Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Department of Mechanical Engineering

Problem Set 4 Solutions

Solution to Problem 1: My Dog Ate My Codebook

Solution to Problem 1, part a.

There are many possible codebooks, and two of them are enumerated in Table 4–3.

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

Table 4–3: 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: Correct Angle on the Rectangle

Solution to Problem 2, part a.

Arranging the code in a 4 x 1 or 1 x 4 rectangle results six parity bits, for a total of ten bits per message. This has a code rate of 2/5. For a 2 x 2 code, there are 5 parity bits, so you have a code rate of 4/9, slightly better. If efficiency were your only concern, you would use the 2 x 2 code.

Solution to Problem 2, part b.

The design does not correct for double errors.

Solution to Problem 2, part c.

The minimum hamming distance required for double error correction is 5. The minimum hamming distance of your code is 3.

Solution to Problem 3: A Different Hamming Code

Solution to Problem 3, part a.

```
>> G = [1 0 0 0 1 1 1; 0 1 0 0 0 1 1; 0 0 1 0 1 0 1; 0 0 0 1 1 1 0];
>> H = [ 1 0 1 1 1 0 0; 1 1 0 1 0 1 0; 1 1 1 0 0 0 1];

>> PATTERN = [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.

```
>> CODEBOOK = mod(PATTERN*G, 2)
 0   0   0   0   0   0   0
 0   0   0   1   1   1   0
 0   0   1   0   1   0   1
 0   0   1   1   0   1   1
 0   1   0   0   0   0   1
 0   1   0   1   1   1   1
 0   1   1   0   1   0   0
 0   1   1   1   0   1   0
 1   0   0   0   1   0   1
 1   0   0   1   0   1   1
 1   0   1   0   0   0   0
 1   0   1   1   1   1   0
 1   1   0   0   1   0   0
 1   1   0   1   0   1   0
 1   1   1   0   0   0   1
 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 0 1 1

>> % 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 1 0 0

>> % 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 0 0 1

>> % 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 0 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 =
1 1 0

>> % 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 0 1 1

>> % Two errors, in positions 3 and 7
>> CODE7(4) = ~CODE7(4);
>> CODE7(7) = ~CODE7(7);
>> CODE7
CODE7 =
0 1 0 1 0 1 1

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

>> % Incorrectly concludes there is an error in the first bit
>> CODE7(7) = ~CODE7(7)
CODE7 =
1 1 0 1 0 1 1

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