LIÈGE université
Sciences Appliquées

ELEN060-2: Source coding, data compression and channel coding

Arthur Louis[1] and **Adrien Saulas**[2]

[1] *alouis@student.uliege.be (s191230)*
[2] *adrien.saulas@student.uliege.be (s184481)*

Course given by **Louis Wehenkel**[1], **Anthony Cioppa**[2] and **Gaspard Lambrechts**[3]

[1] *l.wehenek@uliege.be*
[2] *anthony.cioppa@uliege.be*
[3] *gaspard.lambrechts@uliege.be*

CONTENTS

I. IMPLEMENTATION

A. Binary Huffman Code

The function implementing the binary Huffman encoding takes the probability dictionary in argument and we directly transformed it into a list of probability tuples and created a list of trees with initially one symbol per tree with its probability.

Then using the function `merge` we created to merge the two most probable trees extracted using the `sort` function, we simply had to loop until the list of trees contained only one element. The function merge used the argument `symbols_dict` to keep track of the merges by assigning the correct codewords to each symbol.

At the end of the loop we simply had to reverse the list of codewords contained in each key of the dictionary and convert it to a string to return the expected kind of output.

Applying the Huffman encoding on the exercise 7 of the second exercise session gave us the following result :

```
> Huffman_code("A":0.05,"B":0.10,"C":0.15,"D":0.15,"E":0.2,"F":0.35)
> 'A': '000', 'B': '001', 'C': '100', 'D': '101', 'E': '01', 'F': '11'
```

The result obtained is not the same as in the exercise session but is still correct because the python code chose different sub trees when extracting the most probable ones because some of the probabilities were equal.

If we wanted to extend to any kind of alphabet size, we would have to change our implementation, sending the complete tree list to merge with an additional argument `alphabet_size` that would allow to pop this number of sub trees from the list and looping in the correct order to apply the merge mechanism and computing the correct codewords.

B. On-Line Lempel-Ziv Encoding

We started by initializing the `dict` with a single entry corresponding to the empty string. We then started scanning the sequence by setting the value `next_index` to 1 and keeping track of the current position with the variable `i`.

For each position `i` we started looking in the dictionary for the longest prefix that matches a key in the dictionary until it is no longer an already existing key using the variables `prefix` and `buffered_prefix`. This is done by using another tracker `j` that keeps track of the index of the last element of the prefix. When `prefix` is no longer a key in the dictionary, the function encodes the codeword corresponding to the `buffered_prefix` and adds the character corresponding to the position `i`.

The function then adds the new codeword for the newly seen prefix. When the initial sequence is all encoded, it outputs the dictionary and the encoded sequence.

Applying the function to the example seen during the course gives us the following result :

```
> LZ_online('1011010100010')
> 100011101100001000010
```

Which is the expected result.

C. Comparison of the Lempel-Ziv Algorithms

The main difference between the two versions of the Lempel-Ziv algorithms is that the basic one needs to have the full file in memory before compressing using a sliding-window and a hash table while the online version can start encoding on the fly when it sees the first character of the input sequence.

This results in the basic version having a better compression ratio for many kinds of files (images, texts, ...)

but can be slow for very large file that don't easily fit in the memory or have large repeated sequences. The online version, on the other hand, is well-suited for data that is generated constantly (streaming, ...). Unfortunately, the online version can have a higher compression ratio that the basic version. Indeed, it requires more handling of the data that as already be seen and may need to encode sub strings that were not present in the dictionary when it started encoding.

D. LZ77 Algorithm

Our implementation of the LZ77 algorithm is straightforward, we start by initializing three variables `encoded_sequence`, `sliding_window` and `look_ahead_buffer` respectively to an empty string, another string and the full sequence to encode. We then loop on the `look_ahead_buffer` until it is empty.

In each iteration of the loop, we find the best match between the `look_ahead_buffer` and the `sliding_window` using our function `best_match` that returns the best match encoded and the number of sliding characters. Using these outputs, we update the variable `encoded_sequence` and slide the `look_ahead_buffer` and `sliding_window` by the correct number of elements.

Applying the function to the example from the statement returns the following result:

```
> LZ77(sequence='abracadabrad',window_size=7)
> 00a00b00r31c21d74d
```

Which is the expected result.

II. SOURCE CODING AND REVERSIBLE (LOSSLESS) DATA COMPRESSION

A. Display of Images and analysis of files

The two images were made visible in the Jupyter Notebook using the packages `PIL` and `io` to respectively show the images and read the files. Both images look the same and have the same 512×512 pixels. The two pictures are presented in figure 1 and 2.

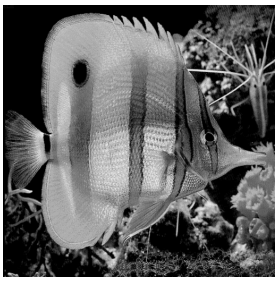


FIG. 1: Raw picture

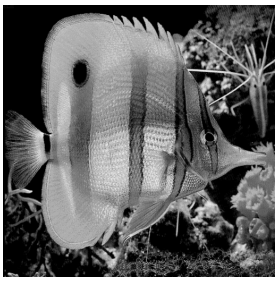


FIG. 2: PNG encoded picture

To represent an image whatever the format, $512 \times 512 = 262144$ symbols are required, one for each pixel. Thus to compute every image possible, there's 256 possibilities per pixel leading to the following formula : 256^{262144} required symbols to represent each and every possible gray scale image.

The raw pictures has a length of 262144 symbols (which is expected) and the PNG encoded one has 160552 symbols which represents an impressive compression.

B. Estimation of the Marginal Probability Distribution of all Symbols from the PNG and Corresponding Huffman Code

Before applying the Huffman encoding to the `PNG.txt` file, a step was made to make every computation in the future easier. The data was transformed in ascii. Indeed, in the current representation, each pixel is represented using two hexadecimal symbols, totalling to 8 bits. We could make our lives easier by turning every symbol into ascii characters that are encoded using 8 bits. Unfortunately, some characters in the ascii representation are used as control characters, thus we also used some characters encoded in UTF-8 using the function `chr` and applied the same method for 1's and 0's as it we make our lives easier with the Lempel-Ziv encoding.

With our newly created ascii string stored in a Python list, we used the function `unique` from `numpy` to create our marginal probability dictionary that we then used to create our binary Huffman Code.

Using our marginal probability dictionary, we encoded the `PNG.txt` and obtained an encoded string of 1289279 bits giving us a compression rate of :

$$\frac{1284416}{1280279} = 1.00323132$$

This result is not really impressive but was expected as the PNG encoding is already really effective to reduce a file size.

C. Expected Average Length for the Huffman Code

By first verifying that the Kraft inequality is respected :

$$\sum_{\text{code}} 2^{-\text{len(code)}} < 1$$

We know that there's a uniquely decodable code for the our encoding and can compute the theoretical average length by multiplying each codeword's length with it's respective probability. Applying this on our codewords dictionary we obtain a length of 7.974232647366582 bits per encoded symbol of the PNG file. In parallel, we also computed Shannon's entropy for our dictionary and obtained 7.957359831200671 bits per encoded symbol.

We then proceeded to compute an empirical by creating 1000 randomly drawn sequences from our probability dictionary, each of length 1000 and encoding them with our Huffman Code. This method gave us an empirical average length of 7.974686999999984 bits per encoded symbol.

By comparing all of these measures we can say that the we are reaching results close to the theoretical lower bound represented by Shannon's entropy that is already close to the 8 bits necessary to represent an image in the RAW representation.

D. Evolution of the Empirical Average Length

By creating random sequences drawn from our probability dictionary as in the previous question, we computed their length and plotted the evolution using `matplotlib.pyplot`, the result can be seen in figure 3. As can be seen on the plot, the empirical length seems to approach the value discussed before and as predicted stays above Shannon's entropy which is the lower bound under which encoding can't go.

E. Encoding the PNG using the On-Line Lempel-Ziv Algorithm

Thanks to the changing of representation to ascii from earlier, the encoding using `LZ_online` was pretty straightforward, the tricky part was just to count the bits required in the Lempel-Ziv representation. We simply had to count the 1's and 0's as 1 bit binaries and the ascii characters as 8 bits. This led to the encoded string being 1537929 bits long. The compression was found by computing :

$$\frac{1284416}{1537929} = 0.835159$$

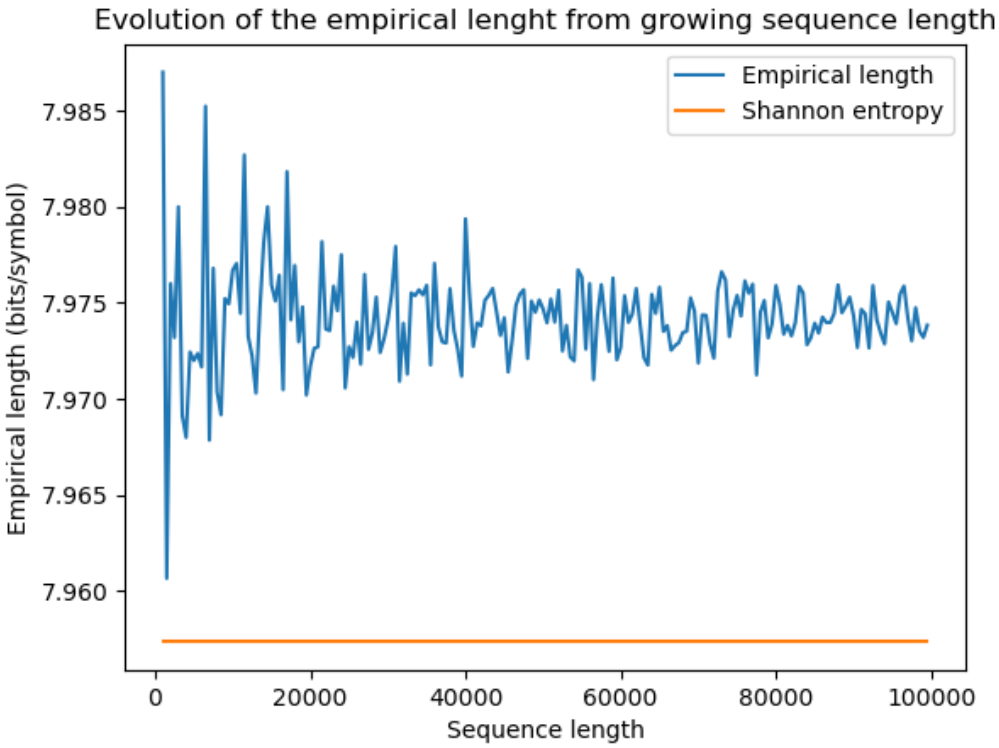


FIG. 3: Evolution of the empirical length

This result was disappointing but expected as the PNG representation of images already use some state of the art compression and compressing the file could only lead to make the file bigger which is not productive, encoding files isn't always a go to option.

F. Encoding the PNG using the LZ77 Algorithm

The preprocessing of the data in ascii also made our lives easier in the process of encoding it using the LZ77 algorithm. Here to count the number of required bits for the encoded representation, we had to think on how many bits will the number associated to the sliding window would be encoded. The answer was 3 bits. Indeed 2 wasn't enough as it could only fit a sliding window of size 4 and 4 would be to big as our sliding window was of size 7 which is smaller than 8 ($= 2^3$). The character associated with the two 3 bits number would be encoded on 8 bits leading to a encoded representation of 2168838 bits. The compress rate can also be found by computing :

$$\frac{1284416}{2168838} = 0.59221$$

Once again, the encoding is not a good idea but this effect was expected has applying LZ77 to a file that was already encoded using the same algorithm (as the PNG does) can only make the file larger. This effect is also amplified by the small window size as a window of width 7 cannot capture recurring patterns in so big files.

G. Combining LZ77 and Huffman Code

Combining LZ77 and Huffman Code is a good idea and is used in state of the art compression as the avoidance of patterns can be identified using LZ77 and Huffman encoding is really good at encoding symbols. The two algorithms can be applied in either first or second option.

H. Applying the combination of LZ77 and Huffman Code to the PNG Sequence

The combination we chose to encode our file was to first apply the Huffman encoding and then proceed to apply the LZ77 algorithm. This would allow us to represent the 1's and 0's after the LZ77 encoding with only 1 bit and the two numbers to identify the window with once again 3 bits since we used the same window of size 7. This led to a encoded file of 2429992 bits and a compression ratio computed with :

$$\frac{1284416}{2429992} = 0.528568$$

I. Lengths and Compression Rates of the Different Algorithms with Different Windows

We computed the lengths and compression rates by looping using different window sizes and applied the two different algorithms presented earlier and the evolution of the length of the files and the compression rates are represented in the plots 4 and 5.

FIG. 4: Evolution of the compression ratios as a function of the sliding window size

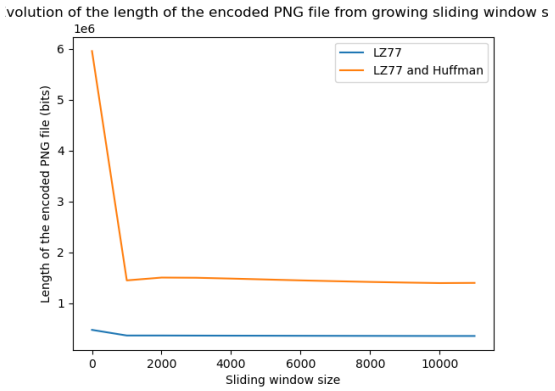


FIG. 5: Evolution of the empirical as a function of the sliding window size

The results in the plots 4 and 5 were expected. Indeed, with the sliding window size equal to 1, we can only increase the size of the file by a large factor and when increasing the window size we tend to quickly reach a plateau that never goes over the value of 1.0 for the compression rate, indicating that both process are counter productive.

J. Encoding Directly the Pixel Sequence

This time we applied our binary Huffman encoding to the `pixel.txt` file. After using the same ascii conversion technique and creating the marginal probability dictionary we obtained an experimental file length of 1584881 bits compared to the original of 2097152 bits. This gave us the following compression ratio :

$$\frac{2097152}{1584881} = 1.32322$$

This result was expected. Indeed we obtained a better compression rate than when encoding the PNG file because the raw had more compression potential than it's PNG formatted copy.

With the computed marginal probability dictionary and the corresponding Huffman binary code we obtained a theoretical average length of 6.04584 bits per encoded symbol, which proves once again the better compression potential of the `pixel.txt` file compared to the `PNG.txt` file.

K. Comparison of the Huffman Code of the PNG and Pixel Sequence

By simply comparing the number of bits needed in both representations, we can conclude that applying the Huffman encoding to the `PNG.txt` file is better than on the `pixel.txt`. Indeed, the first one needs 1289279 bits and the second one 1584881 bits. Even though the compression rate of the encoding done on the `pixel.txt` file seems better, we need to remember that the `PNG.txt` already has encoding done on it and even though the improvement is minimal, the encoding is better on the `PNG.txt`.

III. CHANNEL CODING

A. Reading and Encoding the Text Signal

The implementation to turn the text stored in `text.txt` was done with the function `turn_to_binary`. The size of our text in binary will be multiplied by 8 since in the ASCII code each character is represented by 8 bits. We thus have :

$$2261 \text{ characters } \times 8 = 18088 \text{ bits}$$

B. Channel Effect on the Binary Text Signal

Here's an example of our text with adding a noise of 0.02 corruption per bit :

TH050BOy UHO LiVED(Lr. and Mrc0e Dqpsleù1c of number fïur, Privet Dvive, wure proud to$say that txey were pgrfestly noòmal, thANk$yOu vEry mush. They were the äas4 people youĹ expect to be(involved in an}thinŚtòanGe or mySteri7fus, because txey just dmfnf́ hol$ ÷ith such nnnsense. Mr. Derslåy was the directos of!a fiòm calnud Erunæinos, whil!MAle drillqn he was a cig, beefy man with!herd,y a Y nek, althoug('e did"have!a0very larGe ousuache. Mrs. DUrslEy vas thi. anf blknde ald hid neaòly twice the0usual ammunt of neck,00which caea in vEry usåful00as0she cpent so much Of her timg craning over gardeo fences,0spying on tXe oeighborw. The EursleY s èad a smalì son calle$ Dudley0anf in 4heir gpi.ion there was no fIner boy anywhure Phe DurqleysphAä'everything thay gaoted, bqt they also had a secret, and their gRåatest feab ÷as'thá— sgmebody0would di3ãover it. Thmy dadnt́ think they could beaò it if anYone!f7fund ouu about the Pëtters."Mrs.$Pmttår war Mrs. Lursleyś sisteò- buu tjey hadnt́ met F7fr several ieArs; in facu, Mrs. Dursley pretended she dIdnd́ hawe a qhsder, recause her sister anD hdr go/d-for-noth)ng luscaîd were as unEursleyish as iu waw possmble to"be. The Dwrqleyc sèuddermd(to!tjink w'qt the neighborS would0say if the(10otters qrriv%d yl00the s$regt. T'g 04ursleys kfew tla— the Qottdrs had a smaLl son,"too, bwt t(ey lad ne ep eVen'såen him. This boy 7as anotèer good reaso fOr keeping tie Pmpters cwAy; thmy Dédnt́ vint 04udley mizing with!a child likm ôhat.00Wheî Mr® and Mps. Dursley0woke up on the0dull, gray Tu%sday0our stOr9 starts, thera wa{ nothinf !bout the(cloudy so9 outriDe to {ugeest that rtranwe and mystebio5s thyngs 7oõld00sïon be happenync0all ovep the cotntry."Lv.(Tursley'hulmEd0aw8he pibked out0iis mOst'boring!tie foú workl end Mrs. Du6sldy gossipde aváy happily as she!wrustled a$screaming Dudley ynto his high chamr.'None of t'em!notioEd"a0larïe, tawny owl Flutter pa3t thw window. At (alf p!wT emght, Mr. FursLey pickmd up his bri%fcase.0pecoed Írs.0Dupsley on tie chee+, and tvieä'to kiss Dudlmy çood-bye"but m)ssed, because 04udl%} was0now hivang a tantzum and phrowinç his cereal at the wallw. "Ìmttle t7f{e,"'horTled Mr0e$Dursley as he left(the houóe. he got into his cer anD baaked(out oæ numbeò fourś00dsive.*

We can notice that although the error rate seems low (only one bit out of 50 is changed), in ASCII models, this changes approximately one letter out of six, which greatly impairs the readability of the message.

C. Channel Effect on the Binary Text Signal with Redundancy

The process of encoding the text using the Hamming code was done using the function `hamming_code`. The encoding is done using the encoding matrix G, which is coded as-is in the Notebook and allow to convert any 4 bit message and map into the Hamming(7,4) message. The encoding is done by simply looping on the binary transformed initial text.

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

D. Decoding using the Redundancy

The proces and decoding the text using the Hamming code was done using the function `hamming_decode`.

Firstly, we notice that encoding and decoding the text without adding noise does not change the text itself. Indeed, since in the Hamming(7,4) code, the text is represented by the first 4 bits of each group of 7, it is then easy to retrieve the encoded message.

During decoding, we simply took the first 4 bits of a group representing a part of the message and ran it back through the generator matrix G expressed below.

At the output of this generator matrix, we end up with 2 messages whose first 4 bits represent the bits containing the information, and therefore in this case are identical, and the last 3 bits represent the parity bits that allow us to realize if the message was corrupted during transmission. Finally, by performing an XOR between these 3 parity bits of each message, we can find the syndrome. This syndrome will be null if the message has undergone no modification (or had no more than 2 modifications in the message) and non-null if there has been one or more modifications in the message. Moreover, if we face a non-null syndrome, we can retrieve the corrupted bit and correct it (provided there is only one error in the message). For the correction, we just need to follow the graph given at slide 14 of the channel coding part 1 (figure 6) in the theoretical course, which can be read as expressed in the following table for a seven-bit message $s_1 s_2 s_3 s_4 p_1 p_2 p_3$

TABLE I: Action to take upon detecting a syndrome

syndrome	bit to change
101	s_1
110	s_2
111	s_3
011	s_4
100	p_1
010	p_2
001	p_3

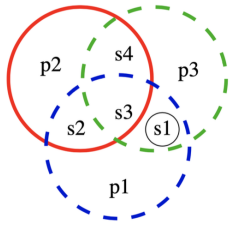


FIG. 6: Parity circles

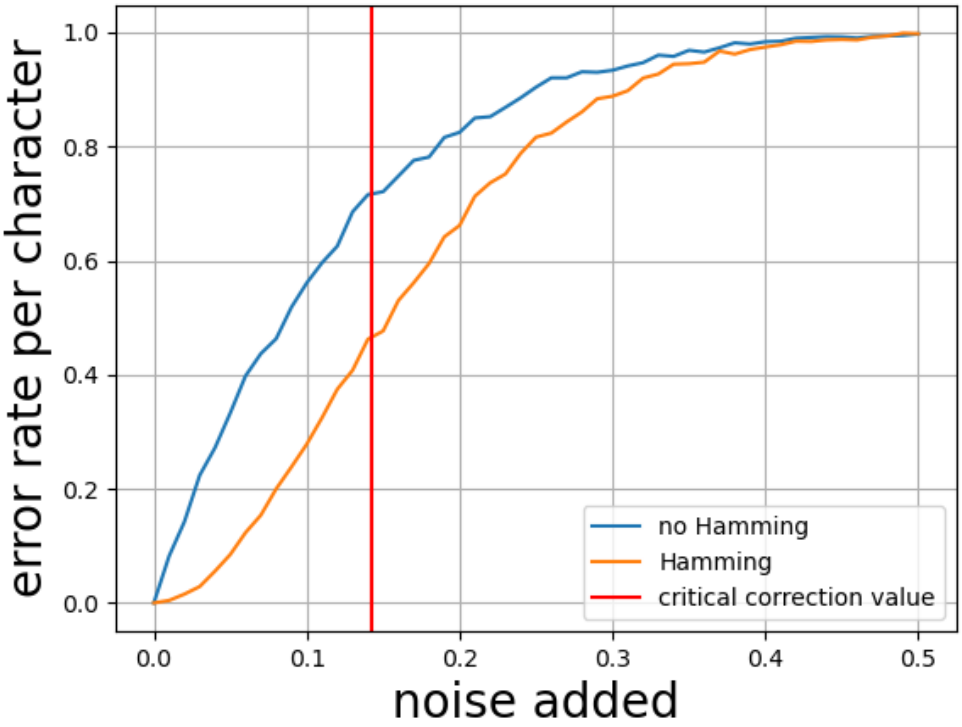
Thanks to this method, we can correct up to one error and detect up to two in our message since the Hamming distance between two possibilities is 3 and get the following decoded text :

*THE BOY WHO L)VED Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal, thank you very much. Thåy were the last people youd́ expect to be involved in anything strange or mysterious, because they just didnt́ hold with such nonsense. Mr. Durslcy was the director of a firm cal,ed Grunnings, which made drills. He was a bigį beefy mao with hardly any neck, although he did!have a very large mustache. Mrs. Dursley was thin and blonde and had nearly twice the usual amount of neck, which came in very useful as shμ spent so much of her time craning over garden fences, spying on the@neighbors. The Dursley s had á small son c±lled Dudley and in their opinion0there was no finer boy anywhere. The Dursleys had everything they wanted, but they also had a secret, and their greatest fear was that somebody would discover it. Theq didnt́ think they could bear it if anyone found out about the Potters. Mrs. Potter was Mrs. Dursleyś sister, but they hadnt́ met for several years; in fact, Mrs. Dursley pretUnded she didnt́ have a sister, because her sister*and her good-for-nothing husband were as unDursle©ish as it was possible to be. The Dursleys shuddered to think what the neighbors would say if the Potters arrived in the street. The Dursloys knew that the Potters ‘ad a small son, too, but they xad never!even seen him. This boy was inother good reason for keeping the PotDers away; they didnt́ want Dudley mixing with a child like tÈat. When Mr. and Mrs. Dursley woke up on the dull, gray Tuesday our story starts, there was nothing adout the cloudy sky outside to suggest that strange and mysterious things would soon be happening all over the*

c0funtry. Mr. Dursley hummed as he picked out his most boring tie for work, and Mrs. Dursley-gossiped away happily as she wrestled a screaming-Dudley into his-high chair. None of them noticed a large, tawny owl flutter past the window. At half past eight, Mr. Dursley picked up his briefcase, tecked Mrs. Dursley on the cheek, and tried to kiss Dudley good-bye but'missed, because Dudley was now having a tantrum and throwing hms cereal at the walls. "Little tyke," chortled Mr. Dursley as he left the house. Be got into his car and backe' out of number fourś drive.

which seems a lot better than without the Hamming coding method. In fact the per character error is only 0.0030959 vs 0.15 without the Hamming method.

E. Transmission over a Binary Symmetric Channel



We can notice that the Hamming (7,4) method allows correcting a part of the errors caused by transmission noise. Unfortunately, since this method only allows correcting one bit of error out of the 7 encoded, when the error rate exceeds $\frac{1}{7}$, the error curve per character of the Hamming method becomes similar to that without encoding. Ultimately, once the noise rate reaches 0.5, the message is completely changed. However, the Hamming method still allows correcting up to about 30% of errors around a 10% noise level. Therefore, the Hamming (7,4) method is well adapted to our transmission problem.

F. Reducing the Probability of Errors

If we want to increase the decoding quality, we can increase the Hamming distance of the different error possibilities by adding a parity bit, for example. However, this will decrease the transmission efficiency. If we want to increase efficiency, we can use the opposite reasoning by removing a parity bit or adding a message bit per code.