Telecommunication Systems I

Report of the 3rd Project

Olga Tsirou 2018030061

Odysseas Stavrou 2018030199

December 2020

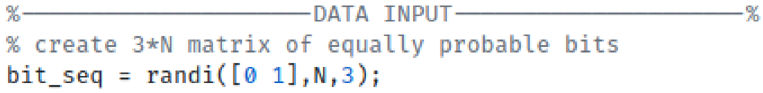
Technical University of Crete

Prof.: A. P. Liavas

A. Study of 8-PSK fundamentals

1. Data Creation:

MATLAB's **randi** function was used to produce the binary sequence, consisting of $3N$ equally probable bits.



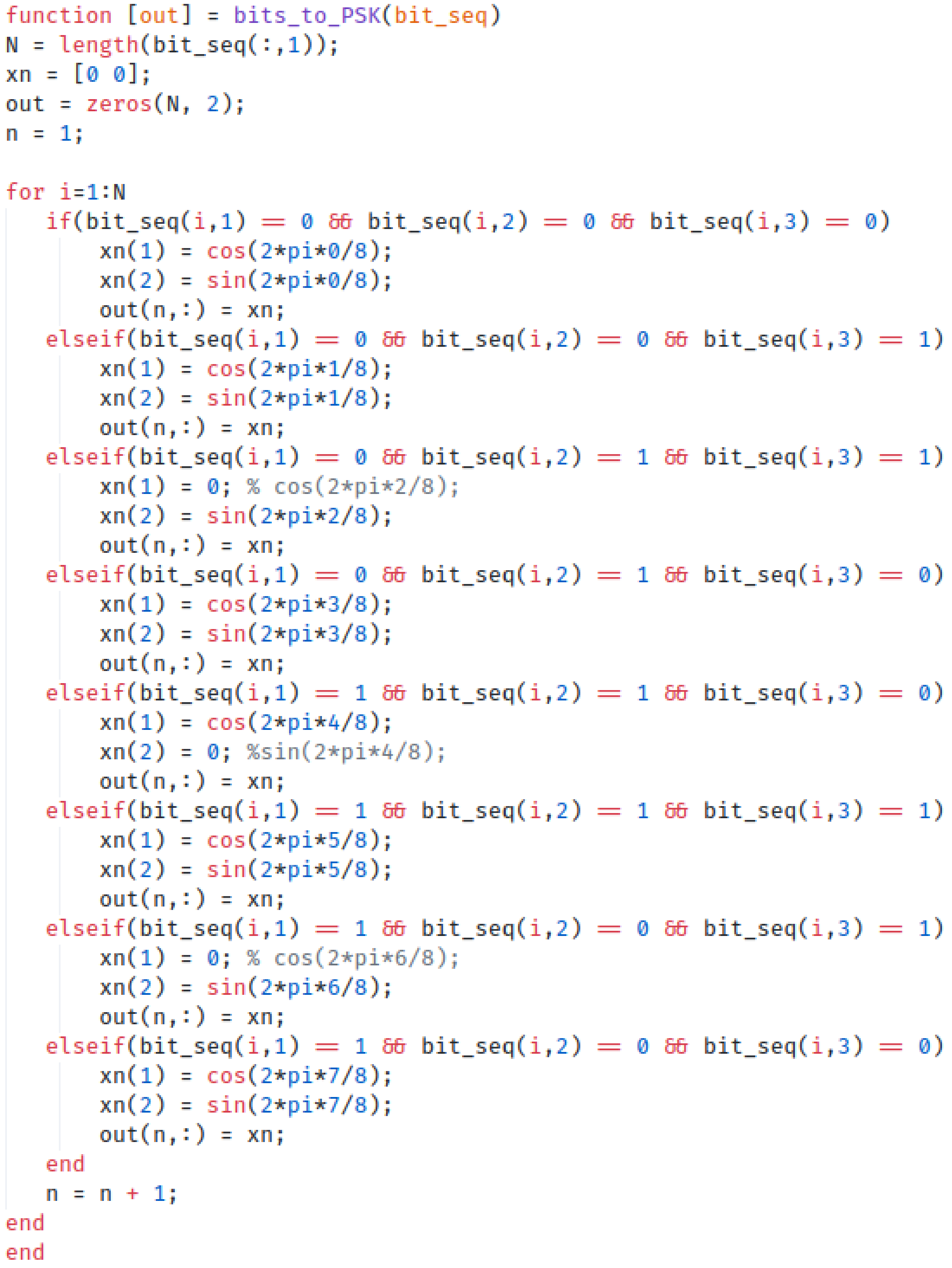
```
%----------------------DATA INPUT----------------------%
% create 3*N matrix of equally probable bits
bit_seq = randi([0 1],N,3);
```

2. Encoding:

The bits were encoded into 8-PSK symbols using the following Gray table and the following function. Each one of the 8 possible symbols is described as a column vector as shown below:

$$\mathbf{X}_n = \begin{bmatrix} X_{I,n} \\ X_{Q,n} \end{bmatrix} = \begin{bmatrix} \cos(\frac{2\pi m}{8}) \\ \sin(\frac{2\pi m}{8}) \end{bmatrix}, \quad m \in [0,7]$$

Bits	X_I	X_Q
000	$\cos(\frac{2\pi 0}{8})$	$\sin(\frac{2\pi 0}{8})$
001	$\cos(\frac{2\pi 1}{8})$	$\sin(\frac{2\pi 1}{8})$
011	$\cos(\frac{2\pi 2}{8})$	$\sin(\frac{2\pi 2}{8})$
010	$\cos(\frac{2\pi 3}{8})$	$\sin(\frac{2\pi 3}{8})$
110	$\cos(\frac{2\pi 4}{8})$	$\sin(\frac{2\pi 4}{8})$
111	$\cos(\frac{2\pi 5}{8})$	$\sin(\frac{2\pi 5}{8})$
101	$\cos(\frac{2\pi 6}{8})$	$\sin(\frac{2\pi 6}{8})$
100	$\cos(\frac{2\pi 7}{8})$	$\sin(\frac{2\pi 7}{8})$

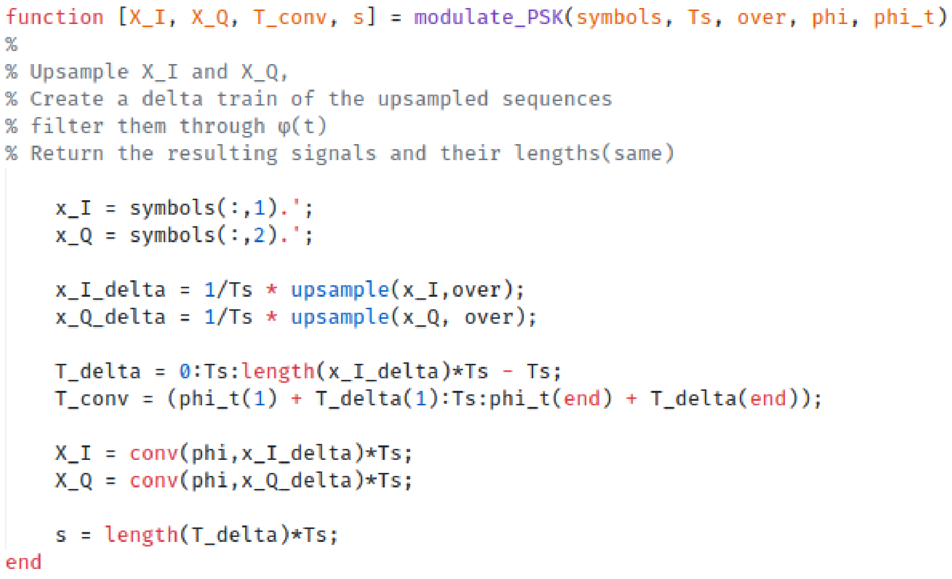


```
function [out] = bits_to_PSK(bit_seq)
N = length(bit_seq(:,1));
xn = [0 0];
out = zeros(N, 2);
n = 1;

for i=1:N
   if(bit_seq(i,1) == 0 && bit_seq(i,2) == 0 && bit_seq(i,3) == 0)
       xn(1) = cos(2*pi*0/8);
       xn(2) = sin(2*pi*0/8);
       out(n,:) = xn;
   elseif(bit_seq(i,1) == 0 && bit_seq(i,2) == 0 && bit_seq(i,3) == 1)
       xn(1) = cos(2*pi*1/8);
       xn(2) = sin(2*pi*1/8);
       out(n,:) = xn;
   elseif(bit_seq(i,1) == 0 && bit_seq(i,2) == 1 && bit_seq(i,3) == 1)
       xn(1) = 0; % cos(2*pi*2/8);
       xn(2) = sin(2*pi*2/8);
       out(n,:) = xn;
   elseif(bit_seq(i,1) == 0 && bit_seq(i,2) == 1 && bit_seq(i,3) == 0)
       xn(1) = cos(2*pi*3/8);
       xn(2) = sin(2*pi*3/8);
       out(n,:) = xn;
   elseif(bit_seq(i,1) == 1 && bit_seq(i,2) == 1 && bit_seq(i,3) == 0)
       xn(1) = cos(2*pi*4/8);
       xn(2) = 0; %sin(2*pi*4/8);
       out(n,:) = xn;
   elseif(bit_seq(i,1) == 1 && bit_seq(i,2) == 1 && bit_seq(i,3) == 1)
       xn(1) = cos(2*pi*5/8);
       xn(2) = sin(2*pi*5/8);
       out(n,:) = xn;
   elseif(bit_seq(i,1) == 1 && bit_seq(i,2) == 0 && bit_seq(i,3) == 1)
       xn(1) = 0; % cos(2*pi*6/8);
       xn(2) = sin(2*pi*6/8);
       out(n,:) = xn;
   elseif(bit_seq(i,1) == 1 && bit_seq(i,2) == 0 && bit_seq(i,3) == 0)
       xn(1) = cos(2*pi*7/8);
       xn(2) = sin(2*pi*7/8);
       out(n,:) = xn;
   end
   n = n + 1;
end
end
```

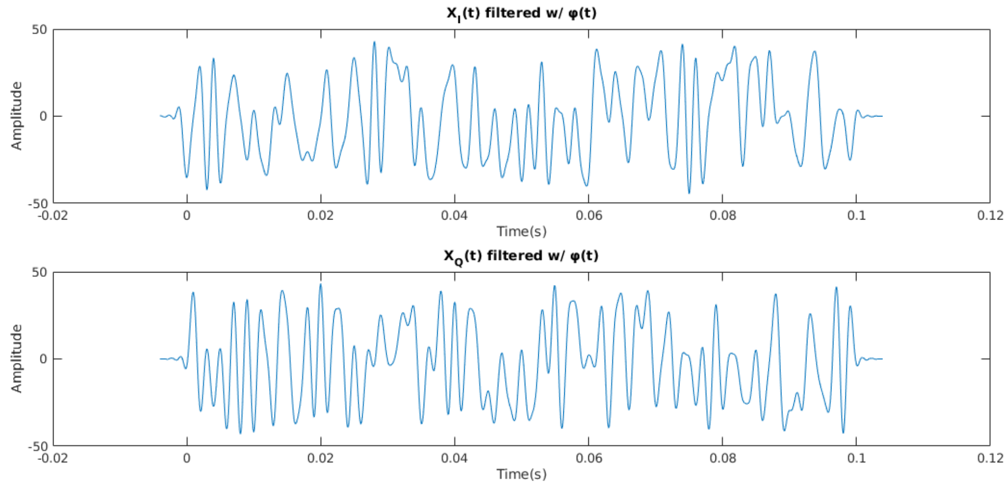
Note that since MATLAB approximates π, some functions like $\cos(\frac{2\pi 2}{8})$ had to be hard-coded to 0.

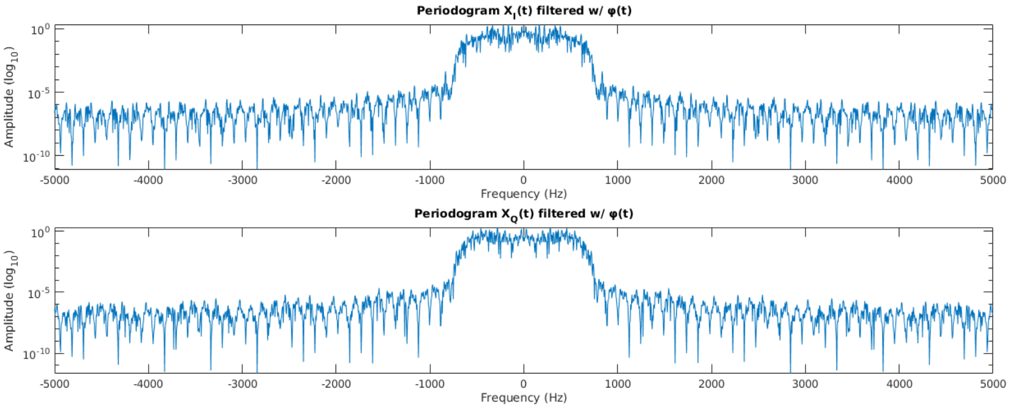
3. Modulation:

   Convolving the symbols into the $\phi(t)$ function, produces the following graph and periodogram.



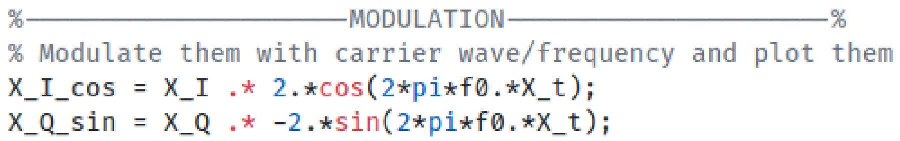
```
function [X_I, X_Q, T_conv, s] = modulate_PSK(symbols, Ts, over, phi, phi_t)
%
% Upsample X_I and X_Q,
% Create a delta train of the upsampled sequences
% filter them through φ(t)
% Return the resulting signals and their lengths(same)

    x_I = symbols(:,1).';
    x_Q = symbols(:,2).';

    x_I_delta = 1/Ts * upsample(x_I,over);
    x_Q_delta = 1/Ts * upsample(x_Q, over);

    T_delta = 0:Ts:length(x_I_delta)*Ts - Ts;
    T_conv = (phi_t(1) + T_delta(1):Ts:phi_t(end) + T_delta(end));

    X_I = conv(phi,x_I_delta)*Ts;
    X_Q = conv(phi,x_Q_delta)*Ts;

    s = length(T_delta)*Ts;
end
```



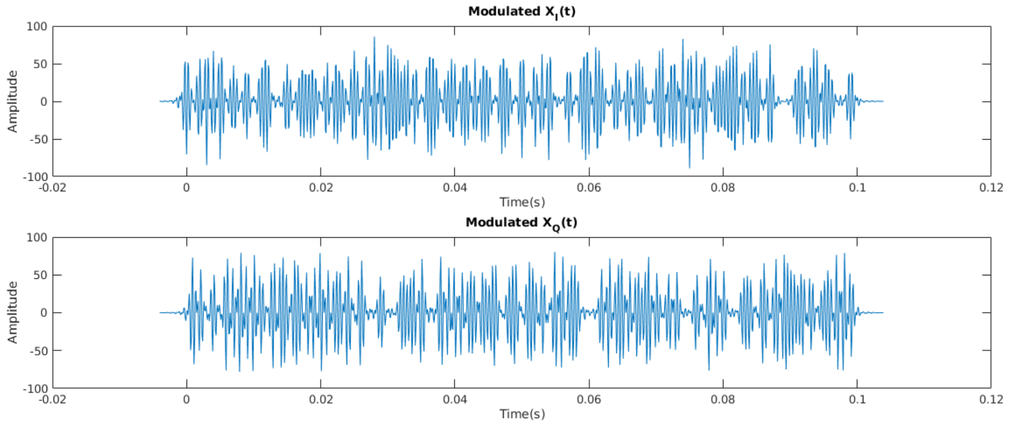


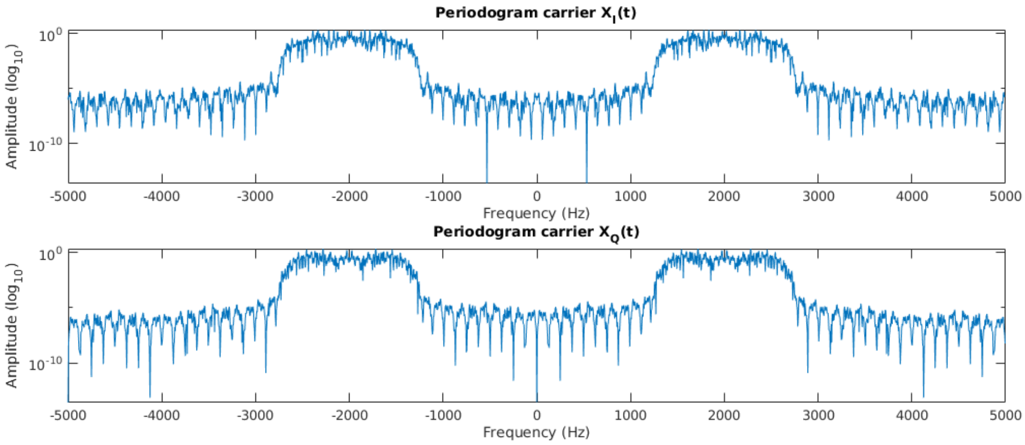
4. Carrier Wave:

   In order to transmit the signal at a certain frequency f_0, it needs to be multiplied with a sine-wave of frequency f_0. Since the information is coded into the phase of the signal, it can be observed that each time there is a phase shift, the transmission symbol has changed.



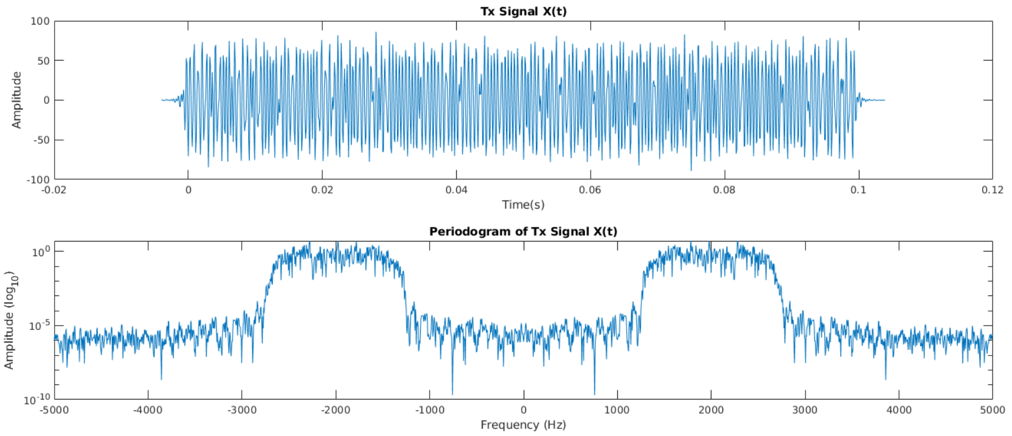
```
%----------------------MODULATION----------------------%
% Modulate them with carrier wave/frequency and plot them
X_I_cos = X_I .* 2.*cos(2*pi*f0.*X_t);
X_Q_sin = X_Q .* -2.*sin(2*pi*f0.*X_t);
```





5. Channel Input:

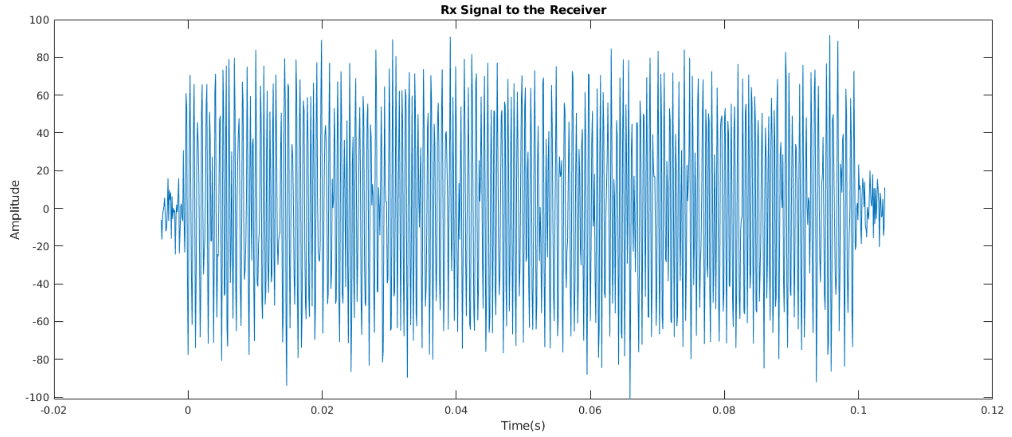
   Adding the two signals together results in the final transmission signal $X(t)$. A slight amplification can be observed due to summing the above signals. Plotting the periodogram of $X(t)$ proves that the frequency is the same as the carrier's $(f_0 = 2KHz)$



6. Assuming an ideal channel, no amplification/attenuation is expected and also perfect synchronization is achieved.

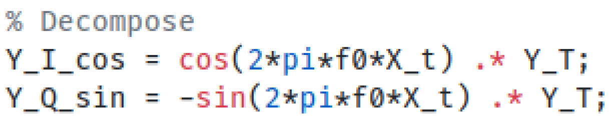
7. On the receiver's side the signal $Y(t)$ arrives with some random noise embedded to it. In this case with a signal to noise ratio of 20dB.

```
%---------------------------RECEIVING END---------------------------%
% Receive Noisy signal
snr_db = 20;
var_w = 1/(Ts * 10^(snr_db/10));

wg_noise = sqrt(var_w) .* randn(1,length(X_t));

% noised signal
Y_T = X_T + wg_noise;
```

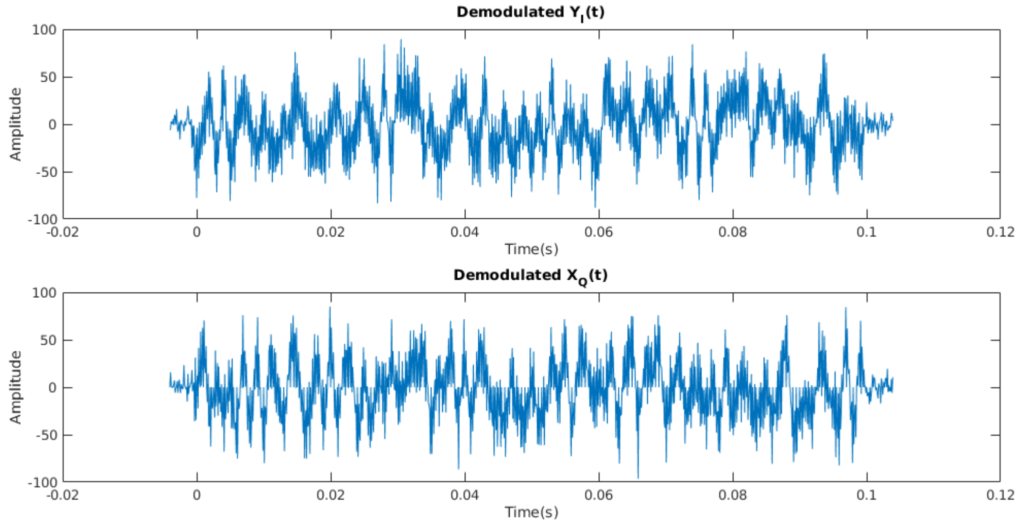


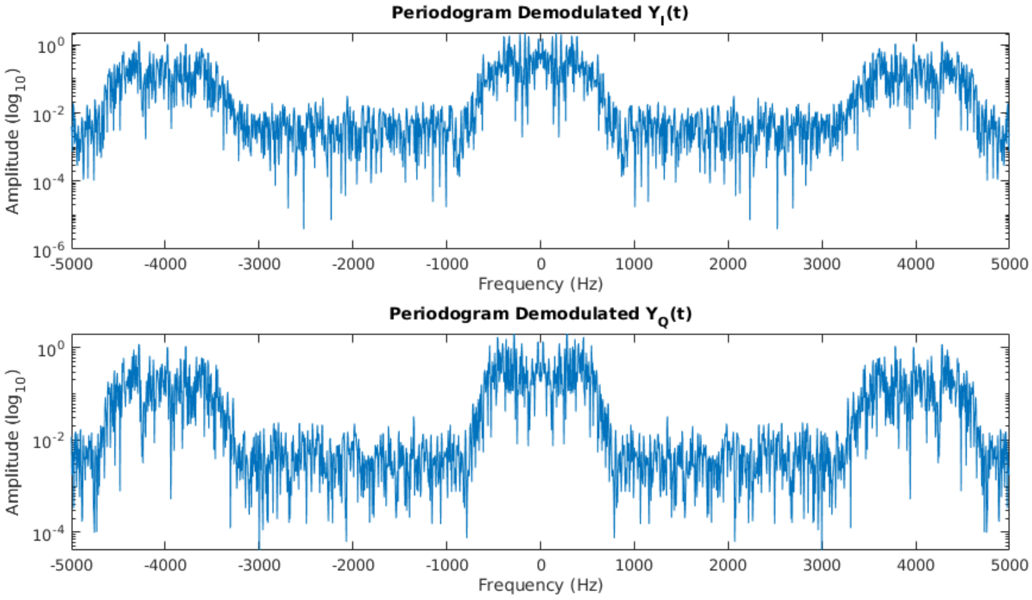
8. <u>Demodulation:</u>

Multiplying $Y(t)$ with the same carrier signals used in (4.), splits the signal so that the symbols can be derrived. Studying their plotted periodograms, one could observe that the signal has returned to its baseband form. There are still some lobes around the frequencies of the carrier waves that should be filtered.



```
% Decompose
Y_I_cos = cos(2*pi*f0*X_t) .* Y_T;
Y_Q_sin = -sin(2*pi*f0*X_t) .* Y_T;
```

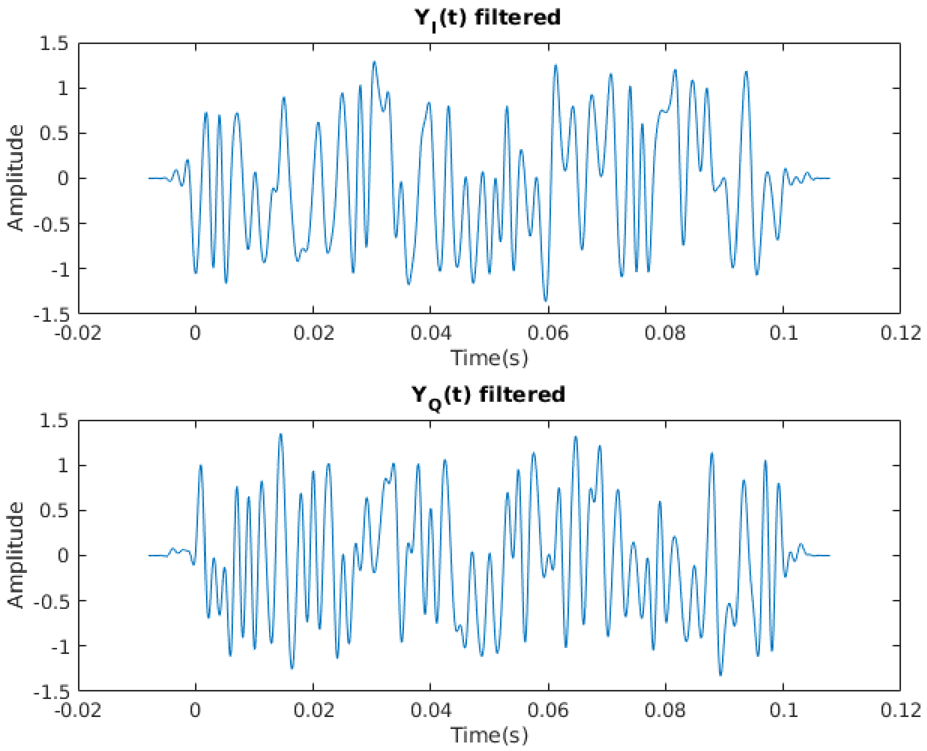


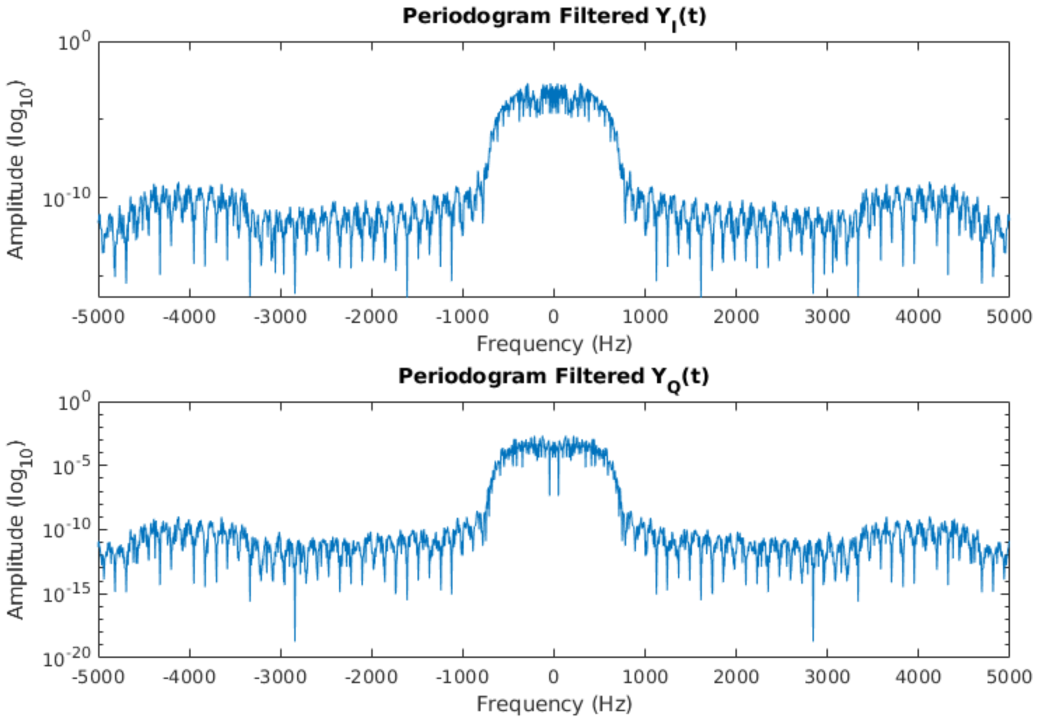


9. Filtering:

   By filtering the two signals that were demodulated in the previous step, almost identical (due to noise) symbols (with (3.)) are derrived.

```
%————————————————————FILTERING————————————————————%
% filter using φ(t)
Y_I_filter = conv(phi, Y_I_cos)*Ts;
Y_Q_filter = conv(phi, Y_Q_sin)*Ts;
Y_t = linspace(phi_t(1) + X_t(1),phi_t(end) + X_t(end), length(Y_I_filter));
```





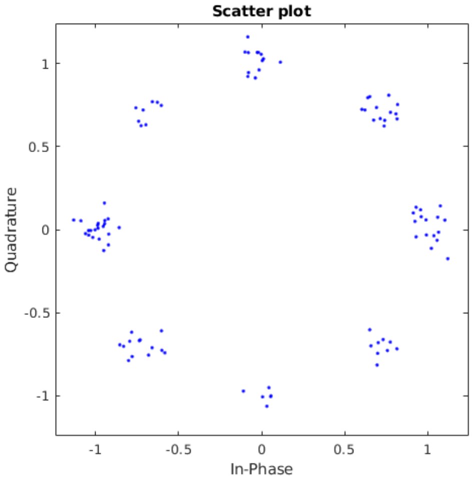
Indeed, after filtering the lobes disappeared.

10. Downsampling:

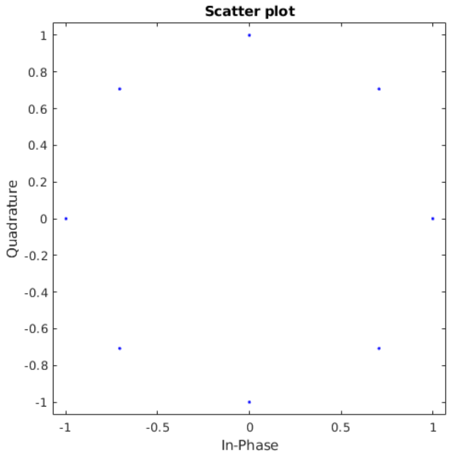
    Using MATLAB's **downsample** and **scatterplot** functions the zeros added by **upsample** are removed and then the symbols are graphed on the unit-circle. Depending on the SNR (signal to noise ratio) the derrived symbols are closer to, or more randomly scattered around their respective regions.

    Note the tail cutting of the convolutions to achieve the correct time vector.

```
%———————————————————————SAMPLING———————————————————————%
% downsample and tail-cut the convolution near-0 points
Y_I_downsmpl = downsample(Y_I_filter(Y_t >= 0 & Y_t < s), over);
Y_Q_downsmpl = downsample(Y_Q_filter(Y_t >= 0 & Y_t < s), over);


scatterplot(Y_I_downsmpl + 1i*Y_Q_downsmpl);
```



(a) Received Symbols

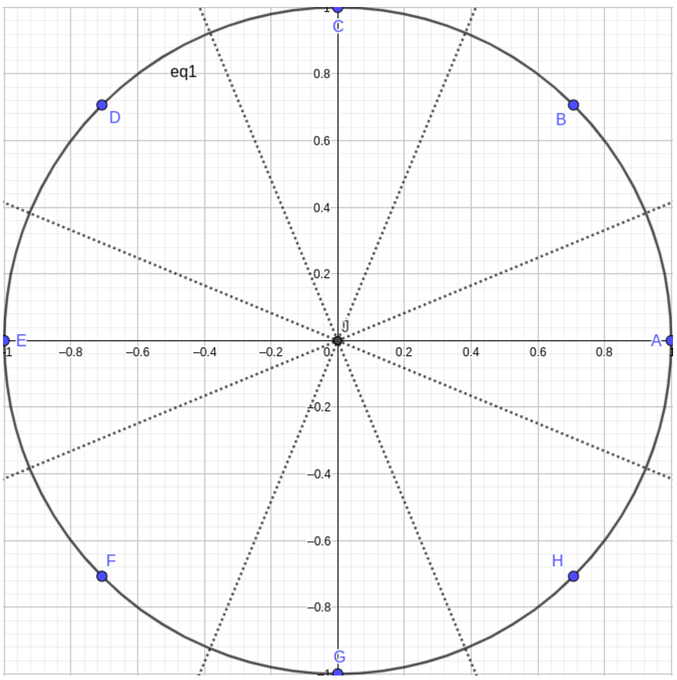


(b) Original Symbols

11. Detecting and Decoding Symbols:

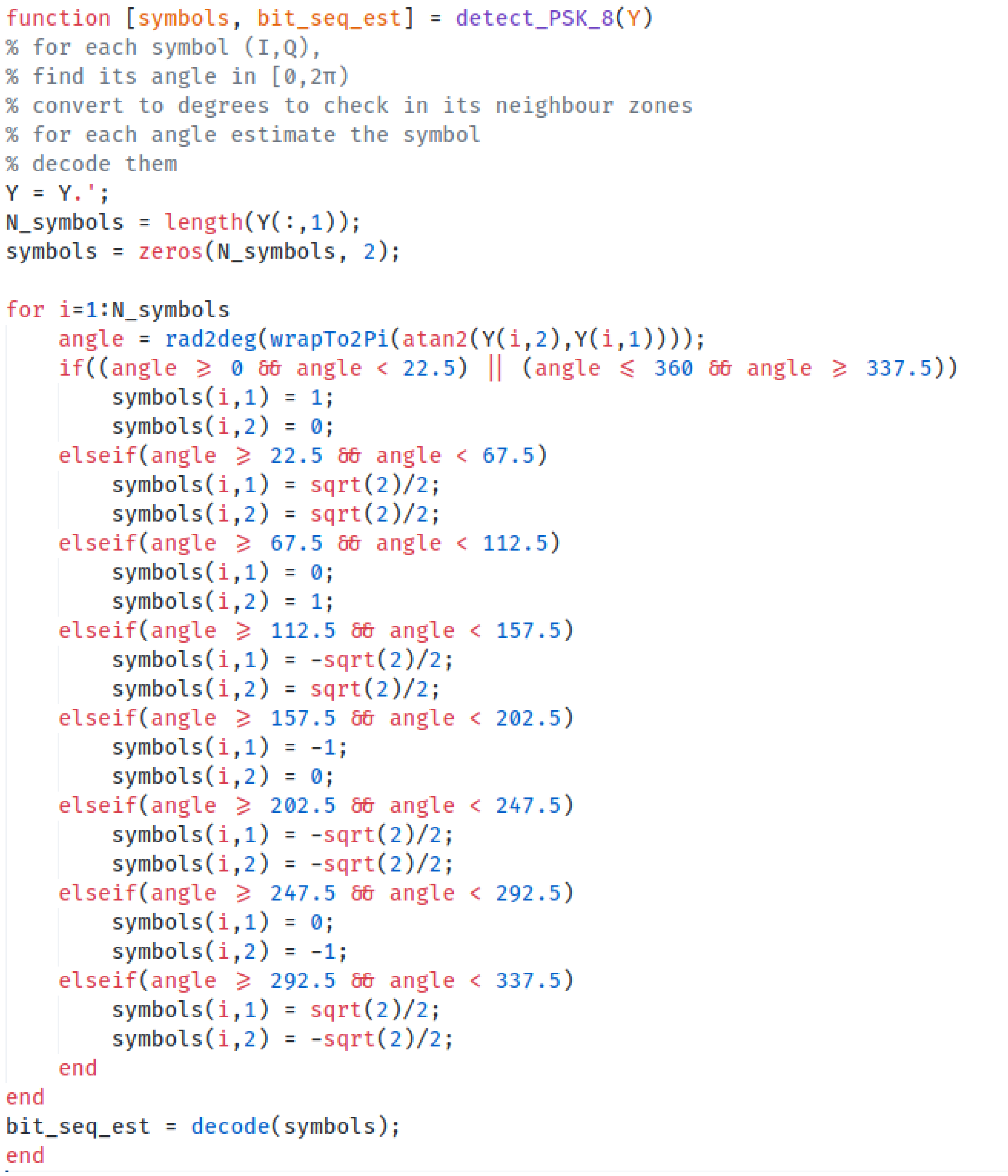
    The following function takes each derrived symbol and by using the "Closest Neighbour Rule" determines the symbol's actual values. To correctly decide the symbol, boundaries are "drawn" on the unit circle depicting their regions. If a symbol falls anywhere in that region, then it automatically gets reassigned the correct symbol values.

    For each of the 8 possible symbols as a centre, a boundary is a straight line from the circle's center that creates a ±22.5° angle from the symbol.

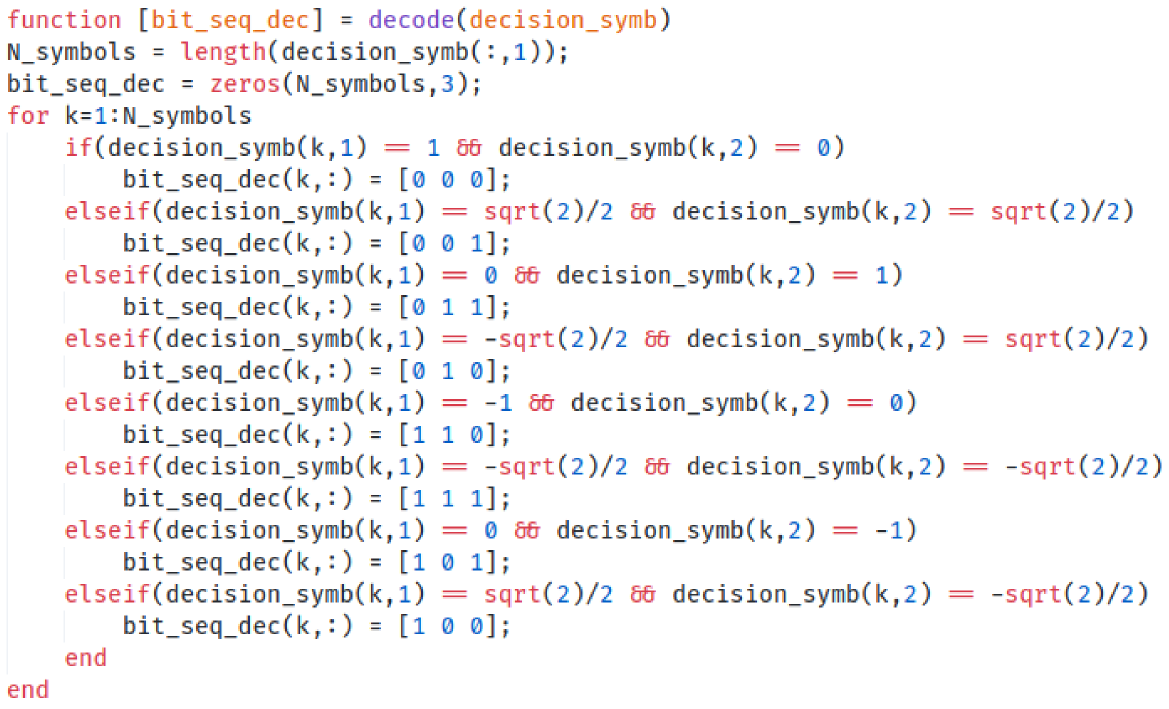


The dottet lines represent the boundary limits of each Symbol in its respective region. MATLAB's **atan2, wrapTo2Pi, rad2deg** functions were used in order to determine the angle in [0, 360°).

After that all symbols are decoded using the table in (2.).



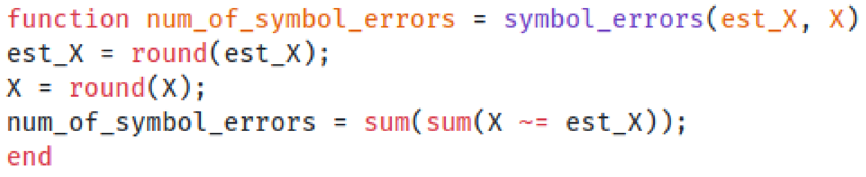
```
function [symbols, bit_seq_est] = detect_PSK_8(Y)
% for each symbol (I,Q),
% find its angle in [0,2π)
% convert to degrees to check in its neighbour zones
% for each angle estimate the symbol
% decode them
Y = Y.';
N_symbols = length(Y(:,1));
symbols = zeros(N_symbols, 2);

for i=1:N_symbols
    angle = rad2deg(wrapTo2Pi(atan2(Y(i,2),Y(i,1))));
    if((angle >= 0 && angle < 22.5) || (angle <= 360 && angle >= 337.5))
        symbols(i,1) = 1;
        symbols(i,2) = 0;
    elseif(angle >= 22.5 && angle < 67.5)
        symbols(i,1) = sqrt(2)/2;
        symbols(i,2) = sqrt(2)/2;
    elseif(angle >= 67.5 && angle < 112.5)
        symbols(i,1) = 0;
        symbols(i,2) = 1;
    elseif(angle >= 112.5 && angle < 157.5)
        symbols(i,1) = -sqrt(2)/2;
        symbols(i,2) = sqrt(2)/2;
    elseif(angle >= 157.5 && angle < 202.5)
        symbols(i,1) = -1;
        symbols(i,2) = 0;
    elseif(angle >= 202.5 && angle < 247.5)
        symbols(i,1) = -sqrt(2)/2;
        symbols(i,2) = -sqrt(2)/2;
    elseif(angle >= 247.5 && angle < 292.5)
        symbols(i,1) = 0;
        symbols(i,2) = -1;
    elseif(angle >= 292.5 && angle < 337.5)
        symbols(i,1) = sqrt(2)/2;
        symbols(i,2) = -sqrt(2)/2;
    end
end
bit_seq_est = decode(symbols);
end
```



```
function [bit_seq_dec] = decode(decision_symb)
N_symbols = length(decision_symb(:,1));
bit_seq_dec = zeros(N_symbols,3);
for k=1:N_symbols
    if(decision_symb(k,1) == 1 && decision_symb(k,2) == 0)
        bit_seq_dec(k,:) = [0 0 0];
    elseif(decision_symb(k,1) == sqrt(2)/2 && decision_symb(k,2) == sqrt(2)/2)
        bit_seq_dec(k,:) = [0 0 1];
    elseif(decision_symb(k,1) == 0 && decision_symb(k,2) == 1)
        bit_seq_dec(k,:) = [0 1 1];
    elseif(decision_symb(k,1) == -sqrt(2)/2 && decision_symb(k,2) == sqrt(2)/2)
        bit_seq_dec(k,:) = [0 1 0];
    elseif(decision_symb(k,1) == -1 && decision_symb(k,2) == 0)
        bit_seq_dec(k,:) = [1 1 0];
    elseif(decision_symb(k,1) == -sqrt(2)/2 && decision_symb(k,2) == -sqrt(2)/2)
        bit_seq_dec(k,:) = [1 1 1];
    elseif(decision_symb(k,1) == 0 && decision_symb(k,2) == -1)
        bit_seq_dec(k,:) = [1 0 1];
    elseif(decision_symb(k,1) == sqrt(2)/2 && decision_symb(k,2) == -sqrt(2)/2)
        bit_seq_dec(k,:) = [1 0 0];
    end
end
```

12. Detecting Symbol Errors:

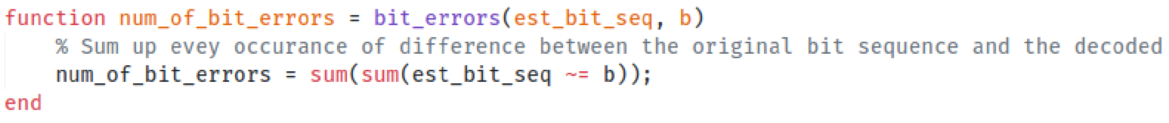
    This function returns how many symbols were falsely Detected. Since MATLAB has different approximations for some functions, rounding was performed to avoid false errors being detected



```
function num_of_symbol_errors = symbol_errors(est_X, X)
est_X = round(est_X);
X = round(X);
num_of_symbol_errors = sum(sum(X ~= est_X));
end
```

13. Detecting Bit Errors:

    Just like above, this function detects how many bits were badly decoded.



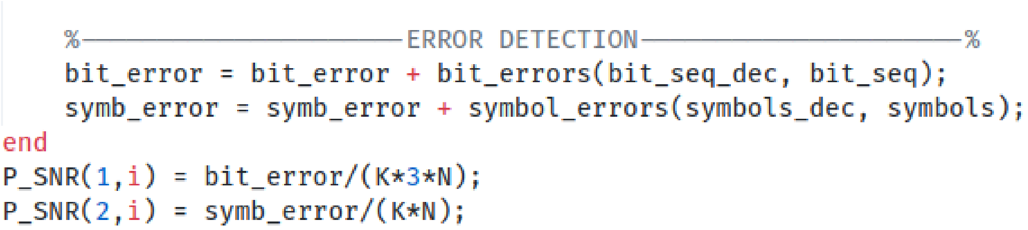
```
function num_of_bit_errors = bit_errors(est_bit_seq, b)
    % Sum up evey occurance of difference between the original bit sequence and the decoded
    num_of_bit_errors = sum(sum(est_bit_seq ~= b));
end
```

In the end all the bits and Symbols were received and decoded successfully with zero errors. However an ideal channel was taken for granted and the SNR was high.

B. Probability of Symbol/Bit error estimation using Monte Carlo Method

1. Calculate the Error Probabilities for Symbols and Bits alike:

   The code used is the excact same as above with some tweeks to calculate the probability of each SNR value (found in monte_carlo.m script).

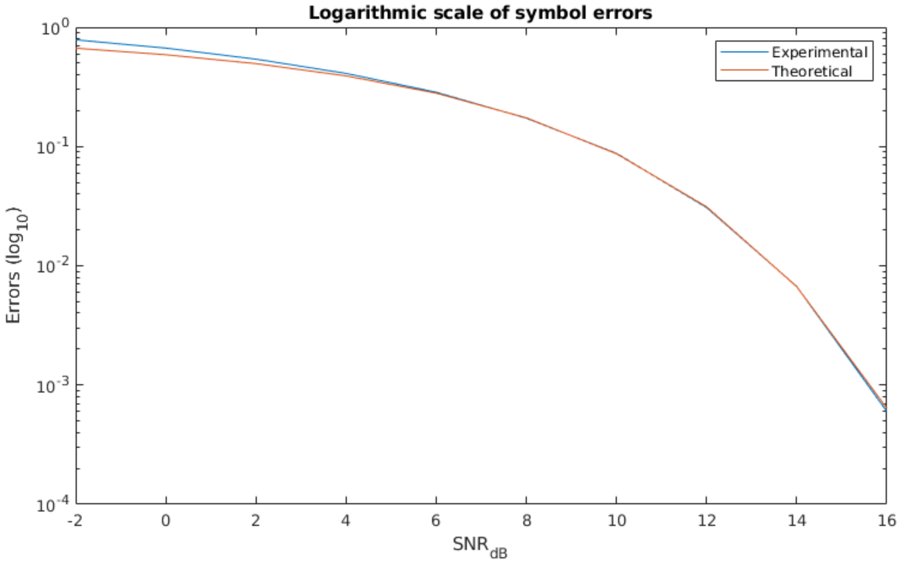


```
    %-------------------------ERROR DETECTION-------------------------%
    bit_error = bit_error + bit_errors(bit_seq_dec, bit_seq);
    symb_error = symb_error + symbol_errors(symbols_dec, symbols);
end
P_SNR(1,i) = bit_error/(K*3*N);
P_SNR(2,i) = symb_error/(K*N);
```

2. Symbol's Error Probability Upper Bound:

   The upper bound for $\mathbf{P}(\mathbf{E}_{symbol})$ is given by the following formula:

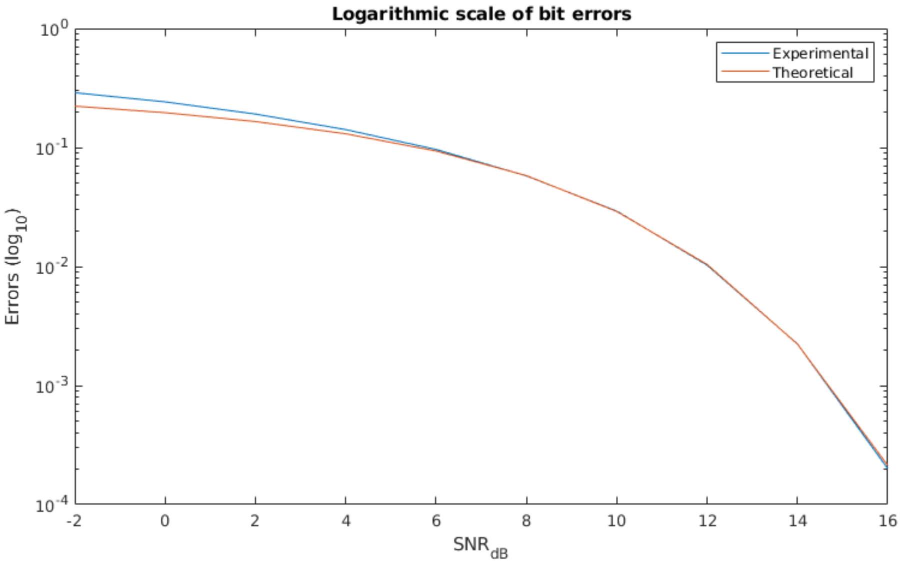
$$\mathbf{P}(\mathbf{E}) \leq 2Q\left(\sqrt{2SNR}\sin\left(\frac{\pi}{8}\right)\right), \qquad SNR = 10^{\frac{SNR_{dB}}{10}}$$



3. Bit's Error Probability Lower Bound:

The lower bound for $\mathbf{P}(\mathbf{E}_{bit})$ is given by:

$$\mathbf{P}(\mathbf{E}_{bit}) \leq \frac{2Q\left(\sqrt{2SNR}\sin\left(\frac{\pi}{8}\right)\right)}{3}, \qquad SNR = 10^{\frac{SNR_{dB}}{10}}$$



Both experimental bounds match with their theoretical counterparts, although a small divergence can be observed for low values of SNR_{db}