Final Project report

Louis Mennrath

November 2024

Contents

1 Introduction

In that report, we aim to implement a **polar code** encoder and decoder to compute the Bit error rate and the Word error rate for different block length and for rate 1/2.

Polar codes are *error correcting linear block codes* that were developed by the Turkish researcher **Erdal Arikan** in 2009. The main principle of polar codes is that there transform the physical channel into many different virtual channels. The virtual channel number increase with the block length of the code and those channel are either considered as very bad or very low, they are *polarized*.

Polar codes are known to be the firsts explicit constructions to achieve **capacity channel** for the binary memoryless symmetric channels. Because of their capabilities and their modest encoding and decoding complexity ($O(n \log n)$), the 3GPP agreed in 2016 to use polar code in the **5G standards** for the control channel and therefore, developing the research on this field.

2 Polar codes

In this report the way we will describe and construct the polar code will be slightly different than the way they are in the *course* [1]. We will base our study on the how the polar codes are constructed in the *LDPC and Polar codes for 5G standards* [2] online course by NPTEL available on YouTube.

2.1 Polar transform $\mathbf{G}_2$

The polar transform of rank 2 is an operation that maps 2 bits to 2 bits and that can be represented by the matrix $\mathbf{G}_2$:

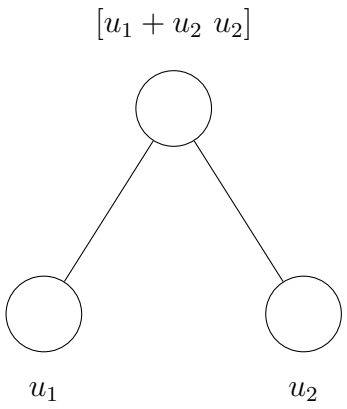
$$\mathbf{G}_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

So we have:

$$\begin{bmatrix} u_1 & u_2 \end{bmatrix} \cdot \mathbf{G}_2 = \begin{bmatrix} u_1 + u_2 & u_2 \end{bmatrix}$$

2.2 Binary tree representation

We can represent this polar transform of rank 2 by a binary tree as follow:



Where you combine the value of the children nodes to the parent node following the rule : [left_child + right_child; right_child] using addition modulo 2.

2.3 Domain extension

The polar transform and its tree representation could be extended to any n dimension where 2^n bits are mapped to 2^n bits.

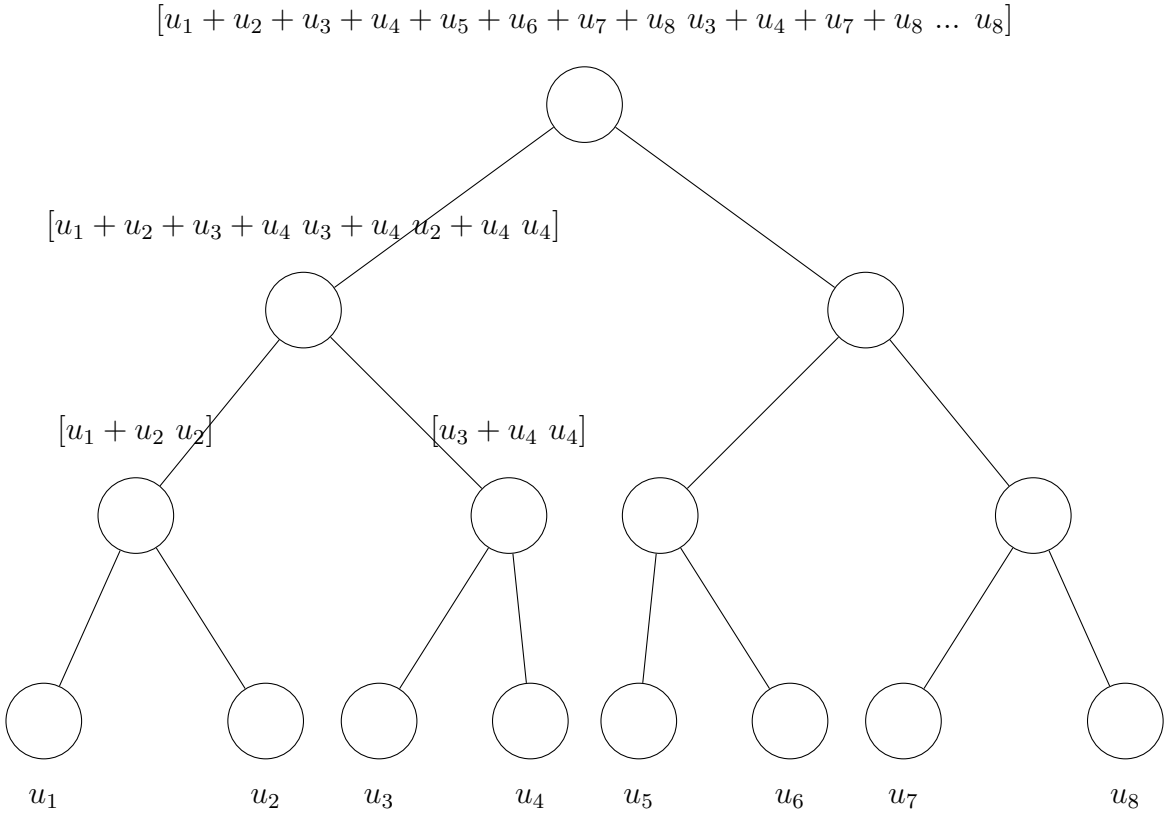
We obtain $\mathbf{G}_{2^n}$ by doing the **Kronecker product** of $\mathbf{G}_2$ by $\mathbf{G}_2$ n times:

$$\mathbf{G}_{2^n} = \mathbf{G}_2^{\otimes n} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{\otimes n}$$

And for the tree representation we just reproduce the 2 bit scheme. Here is the representation for $n = 3$ where we handle 8 bits long sequences:

$$\mathbf{G}_8 = \mathbf{G}_2^{\otimes 3} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

And the tree representation is:


$[u_1 + u_2 + u_3 + u_4 + u_5 + u_6 + u_7 + u_8 \; u_3 + u_4 + u_7 + u_8 \; ... \; u_8]$
$[u_1 + u_2 + u_3 + u_4 \; u_3 + u_4 \; u_2 + u_4 \; u_4]$
$[u_1 + u_2 \; u_2]$
$[u_3 + u_4 \; u_4]$
u_1
u_2
u_3
u_4
u_5
u_6
u_7
u_8

3 Bit channel and polarization

In this section we will explain how the polar code define a new way of considering the channel and to use virtual channels to correct errors.

3.1 Channel splitting

When we transmit a u vector over a channel (here AWGN) we obtain a receive vector $r^{(N)}$ of N bits according to the figure 1. In the traditional way of decoding we will decode $r^{(N)}$ bit by bit as if each bit was transmitted across a single AWGN channel. This decoding is faithful to what happen in the reality but polar decoding comes with a new way of representing the transmission.

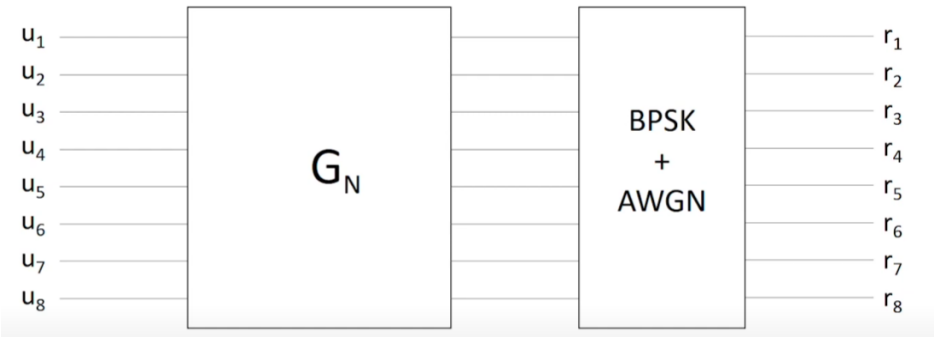


Figure 1: Transmission over AWGN channel

Polar decoding represent the transmission differently, the real channel is separated in N different virtual channels as represented in the Figure 2, where each channel is composed of the received vector and of all the previous bits already decoded.

3.2 Channel reliability

In that way, the output represented are not real outputs, but only a representation that we will use to perform decoding. The reliability of the different channels could vary and can be computed and ordered by reliability. As you can see an example for N=8 3 we can order the reliability of the channel for greater values of N and the reliability of each channel tend to be either really good either really bad when N is increasing, this is the phenomenon of polarization.

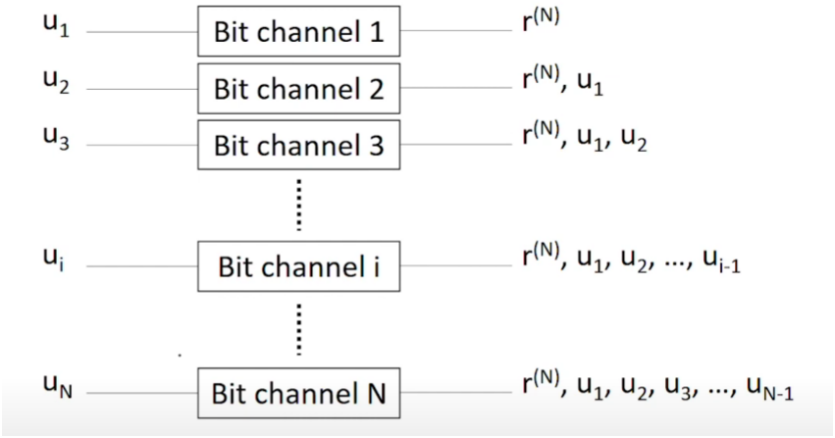


Figure 2: Virtual channel splitting

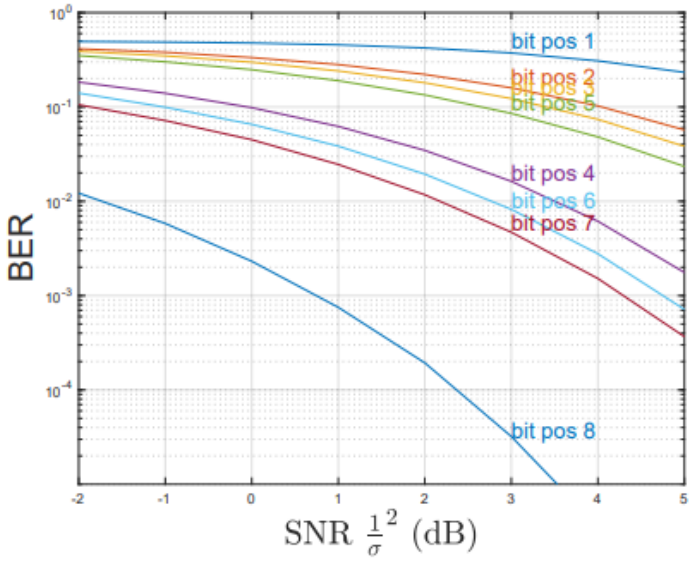


Figure 3: BER for each channel, *from Lecture Notes [1]*

You can see an the reliability of each channel is either really close to capacity or either really far for example if we take N = 1024 which is the size of the sequences used in 5G standards 4

3.3 Reliability sequence

From the previous definition we can order the bit channels by reliability and creating a reliability sequence. There is different ways to compute those relia-

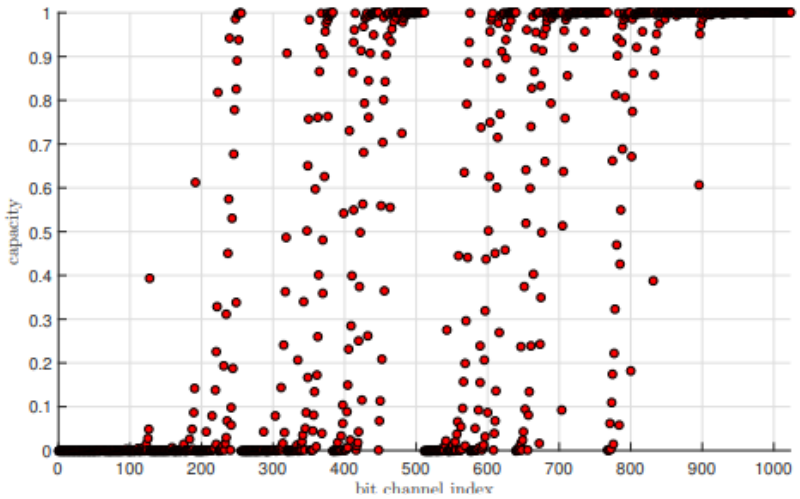
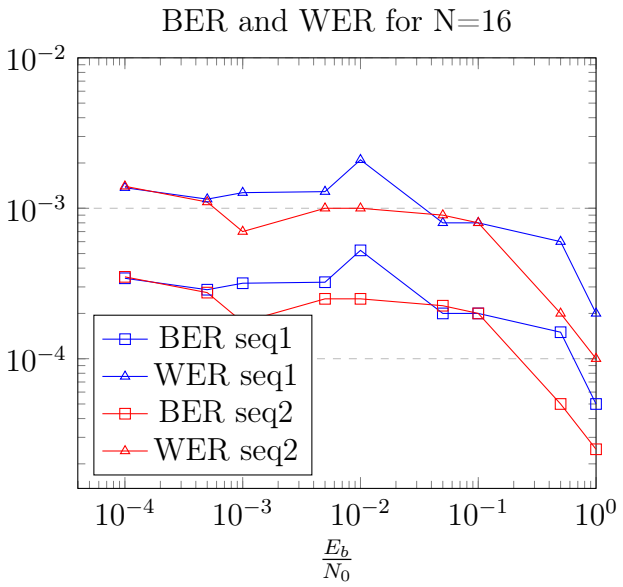


Figure 4: N =1024, *from IEE* [3]

bility sequence. We can either run tests on the physical channel to determine individually the reliability of each virtual channel but we can also calculate the reliability sequence from the parameters of the channel. An other choice is to take a general reliability sequence because the use of a reliability sequence specific to a channel is only needed for the really high performance optimization.

Here is the comparison between the general reliability sequence from the online course and one given by Arikan use in [4] for N=16:



The 2 sequences produces a very similar result so we will take only one reliability sequence and assume that the result and the conclusion we will take are not influenced by the choice of the reliability sequence.

In our case we use the reliability sequence given in the online course [2] for N = 1024.

4 The (N,K) Polar code

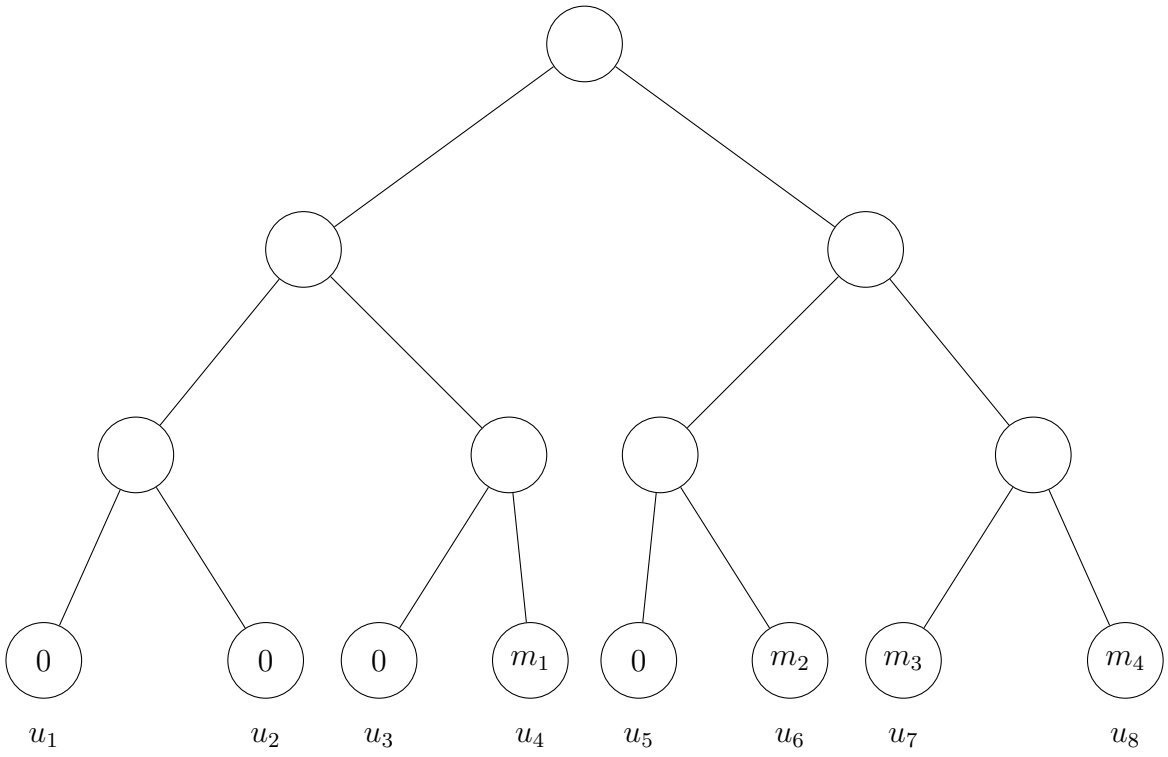
The (N,K) polar code refers to a polar code of a block length of N bits and passing a message of K bits.

4.1 Frozen bits

When performing polar encoding and decoding we want to ensure the best integrity possible for our message so we will use the most reliable bit channels to convey the message. For that we will select the K most reliable bits to put the message and we will take the N-K least reliable bits and initialize theme to 0. Those are called the **frozen bits**. Let's take an example with the (8,4) polar code:

The reliability sequence is : 1, 2, 3, 5, 4, 6, 7, 8 (cf. Figure 3)

The message contains $K = 4$ bits ($m = \begin{bmatrix} m_1 & m_2 & m_3 & m_4 \end{bmatrix}$) and the block length is equal to $N = 8$, so we initialize the N-K less reliable bits channels to 0 and we transmit the message over the K most reliable virtual channels:



After the initialization step we obtain the vector $u = \begin{bmatrix} u_1 & u_2 & \cdots & u_N \end{bmatrix}$ that will be use to perform the encoding.

4.2 Encoding

To perform the encoding we can use 2 different techniques:

- The matrix multiplication
- The tree algorithm

4.2.1 Matrix multiplication encoding

As we seen in the Section 2.3, the polar transform of a vector u of dimension N could be obtained by the multiplication by the matrix $\mathbf{G}_N$ and we obtain the codeword:

$$c_{word} = u \cdot \mathbf{G}_N$$

with $u = \begin{bmatrix} u_1 & u_2 & \cdots & u_N \end{bmatrix}$, $N = 2^n$ and $\mathbf{G}_{2^n} = \mathbf{G}_2^{\otimes n} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}^{\otimes n}$

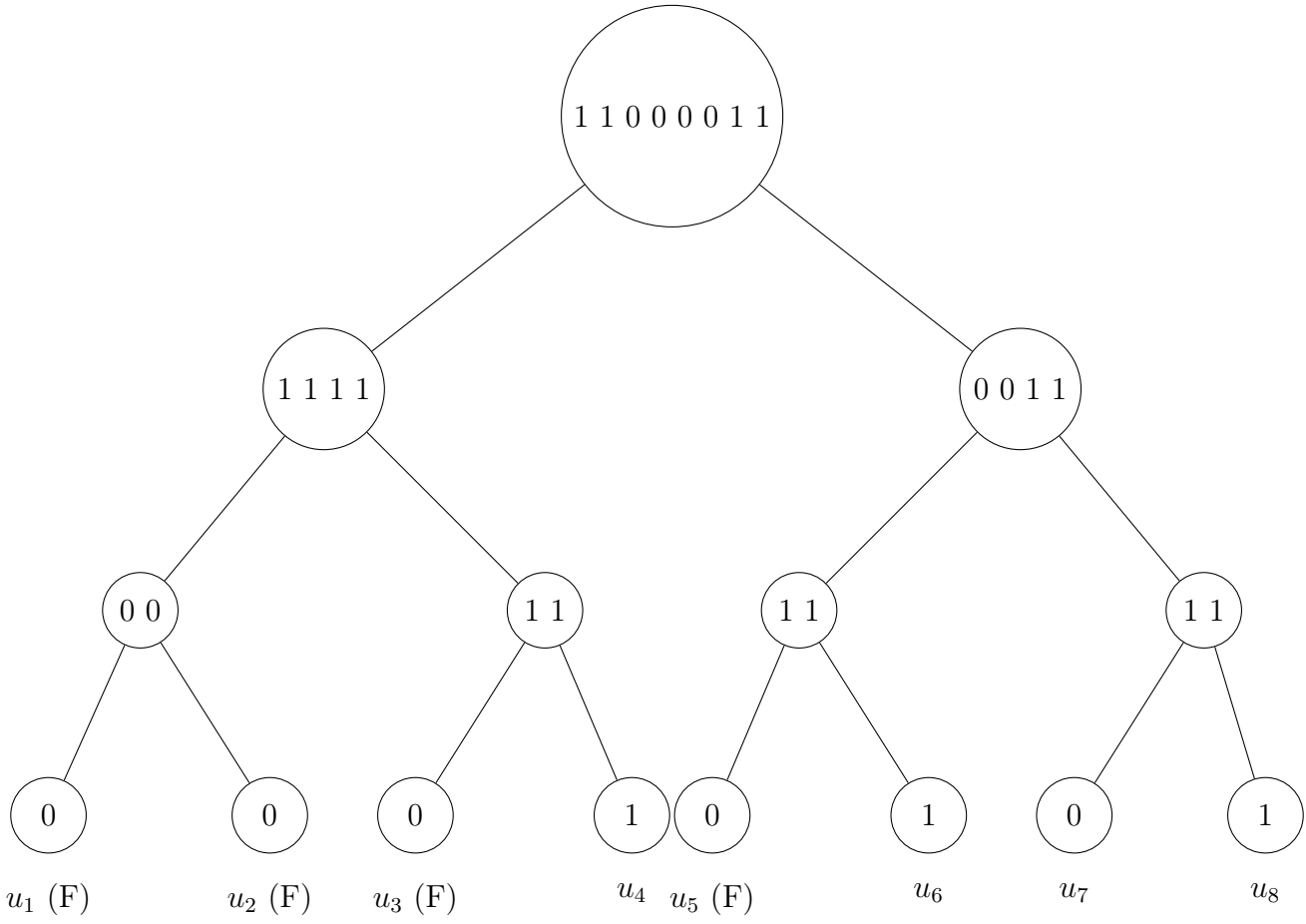
But the need to define $\mathbf{G}_N$ at each encoding and perform the multiplication is not very efficient when we need to scale it to larger values of N.

4.2.2 Tree encoding

Since the beginning of the report we refer to the tree representation and it is the one we will use because it is easy to scale it up. The base of this encoder is the binary tree of dimension 2 describe in the section 2.2, we just repeat the operations at every node of the tree from the leaves to the root.

Let's continue our example with the polar code (8,4). After initializing the leaves nodes, we go to the upper level following the decision rule: $parent_{node} = \begin{bmatrix} child_{left} + child_{right} & child_{right} \end{bmatrix}$

and consider that $m = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$



Step by step we execute the compilation from the leaves node to the root, from left to right. We can remark that at each depth of the tree 8 bits are processed to go from the initialized message $u = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$ to the encoded message $c_{word} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

The complexity of this encoding is in $O(n \log_2(n))$ compared to the matrix multiplication that is of $O(n^2)$, that will make a big difference for the simulation of long block codes (i.e. N=256).

4.2.3 Encoder implementation

I won't describe the code in detail because it is the implementation of the tree algorithm presented in the previous section. But if you want a step by step analysis of the construction of the code you can watch the course video

available on YouTube [2] (the implementation is in Matlab and I translated in python to perform multi thread simulations).

```
def encode(u, N):
    # Define the depth of the tree. We begin at the bottom
    d = int(np.log2(N))
    # nb is the  number of bit to combine at each step
    nb = 1
    while(d>0):

        for i in range(0,N,2*nb):
            left = u[i:i+nb]
            right = u[i+nb:i+2*nb]
            for j in range(nb):
                u[j+i] = node_sum(left, right, nb)[j]
        d -=1
        nb *=2
    return u
```

Listing 1: (N,K) polar code encoder

We store the successive values of each node one the same vector u. To insert properly new values we can define a new function:

```
def node_sum(left, right, l):
    result = np.zeros(l, dtype=int)
    for i in range(l):
        result[i] = np.mod(right[i] + left[i],2)
    return result
```

Listing 2: function $node\_sum()$

5 Transmission simulation

We will simulate the transmission over a AWGN channel and to improve the reliability over noise we add a BPSK modulation.

5.1 AWGN channel parameters

We will transmit the signal across different level of noise on the channel to test the robustness off the code over noise. We will focus our study on **1/2 rate** codes to compare them easily to the most of the other codes studied in the course [1].

For different signal-to-noise ratios $\frac{E_b}{N_0}$ we can find the value of σ to generate the Gaussian noise with the formula:

$$\frac{E_b}{N_0} = \frac{1}{2 * Rate * \sigma^2}$$

So $\sigma = \sqrt{\frac{1}{E_b/N_0}}$ and so we can generate the noise for different values of $\frac{E_b}{N_0}$ after converting the dB values into linear.

5.2 BPSK modulation

To improve noise reliability, we use BPSK modulation and map the bit values as follows:

$$BPSK = \begin{cases} 0 & \longrightarrow & 1 \\ 1 & \longrightarrow & -1 \end{cases}$$

5.3 Transmission implementation

The transmission over AWGN channel and the BPSK modulation are represented in one function for code simplicity:

```
1  def transmitAWGN_BPSK(cword, sigma):
2      mod_u = np.zeros(len(cword))
3      noise = np.random.normal(0, sigma, len(cword))
4      # BPSK modulation
5      for i in range(len(cword)):
6          mod_u[i] = 1-2*cword[i]
7      return mod_u + noise
```

Listing 3: function $transmitAWGN\_BPSK()$

6 Decoding polar codes

In this part we assume that the we receive a sequence of N bits $r = \begin{bmatrix} r_1 & r_2 & \cdots & r_N \end{bmatrix}$ that is a codeword encoding as presented in 4.2.3 and transmitted as in section 5. We will perform decoding to find an estimation of the vector u called $\hat{u} = \begin{bmatrix} \hat{u}_1 & \hat{u}_2 & \cdots & \hat{u}_N \end{bmatrix}$.

6.1 Successive cancellation decoding

To find an estimate $\hat{u}$ of u from r we won't perform as for usual decoder a bit by bit estimation but we are going to use the previous bit estimation to estimate the next bit. This is called the successive cancellation (SC) decoding.

For example, to estimate the value $\hat{u}_3$ we will use r, $\hat{u}_2$ and $\hat{u}_1$ and we assume that the our guess for $\hat{u}_2$ and $\hat{u}_1$ are correct.

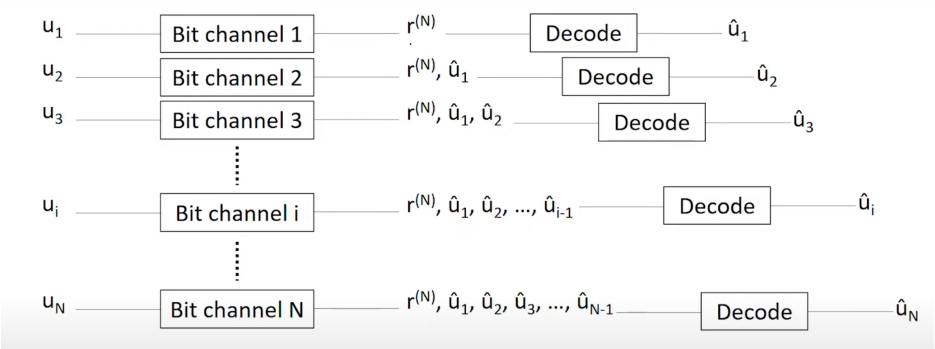


Figure 5: Successive cancellation decoding

Thus the problem is comparable to a single parity check decoder for N bit channels except for the frozen bit positions where the output will be forced to be 0.

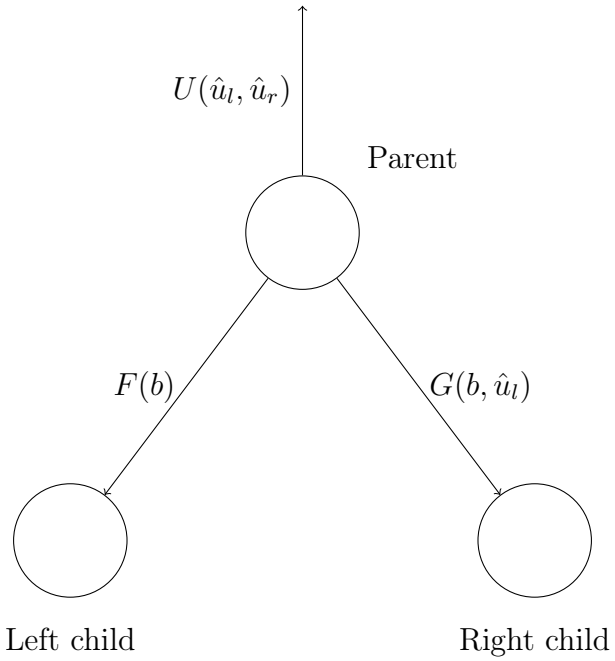
6.2 Tree decoding

In this subsection we will focus on how to implement the SC-decoding on a tree.

6.2.1 Dimension-2 block

The dimension 2 block we are going to describe is constituted of 3 nodes (1 parent and 2 children) and is given 3 functions ($F(b)$, $G(b, \hat{u}_l)$ and $U(\hat{u}_l, \hat{u}_r)$).

This scheme is to be replicated n times where $N = 2^n$.



When the scheme is replicated, each node is both a parent and a child except N+1 exception, at depth 0, the root of the tree is only a parent and at depth $n = \log_2(N)$ the N leaf nodes are only children.

Except for the exception nodes, every node receive a vector from its parent called **belief** which dimension varies according to the depth according to the formula:

$$nb = \frac{N}{2^d} \text{ bits}$$

Where nb is the number of bits in the belief vector d is the depth of the node in the tree.

There are only 3 functions:

- **F() function:** that takes the bits of the belief and pass half of them to the left child
- **G() function:** that takes the bits of the belief and the estimate of the left child and pass it to the right child
- **U() function:** that takes the estimate of the 2 children to pass it to its parent

There are many available function for $F()$ and $G()$ but here are the function we choose, the belief vector is $b = \begin{bmatrix} b_1 & b_2 & \cdots & b_{nb} \end{bmatrix}$ so:

$$F(b) = \begin{bmatrix} f(b_1, b_{\frac{nb}{2}+1}) & f(b_2, b_{\frac{nb}{2}+2}) & \cdots & f(b_{\frac{nb}{2}}, b_{nb}) \end{bmatrix}$$

where $f(a,b) = 2 \times \tanh^{-1}(\tanh(\frac{a}{2}) \times \tanh(\frac{a}{2}))$

$$G(b, \hat{u}_l) = \begin{bmatrix} g(b_1, b_{\frac{nb}{2}+1}, \hat{u}_l) & g(b_2, b_{\frac{nb}{2}+2}, \hat{u}_l) & \cdots & g(b_{\frac{nb}{2}}, b_{nb}, \hat{u}_l) \end{bmatrix}$$

where $g(a, b, \hat{u}_l) = (-1)^{\hat{u}_l} \times b + a$

and $U(\hat{u}_l, \hat{u}_r) = \begin{bmatrix} \hat{u}_l + \hat{u}_r & \hat{u}_r \end{bmatrix}$ where $\hat{u}_l$ and $\hat{u}_r$ are respectively the guesses from the left and right children.

The order of the operation is always in this order from the root to the leaves:

1. **Left child operations**, the parent send the result of **F()** to the left child and wait for its response.
2. **Right child operations**, the parent node send the result of **G()** to its right child and wait for it to return its guess.
3. **Send guess to parent**, the node send to its parent its own guess based on its children guesses via **U()**.

6.2.2 Leaf nodes operations

The leaf nodes don't have any children, their role is to make a hard decision on the information they receive. Because they are located at the base of the tree the size of the belief vector is only one bit, so they have to return to their parent one bit which is the hard decision.

There is only 2 different cases, either the leaf node is located on a frozen bit position and then it return 0 regardless of the value of its belief, either it has to make a hard decision following the classic rule:

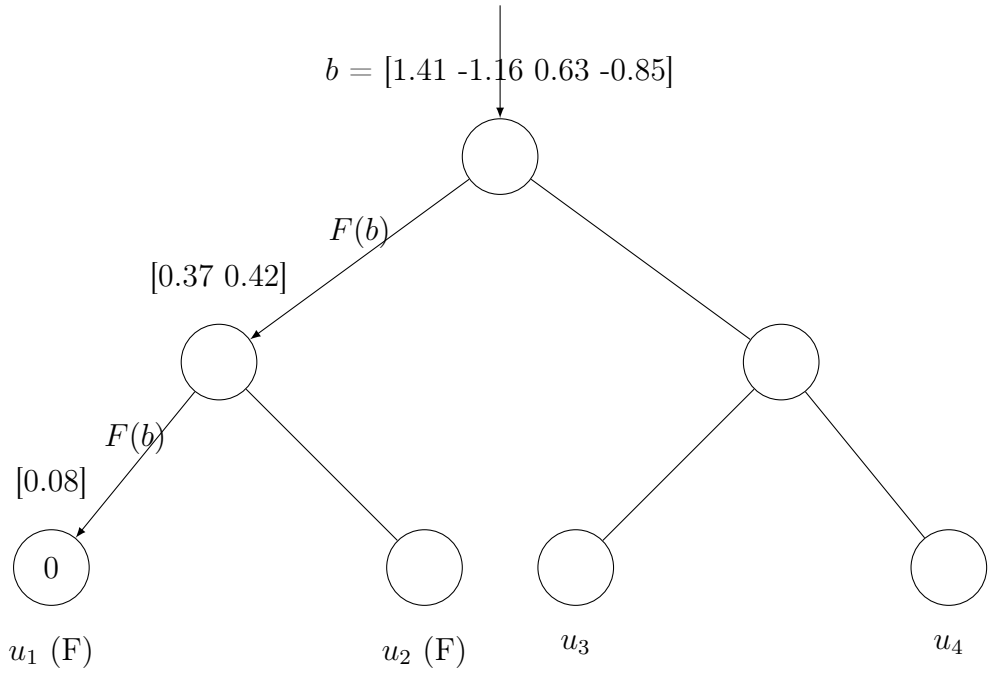
$$\hat{u} = \begin{cases} 0 & \text{if} \quad b > 0 \\ 1 & \text{if} \quad b < 0 \end{cases}$$

where $\hat{u} \in \{0, 1\}$ is the guess returned to the parent and b the belief from the parent.

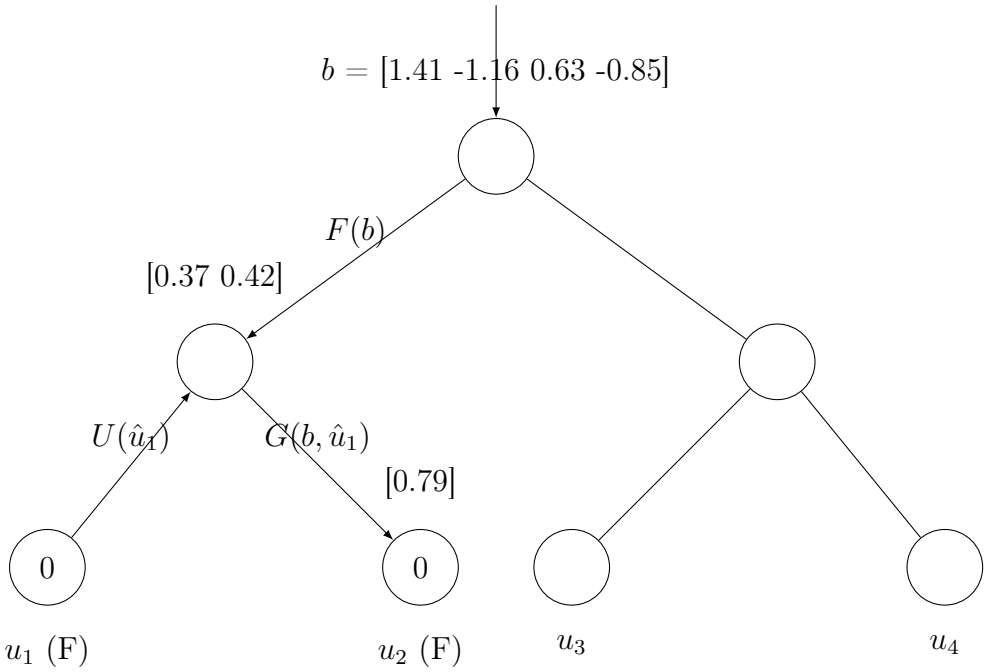
6.2.3 Initialization and algorithm termination

The enter value and the return values are put and read out from the root of the tree, the received vector r is put as the belief for th root and launch the tree decoding. The return value is read out after the U() operation of the root.

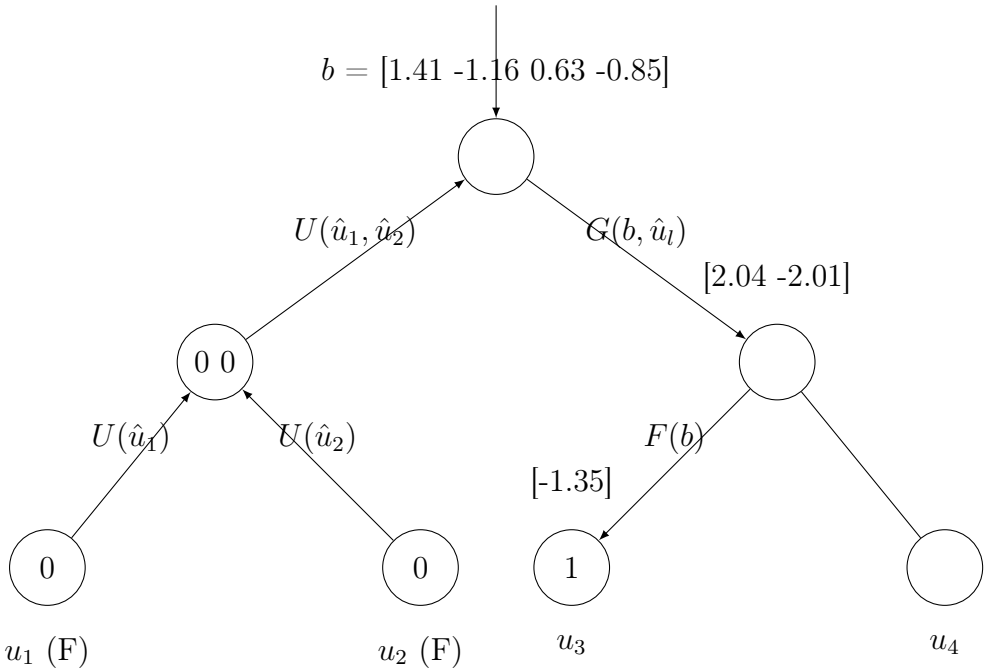
6.2.4 N = 4 decoding example



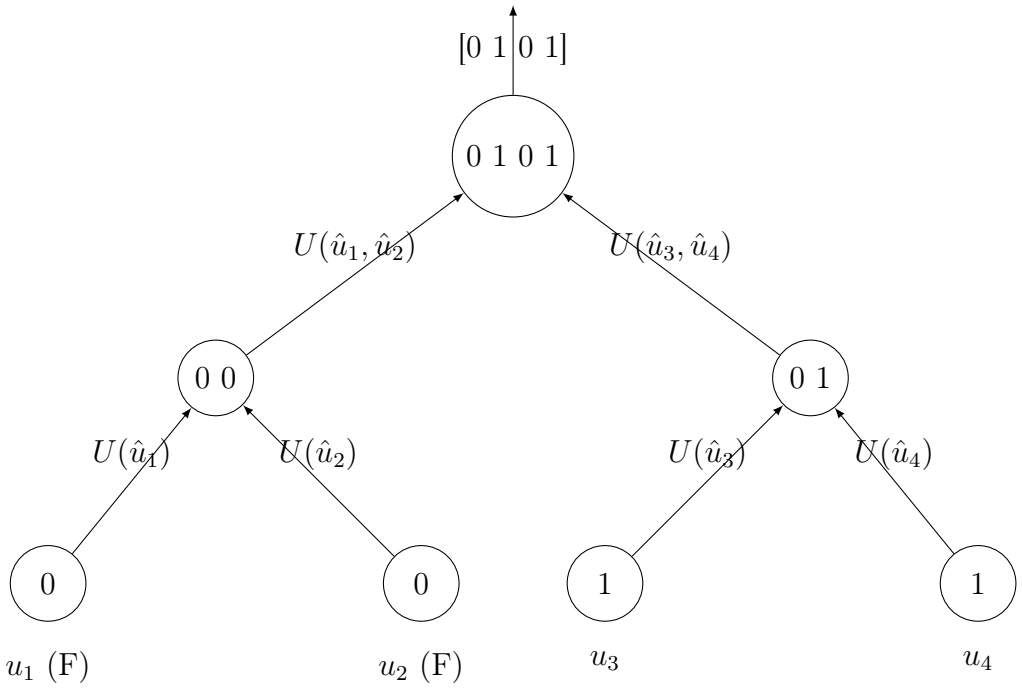
When we receive $r = \begin{bmatrix} 1.41 & -1.16 & 0.63 & -0.85 \end{bmatrix}$ we use it as the belief for the root and then all the left side nodes passes the belief threw the F() function to the u_1 leaf node and because it is a frozen node it return automatically to its parent.



The u_2 bit is also frozen so it will return 0 to its parent and then the parent will return its guess to the root with U() and the root will perform right child operation with G() and the left child will do left and then right child operations:



And after that, all the information go up to the root with U() operations and the output can be read from the root:



And we finally have our output $\hat{u} = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}$ that can be read at the root.

6.3 Decoder implementation

All the steps described in the previous section will be implemented in a long function that will cover every case and will cross the tree:

```
def polar_decode(received, N, K, RS_N):
    max_depth = int(np.log2(N))

    # Storage for the beliefs, size: Depth lines, N columns
    Beliefs = np.zeros([max_depth + 1, N])

    # Storage for the data coming back
    ucap = np.zeros([max_depth + 1, N])

    # Storage for the nodes state: 0 = yet to be activated,
    #1 = finish L, 2 = finish R, 3 = finish U
```

```
    Nodes_state = np.zeros(2 ** (max_depth + 1) - 1)

    # Initialize the roots belief
    Beliefs[0] = received

    # Initialize the crossing of the tree from the root
    node = 0
    depth = 0

    # Starting the decoding while it is not finish
    done = 0
    nearly_done = 0
    while done == 0:
        # Check if we are at the leaf position
        if depth == max_depth:
            # Check if the leaf is a frozen bit
            if (node + 1) in RS_N[:N - K]:
                ucap[depth][node] = 0
            # The non-frozen case is the hard decision for ui
            else:
                if Beliefs[depth][node] > 0:
                    ucap[depth][node] = 0
                else:
                    ucap[depth][node] = 1
            # Check if the decoding is complete
            if node == N - 1:
                nearly_done = 1
            # Going back to parent
            node = int(node / 2)
            depth -= 1
        else:
            # Position of the node in the node state vector
            node_position = int(2 ** depth - 1 + node)

            # Step L and go to left child
            if Nodes_state[node_position] == 0:
                # Length of the incoming belief in bits
                nb = int(N / (2 ** (depth + 1)))

                # Extracting the incoming belief from Beliefs
                Incoming_belief =
```

```
                Beliefs[depth][2*nb*node:2*nb*node+2*nb]
                a = Incoming_belief[:nb]
                b = Incoming_belief[nb:]
                # Going to the left child
                node *= 2
                depth += 1
                # nb *= 0.5
                # Sent the belief to the child
                insert_vector(f(a, b),Beliefs[depth], node*nb)
                Nodes_state[node_position] = 1
            # Step R and go to right child
            elif Nodes_state[node_position] == 1:
                # Length of the incoming belief in bits
                nb = int(N / (2 ** (depth + 1)))
                # Extracting the incoming belief from Beliefs
                Incoming_belief =
                Beliefs[depth][2*nb*node:2*nb*node+2*nb]
                a = Incoming_belief[:nb]
                b = Incoming_belief[nb:]
                # Get the info from the left child
                left_child = 2 * node
                left_child_depth = depth + 1
                #Get the incoming decision from the left child
                u_hat = ucap[left_child_depth]
                [left_child*nb:left_child*nb+nb]
                # Going to the right child
                node = 2 * node + 1
                depth += 1
                # nb *= 0.5
                # Sent the belief to the child
                insert_vector(g(a,b,u_hat),Beliefs[depth],
                node*nb)
                Nodes_state[node_position] = 2
            # Step U and go to parent
            elif Nodes_state[node_position] == 2:
                # Length of the incoming belief in bits
                nb = int(N / (2 ** (depth + 1)))
                # Get the info from the children
                left_child = 2 * node
                right_child = 2 * node + 1
                child_depth = depth + 1
```

```
                #Get the incoming decision from the left child
                u_hat_left = ucap[child_depth]
                [left_child * nb:left_child * nb + nb]
                # Get the incoming decision of the right child
                u_hat_right = ucap[child_depth]
                [right_child * nb:right_child * nb + nb]
                insert_vector(u(u_hat_left, u_hat_right),
                ucap[depth], node * 2 * nb)
                # Going back to parent
                node = int(node / 2)
                depth -= 1
                if nearly_done > 0:
                    if nearly_done == max_depth:
                        done = 1
                    nearly_done += 1
    return ucap
```

This is only the decoder part of the algorithm, you can see the full code on my GitHub.

7 Simulations of BER and WER

On this section we will simulate and compute the Bit Error Rate (BER) and the Word Error Rate (WER) for our encoder. The bit error rate correspond to every bit in $\hat{u}$ different than the bit in c_{word} and the WER correspond to each time the decoded message $\hat{m}$ didn't correspond to the random message m.

From the code point of view, $\hat{u}$ can be read on the first line of the return value of the decoder function *ucap* and $\hat{m}$ can be read on the $\frac{N}{2}$ most reliable positions on the last line of *ucap*.

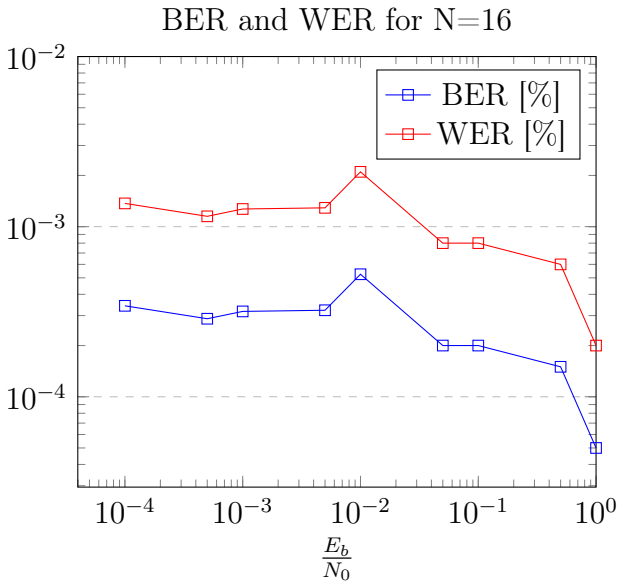
7.1 BER and WER by blocklenght

We can adjust the different simulation parameters to match the best result/computation time ratio for each block length and compute BER and WER for different values of $\frac{E_b}{N_0}$:

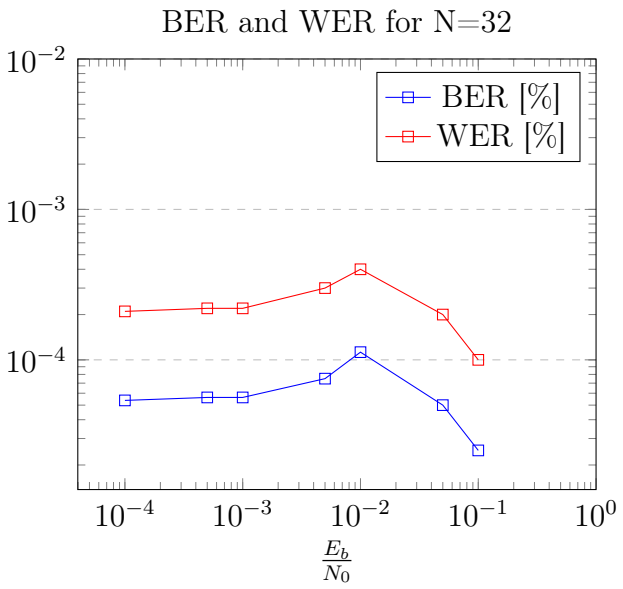
```
N = 64                                  # 16, 32, 64, 128, 256
K = N/2                                 # 8, 16, 32, 64, 128
EbNo_dB_range = [0.0001, 0.0005,
                 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5]
S = 1000000                             # Change according to N

print(simulation(N, S, EbNo_dB_range))
```

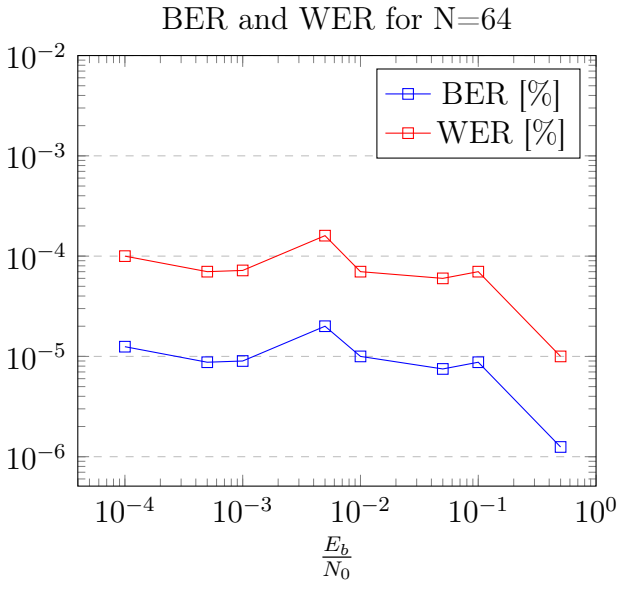
7.1.1 N = 16



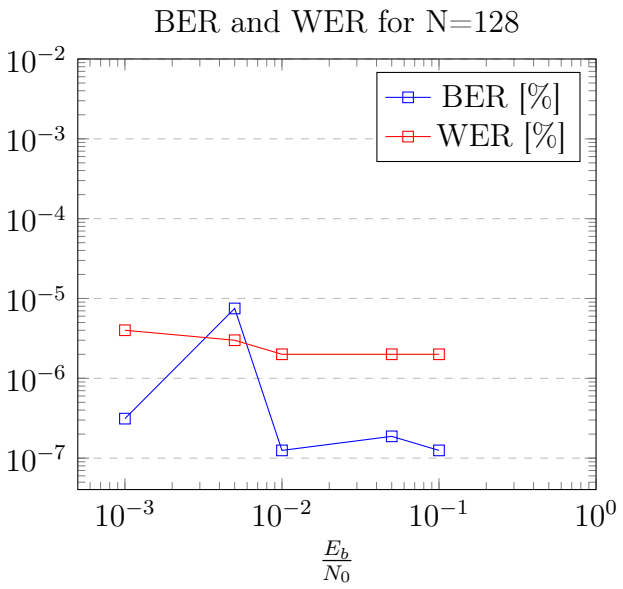
7.1.2 N = 32



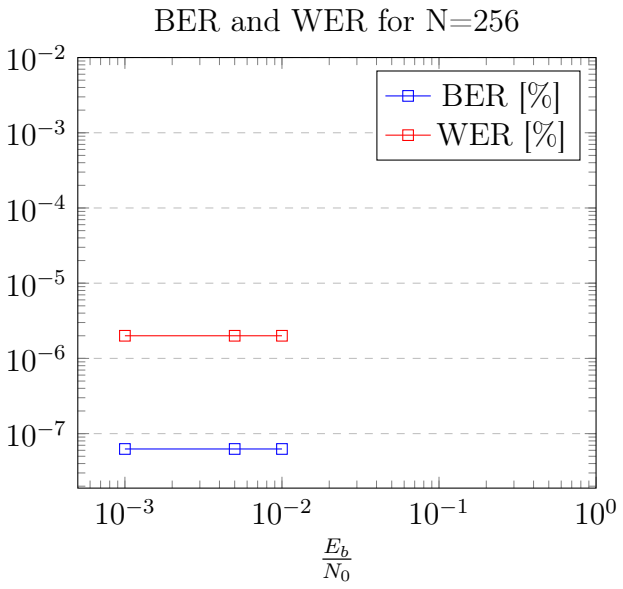
7.1.3 N = 64



7.1.4 N = 128



7.1.5 N = 256



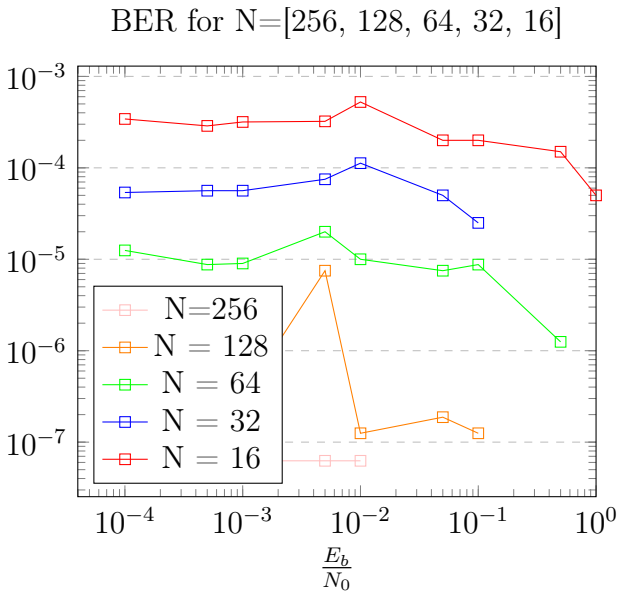
7.1.6 Observations

When we compare the different plots, we remark that the behavior is roughly the same independently of the block-length, as we could expect the **BER and the WER decrease** as the **SNR ratio increase**.

The WER is higher than the BER because if there is a bit error there is *1/2* chance that it will cause a word error because we have a code rate of *1/2* and there is N more bits transmitted than words that's why the **gap between WER and BER increase** as **N increase**.

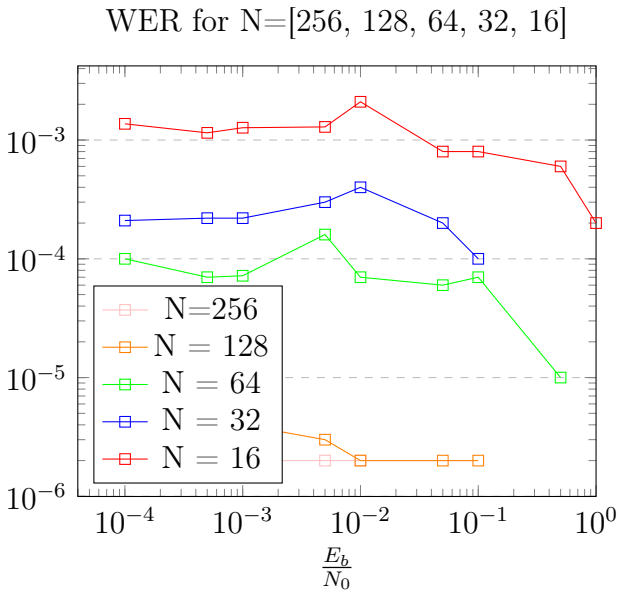
7.2 Blocklenght comparison

7.2.1 BER



We see that the BER decreases when the block-length increase, that means that the strength of the code towards noise increases when the block-length increases. We can also see that for N = 128 and N = 256 BER is so low that we will need to increase the number of simulations to have a good approximation of the real BER.

7.2.2 WER



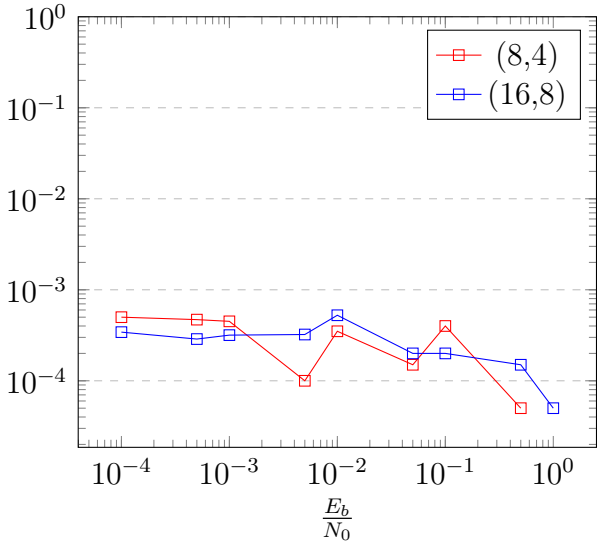
For the WER we came to the same conclusion than for the BER the values are just a bit shifted to the top.

7.3 Comparing to other codes

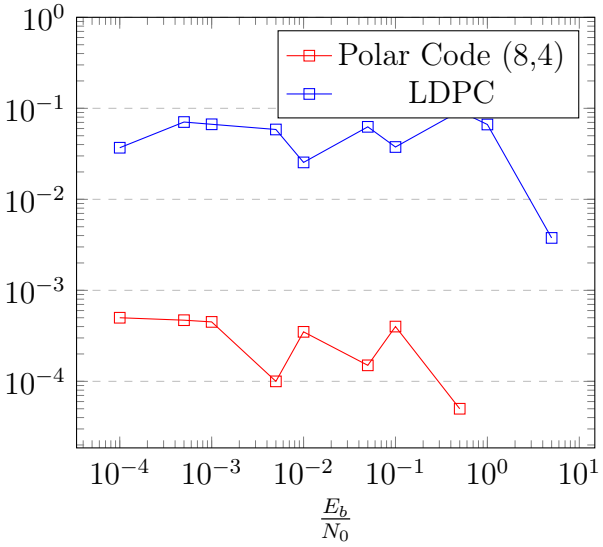
The good results from the polar code could be compared to other codes, we will use the Polar code (8,4) to have a fast computing code to compare to other 8 bit codes we saw during the course [1].

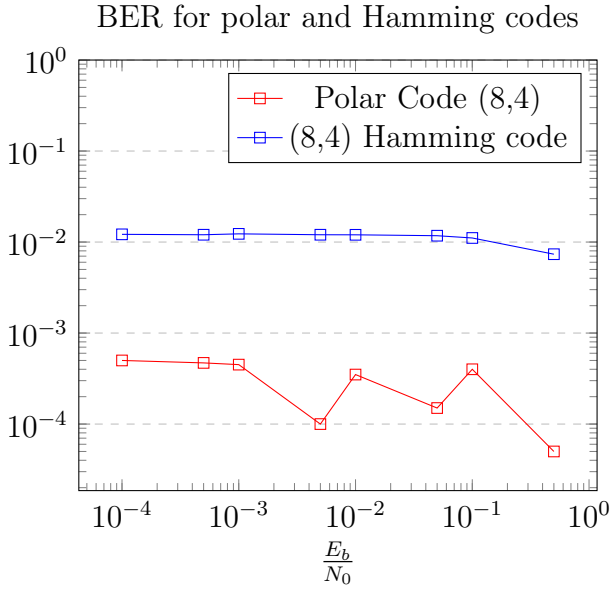
Here is the value of the BER for the (8,4) polar code compared to the (16,8) polar code for reference:

BER for (8,4) and (16,8) polar codes



BER for polar and LDPC codes





The results for LDPC code and Hamming code came from decoder that we have implemented during previous assignments.

8 Conclusion

In that report we saw what are polar code and how to construct polar encoder and decoder by following the tree algorithm described in the videos [2]. This report provide the **only written work** that describe the tree algorithm implementation for polar decoder. It also provide a *Python* version when the video course give a *Matlab* code.

We have also seen that polar codes are really efficient, especially for long block-length, for transmitting messages on low signal to noise ratios comparing to other codes.

Tree decoding provides an efficient and scalable way to decode those polar codes with an algorithmic complexity in $O(n \log_2(n))$. The longer the codes gets, the longer the time of decoding is, that's why for 128 and 256 bits block-length I don't have enough values, the BER and WER are so low that even with 1,000,000 iterations, there is not one bit error and knowing that the computation time for those block-length is 10h that explains that I miss some values to illustrate plainly my statements, but we can deduce that the behavior of long block-length will follow the tendance drawn by block-length

of 64, 32 and 16 bits.

Finally if you want to see the full code, it is available on my GitHub page and if you have any questions don't hesitate to contact me here.

References

[1] Brian M. Kurkoski. *Lattice Coding Theory Lecture Notes*. Course at JAIST, I437E. Nov. 2024. URL: https://dlclms.jaist.ac.jp/moodle/pluginfile.php/125691/mod_resource/content/85/lct.pdf.

[2] *LDPC and Polar Codes in 5G Standard*. Online video course, NPTEL. May 2019. URL: https://www.youtube.com/playlist?list=PLyqSpQzTE6M81HJ26ZaNv0V3Kc.

[3] Valerio Bioglio Member IEEE Carlo Condo Member IEEE Ingmar Land Senior Member IEEE. "Design of Polar Codes in 5G New Radio". In: (2020). URL: https://arxiv.org/pdf/1804.04389.

[4] Saeid Ghasemi and Bartolomeu F. Uchôa-Filho. "An Algorithm for Finding an Approximate Reliability Sequence for Polar Codes on the BEC". In: (2021). URL: SBrT%202021%201570734156.