Department of Electrical and Computer Systems Engineering
Monash University

Information and Networks, ECE3141

Lab 3: Error Correcting Codes

Authors: Dr. Mike Biggar
Dr. Gayathri Kongara
(This update: 23 March 2022, based on an earlier version by Jean Armstrong & David McKechnie)

1. Introduction

This laboratory introduces the concept of error correcting codes, which can be used to deliver a correctly decoded message even if some bit errors occur in transmission. We will take a detailed look at the Hamming code, and upon completion you should have a basic understanding of how such a code works. Hamming codes can correct 1 bit error in a word, but you will also gain some experience with polynomial codes that can be made robust enough to correct multiple bit errors.

In the area of error detection and correction, the Hamming distance is the vector distance between two code words. In other words, it is the minimum number of bit errors required to turn one code word into another. For instance, consider the code words 0110010 and 0101010. These code words have a Hamming distance of two – if two bits (the third and fourth) were "flipped", then one of those code words would be transformed into the other. This is a very important measure, since if errors cause one valid code word to be turned into another valid code word, the receiver cannot detect or correct those errors; the receiver simply receives a code word, confirms that it is a valid one and just assumes all is OK.

A Hamming code[1] is an encoding scheme that can be used to either correct for single bit errors in transmission, or detect one or two bit errors without correcting them. That is to say, a Hamming code can be used for reliable communication if code words suffer 0 (of course!) or 1 bit error, or it can detect (without correction) up to 2 bit errors. The receiver would have to ask for a repeat (ARQ) if the code is to be used only to catch errors. If there are more than 2 bit errors, the receiver may not be able to detect the error at all.[2] Inclusion of the word "may" in the last sentence is important; depending on the error pattern, when 3 or more errors occur in a word we might still detect an error but, importantly, we might not. One valid code word could be converted by these errors into another valid code word.

While the Hamming codes are actually a class of codes, we concentrate in lectures on a particular instance, the Hamming (7,4) code, because the small number of bits is easily manageable. We will spend the first part of this lab focussed on this code. The notation (7,4) indicates that, for every 4-bit message, a 7-bit code word is generated and transmitted. Another way of looking at that is to say that an extra 3 parity bits are added to every 4 data bits before transmission.

[1] Though they are both named after Richard Hamming (https://en.wikipedia.org/wiki/Richard_Hamming), the Hamming distance and Hamming code are not the same thing. The Hamming distance is a measure that can apply to any error protecting code. The Hamming code is just one example of an error protecting code.
[2] However, a higher layer protocol might still recognise an error when it is interpreting the data and could ask for the information to be resent. It could do this by checking some feature over multiple code words. For example, a checksum could be sent periodically.

To investigate the response of error correcting codes when they encounter errors, we need some sort of model of the channel and the errors that are introduced. A Binary Symmetric Channel (BSC) is a very simple channel model that assumes most bits are received correctly, but a small number are "flipped" (a transmitted 0 is received as a 1, or vice versa). Both types of error (0 to 1 or 1 to 0) are assumed to occur with the same probability (hence the "symmetric"). The channel is therefore modelled with a single parameter: the probability of error, which is normally assumed to be relatively small[3]. Under certain circumstances, more complicated real-life channels can be reduced to a BSC, and they are easy to analyse.

A BSC is simplex (transmission only goes one way[4]) and does not favour transmission of ones or zeroes. A BSC determines simply whether an error occurs, but the probability of that error occurring does not depend on whether the bit was transmitted originally as a one or a zero.

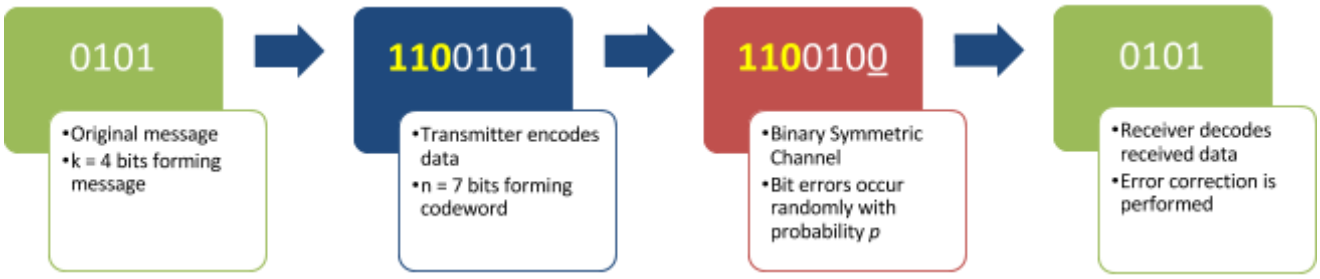


Figure 1. Hamming Coding system

Figure 1 shows the simple binary communication system you will simulate in the first part of this laboratory. The data to be transmitted is divided into blocks of length k bits[5]. n-k bits are added to each block, to form a code word of length n which is transmitted through the channel. The channel is simulated as a BSC in which errors occur randomly with probability p, and an error results in the erroneous bit being flipped.

In this laboratory class, you will carry out the following as you investigate the communication of data across an error-prone channel using error correcting codes:

- Become familiar with the concept of adding error checking bits to a message
- Familiarise yourself with the operation of encoding and decoding using the Hamming (7,4) code
- Use Monte-Carlo simulation techniques for communications systems.
- Understand the implications for error correction from the probability of error and length of a message
- Explore the error correcting capabilities of Hamming and polynomial codes
- Build on your knowledge of MATLAB, particularly in the use of some tools that are available to simulate communication systems.

2. Pre-lab

Prior to the laboratory class, you must read through this entire laboratory description and complete the pre-lab online quiz. The experiment involves simulating the transmission of data protected with error correcting codes, through binary symmetric channels with different

[3] We hope that the probability is small! Mathematically, it can be large as well (up to 50%), but in practice such a channel would be almost useless. Most usable channels have p(bit error)< 10^{-2}.
[4] Of course, we could have another channel (possibly a BSC) going the other way for two-way communication, but the channel model is not dependent on a two-way capability; each direction is modelled separately.
[5] These blocks are called "information bits" or "data bits" in lectures, and "message bits" in the MATLAB documentation.

probabilities of error. So that you can compare your simulation results with predicted values, you need to calculate the error probabilities for the experimental section before you commence the experimental part of the lab (it is part of the pre-lab quiz). When you answer the pre-lab quiz, note the relevant results in the table in Section 5. If you don't do this, you will waste a lot of time and may not complete the laboratory.

Familiarising yourself with the Matlab calls beforehand will also make the lab go smoother and give you a better chance of finishing in the time allowed.

3. Hamming coding of individual messages

a) Generate a random four bit word:

```
>> msg = randi([0 1],1,4)
```

This statement will generate a matrix of one row and four columns, filled with random zeroes and ones. Repeat the process a number of times to verify that different random words are created each time.

> What is the last 4-bit message you created (that is, the one you are leaving in the array as you go on to the next step of encoding it)?
>
> 0100

b) Encode the message:

Generate the code word using a Hamming (7,4) scheme:

```
>> codeword = encode(msg, 7, 4, 'hamming')
```

> Write down the code word, identify the message and parity bits, and check that the parity bits are what you would expect[6]. (Hint: While in lectures we assumed the parity bits were at the end of the transmitted word, sometimes they are put at the start and sometimes they are embedded in between the message bits. You might need to run this command a few times on different message bits to be sure which are the parity bits.)
>
> The code word obtained would be 0110100.
> The message bits would be 0100 ($p_4\ p_5\ p_6\ p_7$) and the parity bits would be 011 ($p_1\ p_2\ p_3$).
>
> Parity Bit Check
> $p_1 = p_4 \oplus p_6 \oplus p_7 = 0$ xor 0 xor $0 = 0$
> $p_2 = p_4 \oplus p_5 \oplus p_6 = 0$ xor 1 xor $0 = 1$
> $p_3 = p_5 \oplus p_6 \oplus p_7 = 1$ xor 0 xor $0 = 1$
>
> Yes, the parity bits are what I would expect.

c) Decode the message:

Decode the code word using the same Hamming(7,4) scheme:

```
>> received = decode(codeword, 7, 4, 'hamming')
```

(If you don't get the original message back, then something is wrong! Go back and check.)

[6] See lectures or a textbook to see how the parity bits are calculated.

4. Decoding errored code words

Now we can explore the effect of changing bits in the encoded word to see if it is still decoded properly. To use consistent notation throughout the experiment, create a new variable `rxinput` which we can use to represent the signal at the receiver input[7], after passing through a BSC.

a) Create rxinput:

```
>> rxinput = codeword
```

Change one of the bits of `rxinput`[8]. Then decode the word and note both the corrupted word and the decoded message. Do it a few times and see if the result is the same when the bit you change is a message bit or a parity bit. Summarise the results. Are they what you expect?

Summary of Result

Codeword = [0110100], expected result = [0100]

n	Corrupted Codeword	Position of bit that is corrupted	Result
1	0100100	3	0100
2	0110000	5	0100

Yes, the result is what I would expect. The decoded results will be the same. If one of the message bits or parity bits is changed, it will not affect the result. This is because the Hamming Code of 3 can detect and correct single bit error. The Hamming code detects error by checking the parity bits and seeing whether the parity bits matched the parity bits that we are supposed to get from the information bits.

What happens if you change more than one bit in `rxinput`? If you're not sure, go back and read about the Hamming Distance in the introduction.

Codeword = [1001011] , expected result = [1011]

n	Corrupted Codeword	Position of bit that is corrupted	Result
1	1111011	2 and 3	1111
2	1001101	5 and 6	1101

The decoded message will be different when more than one bit is changed. Since the Hamming Code only has a minimum distance of 3, it can only detect up to two errors and correct a single bit error. It does not matter when more than 1 information bits or parity bits is changed, the decoded message will not be the same as the original message.

[7] In telecommunications, the abbreviation rx is often used to represent "receiver", and tx to represent "transmitter".

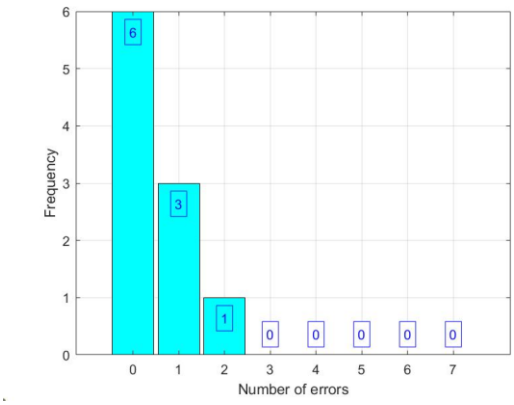
[8] Not sure how to flip bits? You could use the variable editor; select "Open Variable" from the Home tab in Matlab, and select the required array from the available variables. Another approach would be to use the "not" function to flip a bit: e.g. *rxinput(3) = ~rxinput(3)*

5. A Monte Carlo simulation using a Binary Symmetric Channel

MATLAB includes an instruction to simulate a BSC, because it is an important model in telecommunication systems. Try simulating the effect of transmitting a code word through a BSC with a probability of 0.1 a few times using:

```
>> rxinput = bsc(codeword, 0.1)
```

What do you observe?



There are errors in the codeword which occurred a few times. Based on the code that I run, I have observed that there are 6 zero error transmission, 3 one-bit error transmission and 1 two-bit error transmission. This resulted in the codeword not having the exact same original message.

Do you always get errors introduced when you use this function? Why?

No, I will not always get errors introduced when I used this function. The probability of error occurring in the Binary Symmetric Channel (BSC) is 0.1 as given in the provided code above. From here, we could use binomial distribution to get the probability that n bit error would occur in the BSC.

Using the formula $P\ (n\ error) = (^{k}_{n}C) * p^{n} * (1-p)^{k-n}$ that have been derived from the binomial distribution, we will obtain P (0 error) to be 0.48, P (1 error) to be 0.37 and P (2 errors) to be 0.12. As the probability of zero error occurring is 0.48, we cannot say that we will always get errors introduced when we use this function.

The power of MATLAB in handling matrix data makes it simple to code and decode multiple words rather than just the single word we have handled so far.

a) Using `numwords` to define the number of words we wish to process, enter:

```
>> numwords = 100
```

```
>> msg = randi([0 1], numwords, 4)
```

This will create a matrix of 100 random four bit binary words. Encode the words using the Hamming (7,4) encoder:

```
>> codeword = encode(msg, 7, 4, 'hamming')
```

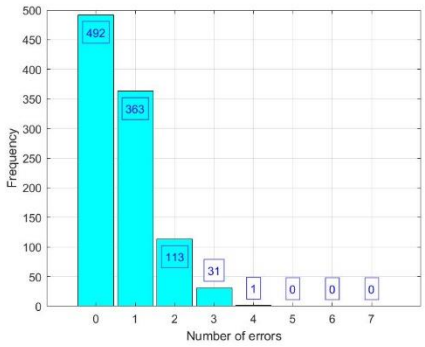
As you are probably aware by now, when you want to use a sequence of more than a few MATLAB instructions, it is easier to create an m-file. The Appendix contains the code for an m-file that you can download from Moodle, which can be used to encode and decode using Hamming codes and introduce errors using a BSC. It also calculates the errors which were unable to be corrected, using the *mod* function to perform modulo-2 arithmetic when comparing the received message to the transmitted one[9].

This is a so-called 'Monte Carlo' simulation, as random data and errors are generated according to some rule, and the performance measured statistically. The title 'Monte-Carlo' refers to the casinos at Monte-Carlo: the experiments are like throwing dice or spinning a roulette wheel. The idea is that we rely on statistical variation to exercise a system under a varied range of conditions, without necessarily trying every possible input bit sequence or error pattern.

b) Read the m-file in the Appendix and try to understand how the code works.

(There is nothing very difficult about this script. You could have written it yourself if you had a bit more time. However, by using this one, you can take advantage of the fact that someone else has gone to some trouble to consolidate the results and present them in a neat way.) Try running the programme. Add code that will label the x and y axis of the plot that is generated (and make any other changes to the plot that you would like!).

From the histogram data presented, explain how we would obtain the number of uncorrected errors using Hamming coding and decoding.



As we can see in the graph above, the number of uncorrected errors would be the number of errors starting from 2 onwards. The total number of errors with values 1 and below would have been corrected so it does not count as the number of uncorrected errors.

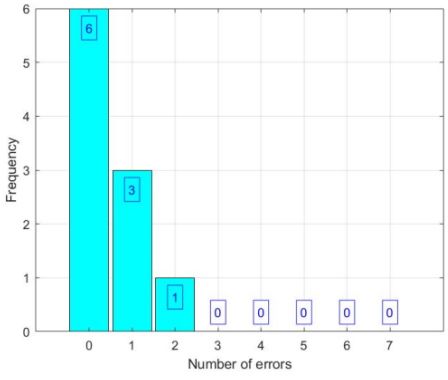
[9] Of course, in a real transmission system we do not have the original data available at the receiver to compare; we just have what we received. In our simulation, we can keep the original data, process it with our encoder, simulated BSC and decoder, and then compare the bits at the end with what was used at the start. Such comparisons allow us to evaluate the performance of error correcting codes such as this, because we know exactly what the received bit sequence should be.

c) Adapt the programme so that it runs a simulation of a (7,4) Hamming code for 10 words, 100 words, and 1,000 words for a probability of bit error p = 0.1. Export the generated plots (noting the filenames) and fill in the simulation values for Table 1 below. **The theoretical values should have been completed as preliminary work.** Note that the theoretical values do not have to be integer numbers. Do you understand why? If you're not sure how to complete the "Number of incorrectly decoded words" column, you should review your answer to part b) above.

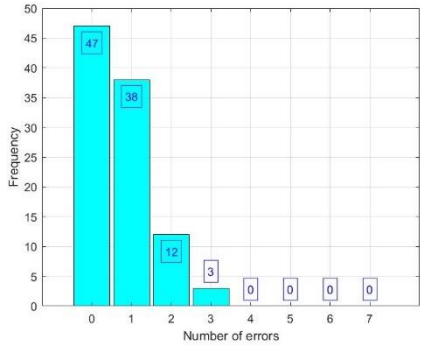
Number of words	Result type	Number of incorrectly decoded words	Number of received words (prior to decoding) with the following number of errors.							
			0	1	2	3	4	5	6	7
10	Theoretical	1.497	4.783	3.720	1.24	0.23	0.03	1.7e-3	6.3e-5	1e-6
	Simulation	1	6	3	1	0	0	0	0	0
100	Theoretical	14.97	47.83	37.20	12.40	2.30	0.26	0.017	6.3e-4	1e-5
	Simulation	15	47	38	12	3	0	0	0	0
1000	Theoretical	149.69	478	372	124	23	2.55	0.17	0.0063	0.0001
	Simulation	156	467	377	132	23	1	0	0	0

Table 1. Hamming Coding ((7,4) codes, p=0.1) Monte Carlo simulation results: comparison with theory

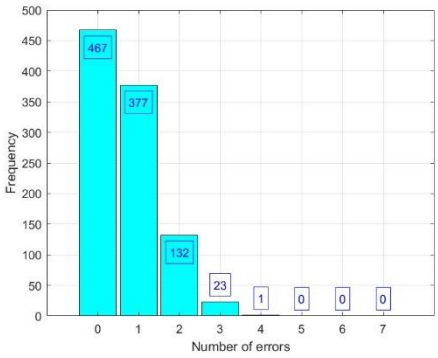
Comment on your findings. Do the simulation results agree with the theoretical ones? Try running the simulations again. Do you get the same results?



when n = 10



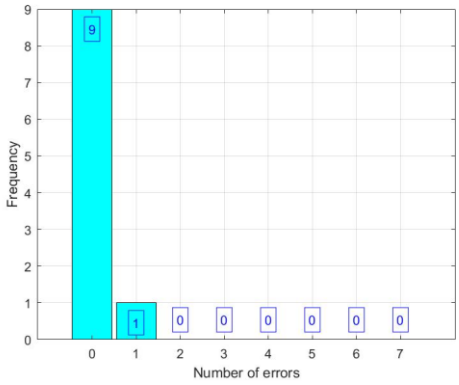
when n = 100



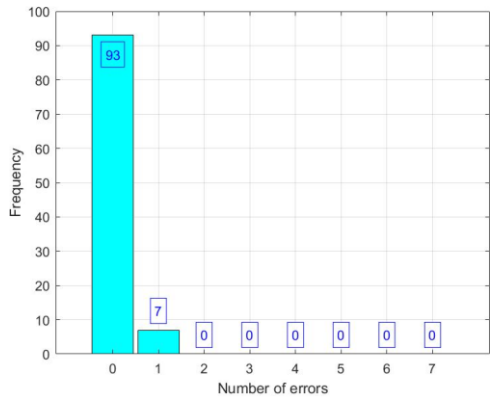
when n = 1000

The simulation results roughly agree with the theoretical ones. If I run the simulation again, I will get a different result but still it does not differ too much away from the theoretical results. There is a difference in result because the Binary Symmetric Channel (BSC) will introduce randomized bit errors into the code word.

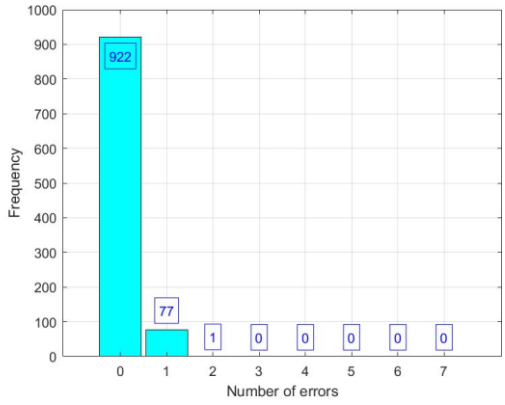
Try running the simulation with a probability of error $p = 0.01$. Comment on how common uncorrectable code words are in this case.



when n = 10



when n = 100



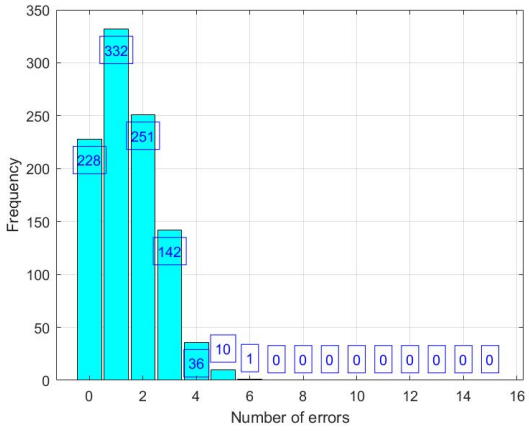
when n = 1000

The number of received codewords with no errors will increase. Since the probability of error, p has decreased, there will be less codewords containing errors.

6. Longer Hamming Codes

Longer Hamming codes, such as the (15, 11) and (31, 26) Hamming codes, are also possible. Do you think they would work well with $p = 0.1$? By doing a few trial-and-error simulations with the (15,11) code, determine the maximum value of p that could be tolerated if we want to keep the chance of a word being incorrectly decoded to less than one in a thousand. How will the accuracy of your answer depend on the number of test code words? Can you find an analytical expression for p that would satisfy this requirement?

(15, 11) Hamming code:



I do not think that the (15,11) code will work well with p = 0.1. Based on the visual observation made for my simulation, the total number of cases with errors is larger than the total number of cases with zero errors. The higher the number of test code words, the more accurate the answer will be. This is due to the law of large numbers which states that as the number of identically distributed, randomly generated variables increases, their sample mean (average) approaches their theoretical mean. We can also explain this observation with the Central Limit Theorem (CLT). When the sample size of the transmitted bit increases, the distribution of bit error would approach a normal distribution. Thus, we can conclude that the longer Hamming code will not work well when p = 0.1.

The analytical expression for p that would satisfy this requirement would be

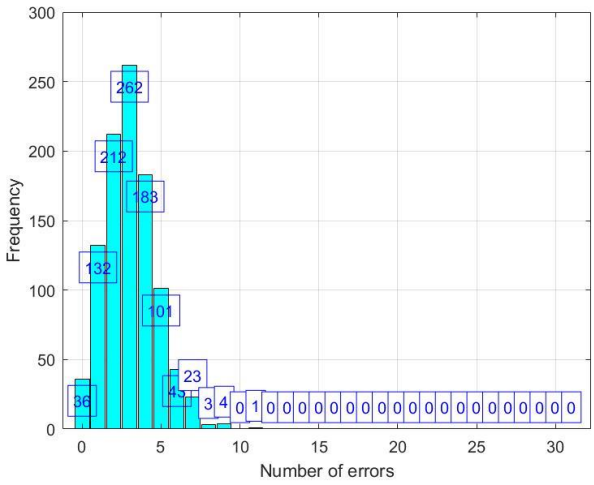
P = 1 – P (0-bit error) – P (1-bit error)

Since k = 15 bits, we need to substitute this into our equation above

1 – (15C0) *p^0*(1-p) ^15 – (15C1) *p^1*(1-p) ^14 <= 1/1000

We would get p to be approximately 0.003

(31, 26) Hamming code: Comment on the suitability of this even longer Hamming code with a high probability of bit error such as p=0.1. (You can just enter a comment if you are confident that you know what's going on. If you're not sure, then run the simulations and calculate the theoretical value again as you did above.)



I do not think that the (31,26) code will work well with p = 0.1. When the length of the Hamming code is increased, there would be a higher probability of the code being decoded incorrectly. Based on the visual observation made for my simulation, the total number of cases with errors is larger than the total number of cases with zero errors. The higher the number of test code words, the more accurate the answer will be.

The analytical expression for p that would satisfy this requirement would be

P = 1 – (31C0) *p^0*(1-p) ^31 – (31C1) *p^1*(1-p) ^30 <= 1/1000

7. Polynomial Error Correcting codes

In lectures and in the above exercise, we have looked in some detail at Hamming Codes as one of the simplest examples of an error correcting code. Many other types of error detecting and correcting codes exist. A common example is the set of "polynomial" (sometimes also called "cyclic") codes. The name "polynomial" comes from the use of binary polynomial arithmetic in the generation of the code words. The binary code word bits are used as coefficients of a polynomial, and the properties and patterns that result from the polynomial arithmetic give us the ability to detect or correct bit errors[10]; similar in principle to Hamming Codes, but more powerful. Polynomial (or "Cyclic Redundancy Check", CRC) codes are very common. They are used, for instance, to detect errors in Ethernet data frames. We don't have time to look into detail about how they work (but you're welcome to research it if you like). However, they follow (as any error correcting code must) the same principle of only allowing certain valid code word patterns to be transmitted and any invalid code word received can be analysed to determine the most likely one that was sent (again, the distance from valid code words determines the corrected word).

Even though we haven't studied them in detail, we can still use polynomial codes and investigate their performance. The *Matlab* function *encode* can also be used to encode using polynomial codes, but we instruct it to encode using "*cyclic*" instead of "*hamming*". The following *Matlab* code comes from *Matlab* documentation, and the numbers n, k are the same as for the Hamming (15,11) code. The "parity check matrix" and "syndrome decoding table" concern the process of error correction, but we do not have to worry about these details for our purposes[11]. (And, in case you're worried, apart from this lab there will be no assessment that requires specific knowledge of polynomial codes.)

```
n = 15;                % Code length
k = 11;                 % Message length
data = randi([0 1],k,1);   % Create a binary message having length k
gpol = cyclpoly(n,k);   % Create a generator polynomial for a cyclic code
parmat = cyclgen(n,gpol); % Create a parity-check matrix by using the
generator polynomial
trt = syndtable(parmat); % Create a syndrome decoding table by using the
parity-check matrix
encData = encode(data,n,k,'cyclic/binary',gpol);      % Encode the data
%    Introduce errors in the 4th and 7th bits of the encoded sequence.
encData(4) = ~encData(4);
encData(7) = ~encData(7);
decData = decode(encData,n,k,'cyclic/binary',gpol,trt); % Decode the
corrupted sequence.
numerr = biterr(data,decData)    % Count bit errors
```

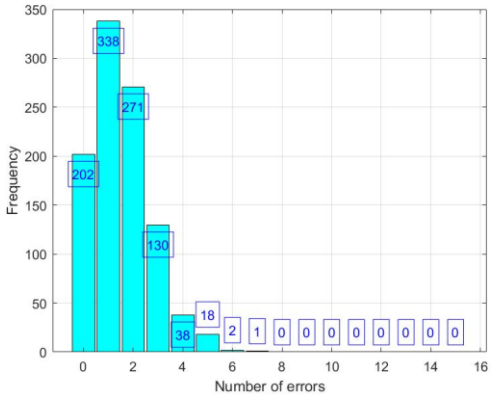
Using some variation on the above script, you can compare the performance of a polynomial code with that of a Hamming code, for the same values of n and k. If you have the time, you could substitute the relevant parts of the above script into the "SimHamming" script so that it will plot results for a large number of code words. However, this is not essential to be able to

[10] This is (obviously) a very high-level description. We'd need to take much more time to explain properly what this polynomial arithmetic is all about. The polynomial arithmetic allows us to design codes that will always detect certain error patterns. Have a look at either of the prescribed textbooks if you're interested to know more, but DON'T PANIC; we do not expect you to be able to do any binary polynomial arithmetic! You don't even really need to understand what it is to complete this part of the lab.

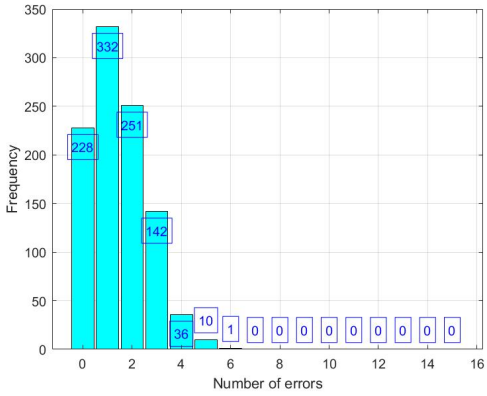
[11] The "syndrome table" is a look-up table that tells us what valid code word to use according to the received bit pattern). See the Tanenbaum and Wetherall textbook ("Computer Networks") for further explanation or do your own research if interested.

answer the following questions; you can get by with just running the script a few times and changing the number and location of the errored bits in the code word.

If you use n=15 and k=11 (as shown above), does the polynomial code perform any better than the Hamming Code? i.e. can it correct more than 1 bit error in a word? Why do you think this is the case?



CRC



Hamming Code

As we can see in the graph above, the polynomial code performs almost the same as the Hamming Code. Both of the code can only detect up to 2 errors and correct up to one error as the Hamming distance is 3. The performance could be improved if the number of parity bits are increased.

Keeping n fixed at 15, by how much do you have to reduce k before you can reliably correct more than 1 error?

We would have to reduce k by 9 or less before we can reliably correct more than 1 error.

Try a few different combinations of values of n, k and the number of errors in the word.

Suppose we want to keep using n=15 bit words but, after error correction, we want to achieve a maximum of 1 in 10^{11} words containing uncorrected errors, and we have a bit error probability p=0.001. We want to find out how many of those 15 bits in each word can actually carry data (that is, what is k?) to achieve this low final word error rate. Finding k will give us the efficiency of this code (where efficiency=k/n).

Hint: You do not need to simulate error coding of 100s of billions of code words. That would take a long time, and your PC might object! You can solve this part-analytically and part-experimentally with the following steps:

- Remembering part 5b where you added histogram values to find the total number of uncorrected errors, and the binomial expression for probability of a given number of errors in a word, find an expression for the probability of more than a certain number of errors, say “E”, occurring in an n-bit word.
- Set that expression equal to the target uncorrected word error rate given above.

- Since you know p(bit error) and the final probability of uncorrected code words ($< 10^{-11}$), you can solve for E (at least approximately – think about what simplifications you can make to make it solvable).
- Then use the Matlab tool (trial and error) to find the value of k which reliably corrects up to that number of errors ("E") using the polynomial coder.

What is the efficiency of the resultant code words?

$({}_{n}^{k}C) * p^{n} * (1-p)^{k-n}$, where p = 0.001 and k = 15

$({}_{E}^{15}C) * 0.001^{E} * (1-0.001)^{15-E}$ <= 10^-11

E would be approximately more than or equal to 4.

We could also use MATLAB code to find the value of k which is based on the given formula

$$P(i \le k) = \sum_{i=0}^{k} {}_{i}^{n}Cp^{i}(1-p)^{n-i} > (1-error)$$

Matlab Code:

```
p=0.001;
n=15;
error=1e-11;
k=0;
Sum=0;
for i=0:n
 Sum=nchoosek(n,i)*p^i*(1-p)^(n-i)+Sum;
 if(Sum>(1-error))
 k=i
 break
 end
end
```

We would get k to be 4.

The efficiency of the resultant code word would be 4/15 * 100 which equates to about 26.67%.

8. Conclusion

This laboratory has given you some hands-on experience using error correcting codes, a Binary Symmetric Channel, the use of Monte-Carlo simulation and relating practical performance to theoretical calculations of error probabilities. You have also seen that more sophisticated error correcting codes can correct more than one bit error in a word, but that this could come at a cost in terms of the efficiency (comparing number of information bits with total transmitted bits).

Before finishing, could each student please click on the "Feedback" icon for this laboratory on Moodle, and record some brief feedback on this laboratory exercise. (Remember – it's completely painless, and anonymous!)

Finally, please submit this completed report via Moodle by the stated deadline. In so doing, please be aware of the following:

- Even if you have had a mark assigned to you during the lab session, this mark will not be registered unless you have also submitted the report.
- Your mark may also not be accepted or may be modified if your report is incomplete or is identical to that of another student.
- By uploading the report, you are agreeing with the following student statement on collusion and plagiarism, and must be aware of the possible consequences.

Student statement:

I have read the University's statement on cheating and plagiarism, as described in the *Student Resource Guide*. This work is original and has not previously been submitted as part of another unit/subject/course. I have taken proper care safeguarding this work and made all reasonable effort to make sure it could not be copied. I understand the consequences for engaging in plagiarism as described in *Statue 4.1 Part III – Academic Misconduct*. **I certify that I have not plagiarised the work of others or engaged in collusion when preparing this submission.**

– END –

Appendix

This is the Matlab script of the file simHamming.m

```
%
% simHamming.m
% A program to simulate the effect of a binary symmetric channel on
% data encoded using a Hamming code
%

clear variables % Good practice to clear the work space at the beginning

% Define the values of parameters to be used in this run. It is best to
% define them all at the beginning of the script as it makes them easier
% to change.
n=7; %Length of the words after coding
k=4; %Number of message bits
numwords=1000; %The number of words to be encoded and decoded
p=0.1 %Probability of bit error in the BSC

msg=randi([0,1],numwords,k); %Generate the random data
codeword=encode(msg,n,k,'hamming'); %Encode the data
rxinput=bsc(codeword,p); %Simulate the BSC
rxdecode=decode(rxinput,n,k,'hamming'); %Decode the received signal
numerr = biterr(msg,rxdecode); % Count number of errored bits after decode
Percent_error_after_decode = 100*numerr/(numwords*n)

% Compare the transmitted data before coding and the received data
% after decoding to see where errors have occurred
erroredbits=mod(msg+rxdecode,2);
errsperword=sum(erroredbits,2); %Add up the errors in each word

% The array errsperword contains the number of errors in each word sent.
% In this case we are interested in whether the word is decoded correctly,
% not how many errors it has, so we want to count up the number of words
% with one or more errors:
numwitherr=length(find(errsperword));

% Look at the statistics of errors introduced by the BSC
erroredbits_before_decode=mod(codeword+rxinput,2); %Find bit errors
errsperword_before_decode=sum(erroredbits_before_decode,2);

figure; %Create a new figure
K=histc(errsperword_before_decode,0:1:n); %Calculate the histogram
bar(0:1:n,K,0.9,'c'); %Plot the histogram
xlabel('Number of errors');
ylabel('Frequency');
grid on;

m=max(K); %Find the max value in K (used to position labels)
for count=1:(n+1) %Add data labels to the plot
    if K(count)>m*0.1 %This value is bigger than 10% of the max
        %Add blue text slightly inside the bar
        text(count-1,K(count)-m*0.06,sprintf('%d',K(count)), ...
            'HorizontalAlignment','center', ...
            'Color','b', ...
            'EdgeColor', 'b');

    else %This value is less than 10% of the max
        %Add blue text slightly outside the bar
        text(count-1,K(count)+m*0.06,sprintf('%d',K(count)), ...
            'HorizontalAlignment','center', ...
            'Color', 'b', ...
            'EdgeColor', 'b', ...
            'BackgroundColor', 'w');
    end
end
```