130011035 Mark-Jan Nederhof Error Correction 27/11/2015

1) Introduction:

This Practical asked that we work with Hamming codes, a technique used in error correction when passing bits through a noisy channel. This would then be built on by using the method of 'interleaving' to help reduce the number of errors when burst errors occur.

2) READ ME:

Using the command line, enter into the SRC folder, which is within the Error Correction folder. Once inside the SRC folder, run

```
' javac HammingCode.java InterLeaver.java BurstErrors.java Main.java'
'cd ..'
'java Codes/Main'
```

3) Implementation:

I initially created the HammingCode class, as this would give me a foundation to work from. It took me a while to understand how the parity bits worked properly, but eventually I managed to create an implementation which would account for them properly. I initially added an array of ints to store the code which would be received, which was set from a value I could enter in the Main method. I then created the basic hamming code using the createHammingCode method. This involved setting the size of the code by working out how many parity bits were needed along with how many data bits were in place and adding them together. From here I iterated through the array and if I found the location of each of the parity bits within the array, I stored them in an array list, while if I found a data bit slot, I would store the value from the code which was entered in the main method.

Once the hamming code was created, I would calculate the parity bits. This would be done in the calculateTheParity method, where I would set how many spaces along the array the parity calculator would move, how many bits from the array it would check, where I was in the array and finally what the total value of the sum was. Then I would iterate through the array, stopping at all of the correct locations in order to calculate the value of the parity bit. Once I had this working I started working on the BurstErrors Class.

In this class I created variables which would store the probability , p, the probability of good to bad, pgb, and the probability of bad to good, pbg. In the initial tests, I created a 'burst code' which was just a separate returnable code which I could compare to the original code word. The code words had the chance to be altered when in the sending bit method, which was my noisy channel. It would initially check what state the code was in, and if it's in a 'bad' state then it would check if the randomly generated float was less than or equal to the value of p. If it was then it would flip the bit and also calculate whether or not it should return to the 'good' state. However, if the code was not in a 'bad' state, then it would calculate whether it should change into this state.

Due to having some of the bits flipped, I implemented the parity checks. This involved checking if anything had actually been flipped in the checkIfCodeCorrect method. If nothing had been changed then there was no need to continue to change the code as nothing had been changed. If something was found to be different, then the error counter would increase. If there was only one error, then the fixTheCode method would attempt to find the flipped bit. However, due to the nature of Hamming Codes, if more than one bit was flipped then no changes would be attemped.

In the fixTheCode method, I created an array list which would store the values of the parity bits.

This was populated by iterating through the array list which contained the locations of the parity bits and retrieving their values from the array. These were then added together to find the location within the array that the error had occurred. This bit was then flipped. To get the correct values in the parity bits, I ran an update the parity method.

This method would calculate the parity bits' values within the incorrect code using a similar strategy to the calculate the parity method. The value of this was then returned to the user once the correct parity values had been recalculated using the calculate the parity double method. This method was the same as the calculate the parity method, except that it would return the new code.

Once these features had been implemented, I made the InterLeaver class. This also meant that I had to alter the Main class from simply reading in the values, putting them through the channel and fixing them. The values became set to run the code ten times to get a good average of values.

In the Inter Leaver class, I created two 2D arrays, which I used to store the initial state of the hamming codes. These were entered using a method called set The Data, which would enter in the array of hamming code to the 2D array. The hamming codes were generated randomly in the Main method. Then there was a separate method which was used to update the values stored in the data variable, this was to avoid also updating the values of dataCheck. There was also a checkErrors method, this compared the values in data to those in data check, and anywhere a number that did not match was found, an error was added on to the total.

Then in the main class, there was a random number generator which created the four data bits which would be used in the hamming codes. I then iterated down each of the columns of the data 2D array sending each bit to the sending bit method in the burst errors method. This would give the potential for each of the bits to change. Then once this had been completed, I read the data out of the 2D array, and used the check if code correct method to fix errors which had occurred to the individual hamming codes. I then iterated through this process 100 times and calculated the total number of errors which occurred, the total number of errors left after being sent for fixing, and the total number of errors corrected.

One error which I discovered early on was that when I entered a number over 10 bits into the main method, an error would be thrown that would add many odd numbers, like 9's and 2's. I could find no reason for it to do this other than a feature in the parseInt method which I had been using at the time. For this reason I only implemented the Hamming(7,4) codes but had ways to implement the other possibilities had they been available.

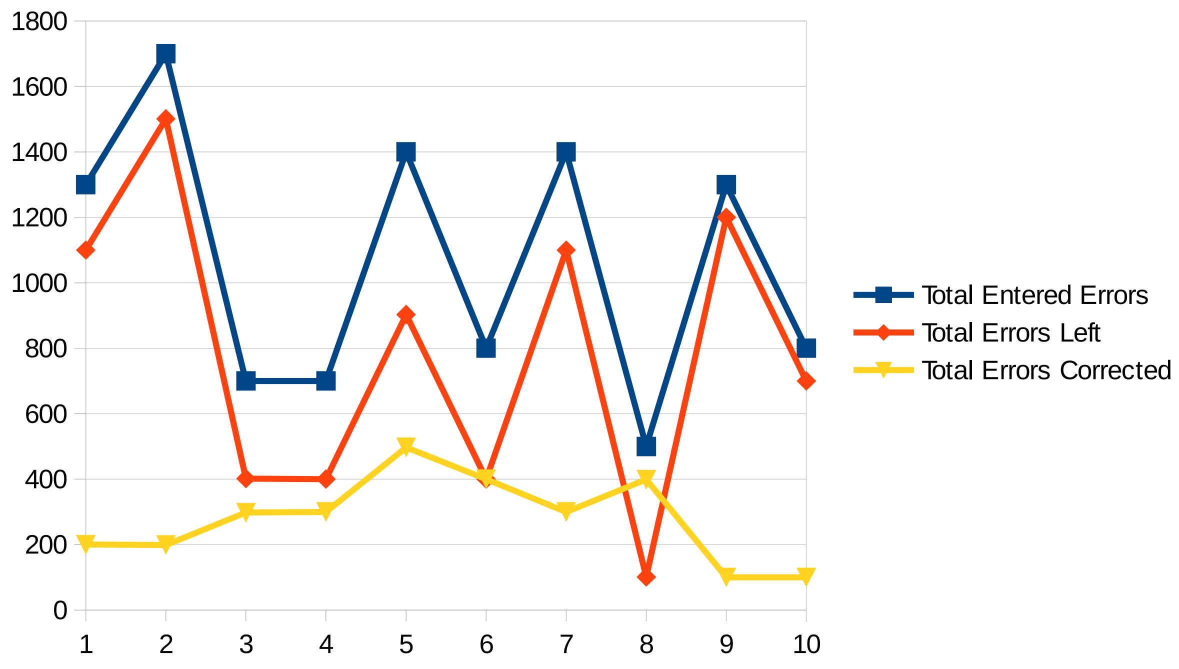
4) Experimentation:

I performed a number of experiments on the data, which will all be included in the 'experiments' file.

1) RESULTS OF P = 0.3, PBG = 0.3 and PGB = 0.1

When I entered in the data in this instance I found that the number of total errors averaged out at 1060, the total errors left after fixing averaged out at 780.7 and the total number of errors which were corrected was 279.3. This shows that in most of the cases there would be more errors left at the end that had been corrected. This would be because the probability was set to 0.3 so once this could explain why the average number of errors *per test* was 10.6 which is relatively low. Also the PBG which could change from the bad state to the good was higher than the PGB which could also explain this, as it would be easier getting back to the good state compared to getting into the bad.

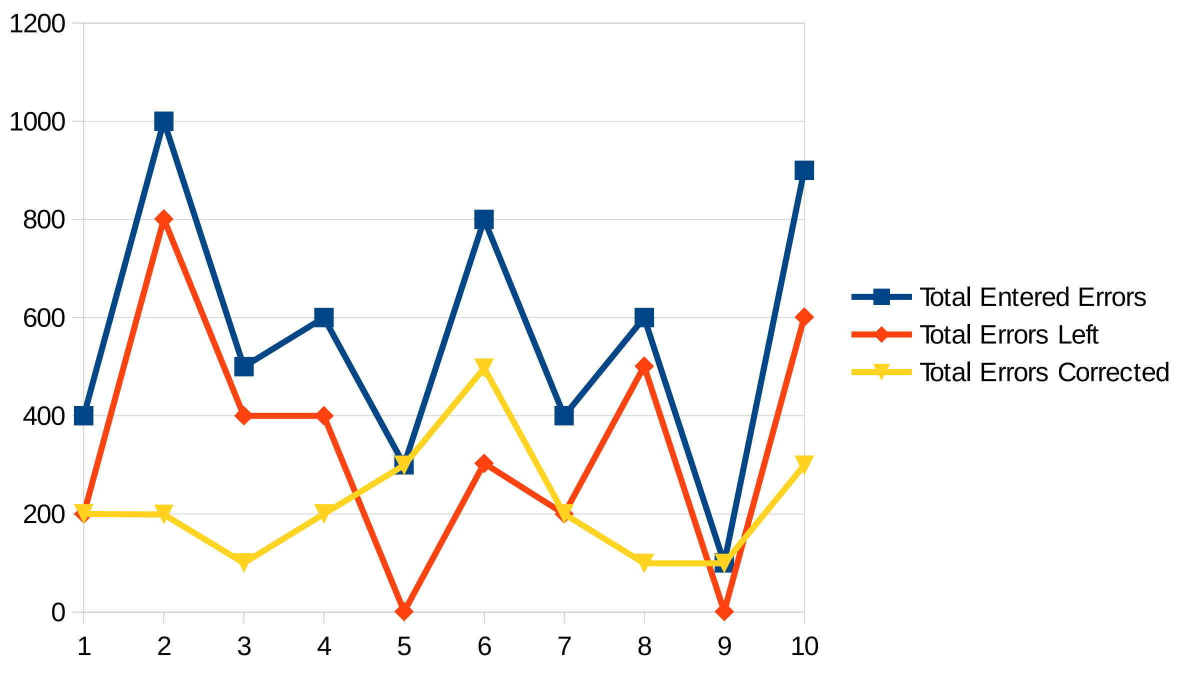
Full results are in sheet one of the experiments spreadsheet.



2)RESULTS OF P = 0.1, PBG = 0.1 and PGB = 0.1

In this experiment, each of the values were set to 0.1. This meant it was unlikely that the state would change, and even once it did it was unlikely that the changes would occur. This could explain why there was a relatively low average of 560 with the highest number coming at 1000 errors. The total number of errors corrected was also a lot closer to the total number of errors left, averaging out at 219.2 and 340.8 respectively. This would be due to the low probability of change, meaning it was harder to get into the bad state for change to occur, resulting in more codes with only one error allowing changes to happen.

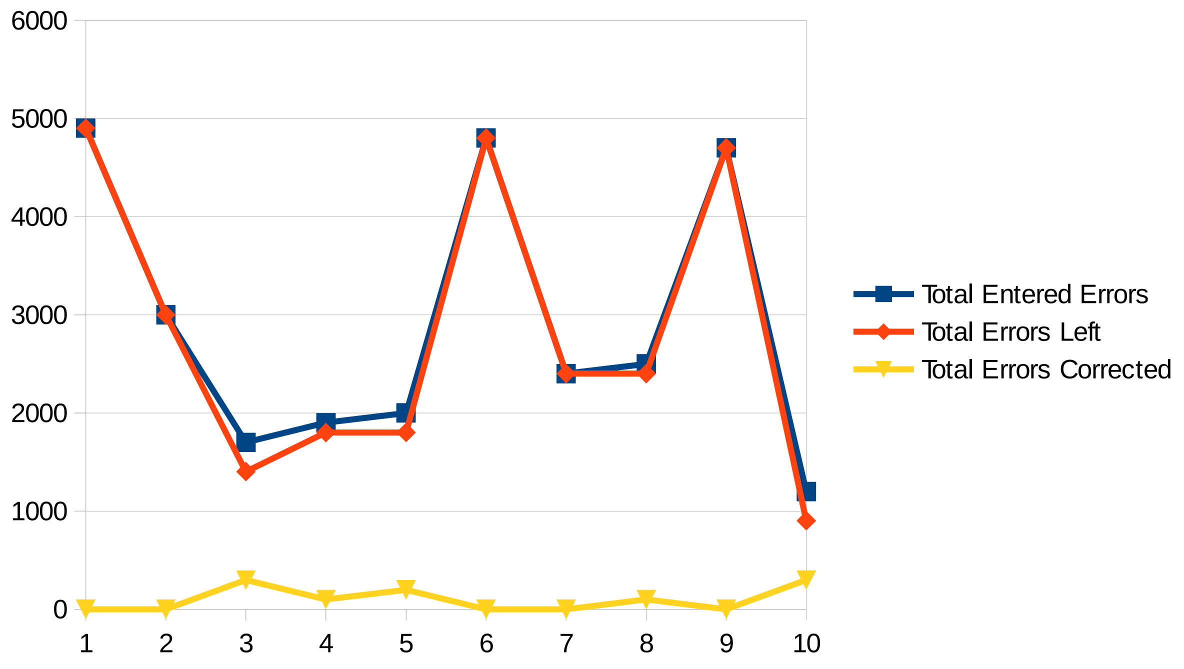
Full results are in sheet two of the experiments spreadsheet.



3) RESULTS OF P = 1, PBG = 0.1 and PGB = 0.1

In this experiment, if the state was changed, it was guaranteed that the code would be changed from that point on, unless in the unlikely case of change back to the good state. This caused a very high number of errors to occur, and due to p = 1, relatively very few errors were corrected. With the total errors averaging at 2910 and the errors corrected averaging at 99.4. There was also a substantial difference in the average number of errors corrected when compared to the errors left, which was averaging out at 2810.6.

Full results are on sheet three of the Experiments spreadsheet

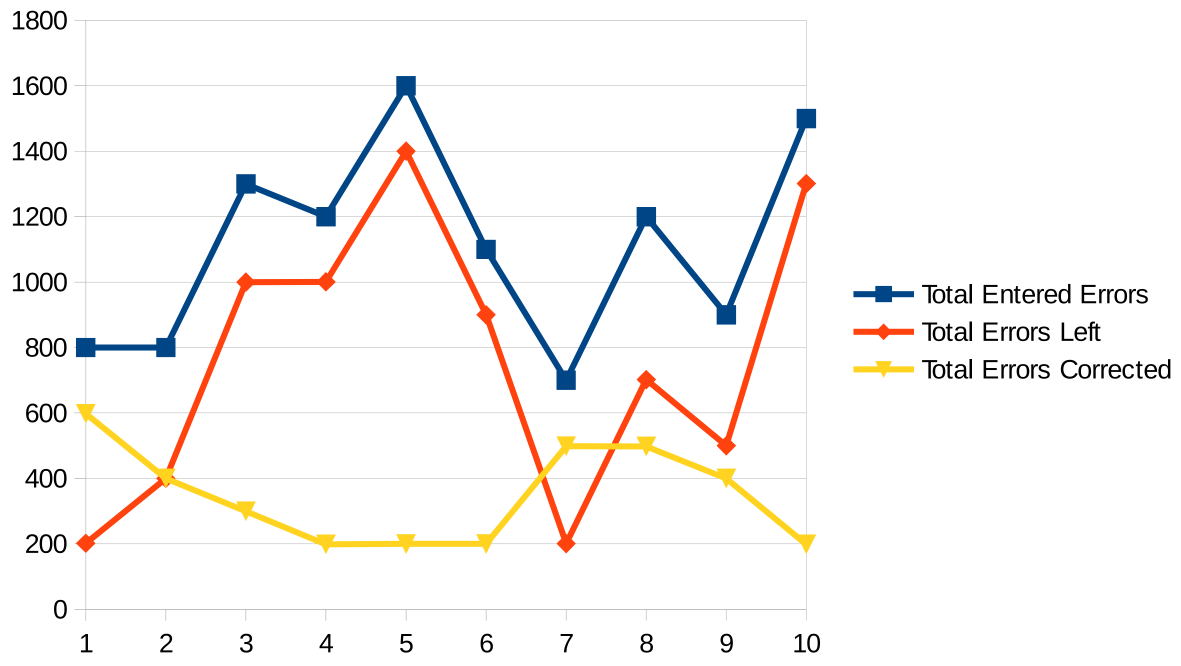


4)RESULTS OF P = 0.3, PBG = 0.5 and PGB = 0.2

In this the chances of the state changing to the bad state was relatively low, while changing from bad to good was substantially higher. This caused a relatively low number of errors, along with a much higher average number of errors corrected. There was also less difference between the errors left and the errors corrected.

Total Entered Errors Average = 1110
Total Errors Left Average = 760.7
Total Errors Corrected Average = 349.3

Full results are on sheet four of the Experiments spreadsheet

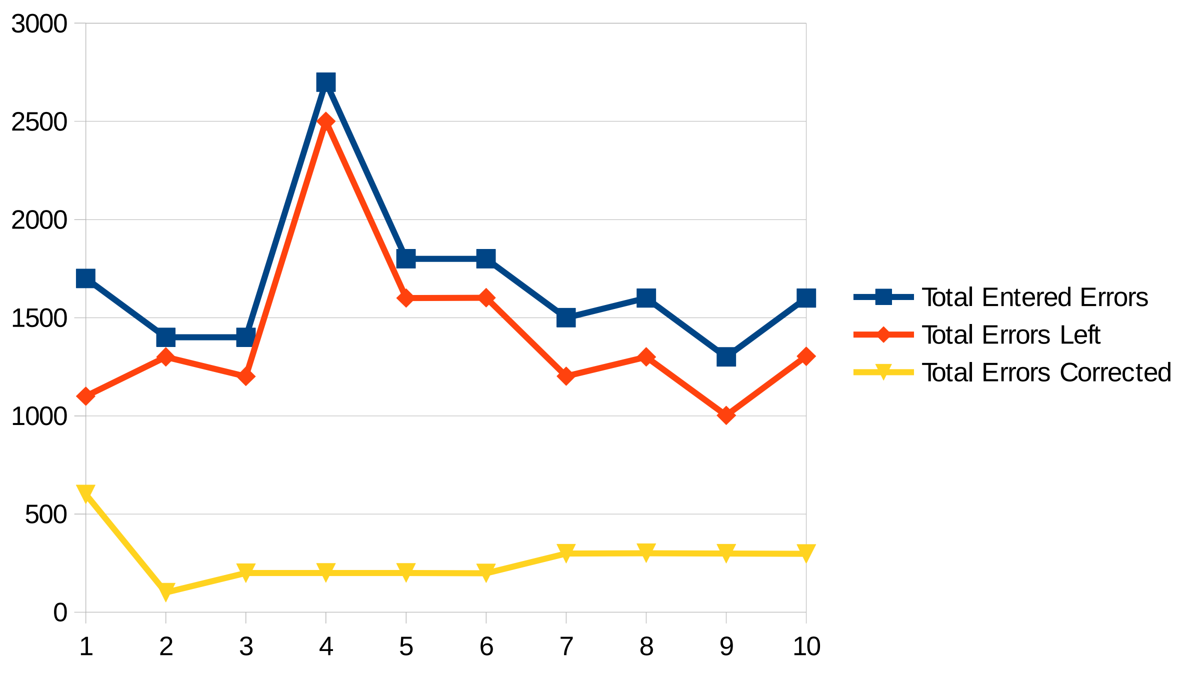


5) RESULTS OF P = 0.3, PBG = 0.3 and PGB = 0.3

In this experiment the values were set to the same relatively low value, giving each instance an equal chance of occurring. This caused a middling value in the total errors entered average, but the total errors left was relatively very high and the gap between the errors left and those corrected was very wide also. I am unsure of this reason, as in this instance I had assumed that due to the even chances, that the values would all be more similar than in other experiments.

Total Entered Errors Average = 1680
Total Errors Left Average = 1411
Total Errors Corrected = 269

Full results are on sheet five of the Experiments spreadsheet



Conclusion:

I felt that this practical went well overall and that the implementation is a good representation of how Hamming codes were meant to be implemented. However, given the results which came from the experiments, I found that they weren't as efficient as they could have been. I am unsure whether my implementation of the interleaving was correct in this instance and whether that could have caused the amount of high errors I received in the experiments, or whether this method of error correction is not as efficient as it could be.