>> CHECK7 = mod(CODE7*H', 2)
CHECK7 =
1 1 0

>> % Incorrectly concludes there is an error in the second bit
>> CODE7(2) = ~CODE7(2)
CODE7 =
0 0 1 0 1 1 1

>> OUTPUT7 = CODE7(1:4)
OUTPUT7 =
0 0 1 0
```

Solution to Problem 3, part g.

We know from the way we have identified errors before that each syndrome will be identical to one of the columns of H , precisely the column where the error occurred in the original bit. This tells us that the table of syndromes will be just the same than H^T :

bit	Syndrome		
1	1	0	1
2	1	1	0
3	1	1	1
4	0	1	1
5	1	0	0
6	0	1	0
7	0	0	1

We note also that the last 3 columns of G are precisely the first four rows of this matrix, that is, the first four columns of H are the transpose of the last 3 columns of G . This observation should hint you on why do Hamming codes work.