# LEMPEL-ZIV ENCODING ALGORITHM

ECSE 436: Signal Processing Hardware

DECEMBER 20, 2016

CHROUK KASEM 260 512 917
MUHAMMAD TAHA 260 505 597
Course Instructor: Professor Jan Bajcsy

# Table of Contents

# Table of Figures

# 1. Project Description and Objectives

## 1.1. Project Description

As the data created in the world is rapidly increasing to reach orders of zettabytes [1], the burden of need of compression to save storage space, increase transmission efficiency, lower transmission time and save bandwidth is compulsory. The father of information theory, Claude Shannon, has paved the way for modern compression when he published in 1948 his paper "A Mathematical Theory of Communication" [2]. His theorem, Shannon Source Theorem, provide basis for uniquely decodable lossless compression algorithms: Huffman, Run-length, Lempel-Ziv and Turbo compression. This project looks at Lempel-Ziv compression algorithm that's used daily by computer users in 'zip' files, as well as GIF images. This project is an extension of the Design Project, where another three implementations of compression algorithms were developed, tested and used with source files [3].

The design process of this project involves designing three versions of encoders and decoders. The first version of the encoder has a fixed window and fixed length matches with pointer values at each match in window. The second version of the encoder has a fixed window and fixed length matches with pointer values at each bit in window. The third version of the encoder has a sliding window and fixed length matches with pointer at each bit in the window. The decoders for all three versions retrieve original data files without loss in an error-free channel.

After successful completion of all three designs of both encoders and decoders, the algorithms are extensively tested. All three encoders and decoders are tested with varying source files; both i.i.d source files with different statistics and Markov source files. The encoders are also tested among each other to detect trends of each implementation with different sources. The encoders are further tested with practical source files: images and text files for reusability of this implementation in real-life. Finally, this project takes a step further by running optimization simulations for all encoders to find the optimal window size and match length that achieve minimum compression.

## 1.2. Motivation and Objectives

This project focuses on Lempel-Ziv encoding algorithm. Lempel-Ziv is a memory, lossless compression algorithm that provides a uniquely decodable encoded string [4]. The motivation of this project is to implement universal Lempel-Ziv compression due its widespread use in everyday compression and its versatility to varying source statistics. The full system of each encoder/decoder is as show in **Figure 1.1**.

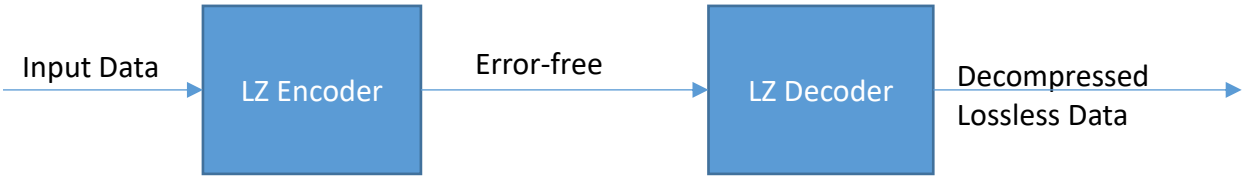


*Figure 1.1.1: Block Diagram for LZ full system*

The objectives of this project are:

1) Successful implementation of three versions of uniquely decodable Lempel-Ziv encoders
2) Successful implementation of related three versions of lossless Lempel-Ziv decoders
3) Testing of Lempel-Ziv encoders/decoders with different sources with varying parameters
4) Optimization simulations for window size and matches sizes for all three versions of encoders

# 2. Universal Lempel-Ziv Encoder Version 1

## 2.1. Introduction

The first version of the Lempel-Ziv encoder is the fixed window (w), fixed matches length (n), encoder. The encoder looks at each n bits and compares them to the matches in the fixed window, if encoder finds a match then it sets the FLAG bit to 1 followed by the pointer to the match in widow. This implementation of the encoder looks at matches in the window at each n bits; $X_0$, $X_n$, $X_{2n}$, …,$X_{w-n}$; making the size of pointer in encoding equal to $\log_2(w/n)$. If the encoder doesn't find a match, then it sets the FLAG bit to 0, and copies the not encoded n bits in encoded string.

An example of the implementation of this version of encoder is shown in **Figure 2.1.** This example has n as 3, with window size w equal to 12. The size of the pointer values is shown in **equation (1)**:

$$\log_2(w/n) = \log_2(12/3) = \log_2(4) = 2 \text{ bits} \quad \textbf{(1)}$$

The window size for i.i.d source in this example with probability of zero,$P_0$ = 0.5, and probability of one, $P_1$= 0.5, can be calculated using **equation (3)**, with H(X) of this binary source given in **equation (2)**:

$$H(X) = P_0\log_2(\frac{1}{P0}) + P_1\log_2(\frac{1}{P1}) = 0.5\log_2(\frac{1}{0.5}) + 0.5\log_2(\frac{1}{0.5}) = 1 \quad \textbf{(2)}$$

$$w = n^2 2^{nH(X)} = (3)^2 2^{3x1} = 72 \quad \textbf{(3)}$$

The example doesn't use window size of 72 for illustrative reasons. It also important to note this window size w and length of match n are not optimal and, as shown in example, will not necessarily yield in compression. Simulations run on varying parameters of w, and n in **Section 5.3** will show the optimal values that will yield the best compression ratio.

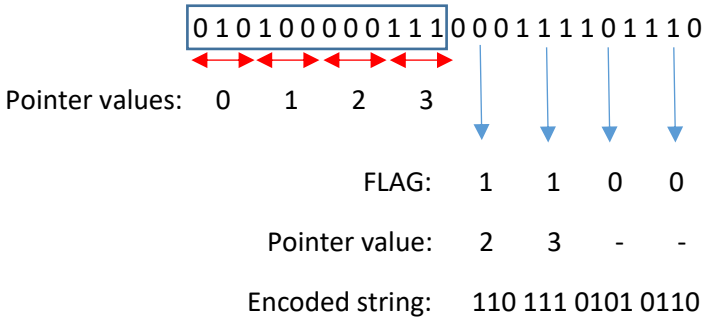


*Figure 2.1: Illustrative example of LZ encoder version 1 with n=3 and w=12*

The encoded string of this example from version 1 of encoder is 11011101010110. The expected length, E(L), of this compression can be calculated using **equation (4)**, where $P_{match}$ is the probability of finding a match in the window (i.e. probability FLAG bit is 1) and $P_{no\text{-}match}$ is the probability that no match is found in the window (i.e. probability FLAG bit is 0):

$$E(L) = P_{match} \times (\frac{1+log2(w/n)}{n}) + P_{no\text{-}match} \times (\frac{1+nH(X)+2log2(n)}{n}) = 0.5 \times (\frac{1+2}{3}) + 0.5 \times (\frac{1+3\times1+2\log2(3)}{3})$$

$$= 1.695 \text{ bits/pixel} \quad \textbf{(4)}$$

## 2.2. Implementation

All versions of the Lempel-Ziv encoder were implemented using MATLAB. The flowchart in **Figure 2.2** shows the implementation logic of first version of LZ encoder.

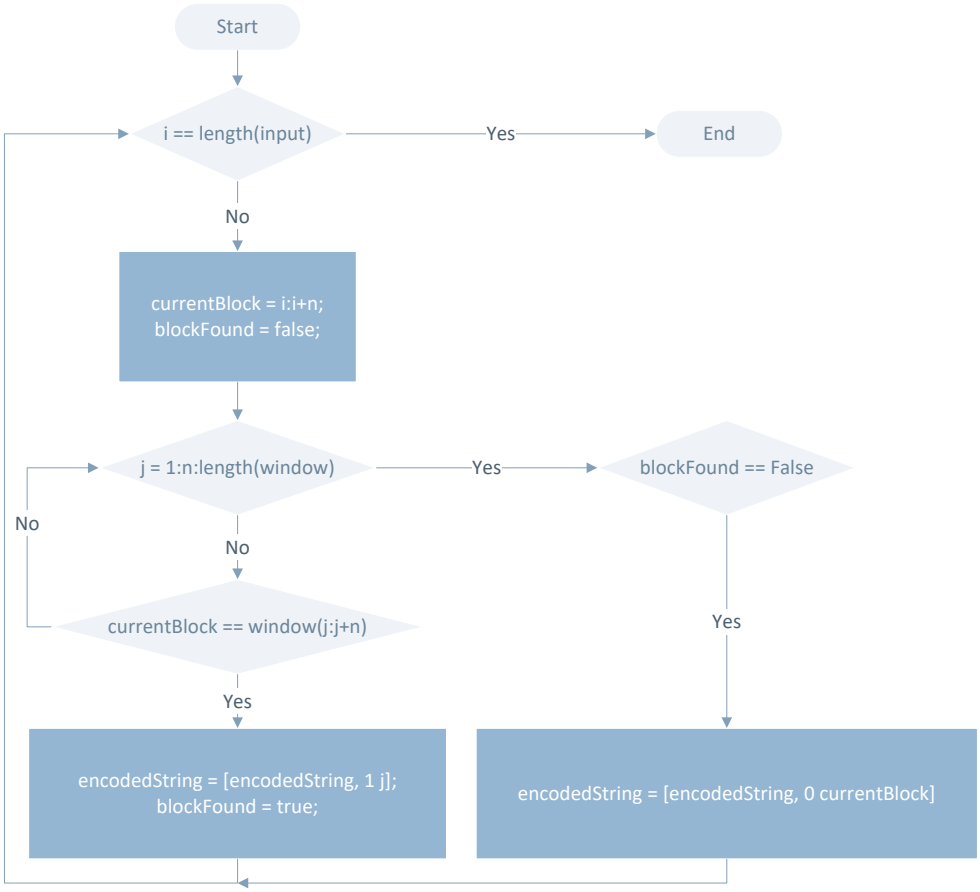


*Figure 2.2: Flowchart for V1 of LZ Encoder*

The encoder loops over all input data bits at increments of n until it reaches end of input string. At each iteration, it sets the currentBlock vector to input data bits starting at i to i+n and sets blockFound to false. The encoder then loops over window at increments of n (ie. window bits at j to j+n) to find a match to currentBlock. If encoder finds a match, it sets blockFound to true and adds FLAG bit 1 to encodedString value along with pointer value (j) of match in window. If encoder doesn't find a match at the end of looping over window matches, it checks if blockFound is false and adds FLAG bit 1 to encodedString along with the currentBlock bits unchanged. This continues until encoder reaches the end of the input data where it outputs an encodedString. The code in Matlab can be found in **Figure 8.1** in Appendix.

# 3. Universal Lempel-Ziv Encoder Version 2

## 3.1. Introduction

The second version of the Lempel-Ziv encoder is the fixed window (w), fixed matches length (n), encoder. The encoder looks at each n bits and compares them to the matches in the fixed window, if encoder finds a match then it sets the FLAG bit to 1 followed by the pointer to the match in widow. This implementation of the encoder looks at matches in the window at each bit; $X_0$, $X_1$, …, $X_n$, $X_{n+1}$, $X_{n+2}$ …, $X_{2n}$, …,$X_{w-n}$, $X_{w-(n+1)}$, $X_{w-(n+2)}$, …, $X_w$; making the size of pointer in encoding equal to $\log_2(w)$. If the encoder doesn't find a match, then it sets the FLAG bit to 0, and copies the not encoded n bits in encoded string.

An example of the implementation of this version of encoder is shown in **Figure 3.1.** This example has n as 3, with window size w equal to 12. The size of the pointer values is shown in **equation (5)**:

$$\log_2(w) = \log_2(12) = 3.584962 = 4 \text{ bits} \quad \textbf{(5)}$$

The window size for i.i.d source in this example with probability of zero,$P_0 = 0.5$, and probability of one, $P_1 = 0.5$, can be calculated using **equation (7)**, with H(X) of this binary source given in **equation (6)**:

$$H(X) = P_0\log_2(\frac{1}{P0}) + P_1\log_2(\frac{1}{P1}) = 0.5\log_2(\frac{1}{0.5}) + 0.5\log_2(\frac{1}{0.5}) = 1 \quad \textbf{(6)}$$

$$w = n^2 2^{nH(X)} = (3)^2 2^{3x1} = 72 \quad \textbf{(7)}$$

The example doesn't use window size of 72 for illustrative reasons. It also important to note this window size w and length of match n are not optimal and as shown in example will not necessarily yield in compression. Simulations run on varying parameters of w, and n in **Section 5.3** will show the optimal values that will yield in best compression ratio.

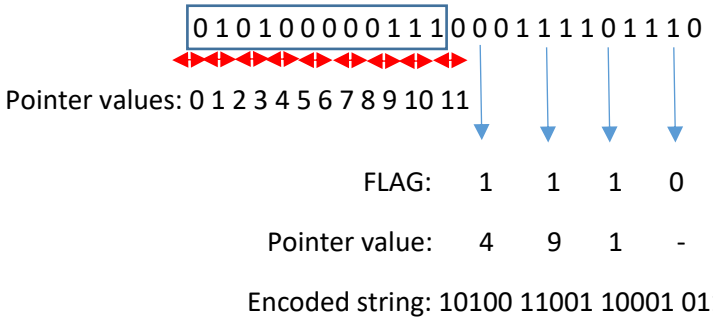


*Figure 3.1: Illustrative example of LZ encoder version 3 with n= 3 and w=12*

The encoded string of this example using version 2 of encoder is 1010011001100010110. The expected length, E(L), of this compression can be calculated using **equation (8)**, where $P_{match}$ is the probability of finding a match in the window (i.e. probability FLAG bit is 1) and $P_{no\text{-}match}$ is the probability that no match is found in the window (i.e. probability FLAG bit is 0):

$$E(L) = P_{match} \times (\frac{1+log2(w)}{n}) + P_{no\text{-}match} \times (\frac{1+nH(X)+2log2(n)}{n}) = 0.75 \times (\frac{1+\log2(12)}{3}) + 0.25 \times (\frac{1+3\times1+2\log2(3)}{3})$$

$$= 1.7437 \text{ bits/pixel} \quad \textbf{(8)}$$

## 3.2. Implementation

All versions of the Lempel-Ziv encoder were implemented using MATLAB. The flowchart in **Figure 3.2** shows the implementation logic of second version of LZ encoder.

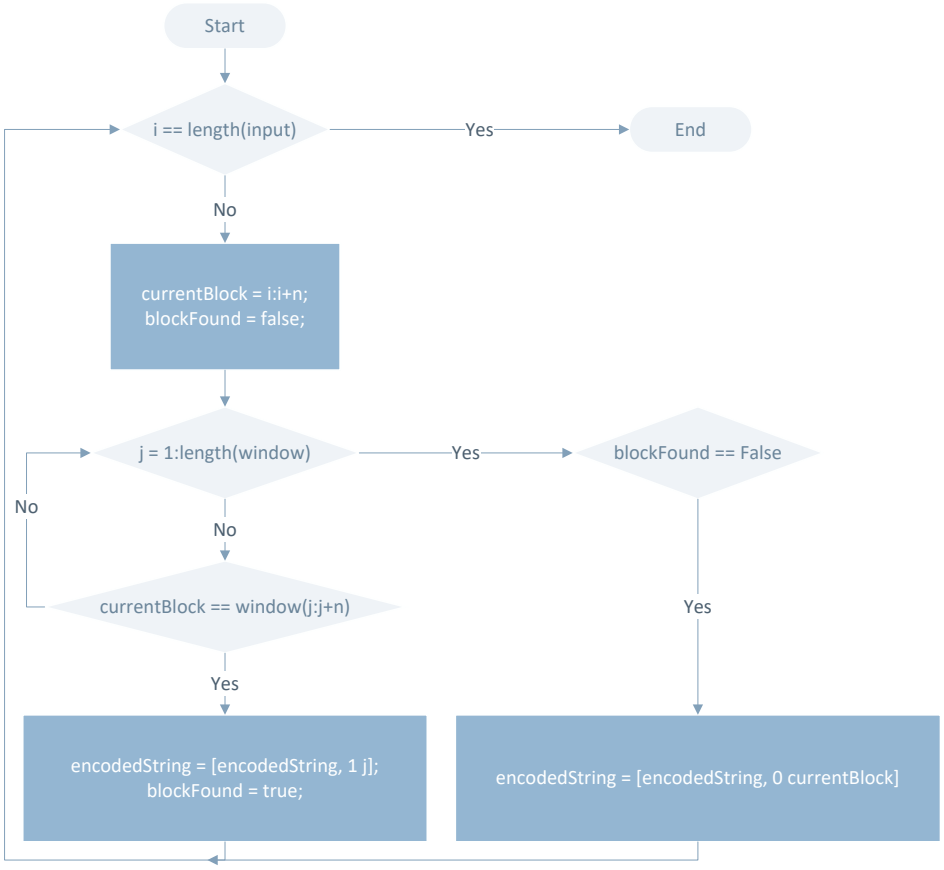


*Figure 3.2: Flowchart for V2 of LZ Encoder*

The encoder loops over all input data bits at increments of n until it reaches end of input string. At each iteration, it sets the currentBlock vector to input data bits starting at i to i+n and sets blockFound to false. The encoder then loops over window at increments of 1(j increases by 1 on each iteration versus by n as in first version of encoder) looking for matches of size n to find a match to currentBlock. If encoder finds a match, it sets blockFound to true and adds FLAG bit 1 to encodedString value along with pointer value (j) of match in window. If encoder doesn't find a match at the end of looping over window matches, it checks if blockFound is false and adds FLAG bit 1 to encodedString along with the currentBlock bits unchanged. This continues until encoder reaches the end of the input data where it outputs an encodedString. The code in Matlab can be found in **Figure 8.3** in Appendix.

# 4. Universal Lempel-Ziv Encoder Version 3

## 4.1. Introduction

The third implementation of the Lempel-Ziv encoder is the sliding window (w), fixed matches length (n), encoder. The encoder looks at each n bits and compares them to the matches in the sliding window, if encoder finds a match then it sets the FLAG bit to 1 followed by the pointer to the match in widow. This implementation of the encoder looks at matches in the window at each bit; $X_0$, $X_1$, …, $X_n$, $X_{n+1}$, $X_{n+2}$ …, $X_{2n}$, …,$X_{w-n}$, $X_{w-(n+1)}$, $X_{w-(n+2)}$, …, $X_w$; making the size of pointer in encoding equal to $\log_2(w)$. If the encoder doesn't find a match, then it sets the FLAG bit to 0, and copies the not encoded n bits in encoded string. The window changes by sliding by one bit after encoding n bits, thus window for each n bits is different. This is useful for sources with evolving statistics such as Markov Sources.

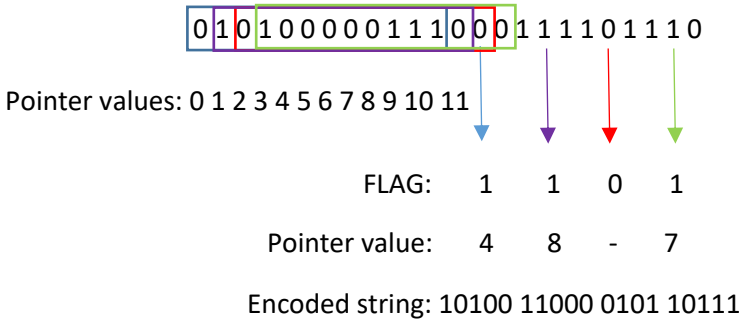


*Figure 4.1: Illustrative example of LZ encoder version 3 with n=3 and w=12*

An example of the implementation of this version of encoder is shown in **Figure 4.1.** This example has n as 3, with window size w equal to 12. The size of the pointer values is shown in **equation (9)**:

$$\log_2(w) = \log_2(12) = 3.584962 = 4 \text{ bits} \quad \textbf{(9)}$$

The window size for i.i.d source in this example with probability of zero,$P_0 = 0.5$, and probability of one, $P_1 = 0.5$, can be calculated using **equation (11)**, with H(X) of this binary source given in **equation (10)**:

$$H(X) = P_0\log_2(\frac{1}{P0}) + P_1\log_2(\frac{1}{P1}) = 0.5\log_2(\frac{1}{0.5}) + 0.5\log_2(\frac{1}{0.5}) = 1 \quad \textbf{(10)}$$

$$w = n^2 2^{nH(X)} = (3)^2 2^{3x1} = 72 \quad \textbf{(11)}$$

The example doesn't use window size of 72 for illustrative reasons. It also important to note this window size w and length of match n are not optimal. Simulations run on varying parameters of w, and n in **Section 5.3** will show the optimal values that will yield in best compression ratio.

The encoded string of this example using version 3 of encoder is 1010011001100010110. The expected length, E(L), of this compression can be calculated using **equation (12)**, where $P_{match}$ is the probability of finding a match in the window (i.e. probability FLAG bit is 1) and $P_{no\text{-}match}$ is the probability that no match is found in the window (i.e. probability FLAG bit is 0):

$$E(L) = P_{match} \times (\frac{1+log2(w)}{n}) + P_{no\text{-}match} \times (\frac{1+nH(X)+2log2(n)}{n}) = 0.75 \times (\frac{1+\log2(12)}{3}) + 0.25 \times (\frac{1+3\times1+2\log2(3)}{3})$$

$$= 1.7437 \text{ bits/pixel} \quad \textbf{(12)}$$

## 4.2. Implementation

All versions of the Lempel-Ziv encoder were implemented using MATLAB. The flowchart in **Figure 4.2** shows the implementation logic of third version of sliding window LZ encoder.

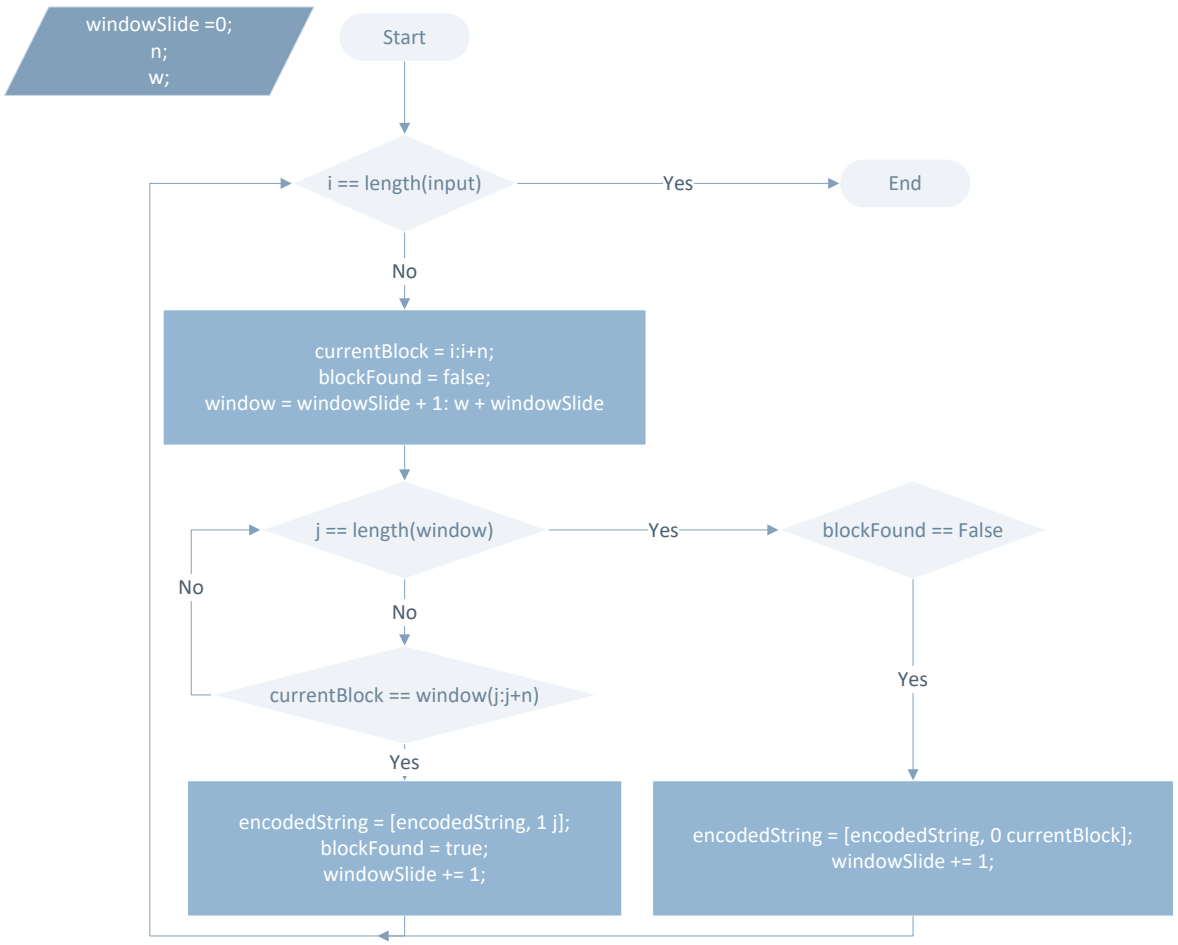


*Figure 4.2: Flowchart for V3 of LZ Encoder*

The encoder loops over all input data bits at increments of n until it reaches end of input string. At each iteration, it sets the currentBlock vector to input data bits starting at i to i+n and sets blockFound to false. Since the window is sliding, this version of encoder also sets the window to w bits plus windowSlide, which is initially zero. The encoder then loops over window at increments of 1(j increases by 1 on each iteration versus by n as in first version of encoder) looking for matches of size n to find a match to currentBlock. If encoder finds a match, it sets blockFound to true and adds FLAG bit 1 to encodedString value along with pointer value (j) of match in window. If encoder doesn't find a match at the end of looping over window matches, it checks if blockFound is false and adds FLAG bit 1 to encodedString along with the currentBlock bits unchanged. Also, it increments the windowSlide by 1, therefore the next window set will also move by one. The process repeats for the next n data bits with the new window. This continues until encoder reaches the end of the input data where it outputs an encodedString. The code in Matlab can be found in **Figure 8.5** in Appendix.

# 5. Results and Testing

In order to evaluate the accuracy of the developed encoders and decoders a basic strategy was followed to ensure their accuracy. After the development process, some basic tests were run through each encoder and decoder to ensure they execute as expected. For these tests, small sized and simple input files were passed through the algorithms and the output from encoder/decoder was also computed by hand to ensure conformity. The algorithms were also stepped through in debug mode during this process to check if each step of the function was properly executed. After this step, some random samples of large file (10^4) were passed through the algorithm and the output from the decoder was compared with the initial input to ensure they are both the same. The next step was to simulate the three algorithms under different sources and parameters and observe their behavior accordingly, which is discussed in the following sections.

## 5.1. I.I.D Sources

For all three algorithms, it was necessary to observe not only how efficient the compression is, but how often the codewords match, under different block sizes (n). The window size was computed according to the formula provided in **eq (3).** This was achieved by having a counter count every time a match was identified (when the flag bit set as 1), and dividing this count by the total number of iterations each algorithm went through to arrive at the encoding. In the graphs shown in **Figures 5.1, 5.2 and 5.3**, this variable is defined as P-match. Each simulation was repeated 50 times and the average value of P-match taken and plotted as shown in the graphs. The simulations were simulated for an i.i.d input file size of 10000.

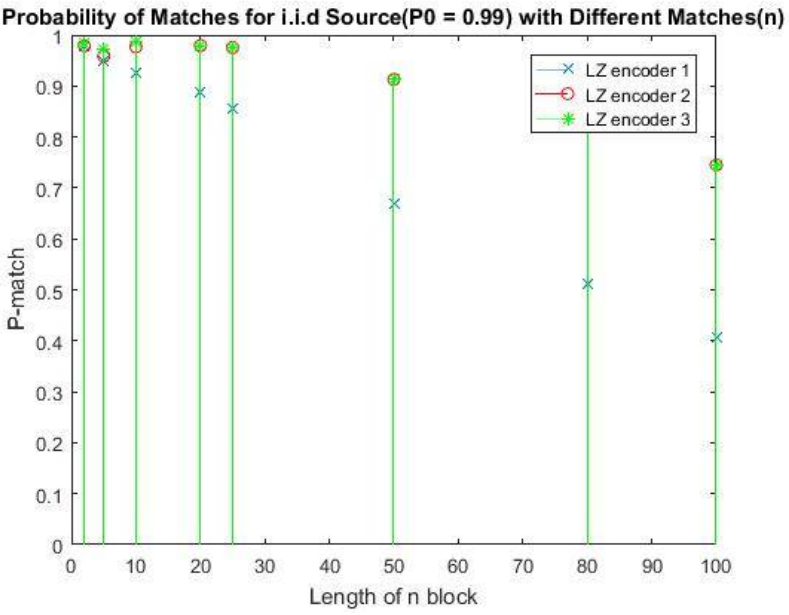


*Figure 5.1: P-match simulation for P0 = 99%, inputSize = 10000*

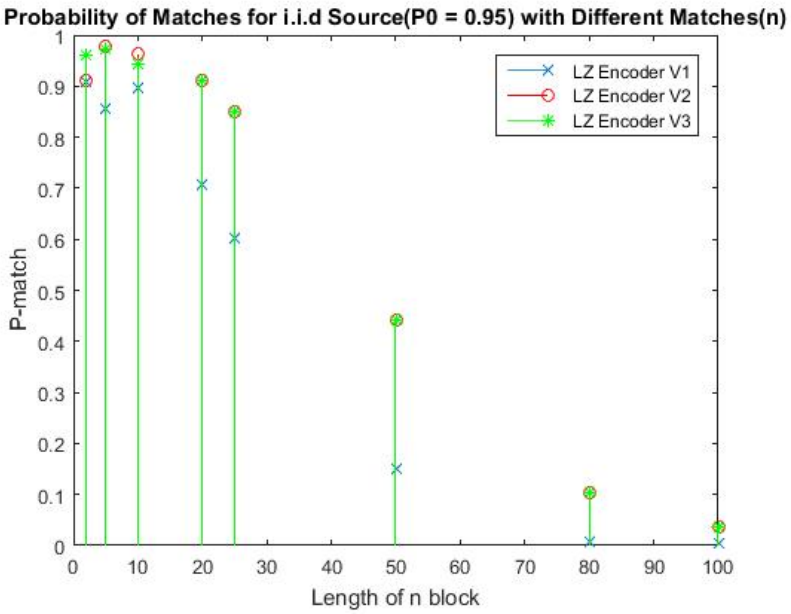


*Figure 5.2: P-match simulation for P0 = 95%, inputSize = 10000*

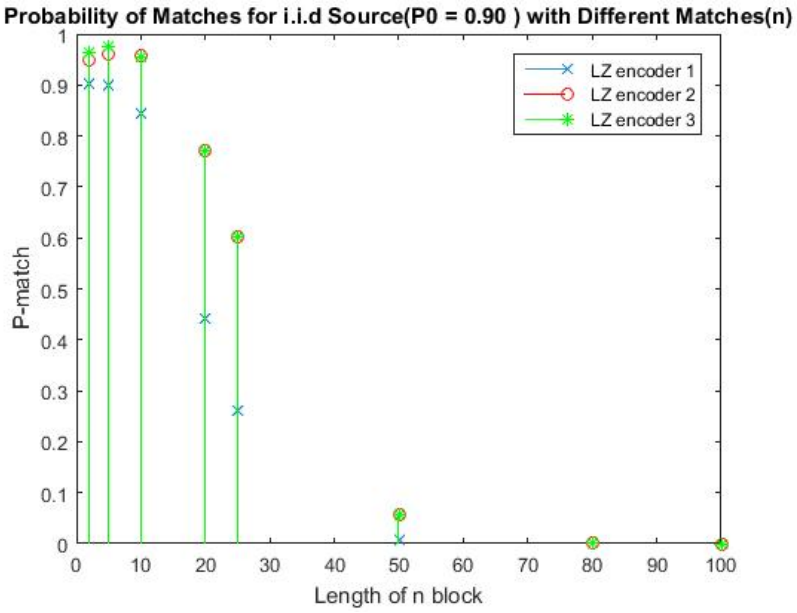


*Figure 5.3: P-match simulation for P0 = 90%, inputSize = 10000*

As can be observed from the graphs above, the value of P-match is high for lower values of N. This is an expected behavior as for small values of N, it is very likely to find the same block of binary string in an input file of 10000 bits. This value sharply decreases with increasing n, as expected, making it more unlikely to find the same block in the file. Another observation to note is that P-match is higher for higher $P_0$, i.e. when the source has higher bias towards 0. For example, when $P_0$ = 99%, P-match for n=80 and 100 are quite high, but are 0 when $P_0$ = 90%. Overall, version 2 and 3 of the algorithm give a much better performance than version 1, with version 3 giving a slightly better performance in certain scenarios. This is because in both versions 2 and 3, overlapping blocks of n are also searched from the window, for

example, blocks of 1…n and 2…n+1 are both searched in the input string, whereas version 1 only searches for consecutives blocks of n, i.e. 1…n and n+1….2n. Version 3 has a better performance due to the sliding window technique, where the window accommodates better for any changes and variations in the source.

The following graphs in **Figures 5.4, 5.5 and 5.6,** show the compression ratio obtained when the tests above were repeated to check for the compression efficiency of the algorithms.

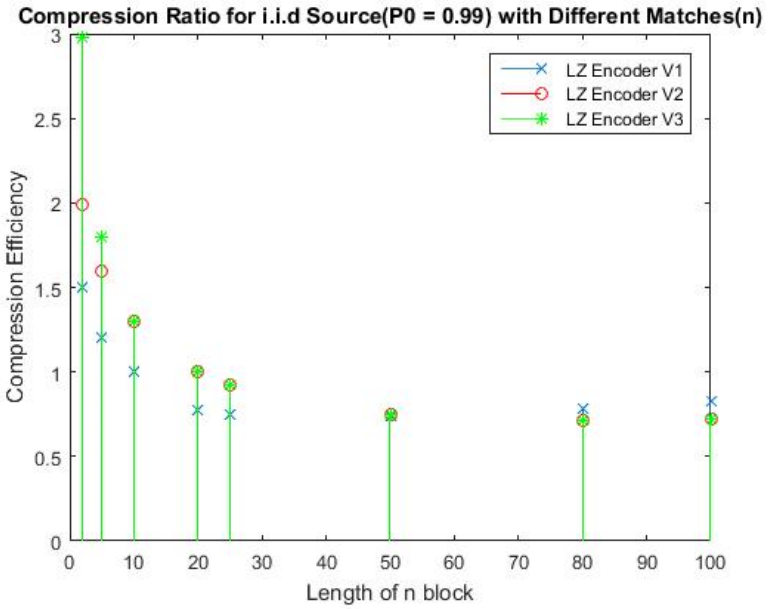


*Figure 5.4: Compression ratio simulation for P0 = 99%, inputSize = 10000*

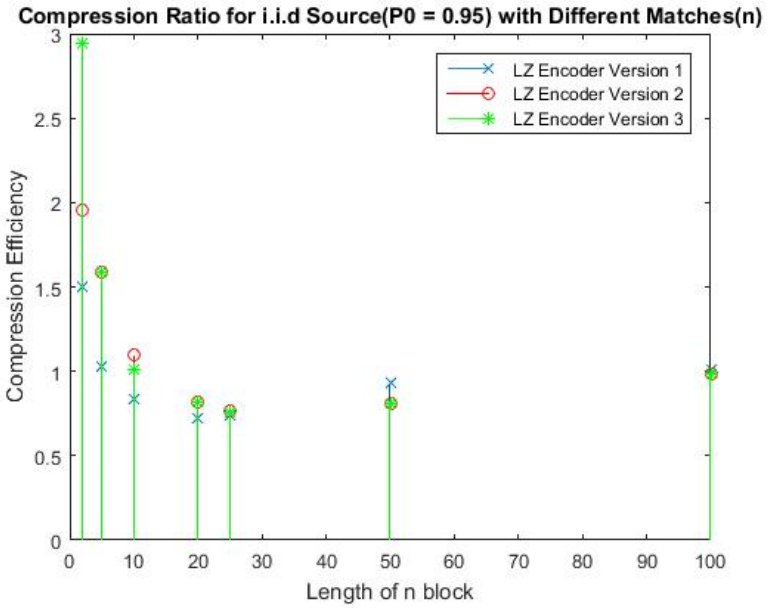


*Figure 5.5: Compression ratio simulation for P0 = 95%, inputSize = 10000*

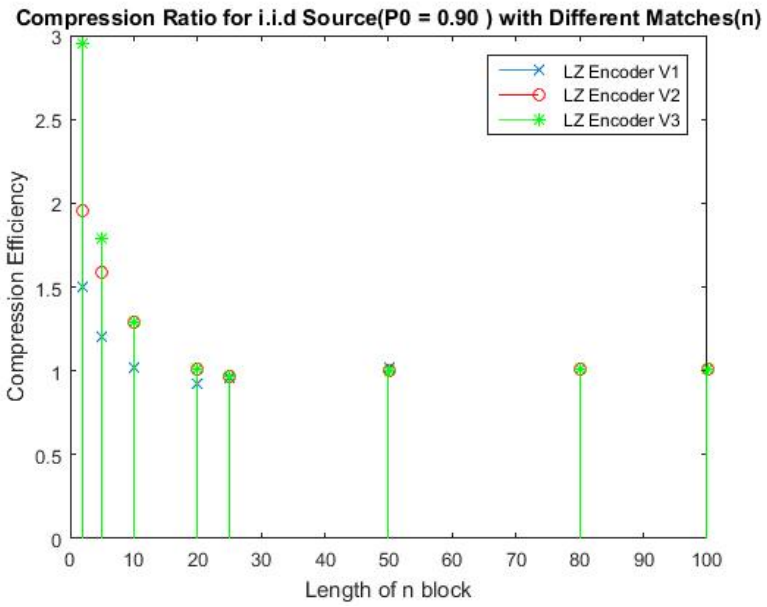


*Figure 5.6: Compression ratio simulation for P0 = 90%, inputSize = 10000*

As can be observed in all 3 graphs above, for lower values of n, compression is not achieved. This is because the block size n is too small and consequently the window is small as well, meaning large sequences which could possibly be matched, will not be matched in this case due to restrictions imposed by the parameters n and w. Another factor which contributes to such a high compression ratio is the fact that the pointers used to refer a matching block on the window happens to be bigger in size than n in these cases. For extremely high values of n such as 80 or 100, the chances of hitting a matching block are extremely low, as shown in the P-match graphs in **Figures 5.1, 5.2 and 5.3.** Therefore, the algorithm performs little to no compression in such cases. It can be noted from these plots that optimal compression is achieved when n is 20 or 25. However, the compression ratio is still quite high, which signifies the fact that the window is too small, to find enough chunks of n sized blocks for the compression to be optimal. A need for further simulations arose, to test the behavior of compression ratio against varying n as well as w, and see which sets of values give the most optimal results. This idea is further discussed in section **5.3.**

## 5.2. Markov Sources

The tests discussed in section 5.1 above were repeated for Markov sources as well. **Figures 5.7 and 5.8** show the P-match plots for Markov sources while **Figures 5.9 and 5.10** show the compression efficiency plots, using the same input size of 10000 bits. The former figures run at probability of zero at zero as 0.95, and probability of one at one as 0.70. The latter figures run at probability of zero at zero as 0.90, and probability of one at one as 0.80.

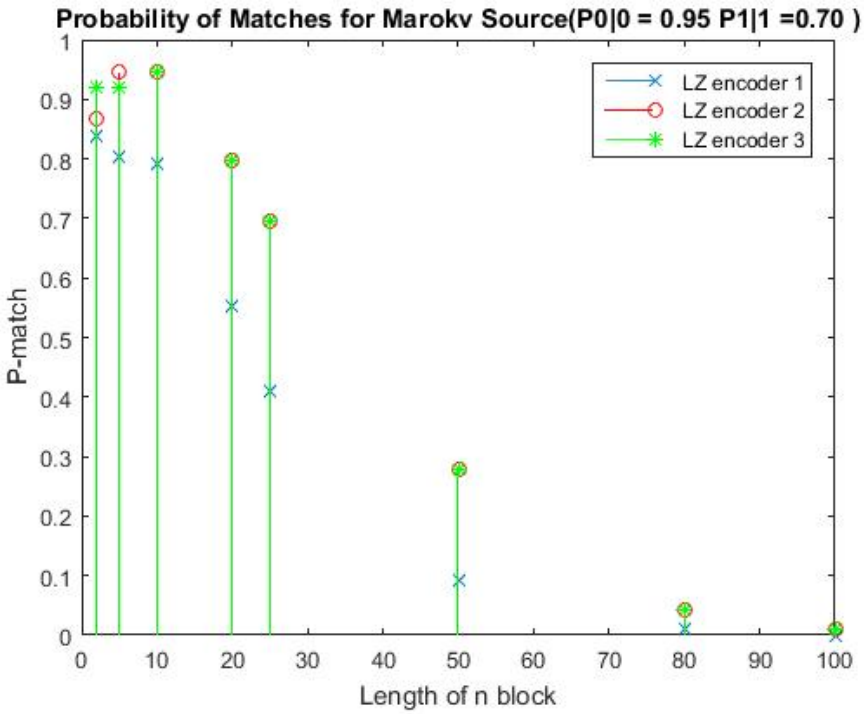


*Figure 5.7: P-match simulation, inputSize = 10000*

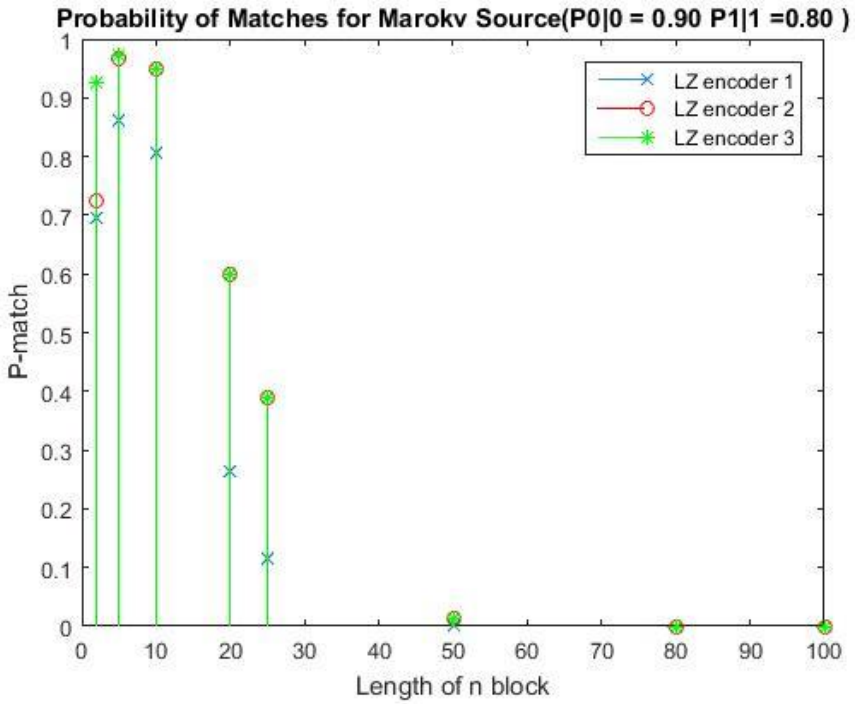


*Figure 5.8: P-match simulation, inputSize = 10000*

Observations similar to i.i.d sources were made for Markov as well, for the P-match plots. In this case however, version 3 of the encoder shows a much better P-match performance overall, due to sliding window technique which allows the algorithm to adjust its window according to variations in source statistics. Another point to note now is that for extremely high values of n, we have almost no match, since Markov sources are in clusters of 0's and 1's.

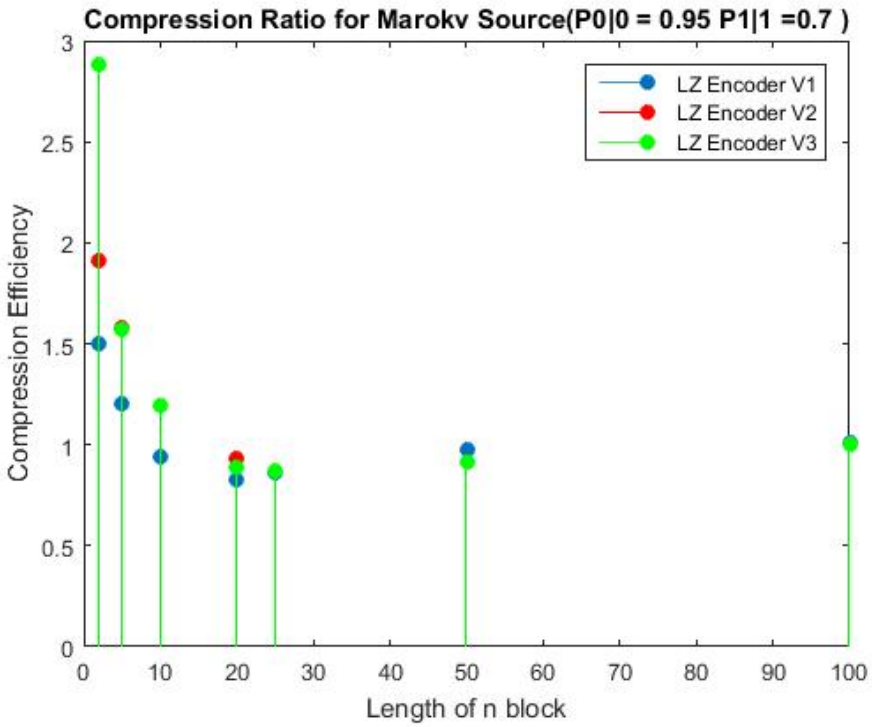


*Figure 5.9: Compression ratio simulation, inputSize = 10000*

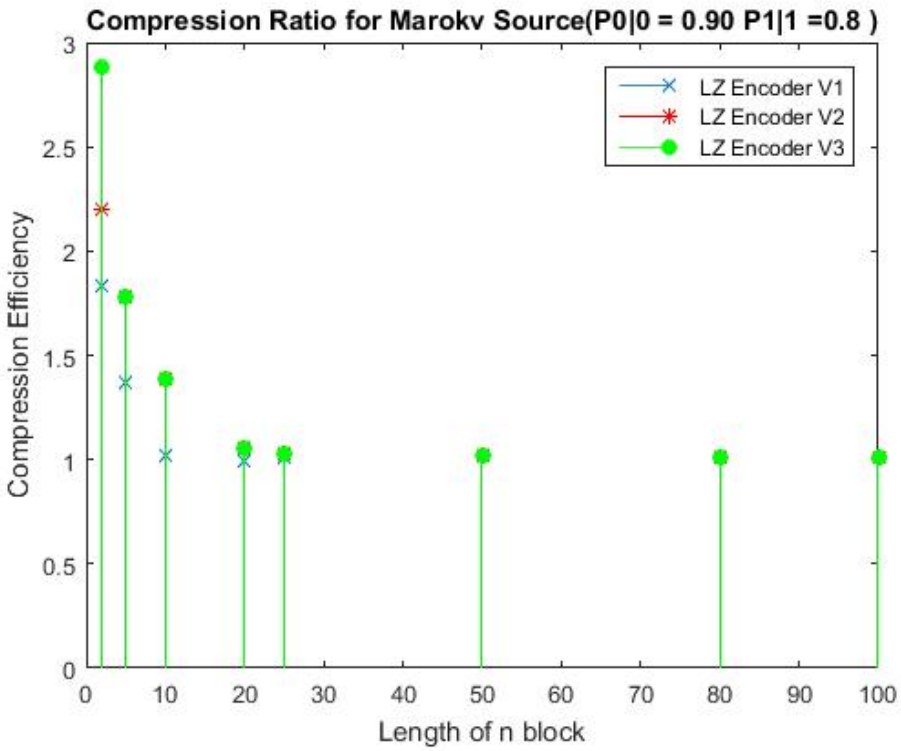


*Figure 5.10: Compression ratio simulation, inputSize = 10000*

The compression efficiency plots also show similar results to that obtained in simulating i.i.d sources. Version 3 however shows overall better performance, due to sliding window capabilities.

## 5.3. Optimization Simulations

As can be noted in sections 5.1 and 5.2, the values for block size n and window size w could be optimized to achieve better compression. In order to arrive at these optimized results, some exhaustive simulations were performed with varying values of n and w, and the compression ratio plotted at each variation in 3D stem plot. These simulations were executed for n varying from block size 3 to 200, and w varying from 2n to 20n in steps of n, for each n, giving a maximum size of 4000 for w. Each simulation was repeated 10 times for random i.i.d source with $P_o$ as 0.99, and the average values for compression ratio taken and plotted. This process was repeated for each encoder and a similar plot was also made for P-match. **Figures 5.11 and 5.12** show the optimized simulations of compression ratio and P-match respectively for encoder version 1. The results show that the minimum achievable compression is 0.262 at n=14 and w=28, while P-match is maximized at w=45 and n=3.

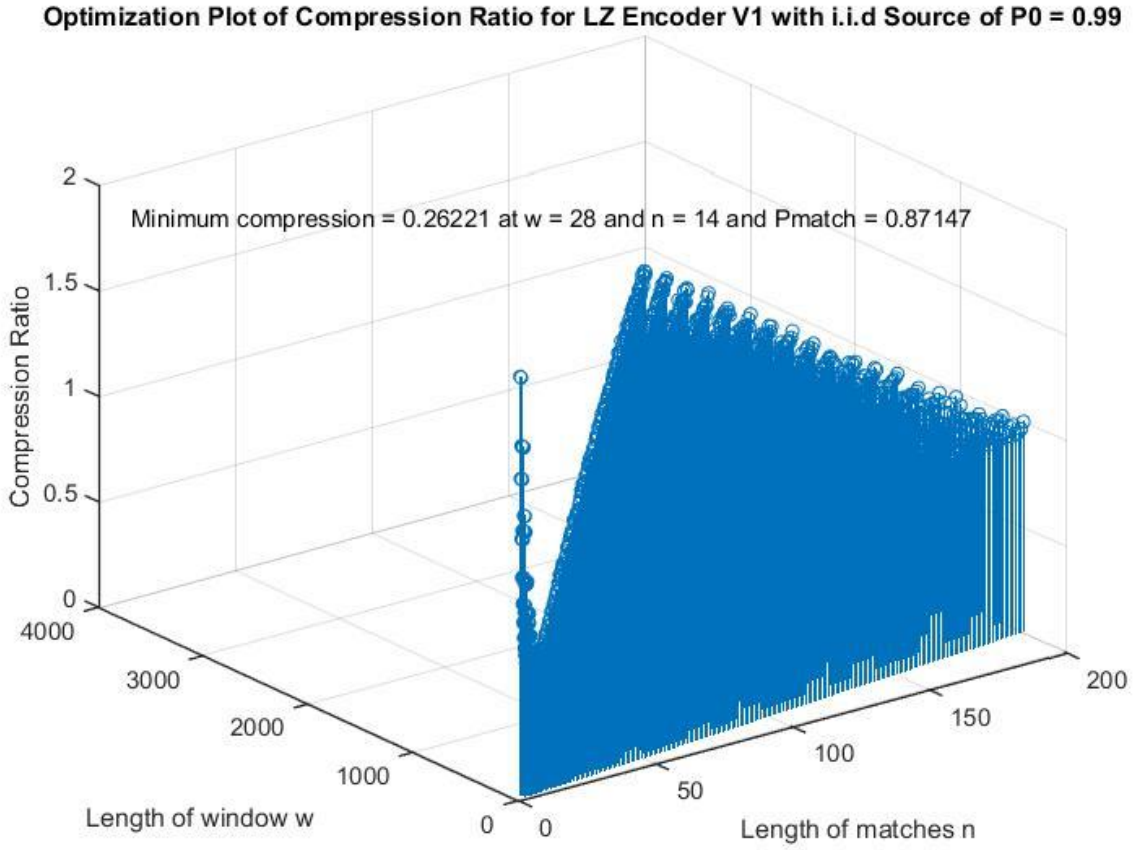


*Figure 5.11: Optimization plot for compression ratio, Encoder v1*

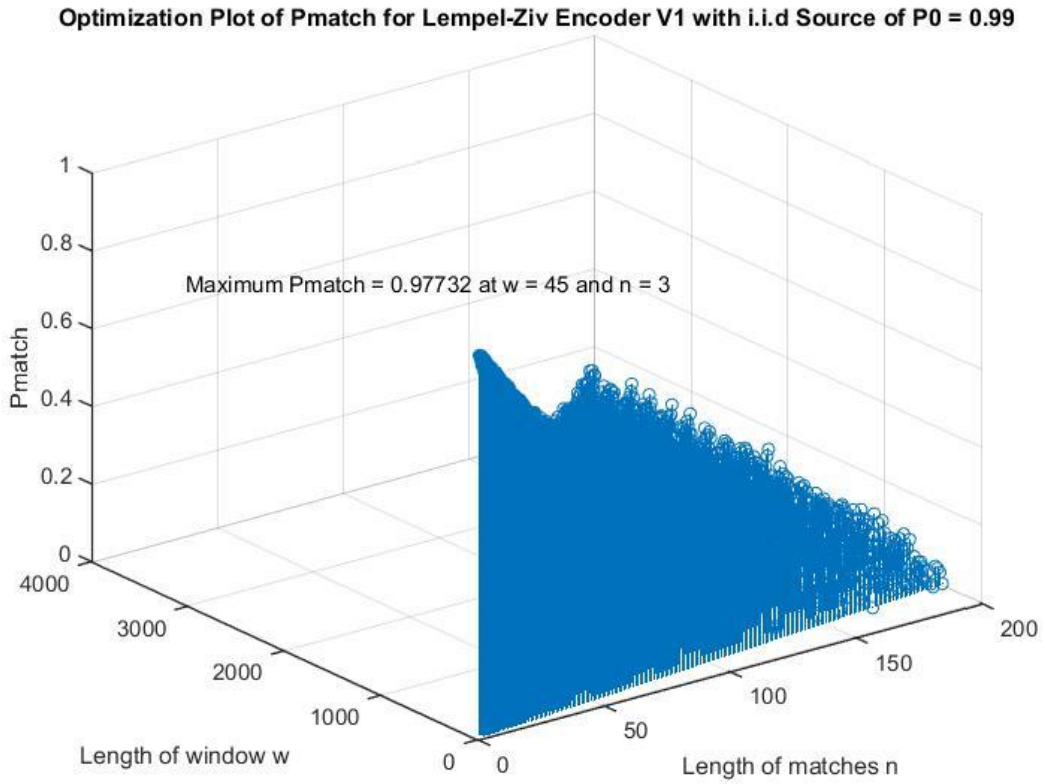


*Figure 5.12: Optimization plot for P-match, Encoder v1*

**Figures 5.13 and 5.14** show the optimized simulations of compression ratio and P-match respectively for encoder version 2. The results show that the minimum achievable compression is 0.267 at n=51 and w=408, while P-match is maximized at w=171 and n=9.

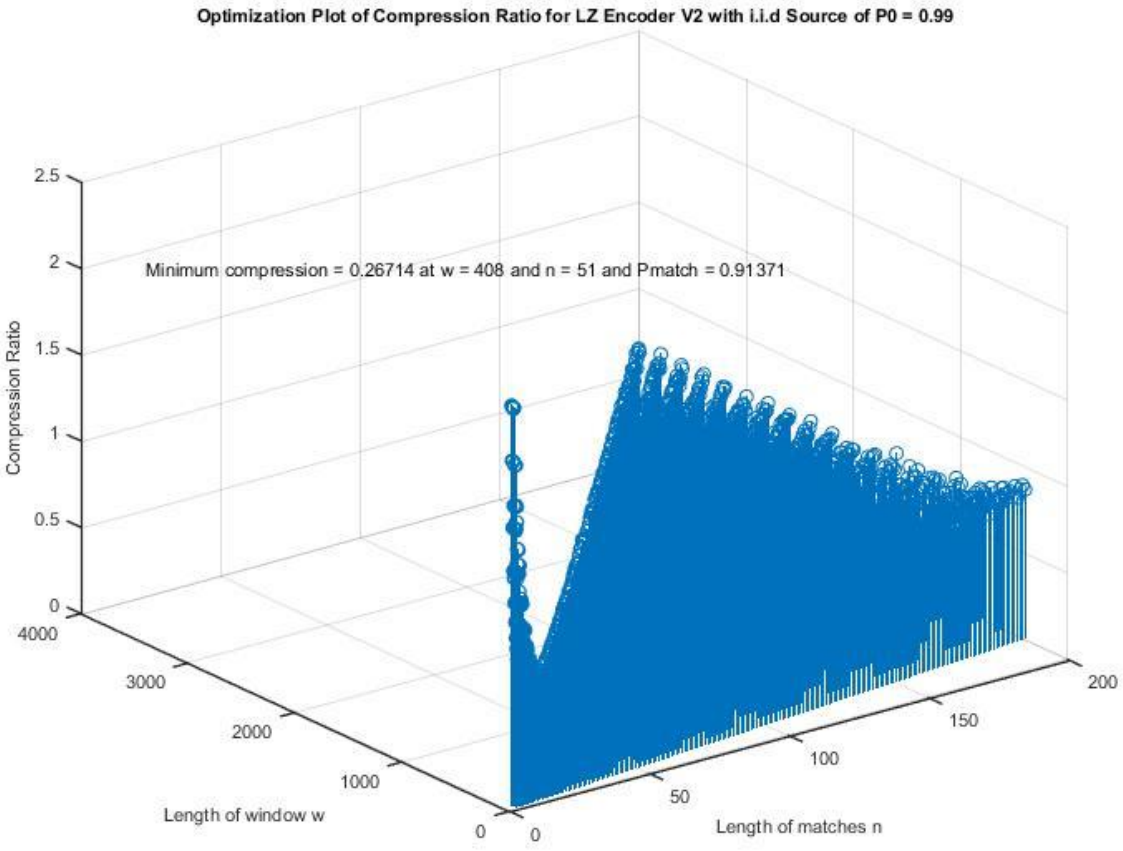


*Figure 5.13: Optimization plot for Compression Ratio, Encoder v2*

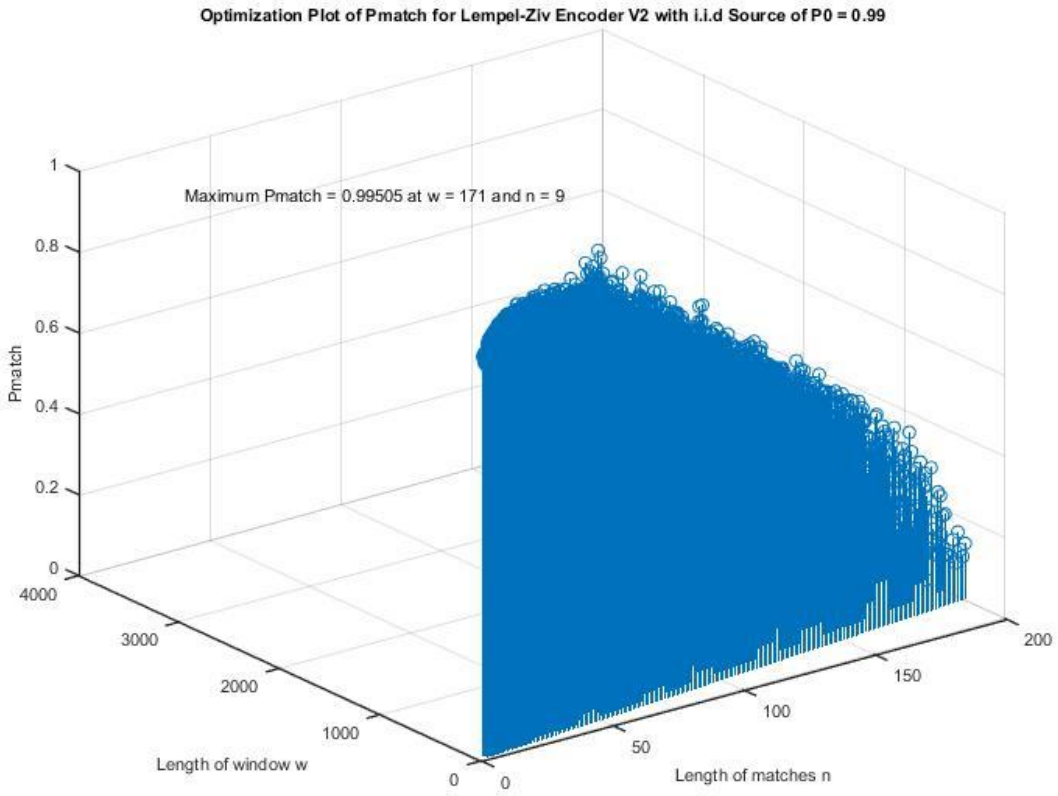


*Figure 5.14: Optimization plot for P-match, Encoder v2*

**Figures 5.15 and 5.16** show the optimized simulations of compression ratio and P-match respectively for encoder version 3. The results show that the minimum achievable compression is 0.272 at n=51 and w=408, while P-match is maximized at w=45 and n=3.

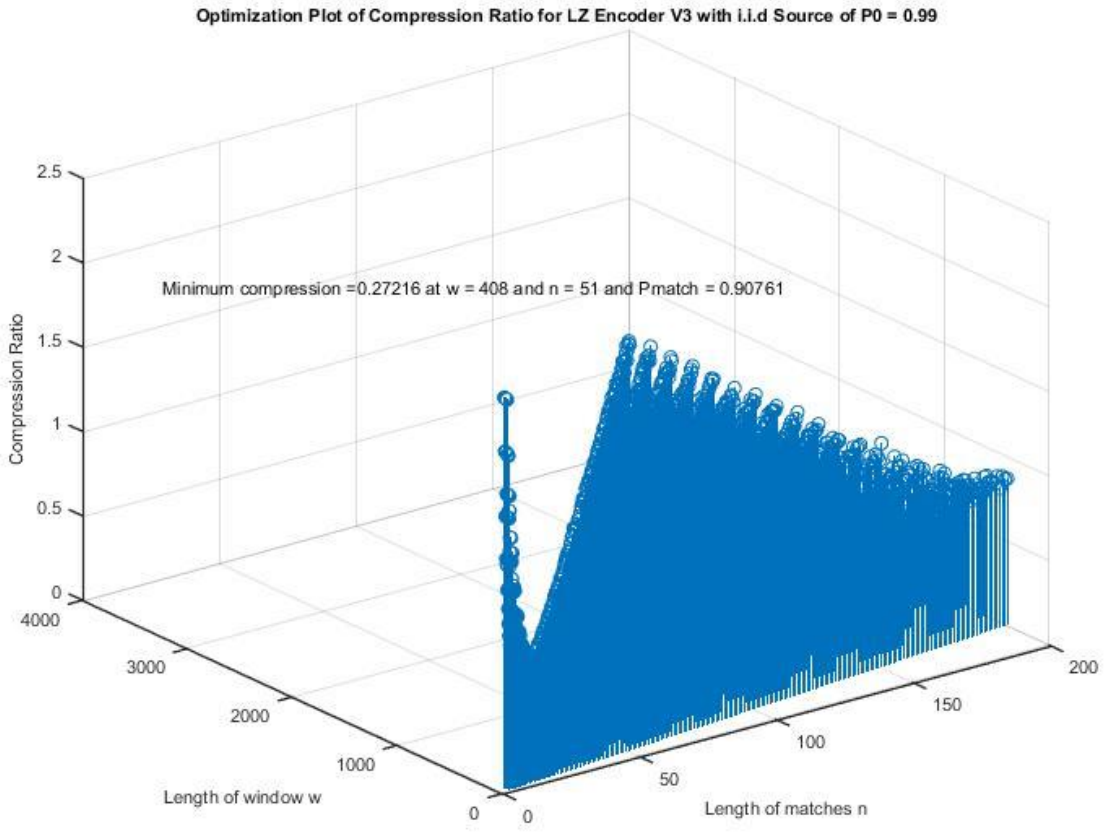


*Figure 5.15: Optimization plot for Compression Ratio, Encoder v3*

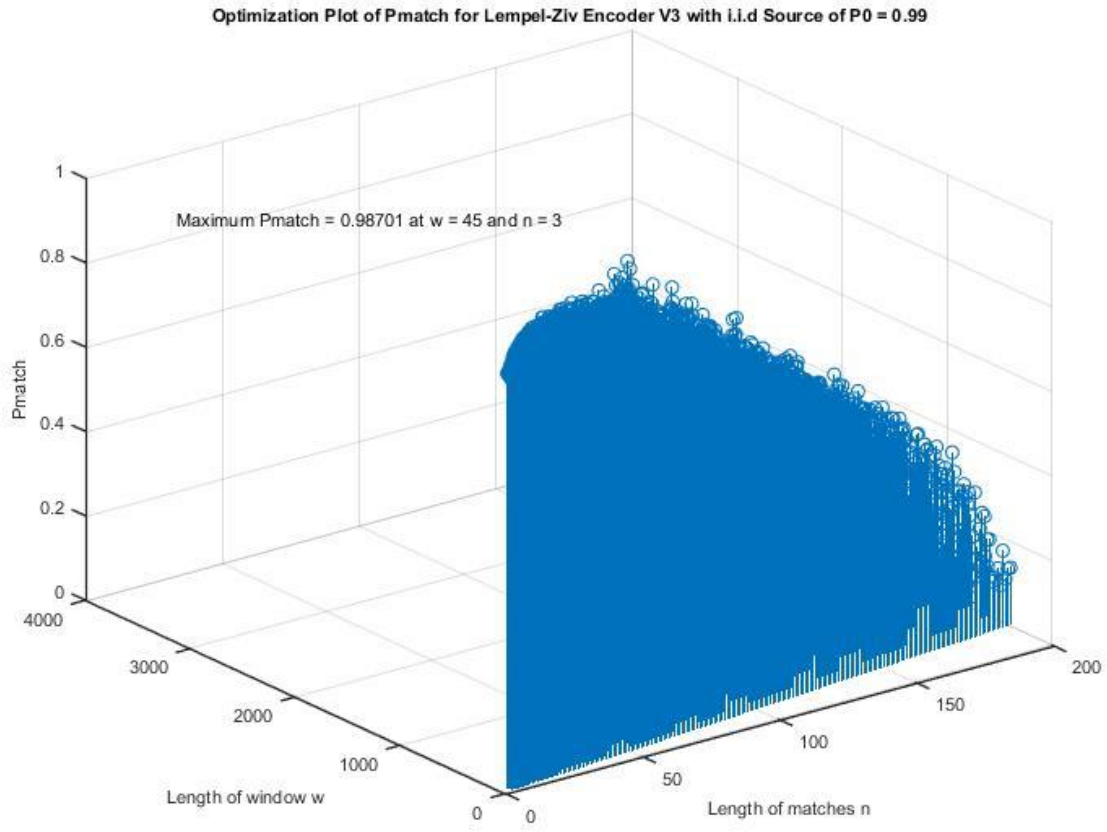


*Figure 5.16: Optimization plot for P-match, Encoder v3*

In plots for compression ratio and $P_{match}$ for all three encoders, the optimized n and w are different in compression ratio in comparison to the $P_{match}$ plot. This is observed due to the fact that $P_{match}$ is not the only parameter that optimizes the compression, very high $P_{match}$ might be a in encoders with very small value of n which in turn isn't the most optimal. The efficiency depends on these independent ($P_{match}$, n, and w) values given in **equation (13)** of the expected length:

$$E(L) = P_{match} \times \left(\frac{1+log2(w)}{n}\right) + P_{no\text{-}match} \times \left(\frac{1+nH(X)+2log2(n)}{n}\right) \quad \textbf{(13)}$$

Thus, these optimization results aim to find minimum compression ratio of each encoder that ideally optimizes the trade-off between the three parameters (w,n, and $P_{match}$).

## 5.4. Practical Source Files

The next step was to try all three algorithms in order to compress images and text documents. The results obtained from section 5.3 for values of n and w were used to achieve the optimal compression for each algorithm. **Figure 5.17** below shows the image used to test the compression algorithms.

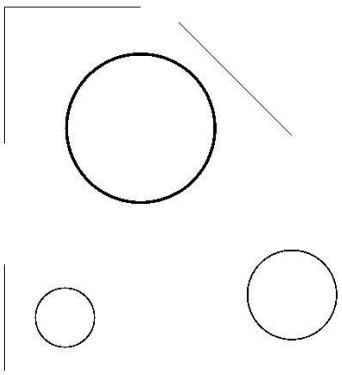

*Figure 5.17: Test image for compression*

It was observed that the compression ratio for encoder version 1 was 0.183 while encoders 2 and 3 compressed the file to 0.296 of its original size.

For the next test, a text document was read in from MATLAB and then passed through the encoders to check the performance. This test document was filled with the phrase “The quick brown fox jumps over the lazy dog”, repeated for 51 paragraphs with each paragraph containing 51 sentences, making a total of 27 pages. This document was generated using MS-Word filler text utility. The text was converted to a binary string, which was subsequently encoded and decoded. The decoded results were converted back to string and checked to ensure successful decoding. Encoder version 1 compressed this document to 0.647 of its original size while encoders version 2 and 3 compressed it to 0.710 and 0.726 of the original size, respectively.

# 6. Conclusion

Lempel-Ziv is a compression algorithm which is widely used in the industry today due to its versatility with varying source statistics. This made the study and implementation of this algorithm a viable as well as a challenging final project for this course. The following highlights the successful implementation and achievements made during this project:

- Successfully implemented three different versions of Lempel-Ziv encoders and decoders
- Simulated all three encoders under different sources as well as with varying parameters
- Successfully encode, compress and decode text and image files with all three algorithms

# 7. References


[1] P. Chanda, E. Elhaik and J. S. Bader, "HapZipper: sharing HapMap populations," *Nucleic Acids Research ,* vol. 40, no. 159, pp. 1-7, 2012.

[2] S. W, "Historical notes: History [of data compression]," 2002. [Online]. Available: https://www.wolframscience.com/reference/notes/1069b. [Accessed 28 03 2016].

[3] C. Kasem, M. Taha and M. H. Mann, "Report 1: Data Compression Algorithms for Information Sources," McGill University, Montreal, 2016.

[4] J. G. Proakis and M. Salehi, Communication Systems Engineering, New York: Pearson, 2001.

[5] R. G. Gallager, Principles of Digital Communication, New York : Cambridge University Press, 2008.

# 8. Appendix

```
function [encoded, P_match] = LempelZivEncoder1(input, n, w)
% This function applies LempelZiv encoding on the input. This is the first
% implementation version, so input window size is fixed along with block
% size. n indicates the block size, which will be compared. w sets the max
% window size
%% Begin encoding

% Intialize window to size w. Assumption is that the window would contain all
% possible combos in input?
window = [];

% result of encoding. Its a matrix, where size of each row is 1 + log2(W/n) . First bit
% is the flag, which is 1 if the block is a repition from before (exists in
% window) and 0 if its a new block symbol. The next bit indicates the
% relative position of the block in window
encoded = [];
pointer = [];
% value to slide our window with
windowSlide = 1;
%Probability that the symbol matches a word
P_match = 0;
% loop over input by incrementing in steps of n
for i = 1 : n : length(input)
    % current block is n bits from i.
    currentBlock = input(i:i+n-1);

    % this variable is true when the currentBlock is found in window
    blockFound = false;

    for j = 1 : n: length(encoded)
        % loop over encoded string and look if it exists in window
        if ((j+n-1 <= length(window)) & (currentBlock == window(j:j+n-1)))
            % current block found in dictionary. Stop looking any further
            % and store the result in output. Set flag to 1 and the
            % pointer in window at each n block of where the block exists
            pointer = decimalToBinary(ceil(j/n) - 1, ceil(log2(w/n)));
            encoded = [encoded, [1 pointer]];
            blockFound = true;
            P_match = P_match + 1;
            break;
        end
    end

    % case where block is not found in encoder. Set flag to 0 and the next
    % n bits to the currentBlock
    if (~blockFound)
        encoded = [encoded, [0 currentBlock]];
    end

    % add current block to window. Window max size has to be w, after that
    % it should slide. Window should be smaller than the input data
    if (i < w)
        window = [window, currentBlock];
    end
end
% Divide the number of matches by total number of symbols
P_match = P_match / (length(input)/n);
%% Encoding done! output should be encoded binary.
end
```

*Figure 8.1: LZ Encoder version 1 MATLAB code*

```
function decoded = LempelZivDecoder1(encoded, n, w)
% decode based on ecoded results created by
% LempelZivEncoder1
decoded = [];
pointerVector = [];
% pointer used for decoding
pointerIdx = 1;
% loop over encoded vector
while (pointerIdx < length(encoded))
    if (encoded(pointerIdx) == 0)
        % if the first bit is 0, then we know the next n bits are the
        % original data bits
        decoded = [decoded, encoded(pointerIdx+1:pointerIdx+n)];
        % in next iteration of the loop we should be looking 3 bits later
        pointerIdx = pointerIdx + n + 1;
    else
        % when the first bit of encoder is 1, the next bit is a pointer to
        % where that value can be retrieved from encoded stream. The encoded(pointerIdx+1)
        % value indicates where to go to fetch the n repeated bits
        pointerVector  = [pointerVector, encoded(pointerIdx+1:pointerIdx+ceil(log2(w/n)))];
        pointer = bi2de(pointerVector);
        % Bear in mind that the encoded string has 1 bit between each n
        % bits that could be matched. So we have to determine how many
        % blocks of n later our match exists. This is done by using pointer
        % times chunk sisze(n).
        decoded = [decoded, decoded(pointer*n+1:pointer*n+n)];
        % increase pointerIdx by 2. One increment is to skip over the pointer,
        % the next increment to go to the new flag in encoded stream
        pointerIdx = pointerIdx + ceil(log2(w/n)) + 1;
        pointerVector = [];
    end
end
end
```

*Figure 8.2: LZ Decoder version 1 MATLAB code*

```
function [encoded, P_match] = LempelZivEncoder2(input, n, w)
% This function applies LempelZiv version 2 encoding on the input.
% Input window size is fixed along with block size. n indicates the block size,
% which will be compared. In this version, we try to get a match of all
% block's of size n. The comparison is performed for every next block of
% size, i.e. block x1--xn, xn+1--x2n and so on
% w sets the max window size

%% Begin encoding

% Intialize window to size w. Assumption is that the window would contain all
% possible combos in input?
window = [];

% result of encoding. Its a matrix, where size of each row is 1 + log2(w) . First bit
% is the flag, which is 1 if the block is a repition from before (exists in
% window) and 0 if its a new block symbol. The next bit indicates the
% relative position of the block in window
encoded = [];
pointer = [];
% value to slide our window with
windowSlide = 1;
%Probability that the symbol matches a word
P_match = 0;
% loop over input by incrementing in steps of n
for i = 1 : n : length(input)
    % current block is n bits from i.
    currentBlock = input(i:i+n-1);

    % this variable is true when the currentBlock is found in window
    blockFound = false;
    % loop over encoded string and look if it exists in window
    for k = 1 : length(window)
        % loop over window and search for the block in window
        if ((k+n-1 <= length(window)) & (currentBlock == window(k:k+n-1)))% & length(window) >= w)
            %ceil(k) - 1
            pointer = decimalToBinary(ceil(k) - 1, ceil(log2(w)));
            encoded = [encoded, [1 pointer]];
            blockFound = true;
            P_match = P_match + 1;
            break;
        end
    end


    % case where block is not found in encoder. Set flag to 0 and the next
    % n bits to the currentBlock
    if (~blockFound)
        encoded = [encoded, [0 currentBlock]];
    end

    % add current block to window. Window max size has to be w, after that
    % it should slide. Window should be smaller than the input data
    if (i < w)
        window = [window, currentBlock];
    end
end
%% Divide the number of matches by total number of symbols
P_match = P_match / (length(input)/n);
%% Encoding done! output should be encoded binary.
end
```

*Figure 8.3: LZ Encoder version 2 MATLAB code*

```
function decoded = LempelZivDecoder2(encoded, n, w)
% decode based on ecoded results created by
% LempelZivEncoder1
decoded = [];
pointerVector = [];
% pointer used for decoding
pointerIdx = 1;
% loop over encoded vector
while (pointerIdx < length(encoded))
    if (encoded(pointerIdx) == 0)
        % if the first bit is 0, then we know the next n bits are the
        % original data bits
        decoded = [decoded, encoded(pointerIdx+1:pointerIdx+n)];
        % in next iteration of the loop we should be looking 3 bits later
        pointerIdx = pointerIdx + n + 1;
    else
        % when the first bit of encoder is 1, the next bit is a pointer to
        % where that value can be retrieved from encoded stream. The encoded(pointerIdx+1)
        % value indicates where to go to fetch the n repeated bits
        pointerVector  = [pointerVector, encoded(pointerIdx+1:pointerIdx+ceil(log2(w)))];
        pointer = bi2de(pointerVector);
        % Bear in mind that the encoded string has 1 bit between each n
        % bits that could be matched. So we have to determine how many
        % blocks of n later our match exists. This is done by using pointer
        % times chunk sisze(n).
        decoded = [decoded, decoded(pointer+1:pointer+n)];
        % increase pointerIdx by 2. One increment is to skip over the pointer,
        % the next increment to go to the new flag in encoded stream
        pointerIdx = pointerIdx + ceil(log2(w)) + 1;
        pointerVector = [];
    end
end
```

*Figure 8.4: LZ Decoder version 2 MATLAB code*

```
function [encoded,P_match] = LempelZivEncoder3(input, n, w)
% This function applies LempelZiv version 2 encoding on the input.
% Input window size is fixed along with block size. n indicates the block size,
% which will be compared. In this version, we try to get a match of all
% block's of size n. The comparison is performed for every next block of
% size, i.e. block x1--xn, xn+1--x2n and so on
% w sets the max window size

%% Begin encoding

% Intialize window to size w. Assumption is that the window would contain all
% possible combos in input?
window = [];

% result of encoding. Its a matrix, where size of each row is 1 + log2(w) . First bit
% is the flag, which is 1 if the block is a repition from before (exists in
% window) and 0 if its a new block symbol. The next bit indicates the
% relative position of the block in window
encoded = [];
pointer = [];
% value to slide our window with
windowSlide = 1;
%Probability that the symbol matches a word
P_match = 0;
% loop over input by incrementing in steps of n
for i = 1 : n : length(input)
    % current block is n bits from i.
    currentBlock = input(i:i+n-1);

    % this variable is true when the currentBlock is found in window
    blockFound = false;
    for k = 1 : length(window)
        % loop over window and search for the block in window
        if ((k+n-1 <= length(window)) & (currentBlock == window(k:k+n-1)))
            %ceil(k) - 1
            pointer = decimalToBinary(ceil(k) - 1, ceil(log2(w)));
            encoded = [encoded, [1 pointer]];
            blockFound = true;
            P_match = P_match + 1;
            break;
        end
    end


    % case where block is not found in encoder. Set flag to 0 and the next
    % n bits to the currentBlock
    if (~blockFound)
        encoded = [encoded, [0 currentBlock]];
    end

    % add current block to window. Window max size has to be w, after that
    % it should slide. Window should be smaller than the input data
    if (i < w)
        window = [window, currentBlock];
    else
        % slide the window
              window = input(1+windowSlide:w+windowSlide);
              windowSlide = windowSlide + 1;
    end
end
%% Divide the number of matches by total number of symbols
P_match = P_match / (length(input)/n);
%% Encoding done! output should be encoded binary.
end
```

*Figure 8.5: LZ Encoder version 3 MATLAB code*

```
function decoded = LempelZivDecoder3(encoded, n, w)
% decode based on ecoded results created by
% LempelZivEncoder1
decoded = [];
pointerVector = [];
% pointer used for decoding
pointerIdx = 1;
windowSlide = 0;
windowIdx = 0;
% loop over encoded vector
while (pointerIdx < length(encoded))
    if (encoded(pointerIdx) == 0)
        % if the first bit is 0, then we know the next n bits are the
        % original data bits
        decoded = [decoded, encoded(pointerIdx+1:pointerIdx+n)];
        % in next iteration of the loop we should be looking 3 bits later
        pointerIdx = pointerIdx + n + 1;
    else
        % when the first bit of encoder is 1, the next bit is a pointer to
        % where that value can be retrieved from encoded stream. The encoded(pointerIdx+1)
        % value indicates where to go to fetch the n repeated bits
        pointerVector  = [pointerVector, encoded(pointerIdx + 1:pointerIdx+ceil(log2(w)))];
        pointer = bi2de(pointerVector);
        % Bear in mind that the encoded string has 1 bit between each n
        % bits that could be matched. So we have to determine how many
        % blocks of n later our match exists. This is done by using pointer
        % times chunk sisze(n).
        decoded = [decoded, decoded(pointer+1 +windowSlide:pointer+n + windowSlide)];
        % increase pointerIdx by 2. One increment is to skip over the pointer,
        % the next increment to go to the new flag in encoded stream
        pointerIdx = pointerIdx + ceil(log2(w)) + 1;
        pointerVector = [];
    end
    windowIdx = windowIdx + n;
    if(windowIdx > w )
        windowSlide = windowSlide + 1;
    end
end
end
```

*Figure 8.6: LZ Decoder version 3 MATLAB code*