Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering
Telecommunications Division

Communication Systems II

Research Project

Error Analysis and Detection Algorithms for Hexagonal-QAM Constellations

Group 69
Liouliakis Nikolaos - 10058
Papadopoulou Aikaterini - 10009

May 2022

Abstract

The purpose of this research project is to model and simulate Hexagonal QAM - HQAM constellations, (also known as Triangular QAM) and propose a low complexity detection algorithm, especially for higher order TQAM constellations. Moreover, the simulations were used to check and verify that the upper bound for the Symbol Error Probability (SEP) which was proposed in [1] is valid.

All simulations were coded and run using Matlab.

Contents

1 Modeling **1**
- 1.1 Creating the constellations . . . 1
  - 1.1.1 Hexagonal Grid & Coordinate system . . . 1
  - 1.1.2 Creating the constellations of even power . . . 1
  - 1.1.3 Creating the constellations of odd power . . . 1
  - 1.1.4 Creating the constellation with power n = 3 . . . 2
  - 1.1.5 Calculating the energy efficiency Es/d^2 . . . 2
- 1.2 Bit mapping the constellations . . . 2
  - 1.2.1 Creating a one dimensional Gray Code . . . 3
  - 1.2.2 Checking the one dimensional Gray Code . . . 3
  - 1.2.3 Gray mapping constellations of even power . . . 4
  - 1.2.4 Gray mapping constellations of odd power . . . 6
  - 1.2.5 Gray mapping the constellation with order n=3 . . . 9
  - 1.2.6 Calculating the Gray code penalty Gp . . . 9

2 Symbol Detection **10**
- 2.1 Linear detection . . . 10
- 2.2 Our detection . . . 11
  - 2.2.1 Method of our detection . . . 11
  - 2.2.2 Verification of our detection . . . 12

3 Simulations **17**
- 3.1 Simulations key points . . . 17
- 3.2 Matlab code and files . . . 18
  - 3.2.1 LinearGrayCode.m . . . 18
  - 3.2.2 SetBits.m . . . 18
  - 3.2.3 GrayCodeAndSetBitsTesting.m . . . 19
  - 3.2.4 RegularHQAM.m . . . 19
  - 3.2.5 EvenGrayMappingTest.m . . . 19
  - 3.2.6 DetectionLinearSlow.m . . . 19
  - 3.2.7 DetectionLinearSlowTesting.m . . . 19
  - 3.2.8 DetectionLinearSlow2D.m . . . 19
  - 3.2.9 DetectionFast.m . . . 20
  - 3.2.10 DetectionFastTesting.m . . . 20
  - 3.2.11 DetectionTesting.m . . . 20
  - 3.2.12 Main_Simulation.m . . . 20
  - 3.2.13 Visual_Simulation.m . . . 20
  - 3.2.14 Final_SEP_BER_graph_of_all_data.m . . . 20
  - 3.2.15 Bounds_and_More_graphs.m . . . 20

4 Data Analysis and Conclusions **21**

5 Further Improvements **30**

1 Modeling

1.1 Creating the constellations

In order to run simulations with HQAM constellations on Matlab, it was necessary to create them first. The constellations have $M = 2^n$ symbols in order to make it possible to map them to binary numbers. Any M-ary HQAM constellation is defined by a set $S_M = \{s_i \in \mathbb{C}, i = 0, 1, ..., M-1\}$ of M symbols, where S_M is a subset of the infinite grid $S = \{\mathbf{v} \in \mathbb{C} : \mathbf{v} = c_1\mathbf{v_1} + c_2\mathbf{v_2} - \mathbf{v_3}\}$, $\mathbf{v_1} = [d_{min}, 0]$ and $\mathbf{v_2} = [\frac{1}{2}d_{min}, \frac{\sqrt{3}}{2}d_{min}]$ are the basis vectors of the 2D grid and $\mathbf{v_3} = [h_o, v_o]$ is the offset vector which centers the symbols around (0,0), d_{min} is the minimum distance between symbols, $c_1, c_2 \in \mathbb{N}$ and $h_o, u_o \in \mathbb{R}$ are respectively the horizontal and vertical offset of the grid. When $c_1, c_2 \in \mathbb{Z}$ and $\mathbf{v_3} = \mathbf{0}$ then the infinite grid S represents the Eisenstein integers, sometimes also called the Eisenstein-Jacobi integers, as explained in more detail in [2].The linear combination of the basis vectors creates all the potential relative positions of the symbols in the complex plane. With the proper coefficients c_1, c_2 and offset $\mathbf{v}_3$, any desired HQAM constellation can be created.

1.1.1 Hexagonal Grid & Coordinate system

Triangular QAM constellations have distinctive hexagonal decision regions, hence the name Hexagonal QAM which is also used in bibliography.
The particularity of the hexagonal decision regions demands a special handling, which differentiates them from QAM constellations that have rectangular decision regions.
To store the symbols, we used a 2D array where the indexes are chosen such that they correspond to c_1 and c_2. Note that the symbols that are in a vertical zig-zag line do not have the same index, since to go from one to the other one that is 2 symbols up we have to increase c_2 by 2 but decrease c_1 by 1. Thus, the dimensions of the 2D array are bigger than one would expect, since they are stored in a way to make the detection easier.

1.1.2 Creating the constellations of even power

For even order of constellation, the number of symbols is a perfect square. The positions of the symbols resemble a square with $\sqrt{M}$ symbols on each side. Let $a_{even} = \sqrt{M} = 2^{n/2}$. The sizes of the 2D array to store the symbols are $a_{even} + \frac{1}{2}a_{even} - 1$, a_{even}. Notice as we noted before, the first dimension is bigger than a_{even}, but the other one is exactly a_{even}. To create them, we used a double for loop for the coefficients c_1 and c_2 and a helping variable to account for the offset we mentioned before, since every other vertical symbol should be moved to the left (which means reducing c_1 by 1). Also, since we needed to center the constellation around (0,0) we must choose the proper $\mathbf{v_3}$= [h_o, v_o] to achieve this. Since the initial coordinates of the first symbol we create, which is the bottom left one, are (0,0) (for c_1=0 and c_2=0) the offset can be split in two terms. The first term of the offset corresponds to the main offset (after which there is a symbol at (0,0)) and the second term offsets the constellation to have the center of two symbols at (0,0). Thus, after some calculations, we have:

$$h_o = \frac{1}{2}a_{even}\,d_{min} - \frac{1}{4}d_{min}$$

$$v_o = \frac{1}{2}a_{even}\frac{\sqrt{3}}{2}d_{min} - \frac{\sqrt{3}}{4}d_{min}$$

which can also be written as $\mathbf{v_3} = \frac{1}{2}a_{even} \cdot (\frac{1}{2}\mathbf{v_1} + \mathbf{v_2}) - \frac{1}{2}\mathbf{v_2}$.

1.1.3 Creating the constellations of odd power

For odd order of constellation, the number of symbols is not a perfect square, the symbols form a cross. Let $a_{odd} = \sqrt{M/8} = 2^{\frac{n-3}{2}}$. The cross is a subset of the symbols that form a bigger square with

sides $3a_{odd}$ without the four corner squares with side $\frac{1}{2}a_{odd}$. The sizes of the 2D array to store the symbols are $3a_{odd}+\frac{3}{2}a_{odd}-1, 3a_{odd}$. To create them, we used a double for loop for the coefficients c_1 and c_2 and a helping variable to account for the offset we mentioned before, since every other vertical symbol should be moved to the left (which means reducing c_1 by 1) and with the proper condition we skip the symbols that belong to the corner squares. Also, since we needed to center the constellation around (0,0) we must choose the proper $\mathbf{v_3} = [h_o, v_o]$ to achieve this. With similar calculations like before but with the small change that the symbol that corresponds to (0,0) (for c_1=0 and c_2=0) is the first symbol we skipped (again the bottom left one) we conclude that:

$$h_o = \frac{3}{2}a_{odd}\,d_{min} - \frac{1}{4}d_{min}$$
$$v_o = \frac{3}{2}a_{odd}\,\frac{\sqrt{3}}{2}d_{min} - \frac{\sqrt{3}}{4}d_{min}$$

which can also be written as $\mathbf{v_3} = \frac{3}{2}a_{odd}\cdot(\frac{1}{2}\mathbf{v_1}+\mathbf{v_2}) - \frac{1}{2}\mathbf{v_2}$.

1.1.4 Creating the constellation with power n = 3

For M=8 which is a special case since it does not follow any pattern it needs special handling. The constellation is close to a square with side 3 with the middle left symbol missing, so like before with a double for loop we create the symbols of the square and with a condition we skip that symbol. To center the symbols is much simpler since it is only for this specific case, so again after some simple calculations for the offset vector $\mathbf{v_3} = [h_o, v_o]$ we conclude:

$$h_o = d_{min}$$
$$v_o = \frac{\sqrt{3}}{2}d_{min}$$

which can also be written as $\mathbf{v_3} = 1\cdot(\frac{1}{2}\mathbf{v_1}+\mathbf{v_2})$.

1.1.5 Calculating the energy efficiency Es/d^2

The energy efficiency is a characteristic of a constellation, and it is useful to check that it is created correctly. So to validate them, we calculated it for those that our model generates. Which (for the most part) agree with the ones in [3].

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
E_s/d^2	2	4.5	9	17.75	37	72	149	289	597	1157	2389	4629	9557	18517

Table 1: The energy efficiency values of our constellations.

1.2 Bit mapping the constellations

Each symbol represents some bits, which are the information that is the ultimate goal to transmit. When the number of symbols in a constellation are M (which is a power of 2) they are mapped to $n = \log_2 M$ bits. Our end goal is the final received information/bits to have as little errors as possible. To achieve this, we map the symbols in a very particular way, such that neighboring symbols have as little as possible bit differences. That way, when the receiver detects a symbol incorrectly and the symbol detected is a neighboring symbol (which is most likely to happen) the error in the information

transmitted is minimized.
When neighboring symbols have exactly one bit difference, it is called Gray mapping and is optimal. For HQAM constellations, perfect Gray mapping is not possible, so something as close to Gray mapping with the best performance is implemented.

1.2.1 Creating a one dimensional Gray Code

Foremost, we need to create an array of binary numbers, which represents the bits, that have one bit difference from each other. So we derived a formula that finds for any number of bits the g_{code} which is the decimal form of the i-th binary number of the Gray code table. The outstanding characteristic of our approach is that we can determine for each bit of any table index whether it is 1 or 0 independently of all the other bits of the number and the other numbers of the table.

$$a = \left\lfloor \frac{2^{n-j} \bmod i}{2^{n-j-1}} \right\rfloor \tag{1}$$

$$b = 2 \bmod k,\ k = \left\lfloor \frac{i}{2^{n-j}} \right\rfloor \tag{2}$$

$$b_j = \mid a - b \mid \tag{3}$$

$$g_{code} = \sum_{j=0}^{n-1} b_j 2^{n-1-j} \tag{4}$$

Where i denotes the index of the Gray code in the Gray code table and j the index of the bit in the binary number. Note that $a, b \in \{0,1\}$. Furthermore, b_0 is the MSB (most significant bit) and b_{n-1} is the LSB (least significant bit).

```
1      codeDecimal = zeros(1,2^n, 'int32');
2      for i = 0:2^n-1
3          for j = 0:n-1
4              a = idivide( mod( int32(i) , 2^(n-j) ) , 2^(n-j-1 ) );
5              b = mod( idivide( int32(i) , 2^(n-j) ) , int32(2) );
6              codeDecimal(i+1) = codeDecimal(i+1) + bitshift( abs(a-b) , n-j-1 );
7          end
8      end
```

Matlab code to generate the Gray code

1.2.2 Checking the one dimensional Gray Code

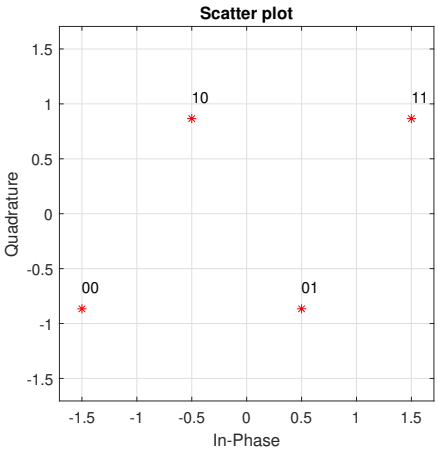
After the creation of the Gray code, we created a special script with the purpose of checking that our approach is indeed correct. To check, we need a function that counts the number of set bits (the number of bits in a binary number that are 1). After some research, we chose Brian Kernighan's Algorithm because of its low complexity that is equal to the number of set bits (at most n) since on our checking we expect it to be precisely 1. To verify that the array of binary numbers have exactly one bit difference we needed to count the bit difference between neighboring numbers, which is done using the XOR operator on those numbers before calling the *SetBits* function, and they must be exactly 1. If two neighbors with more than 1 bit difference are found, then *errors* is not zero at the end, which indicates a problem (obviously *errors* is zero at the end of the execution of our code).
This process is used to verify that the creating method is valid for a big enough set of Gray codes (different values for n) that will be used later.

```
    errors = 0;
    for n=1:31
        GrayCode = LinearGrayCode(n);
        for i= 1:2^n-1
            if SetBits( bitxor( GrayCode(i),GrayCode(i+1) ))≠ 1
                errors = errors + 1;
            end
        end
    end
```

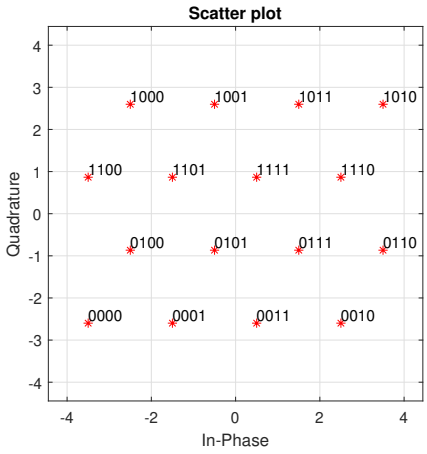
Matlab code to check the one dimensional Gray code

1.2.3 Gray mapping constellations of even power

Since the symbols form a perfect square and each side has a_{even} symbols, which is a power of 2, the optimal way to do the bit mapping is to apply the one dimensional Gray mapping to each of the two dimensions and then combine them together. The one dimensional Gray code has n/2 bits which are combined to give a binary number with n bits. The first n/2 bits, which are the most significant bits, are from the first dimension and the rest n/2 bits, which are the least significant bits, are from the other.
In Figures 1a, 1b, 1c, 1d the final bit mapping is shown. Note that in Figure 1d is used the decimal form of the number for practical purposes.

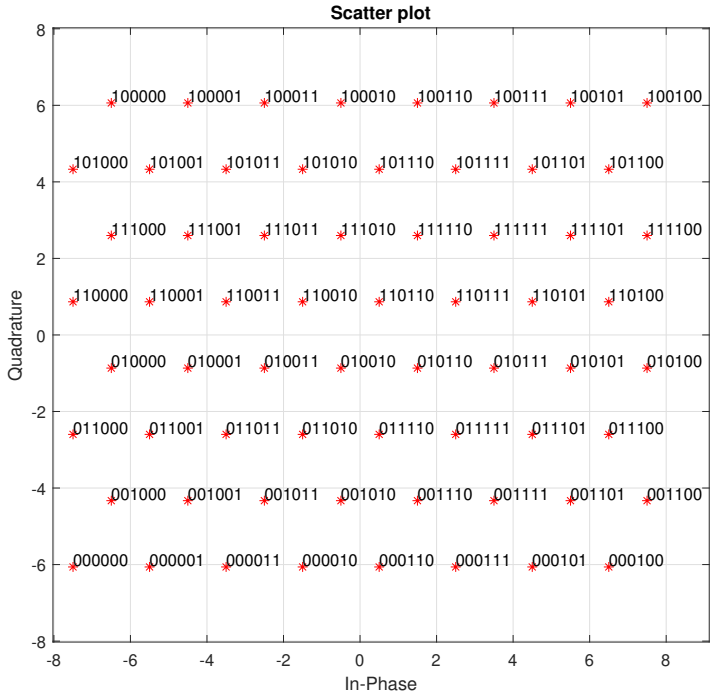


(a) Binary code for n=2, M=4,

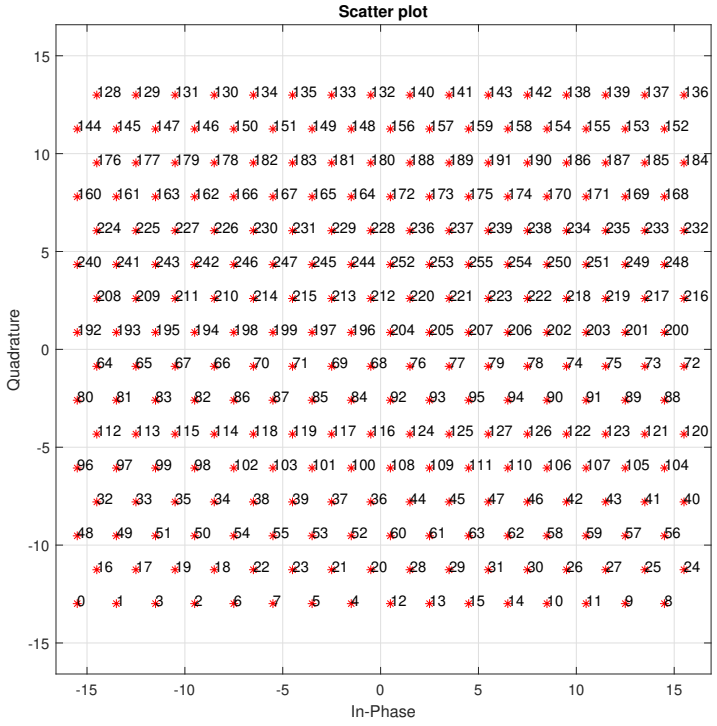


(b) Binary code for n=4, M=16

Figure 1



(c) Binary code for n=6, M=64

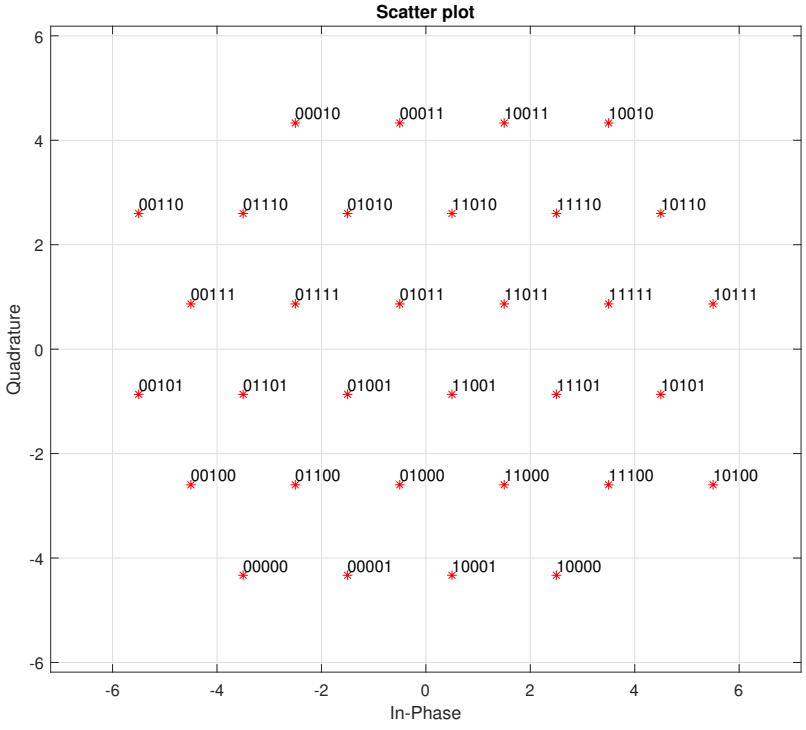


(d) Decimal code for n=8, M=256

Figure 1: Gray mapping of constellations of even power

1.2.4 Gray mapping constellations of odd power

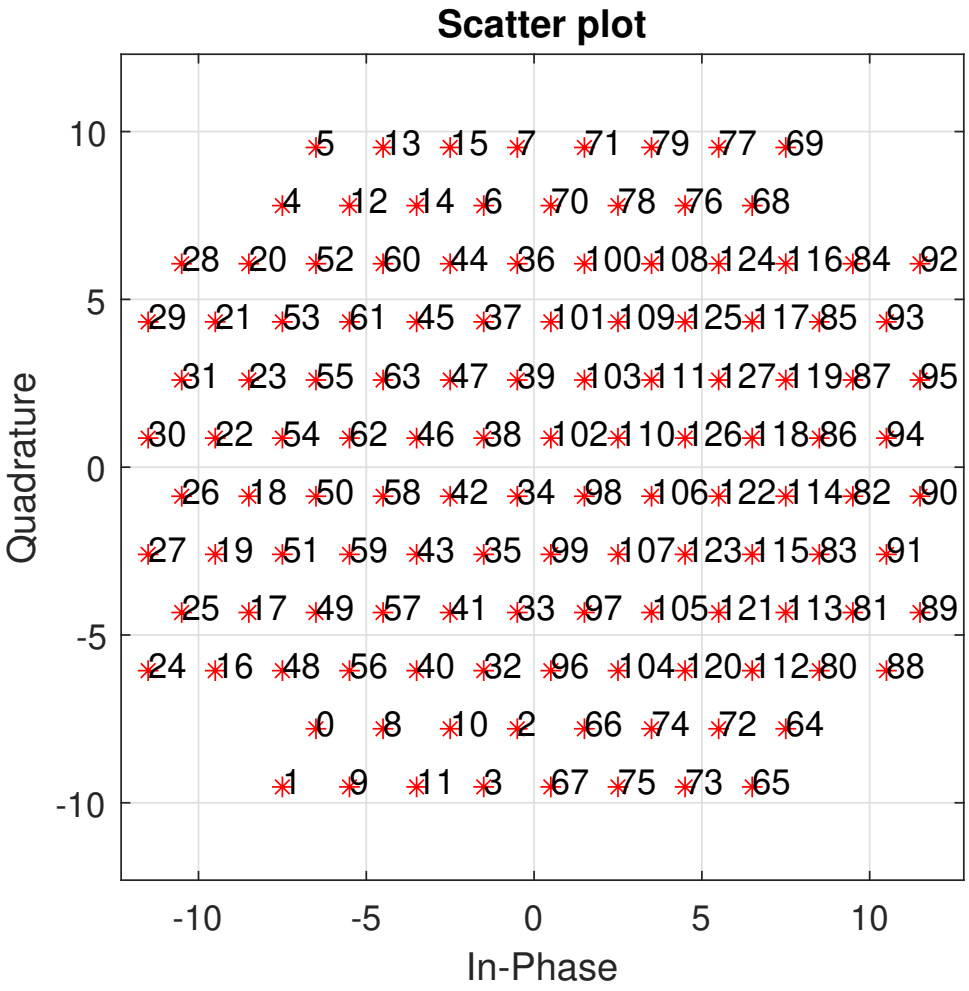
After extensive research, we found paper [4] which was on Cross QAM, so it needed the appropriate changes. To start, we create a rectangular two-dimensional Gray code like before, but with sides $4a_{odd}$ and $2a_{odd}$. The two one-dimensional codes have a different number of bits. The small one (which is in the direction with side $2a_{odd}$) has $\lfloor n/2 \rfloor$ and the other one has $\lceil n/2 \rceil$ [1]. The two-dimensional Gray code of the centered sub rectangle with sides $3a_{odd}$ and $2a_{odd}$ is translated to the symbols. After that bit mapping we are left with the symbols that are at the top and bottom of the constellation which need to be mapped to the leftover Gray code which are the left and right rectangles with sides $a_{odd}/2$ and $2a_{odd}$. Then we split the left rectangle in two smaller rectangles, one on top and one at the bottom with sides $a_{odd}/2$ and a_{odd}. The top one after some more steps will be translated to the top left rectangular symbols (with sides a_{odd} and $a_{odd}/2$) and the bottom one to the bottom left rectangular symbols. Each of those rectangles (with sides $a_{odd}/2$ and $2a_{odd}$) are split in the middle into two squares with side $a_{odd}/2$. The top square of the top left rectangle is first flipped over the x-axis and then is finally translated to the top left square of the top left rectangular symbols (with sides a_{odd} and $a_{odd}/2$) of the constellation. The bottom square of the top left rectangle is first flipped over the y-axis and then is finally translated to the top right square of the top left rectangular symbols (with sides a_{odd} and $a_{odd}/2$) of the constellation. Performing similar operations on the remaining 3 rectangles, the bit mapping is complete.
In Figures 2e, 2f, 2g the final bit mapping is shown.



(e) Binary code for n=5, M=32

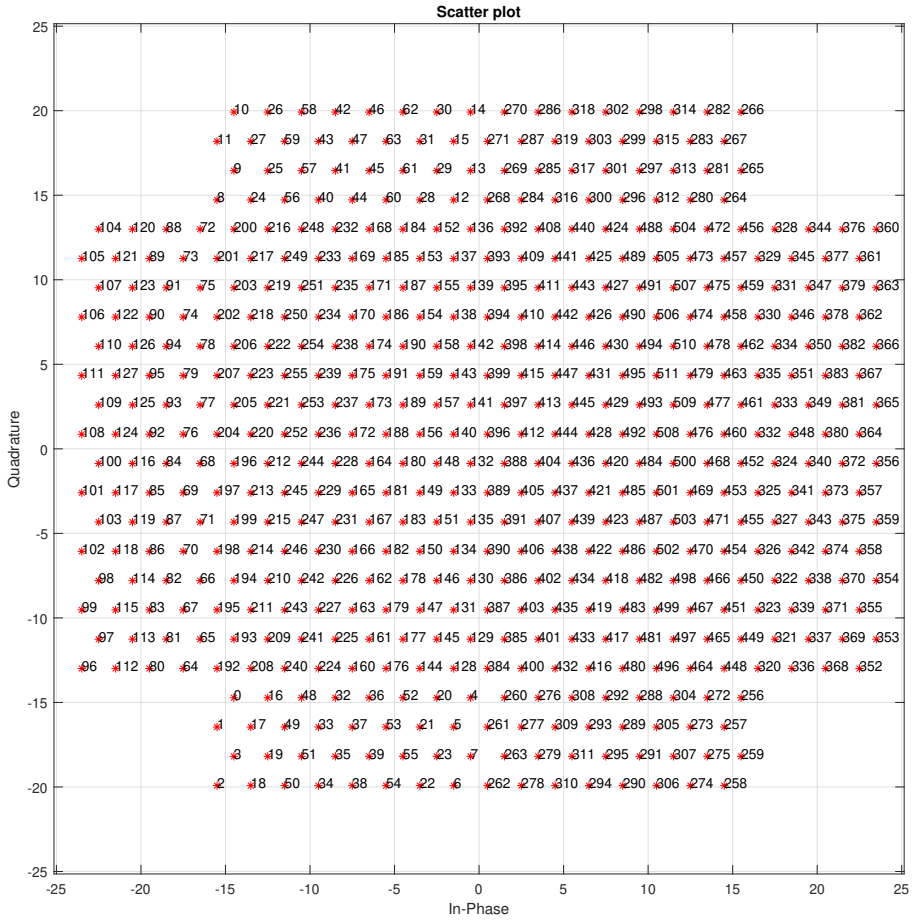
Figure 2

[1] Note that when n is odd $\lfloor n/2 \rfloor + 1 = \lceil n/2 \rceil$ and $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$



(f) Decimal code for n=7, M=128

Figure 2

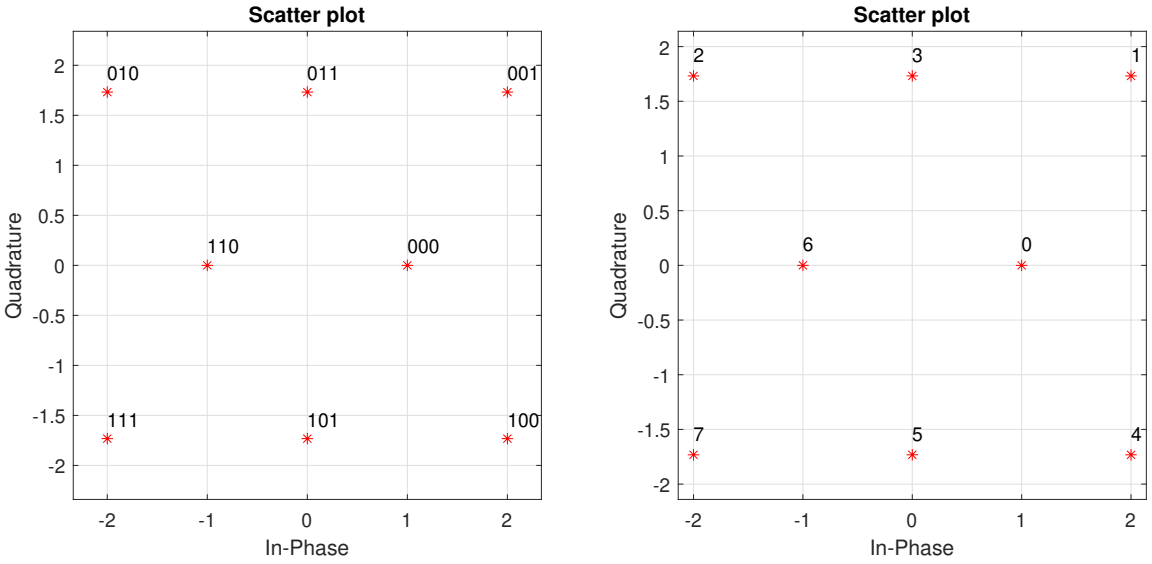


(g) Decimal code for n=9, M=512

Figure 2: Gray mapping of constellations of odd power

1.2.5 Gray mapping the constellation with order n=3

The Gray mapping in this case does not follow some pattern like before, and since there are only a couple of symbols, we hard coded them based on an optimal mapping and is shown in Figures 3h and 3i both in binary and decimal form.



(h) Binary

(i) Decimal

Figure 3: Gray mapping of constellation for n=3, M=8

1.2.6 Calculating the Gray code penalty Gp

The Gray code penalty is a useful characteristic of a bit mapping that represents the average bit difference between neighboring symbols. It was first introduced in Smith's paper [5] and is defined as follows:

$$G_p = \frac{1}{M} \sum_{i=0}^{M-1} G_{p,s_i} \tag{5}$$

$$G_{p,s_i} = \frac{1}{k} \sum_{j=1}^{k} b_{diff} \tag{6}$$

Where G_{p,s_i} is the average bit difference of the symbol s_i with its neighbors, k is the number of neighbors and b_{diff} is the number of bits that the symbol s_i differs from the j-th neighbor (which can be calculated with the *SetBits* function [2] after using XOR on the Gray codes of the symbols).

For HQAMs the maximum number of neighbors (Ns) is 6 and a minimum number of 2 (for the symbols in the corners) as confirmed in table 4. Since there are 2 dimensions and the maximum number of neighboring symbols is more than 4 (= 2 · number of dimensions) there can not be perfect Gray code mapping, which would mean $G_p = 1$. For high order constellation where most of the symbols are inside the constellation (because most of the symbols have 6 neighbors) the G_p converges asymptotically to 4/3, which is equal to G_{p,s_i} for most of the inside symbols of the constellation. For odd power constellations, some inside symbols (those that are in the perimeter of the 8 small squares with sides

[2] Using Brian Kernighan's Algorithm that was mentioned in 1.2.2

$a_{odd}/2$) have G_{p,s_i} higher than 4/3.
After quite some thinking, we derived a closed form for the G_p of even power constellations, which is very easy to evaluate and that was arithmetically verified.

$$G_{p,even} = \frac{\frac{4}{3}(a_{even}-2)^2 + \frac{147}{30}(a_{even}-2) + \frac{14}{3}}{M} \tag{7}$$

Using the closed form for the $G_{p,even}$ given in 7 we can take the limit as M approaches infinity and as mentioned above conclude that the $G_{p,even}$ approaches 4/3.
With similar reasoning, a closed form for G_p of odd power constellations can be derived. A closed form to evaluate the G_p is useful, especially for higher order constellations, since there is no need to model them and run computations for high number of symbols that would require a lot of time and hardware resources to run (a lot of RAM to store the symbol coordinates and their Gray code).

n	2	3	4	5	6	7	8	9	10
M	4	8	16	32	64	128	256	512	1024
G_p	1.1667	1.2750	1.2375	1.3885	1.2823	1.3635	1.3070	1.3516	1.3200

Table 2: The G_p values of our optimal Gray coding for n=2 to n=10

n	11	12	13	14	15	16	17	18	19	20
M	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
G_p	1.3432	1.3266	1.3385	1.3300	1.3360	1.3316	1.3347	1.3325	1.3340	1.3329

Table 3: The G_p values of our optimal Gray coding for n=11 to n=20

2 Symbol Detection

Due to AWGN (Additive white Gaussian noise), the receiver does not receive exactly the symbols sent but something close to them (depending on the SNR). To compensate with this, the receiver must decide which symbol was sent using the limited information given. The receiver in the general case must be MAPD (Maximum a-posteriori probability detector) but since each symbol is equally likely to have been sent the MAPD is equivalent to MLD (Maximum likelihood detector). There are multiple ways of implementing a detection method, each having their advantages and disadvantages. Those can be split into 3 basic categories, the time complexity of the method, the space complexity and the complexity of the idea. In practice, the time complexity is the most important that makes an algorithm stand out (given that it has reasonable space complexity).

2.1 Linear detection

The first method we implemented was the simplest. It uses the definition of MLD to decide which symbol was sent. To implement this, it checks the distance of the received symbol from all the symbols of the constellation. This has a time complexity of $\mathcal{O}(M)$ and since it does not require extra storage except from an array with the symbols of the constellation (which is the bare minimum) it has a space complexity of $\mathcal{O}(M)$.

To improve it a bit we used a condition that stops searching if the distance of the received symbol from the symbol of the current iteration is less than $d_{min}/2$. This improvement reduces the average execution time in half for decent SNR values. The improved method has average and worst time complexity of $\mathcal{O}(M)$ and best case of $\mathcal{O}(1)$ (when the symbol sent was the first symbol we check with probability of 1/M).

2.2 Our detection

To create a more sophisticated detection method, we first looked at an easier form of the problem, the QAM constellations. In QAM the decision boundaries are squares which makes the detection very easy to implement and understand. Basically, the received symbol is normalized with d_{min} and then with the proper rounding on each axis the received symbol can be mapped to the corresponding symbol of the constellation (it helps the symbols of the constellation to be stored in a 2D array).

2.2.1 Method of our detection

Having in mind all of the above, we did a lot of research and were quite optimistic from the start that there would be a fast way to detect the symbols since the decision regions are regular hexagons and the symbols of the constellation are created in a very particular way as mentioned in 1.1.
Our first thought was to utilize our knowledge of linear algebra to find the coordinates c_1 and c_2 which is quite easy to do, after offsetting the received symbol by $\mathbf{v_3}$ and multiplying with the inverse transformation matrix. For relatively high SNR values when the received symbol is close to the symbol sent, the values we get for c_1 and c_2 are expected to be very close to an integer, which are the ones of the symbol sent. However, there are some cases where using this approach would not work as it is and would result in a neighboring symbol.

Therefor, we decided that a slightly different approach is needed to counteract this problem. The main idea is to use 3 vectors at 120 degrees with each other that add up to zero. Then we calculate the coefficients for each vector such that the linear combination of them would give us the received symbol. To calculate them, we find the projections of the received symbol to those vectors. Using 3 vectors in a 2D plane to represent the points might be counterintuitive, since the third is redundant because it is a linear combination of the other two. After calculating those three coefficients, that with the way we calculated them sum up to zero, we round them. There is a need to add a simple yet quite effective condition to make an adjustment to the coefficients. First we find which of the three has the biggest difference before and after the rounding, and then we recalculate it using the others (it is the opposite of the sum of the other two after the rounding). After calculating those rounded coefficients, we only need to use two of them to find the indexes of the 2D array we stored the constellation symbols.

The above have a simple assumption that the decision boundaries are regular hexagons, which is true only for the internal symbols. For the outer symbols the decision boundaries are a superset of those hexagons so if the received symbol is outside those hexagons, which is unlikely for not very low SNR values, we then check if the neighboring hexagons contain any symbols. If there are symbols, it finds to which of those the received symbol is closest to. In the very rare scenario that there is not a symbol in the neighboring hexagons (which means the noise is very high, and the communication channel is basically useless) we used the slow linear detection.

The time complexity of our proposed detection algorithm is $\mathcal{O}(1)$ when we do not consider the very rare case that the symbols are very far from the constellation. For very low SNR values, since we call the linear detection algorithm, the time complexity becomes $\mathcal{O}(M)$. The space complexity is also $\mathcal{O}(M)$ since we need to store the symbols in a 2D array. The complexity of the idea is higher than before, which makes it harder to understand, implement and troubleshoot, but in the end it is worth

it since it improves the time beyond any expectations.

2.2.2 Verification of our detection

After proposing a detection method, it is necessary to make sure it is working as intended. Making sure it works for the constellation symbols is not enough, it should also work for any point of the complex plane. So that is exactly what we did, we made sure that for a very fine grid of points that was larger than the constellation it was able to correctly find the proper decision region it belonged to and map it to the corresponding symbol of the constellation. The question is how do we know if it decided correctly. This is done using the linear detection method, which is always correct. Then we plot the points that do not get mapped correctly, with some color code for each case. A blue dot is printed if the symbol detected by the fast method is not the same as the one detected by the linear method and those symbols are thought to be inside the hexagonal decision regions of the constellation. A magenta dot is printed if the symbol detected by the fast method corresponds to one of the extra cells the 2D array that we store the symbols has. Finally, a green dot is printed if the symbol detected by the fast method is outside the 2D array that we store the symbols.
In Figure 4 we can see that there was an offset error which can also be seen in Figure 5. It was very useful to be able to visualize those errors, since it made understanding and fixing them much easier.

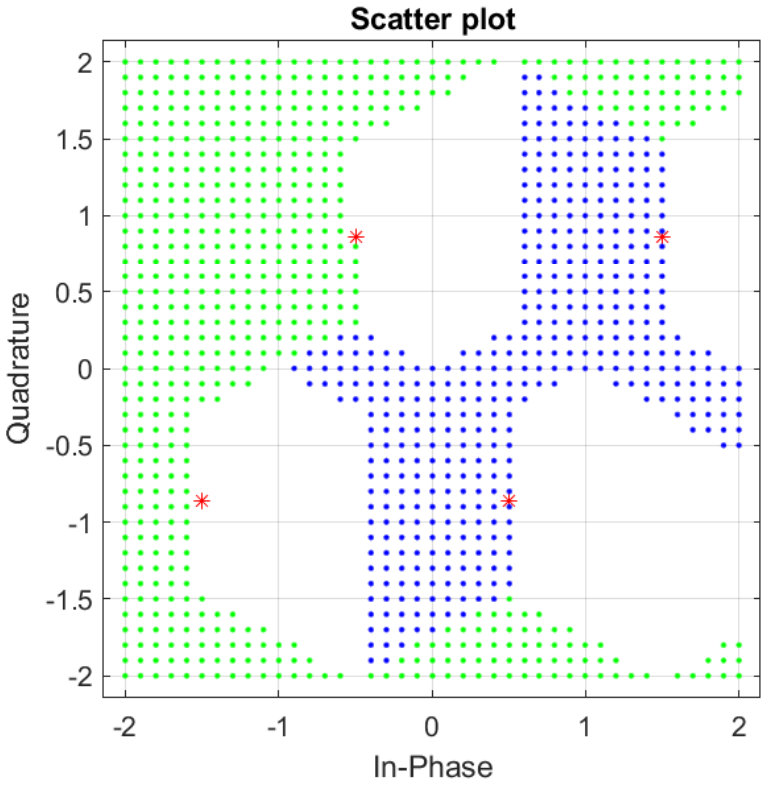


Figure 4: n=2, M=4

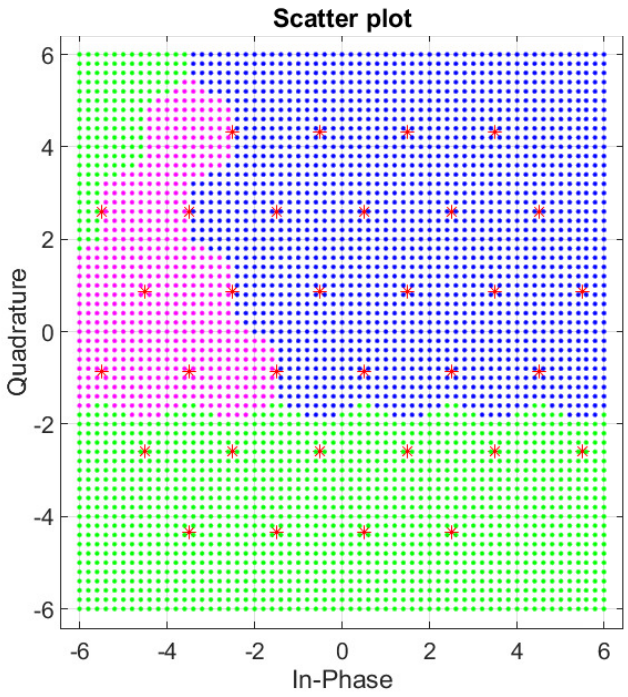


Figure 5: n=5, M=32

After fixing the offset problems, which was done using an expression that takes into consideration the order of the constellation (not hard coded), then the final form of the detection method was very close and the blue dots were almost gone as it can be seen in Figure 6, Figure 7 and Figure 8.

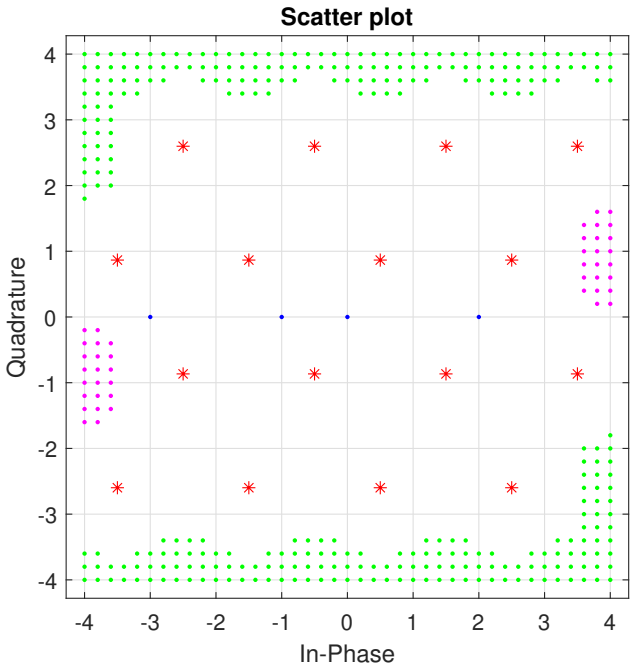


Figure 6: n=4, M=16

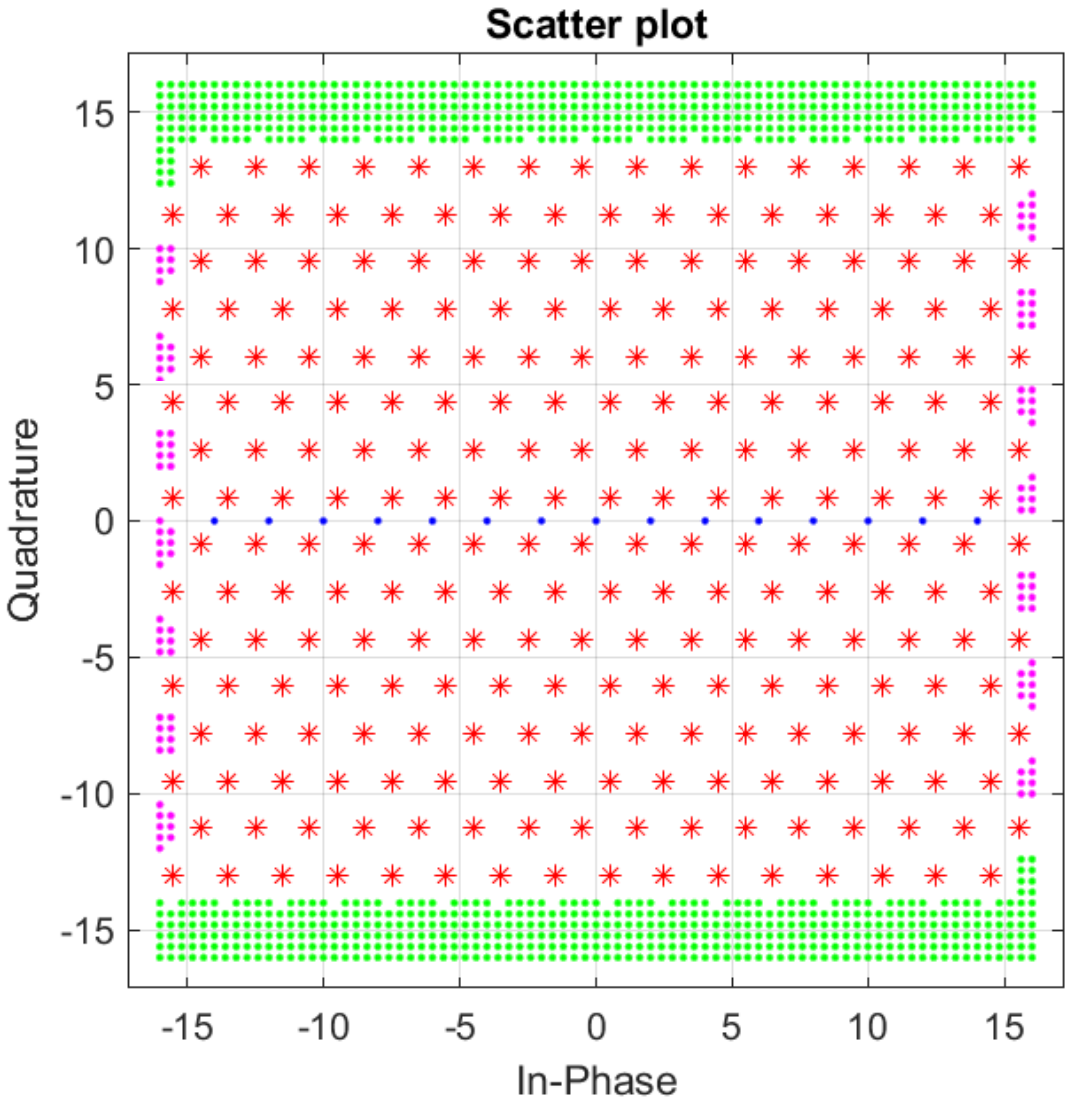


Figure 7: n=8, M=256

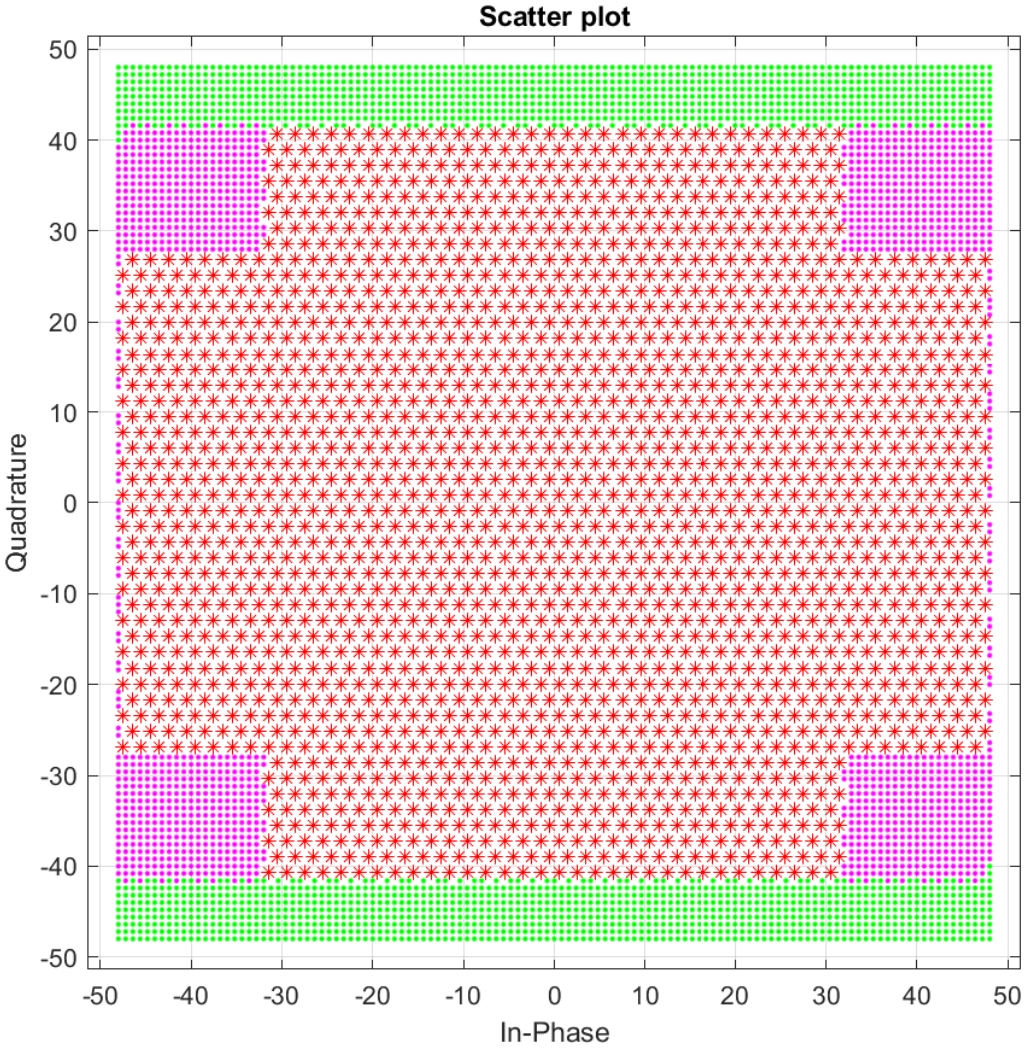


Figure 8: n=11, M=2048

After seeing those results which were expected given the state of the detection method at that point in time, it was time to fix the detection for the points that were outside the hexagonal decision regions. As mentioned in 2.2.1 the idea was to check all neighboring hexagons in the hopes that the symbol we were looking for was in it (which is very likely when the SNR in not very low). Although, there are still points that were not mapped, so if all the previous do not work we use the linear detection method.

In Figure 9 and Figure 10 we can see a lot of blue dots which might be alarming at first but at closer inspection we notice that those dots are in the boundaries of the decision regions and there is more than one symbol that it can be mapped to. This leads to some disagreement between different detection methods, but it is not important, since it is highly unlikely for a received symbol to be exactly on those boundaries, and it does not matter to which symbol those points are mapped.

Finally, after fixing all the problems mentioned above, the final detection method arose. It can detect all the points of the 2D plane correctly as the MLD would but with much much better time complexity.

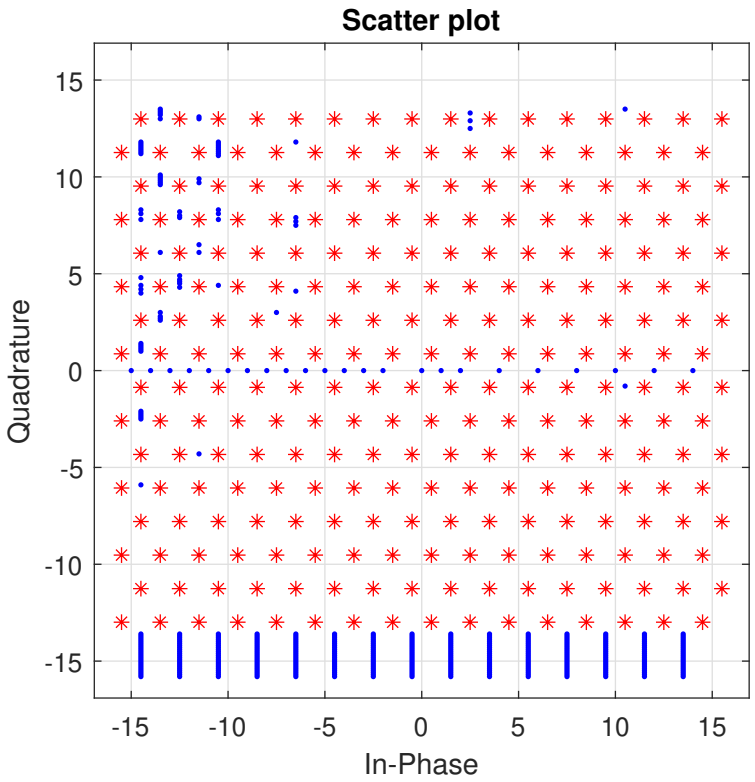


Figure 9: n=8, M=256

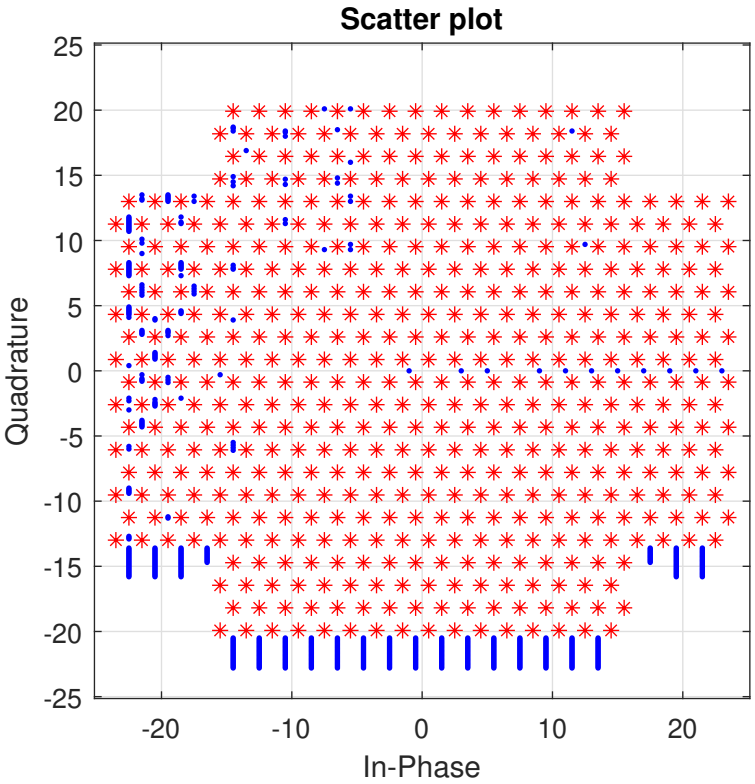


Figure 10: n=9, M=512

3 Simulations

3.1 Simulations key points

The target goal for the SEP (Symbol Error Probability) is to reach as low as 10^{-k} , where k is any desired positive integer. In our case was at least 10^{-5} but we opted for at least 10^{-7}. This requires the number of symbols sent (NOS) to be big enough to be able to produce the desired SEP. Since the SEP is the number of wrongly detected symbols due to noise divided by the NOS we conclude that the NOS must be at least 10^k.

In our simulations, we wanted to be able to run them in parts and combine the final results because it would be more manageable to run them on multiple computers, for short periods of time, for different NOS each time. To achieve this, we had to save the results of each time we ran the simulations and add it to the previous results. We used a matrix to save them that stored multiple data that gives us the ability to extract more information after some data analysis. In particular, we used a 4D matrix that stores for each order of constellation and SNR value and for each time we ran the simulations how many symbols and bits were detected wrong (due to noise), the NOS, the time it took to simulate, who ran them, with which algorithm and when.

An important subtle detail that makes combining the results accurate is that instead of saving the SEP and BER for each time we run a simulation, which would be a double, we save an integer which is how many symbols and bits were detected wrong and the NOS so that later after adding those up for different runs the final results do not have multiple division errors stacking up (and also needing a proper weighted sum).
Similarly with before, to calculate the G_p of the constellation and not have multiple division errors stacking up for each symbol, we multiply with the Minimum Common Multiple of the possible values of the divisor k in the G_{p,s_i} in equation 6, which is equal to 60. This way the $60G_{p,s_i}$ is going to be an integer, and then we divide with $60M$ in the end.

Another optimization strategy that we used for the implementation of 2.2.1 is the *try catch* statement. It is used instead of the *if-else* statement when we expect the number of times it would fall in the *else* to be just a few. When the SEP is not very big, this is a quite effective way to get a small but after all significant improvement [3].

Last but not least, a vital optimization that has a significant performance boost when running the simulations is parallelizing them. Implementing this after some extensive research was relatively easy because, due to the nature of the problem and the structure of our program, it was ideal for parallelization. In Matlab this was implemented using the command *parfor* instead of a regular *for* loop[4]. The speed up varies from one computer to another, since it depends on the number of cores each CPU has. The maximum expected speed up is as the number of cores when all of the code can be parallelized. In our case the simulation is by its nature parallelizable, so we were expecting the maximum speed up, since our computers had 8 cores, to be 8 times faster [5]. In reality, it can not be exactly 8 but close to 8 because it has to send information between each process, which in our case was very little at the start when we sent the information for the constellation and at the end when we collect all the data from each process. After considering all of the above, the approximately 7.6 times speed up that we got was very significant.

[3] When using the *try catch* statement, someone should be quite careful since it would catch not only intentional errors but also unintentional ones. A way to limit this is to specify which error types to catch.

[4] Not every *for* loop can be replaced with a *parfor*, it has to satisfy some very important qualifications.

[5] It is important to note that, when Matlab starts the parallel pool it takes several seconds to minutes unless it was already started in a previous run or manually (it also shuts it down automatically).

3.2 Matlab code and files

After multiple months of intriguing coding for this research project, the following Matlab files were created.

1. LinearGrayCode.m
2. SetBits.m
3. GrayCodeAndSetBitsTesting.m
4. RegularHQAM.m
5. EvenGrayMappingTest.m
6. DetectionLinearSlow.m
7. DetectionLinearSlowTesting.m
8. DetectionLinearSlow2D.m
9. DetectionFast.m
10. DetectionFastTesting.m
11. DetectionTesting.m
12. Main_Simulation.m
13. Visual_Simulation.m
14. Final_SEP_BER_graph_of_all_data.m
15. Bounds_and_More_graphs.m

Furthermore, we also created a few data files that store the results of multiple simulation runs.

1. MultiRunsValuesForSEP&BER_Nick_Old.mat
2. MultiRunsValuesForSEP&BER_Nick.mat
3. MultiRunsValuesForSEP&BER_Katerina.mat
4. GpValues.mat

3.2.1 LinearGrayCode.m

It is a function that implements the one dimensional Gray Code mentioned in 1.2.1. The input is an integer n which is the number of bits the binary code has. It was implemented using 'int32' (for memory saving, instead of 'int64', which can be easily changed in case we want to simulate even bigger order constellations), thus the maximum value of n is 31 because the first bit represents -2^{31}. The function returns an array of length 2^n which is the Gray code with n bits. The important section of the code is shown in 1.2.1.

3.2.2 SetBits.m

It is a function that counts and returns the number of set bits (number of bits that are 1) in a given integer n. As mentioned above in 1.2.2, we use the Brian Kernighan's algorithm.

3.2.3 GrayCodeAndSetBitsTesting.m

This is a script which tests that the Gray code created by 3.2.1 is correct. That means that neighboring numbers of the 1D array have exactly 1 bit difference, as thoroughly explained in 1.2.2. The main code is shown in 1.2.2. This script requires the functions LinearGrayCode.m and SetBits.m.

3.2.4 RegularHQAM.m

This function creates all the regular HQAM constellations and makes the bit mapping, as explained in section 1. It has as input the order of the constellation n and the minimum distance between neighboring symbols. It returns a 2D array with the symbol coordinates as complex numbers, a 2D array with the Gray code of the corresponding symbols, a vector with all the symbol coordinates (as complex numbers), a vector with the Gray code of the corresponding symbols and the average energy of the constellation. The maximum value of n that can be used is 31 as explained in 3.2.1 (with some changes it can be increased to 63). The function requires the LinearGrayCode.m.

3.2.5 EvenGrayMappingTest.m

This script calculates the G_p value for every order of constellation (in a selected range) and the values in table 4. The calculation is done using the constellations created in RegularHQAM.m and for the even power constellations it also calculates the G_p using the closed formula 7 (both values were exactly the same without a single decimal point difference). The values are stored in *GpValues.mat*. There is the option to plot the constellations with the Gray code of the symbols in binary or decimal form by changing the boolean variable named *plot* to true. It requires the RegularHQAM.m and the SetBits.m.

3.2.6 DetectionLinearSlow.m

This function implements the Linear detection. It has input the constellation vector and the Gray code vector as created by RegularHQAM.m, the coordinates of the symbol received (with AWGN noise), the Gray code of the symbol sent, the number of symbols M and the minimum distance. It returns two integers, the SER, which is 0 or 1 depending on whether the symbol was detected correctly with MLD, and the BER, which is the number of bits that were wrong. It requires the SetBits.m function.

3.2.7 DetectionLinearSlowTesting.m

This function is a variation of the DetectionLinearSlow.m function. The only addition is that this function also returns the Gray code of the symbol detected. This could be the only function used instead of DetectionLinearSlow.m since it does something more, but exactly because it does something more it is a bit slower (by a few clock cycles) and when we run the simulations for tens to hundreds of billions of symbols every clock cycle counts.

3.2.8 DetectionLinearSlow2D.m

This function implements the linear detection using the 2D array with the symbol coordinates instead of the vector. This is done because we didn't want to have extra arguments in the DetectionFast.m since it would only be called for low SNR values. It returns the same output as the DetectionLinearSlowTesting.m and has input arguments similar with it, with the main difference that instead of vectors it has the two 2D arrays with the symbol coordinates and their Gray codes.

3.2.9 DetectionFast.m

This function implements our detection algorithm, as mentioned in extensive detail in 2.2.1. The input arguments are the 2D arrays, the information about the symbol sent, the minimum distance and the helping variables p and a. The input p takes one of the following values 1,2,3 depending on the order of the constellation and a corresponds to a_{even} and a_{odd}. The output are the SER and BER which are the same as in DetectionLinearSlow.m. It requires the functions DetectionLinearSlow2D.m and SetBits.m.

3.2.10 DetectionFastTesting.m

This function is a variation of the DetectionFast.m function. It has the same differences as the DetectionLinearSlowTesting.m has from DetectionLinearSlow.m for the exact same reasons.

3.2.11 DetectionTesting.m

This script implements the concepts explained in detail in 2.2.2. One of the main parameters that can be changed in this file is the variable *step*, which is how fine the grid of the test points is. An important note here is that when deciding the *step* the *distance* should also be taken into account. It requires the functions RegularHQAM.m, DetectionFastTesting.m and DetectionLinearSlowTesting.m.

3.2.12 Main_Simulation.m

This script is the heart of the Simulations. It runs simulations for a wide range of constellation orders and for many values of SNR. Some of the main parameters that can be changed are the *totalSymbolsSent* which relates to NOS as NOS = $2^{totalSymbolsSent}$, the save boolean *saveResults* and the name of the file that it stores the data *nameOfThePointsFile*. Also, by changing some lines of code where the detection happens, it is very easy to change the detection algorithm. When it is changed and the results are to be saved it is important to remember to also change the information stored about that data, like with which algorithm it was run and an id of the computer it was simulated in. Those could have been implemented with code to make the changes easier, but it would make it a bit slower to simulate. It requires the function RegularHQAM.m and the various detection algorithms as DetectionLinearSlow.m and DetectionFast.m.

3.2.13 Visual_Simulation.m

This script simulates the symbols passing through the channel (adding noise) and then plots them one by one on a diagram with the constellation. It also produces a diagram of the constellation. It is a nice to have that does not interact with the rest of the files in any meaningful way. Some of the main parameters that can be changed are *n* (the power of the constellation), the *SNR* and *NOS* which is how many symbols will be simulated. It requires the function RegularHQAM.m.

3.2.14 Final_SEP_BER_graph_of_all_data.m

This script combines all the data of the simulations that are stored in *mat* files. It uses an array called *namesOfPointsFiles* with the names of the files that contain the data (if more files with data are to be included, their names should be added to the array). It also produces two graphs from the final combined data, one is the SEP versus received SNR in Figure 11 and the other is BER versus received SNR in Figure 12. It requires that the data in the files are stored appropriately.

3.2.15 Bounds_and_More_graphs.m

This script at the start calls the Final_SEP_BER_graph_of_all_data.m script (which, as already mentioned, combines all the data and produces two graphs for SEP and BER). It produces the rest of the

diagrams that are useful for data analysis, like relative and absolute errors and average time. It only requires the Final_SEP_BER_graph_of_all_data.m script.

4 Data Analysis and Conclusions

The final graphs that are produced from Bounds_and_More_graphs.m are shown and explained in this section.
In Figure 11 the simulated SEP versus received SNR can be seen. As expected, the SEP is decreasing as the SNR increases and agrees with the results in [1].

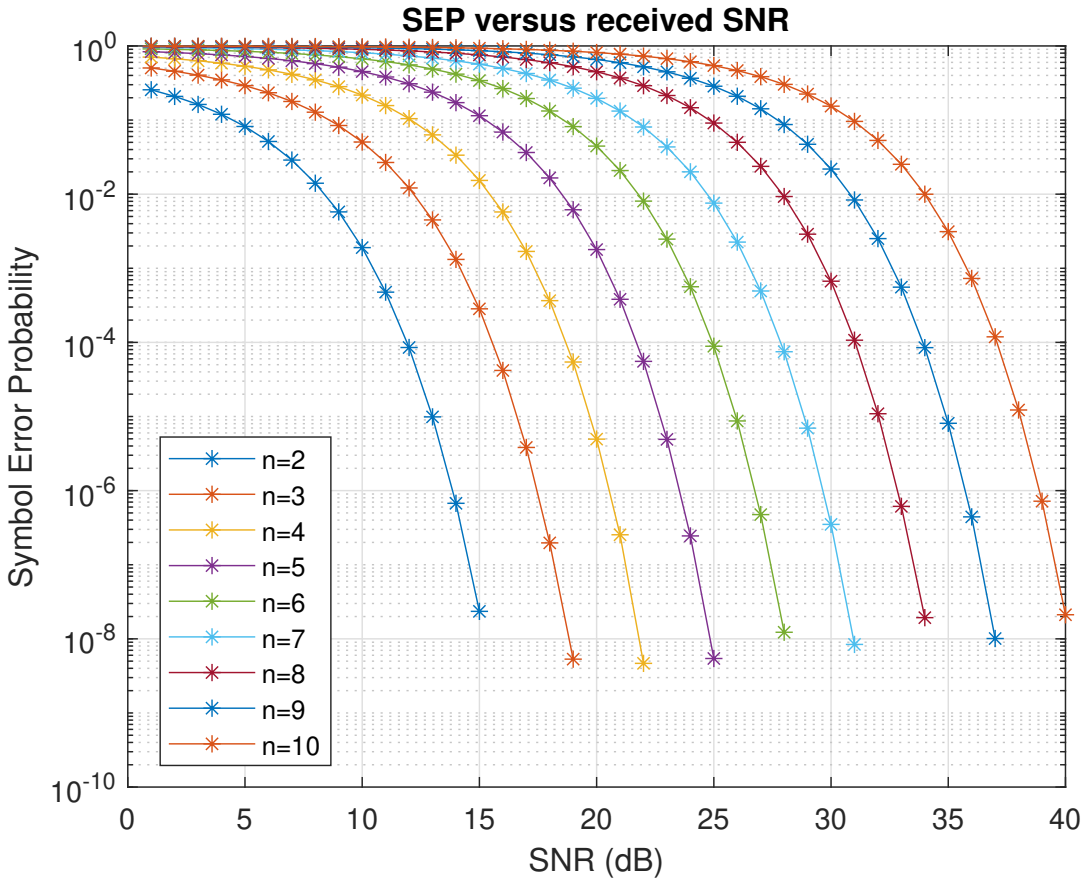


Figure 11

In Figure 12 the simulated BER versus received SNR can be seen. As expected, the BER is decreasing as the SNR increases and the values are lower than the corresponding SEP values.

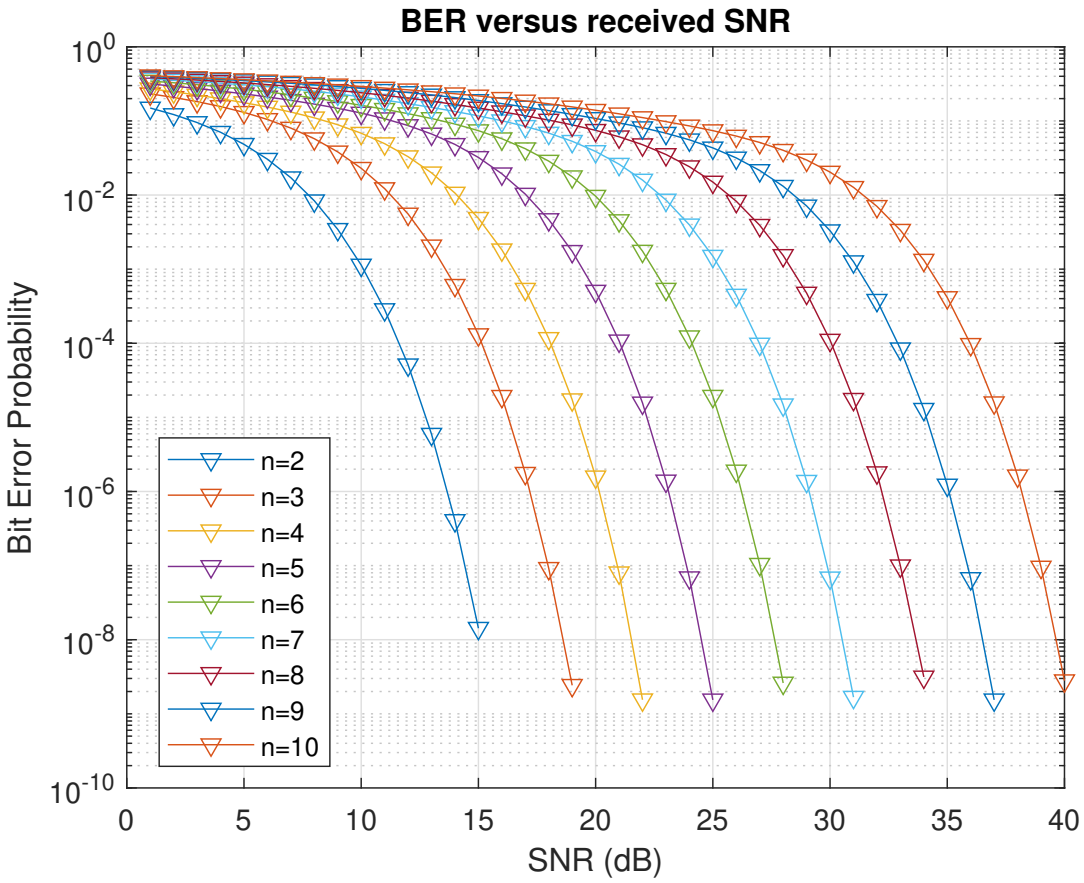


Figure 12

Figure 13 shows the simulated SEP, the SEP approximation and upper bounds versus received SNR. The upper bounds and the approximations were calculated using the proposed method and parameter values in [1] and are indeed upper bounds. This will be more thoroughly examined in the following figures. As for the approximation for the SEP, it can be seen that the values are also very close to the simulation results. For low SNR the approximation values are above the simulated SEP and at some point cross them and end up below them. Note that for n=10 M=1024 the approximation crosses the simulated results 3 times, which is not true if different parameters are used.

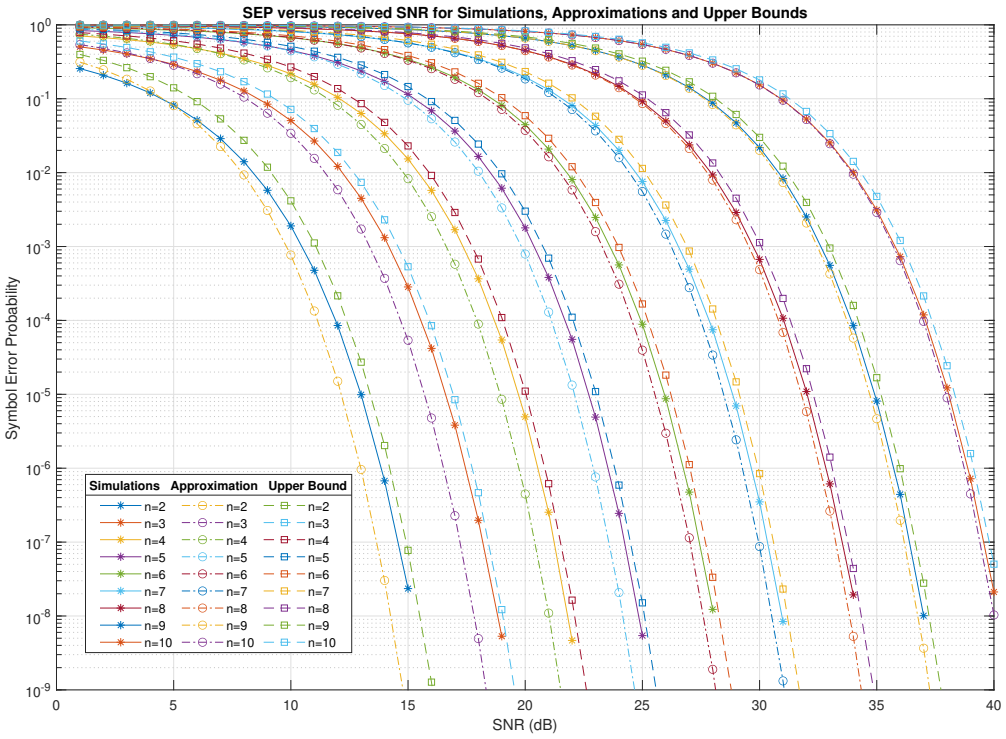


Figure 13

In Figures 14 and 15 the absolute error of the upper bounds relative to the simulated SEP can be seen. In Figures 16 and 17 the relative error of the upper bounds compared to the simulated values for SEP are showed. In Figures 18 and 19 the relative error of the SEP approximation compared to the simulated values for SEP are showed. For the approximations and the upper bounds in Figures 14, 16, 18, we used the values for the parameters k and b[6] that were suggested in [1]. For the approximations and the upper bounds in Figures 15, 17, 19, we used the values for the parameters k that [1] suggested with the values for b that we found, that can be seen in table 4.

[6]The parameter b is the number of external symbols of the constellation.

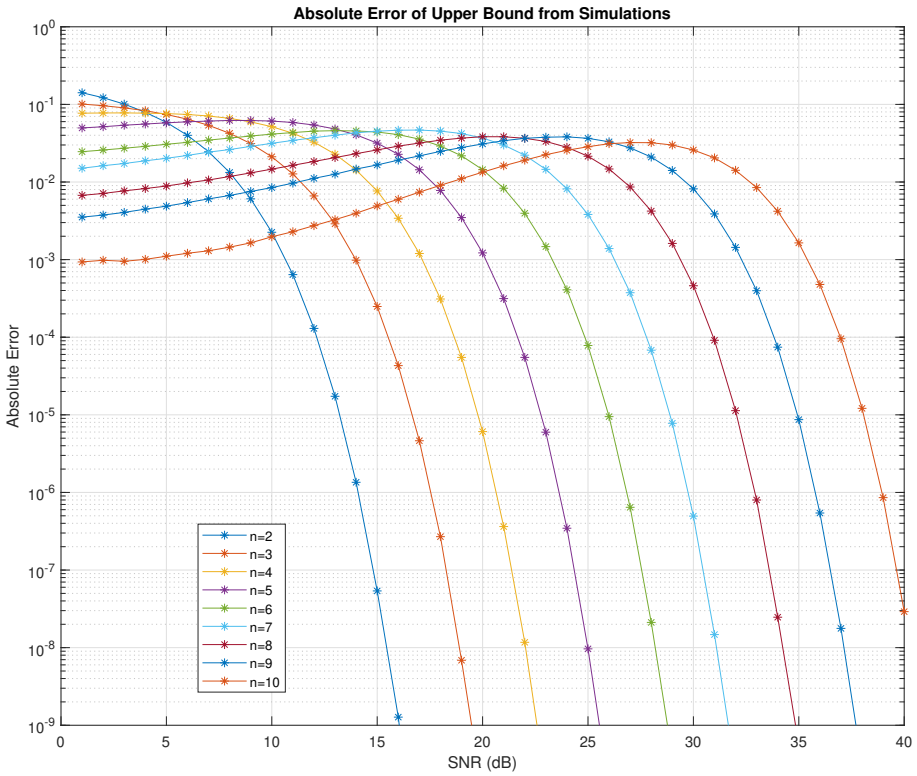


Figure 14: With b values proposed in [1].

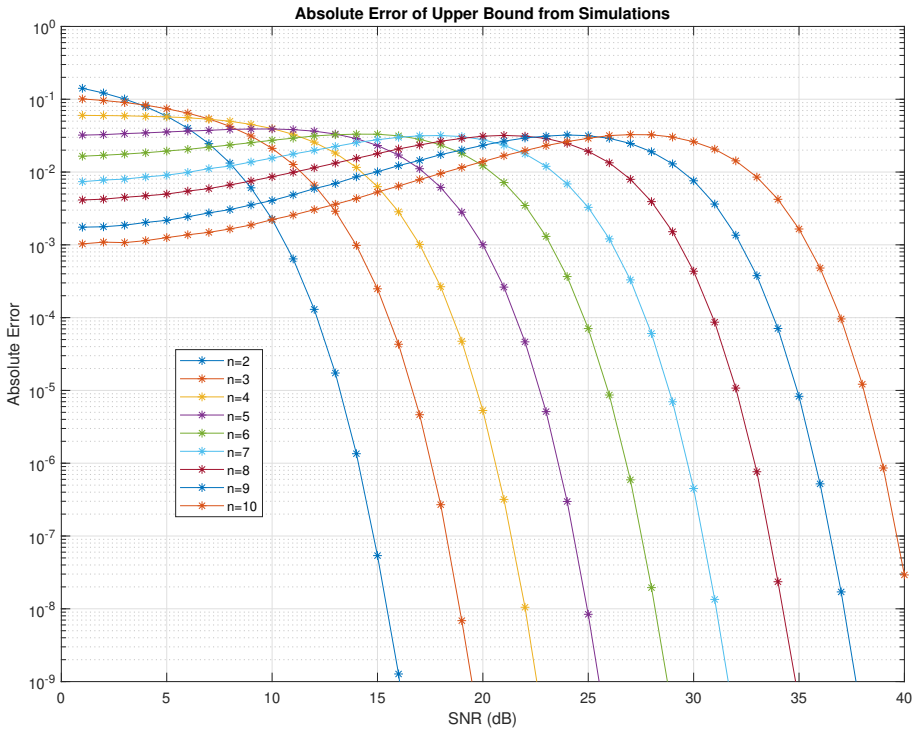


Figure 15: With b values proposed in table 4.

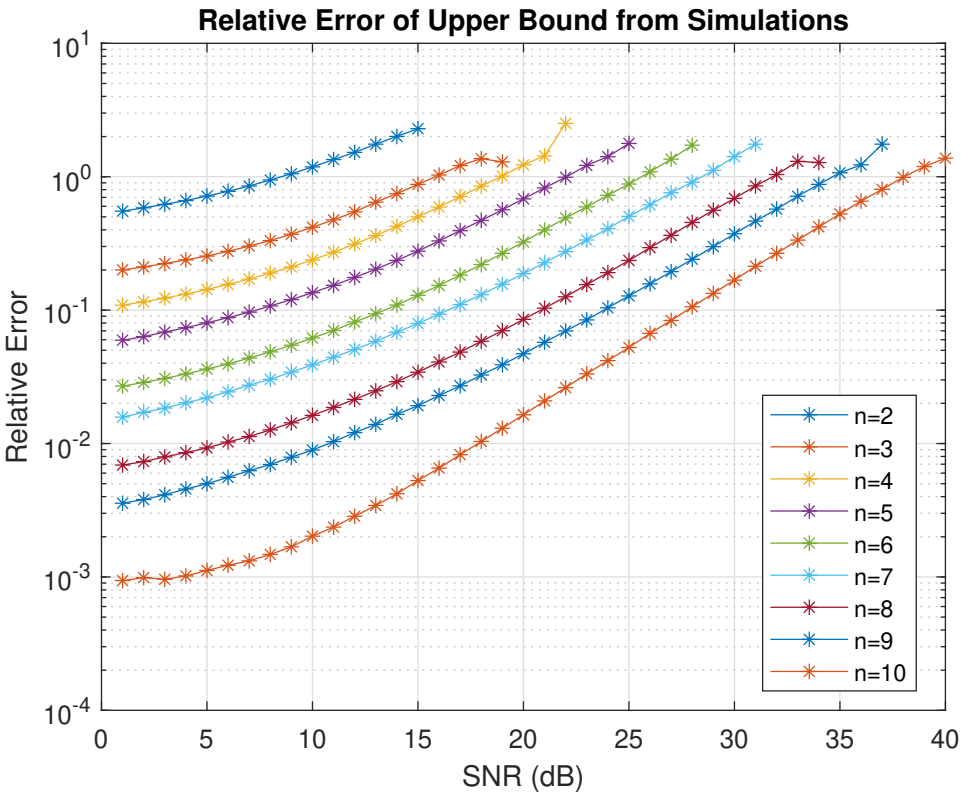


Figure 16: With b values proposed in [1].

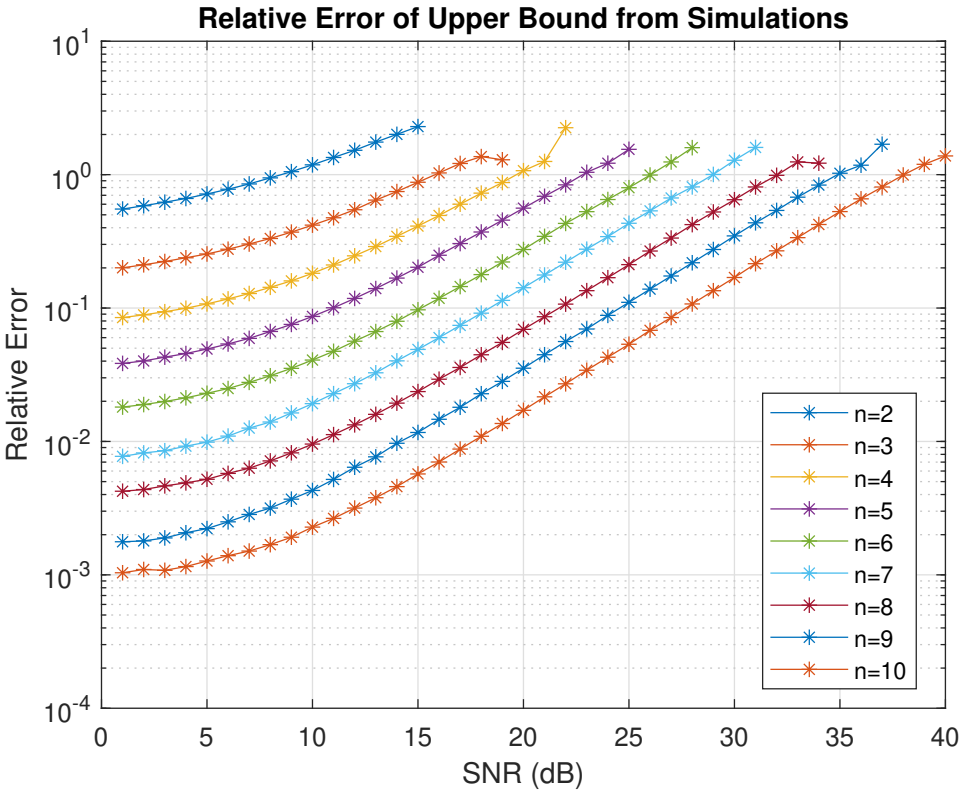


Figure 17: With b values proposed in table 4.

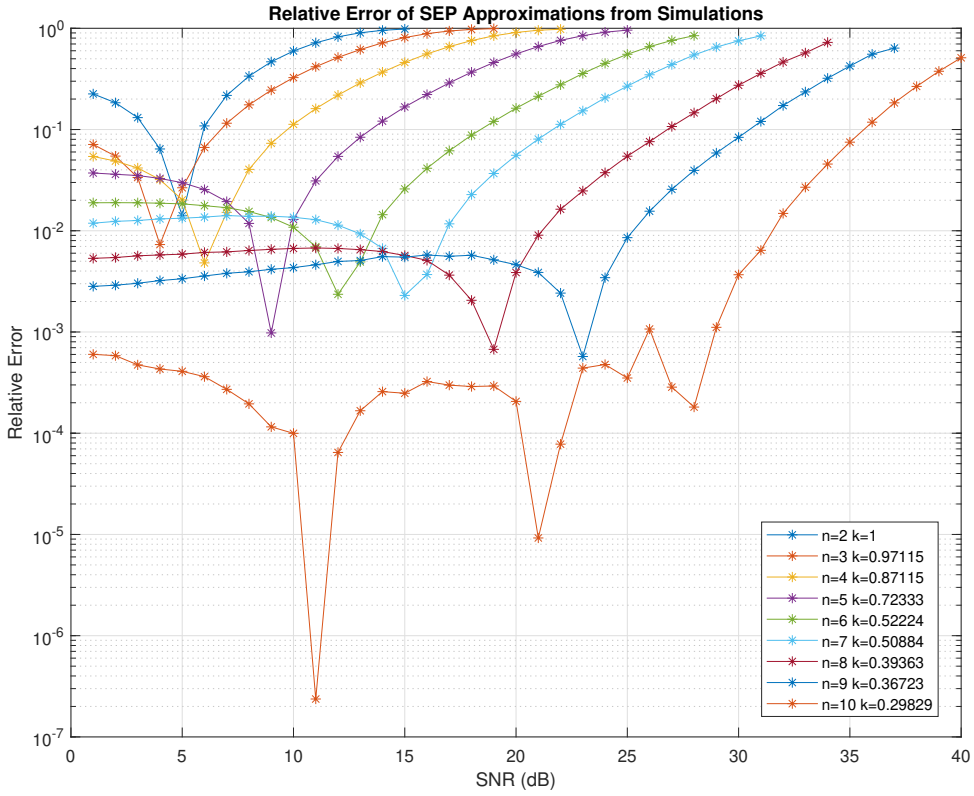


Figure 18: With b values proposed in [1].

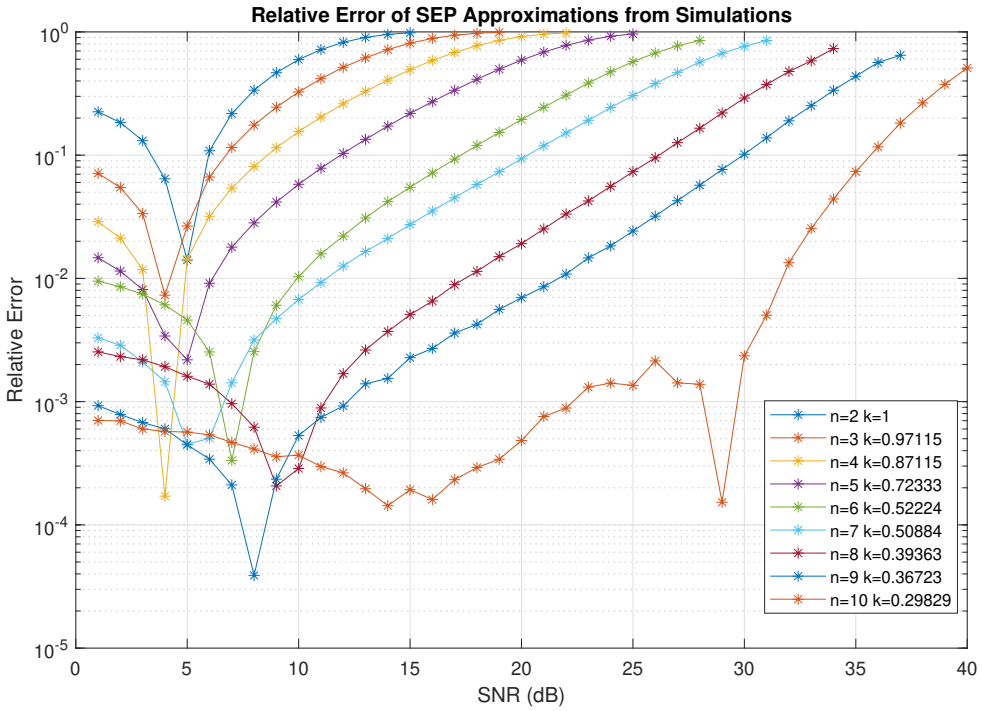


Figure 19: With b values proposed in table 4.

Foremost, in Figures 14 and 15 we can see that all values are positive (as expected) since the upper bounds are above the simulated SEP values (if they were not then there would be a warning when trying to plot a negative number in a logarithmic scale and there would be points missing from the diagram). We can not conclude from these diagrams how tight the upper bounds are, since the SEP values are significantly decreasing as the SNR increases. To estimate how tight the upper bounds are we need Figures 16 and 17. The important takeaway from those graphs is that the bounds are better for higher order constellations, and another obvious takeaway is that as the SNR increases, the relative error of the upper bounds exponentially increases. From Figures 18 and 19 we can clearly observe that the SEP approximation and the simulated results cross exactly one time for most orders of the constellations in figure 18, since there is one dip in the curve (close to the point they cross). But for n=10 M=1024 there are 3 dips, which as mentioned before are because the curves cross 3 times in Figure 13. In Figure 19 for all orders of constellations the SEP approximation and the simulated results cross exactly one time. It can also be concluded that the approximations are better for higher order constellations (since the curves are lower).
Comparing the results from using different parameters of the number of outer symbols of the constellations (the parameter b), we can clearly see some significant differences between the above diagrams. For the upper bounds using the values in table 4, we can conclude that the absolute and relative error is more evenly distributed for differed order constellations. Also, the relative error, especially for low SNR values. Thus, the upper bounds are tighter. For the approximations, the thing that stands out the most is that the dip in the curves (which is close to the point it crosses the curve for the simulated values of SEP) are for lower SNR and that there is only one dip in the curve for n=10, which means that is crosses the curve only one time.

In Figure 20 the simulated BER values and the approximated ones from the SEP and G_p can be seen. To approximate the BER we used the expression $BER = \frac{SEP G_p}{n}$. In Figure 21 the relative error of the BER approximation from the simulated values can be seen.
It is quite obvious that the approximation is very good, especially for higher values of SNR since in Figure 20 the points of the simulations (shown with the asterisk) are inside the points of the approximation (shown with the circle).
With Figure 21 we can study the approximation in more detail. We can notice that for low SNR the higher the order of the constellation (which means more bits corresponding to each symbol) the relative error is bigger. For n=2 M=4 is a very special case since the G_p and G_{p,s_i} are equal. So when an error occurs, it is accurately represented. Also, another thing to notice, that is not a coincidence, is that the curves look very similar with the SEP curves since as the SEP decreases the G_p becomes more relevant and at some point the curves have the minimum relative error. As expected for high SNR the relative error is lower. The values of each curve for high SNR have some deviation from the expected results because the number of symbols that are detected wrong due to noise in the simulations is very small (in some cases even less than 100) despite the NOS being in the order of billions for some points in the graph. Those last data points were a lot worse after some initial simulations, so we made the proper adjustments to the code and run way more simulations (for a few hours) than we had before, and the final result is way better than in early results.

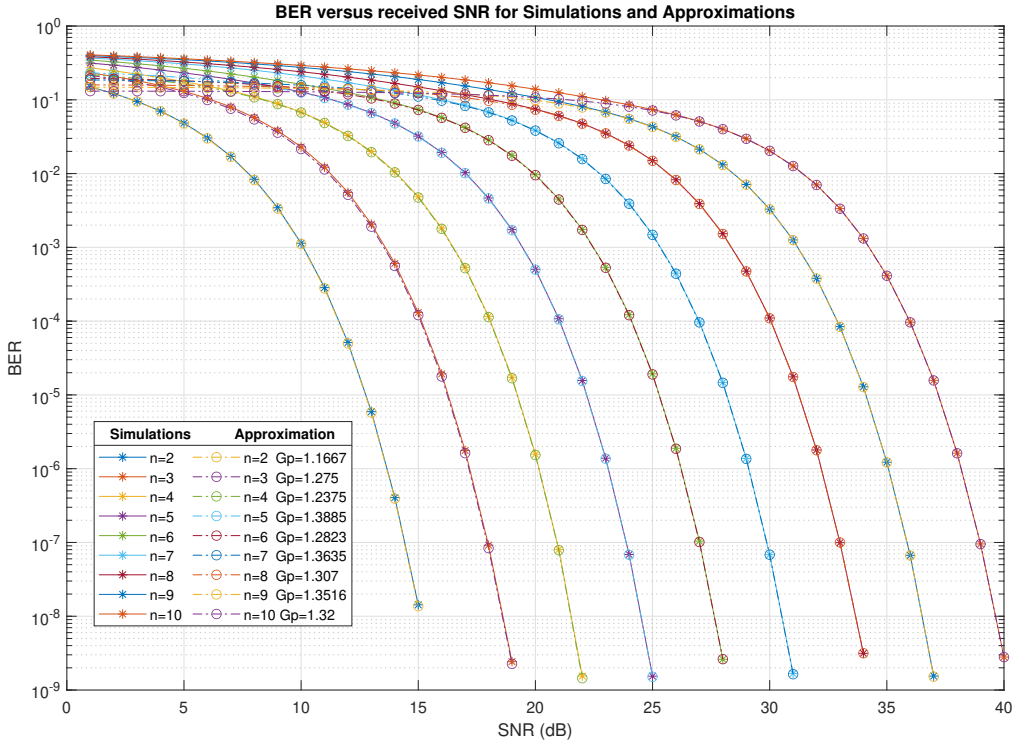


Figure 20

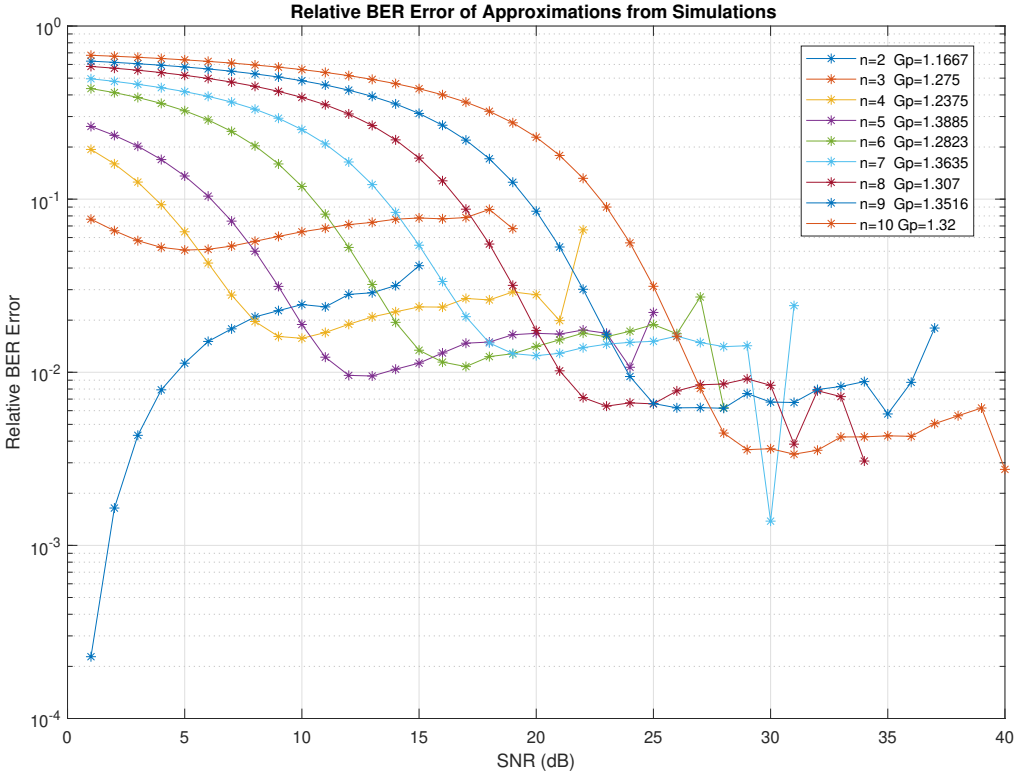


Figure 21

In Figure 22 is showed the average time it took to detect one symbol (with noise) with the Linear detection across all the simulations from multiple computers.
In Figure 23 is showed the average time it took to detect one symbol (with noise) with Our detection across all the simulations from multiple computers.
For the Linear detection we can see the average time remains relatively the same and depends mostly on the order of the constellation. For low SNR values (since we used a variation instead of a simple linear search that when the symbol received is closer than $d_{min}/2$ it stops searching) the time per symbol is higher since it has to find the distance from all symbols. This is not true for the high SNR because it is very probable the received symbol to be very close to the one sent. Also, as the order gets higher, the time it takes to detect a symbol increases since it has to check more symbols. For low order constellations n = 2,3,4 the time is very close since the difference between 4,8,16 is small and the average time is mainly affected by the other parameters.
For Our detection our main goal was to achieve fast detection speeds for high SNR values where the telecommunication systems are expected to operate. This is clearly achieved since the average time is an order of magnitude lower than the Linear detection. Also, as expected for low SNR values, since we use a variation of the Linear detection which actually is a bit slower, the final time per symbol is higher.
A very interesting conclusion from Figure 23 is that for **higher** order constellations, the average time to detect a symbol gets **lower**, which is counterintuitive and not true for most detection algorithms.

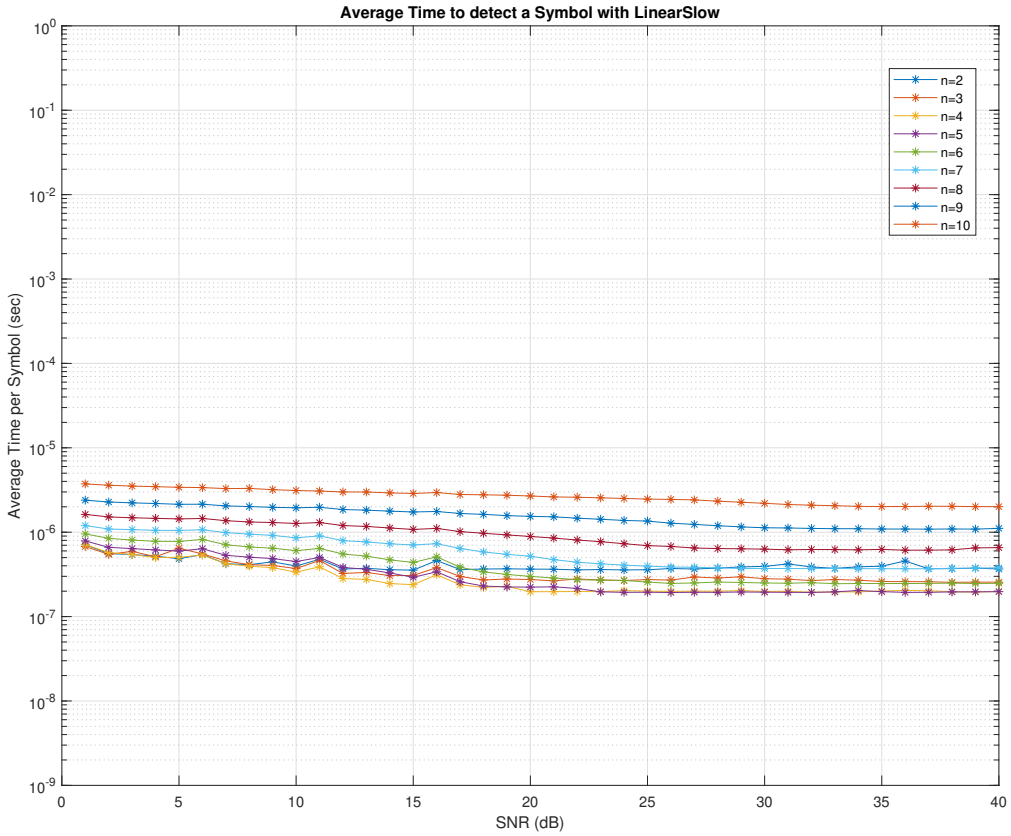


Figure 22

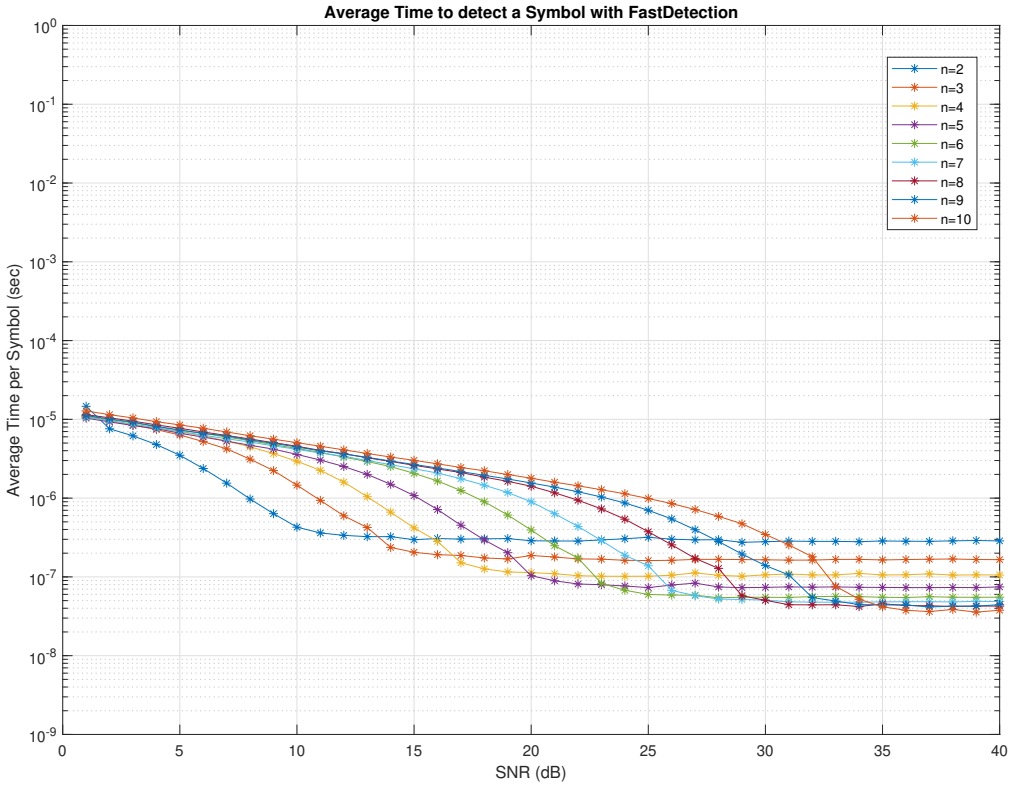


Figure 23

5 Further Improvements

Taking into account all the above data, we conclude that it would be beneficial to improve the performance of our detection algorithm for low SNR values to make simulating faster. An easy and effective improvement would be to use a different function for the not so rare case in simulations (for low SNR) than the variation of the Linear detection. This could be done using a better variation that does a linear search only on the outer symbols of the constellation (since if it was inside the constellation, it would be detected in $\mathcal{O}(1)$ by the main part of the algorithm). For the constellations we calculated how many symbols have 1 up to 6 neighbors, and how many symbols have less than 6 (those are the outer symbols of the constellation) and are showed in table 4. The speed up in this case would be significant as the constellation order increases, since it would skip a grate portion of the calculations (the symbols that have 6 neighbors).

To further improve its performance, we can choose a different algorithm. A variation of the algorithm that does binary search would be faster, since the $\mathcal{O}(log_2 M)$ is better than the $\mathcal{O}(\sqrt{M})$ that the above linear search variation has.

We also had multiple other ideas that we did not fully form and implement, some because we did not have the time and some because they were not very appealing compared to the detection algorithm we proposed. The main idea for one of them is to split the plane in rectangles. By making their size comparable with the minimum distance between symbols, we can make it such that each rectangle

contains points that would be mapped to at most 3 symbols. To implement it, we store in an array which symbols could correspond to the points inside each rectangle. When we receive a symbol we first find in which rectangle is inside, something very easy to do in $\mathcal{O}(1)$ complexity, and then we check with which one of the symbols that correspond to that rectangle is closest again with $\mathcal{O}(1)$ complexity. This approach is ideal for QAM constellations, but for HQAM constellations the space complexity to make it work is quite significant, and can vary drastically depending on the size of the rectangles we choose. The smaller the rectangles used, it would be more probable the received symbol to be inside a rectangle that corresponds to only one symbol, which would make a bit faster the second part but with way more storage requirements.

Another idea is an algorithm that its main objective is to find SEP values using Monte Carlo simulations. It is not a detection algorithm, but an algorithm that checks if the symbol would be detected correctly using MLD and the information of what symbol was sent. This works by checking if the symbol received is closer to the symbol sent (which is information used as the goal is to find the SEP, not a detection algorithm) compared to all of its neighbors. This way, we can deduce if it would be correctly detected by any detection algorithm using MLD. It is very easy to implement and has a time complexity $\mathcal{O}(1)$. The down side of this is that it does not calculate the BER and this is the main reason we did not use it.

Ns\n	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	0
2	0	2	4	2	2	2	4	2	4	2
3	0	2	0	4	6	8	10	16	22	32
4	0	0	2	4	6	12	16	28	40	60
5	0	0	2	2	4	6	12	14	24	30
6	0	0	0	4	14	36	86	196	422	900
<6	0	4	8	12	18	28	42	60	90	124

Table 4: Number of symbols and how many neighboring symbols they have.

Despite not modeling the irregular HQAM constellations, we thought a lot about the minor additions that would need to be made to our proposed detection algorithm to work for them as well. The changes are basically only two. The first part needed to change is the offset we reverse, such that when the detection takes place a symbol is at the origin of the complex plane. The second change is that we would either need to find a closed formula that works for every constellation order for the offsets, that tells us the indexes the detected symbol is stored or using an easier, more practical approach to calibrate those values with a few test symbols (theoretically one is enough) to find those offsets for each order. Another important note is that the algorithms that calculate the G_p values, the values shown in table 4 and the Linear detection, which are the majority of the code that uses constellations to work, will not need changes to work for irregular HQAM constellations as well.

References

[1] T. Oikonomou, S. Tegos, D. Tyrovolas, P. Diamantoulakis, and G. Karagiannidis, "On the error analysis of hexagonal-qam constellations," *IEEE Communications Letters*, pp. 1–1, 01 2022.

[2] E. W. Weisstein, "Eisenstein integer," https://mathworld.wolfram.com/EisensteinInteger.html, June 2022.

[3] P. K. Singya, P. Shaik, N. Kumar, V. Bhatia, and M.-S. Alouini, "A survey on higher-order qam constellations: Technical challenges, recent advances, and future trends," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 617–655, 2021.

[4] P. Vitthaladevuni, M.-S. Alouini, and J. Kieffer, "Exact ber computation for cross qam constellations," *Wireless Communications, IEEE Transactions on*, vol. 4, pp. 3039 – 3050, 12 2005.

[5] J. Smith, "Odd-bit quadrature amplitude-shift keying," *IEEE Transactions on Communications*, vol. 23, no. 3, pp. 385–389, 1975.

[6] A. Patel, "Hexagonal grids," https://www.redblobgames.com/grids/hexagons/, June 2022.