Digital Communications - HW4 - MATLAB Code

Jacopo Pegoraro, Edoardo Vanin

04/06/2018

AWGN, DFE, OFDM systems

```
clear all; close all; clc;
format long g;

b_l = [PN(20); PN(20)];
sstep = 32400;
num_bits = floor(length(b_l) / sstep) * sstep;
b_l = b_l(1:num_bits + 54);

sigma_a = 2;

% ENCODE VIA LDPC

%create LDPC encoder with the dafault matrix (rate=1/2)
encoderLDPC = comm.LDPCEncoder;
enc_b_l = zeros(2*length(b_l),1);
for i = 0:(floor(length(b_l)/sstep))-1
    %encodes block by block the input bits
    %block length is equal to 32400
    block = b_l(i * sstep + 1:i * sstep + sstep);
    enc_b_l(2 * i * sstep + 1:2 * i * sstep + 2 * sstep) = step(encoderLDPC, bl
end

% INTERLEAVING
interl_b_l = interl(enc_b_l);

% MAP 0 TO -1
interl_b_l = 2 * interl_b_l - 1;

a = zeros(floor(length(interl_b_l)/2),1);
for i=1:2:(length(interl_b_l) - 1)
   a((i+1)/2) =  interl_b_l(i) + 1i * interl_b_l(i+1);
end

clear pn;
```

```
SNR_vector = [0.4:0.05:1];
Pbit_AWGN = zeros(length(SNR_vector),1);

for snr_index = 1:length(SNR_vector)
SNR = SNR_vector(snr_index);
% Add Gaussian Noise
snrlin = 10^(SNR/10);
sigma_w = sigma_a * E_qc / snrlin;
w = wgn(length(a), 1, 10 * log10(sigma_w), 'complex');
y_k = a + w;

% Compute Log Likelihood Ratio

llr = zeros(2*length(y_k),1);
llr(1:2:end, 1) = -2*real(y_k)/(sigma_w/2);
llr(2:2:end, 1) = -2*imag(y_k)/(sigma_w/2);
new_length = floor(length(llr) / 64800) * 64800;
llr = llr(1:new_length, 1);

% Decode the bits
llr = deinterleaver(llr); % Deinterleave the loglikelihood ratio first

decoderLDPC = comm.LDPCDecoder;

dstep = 2 * sstep;
tic
for i = 0:(floor(length(llr)/dstep)) - 1
    %decodes block by block the input bits
    %block length is equal to 64800
    block = llr(i * dstep + 1:i * dstep + dstep);
    dec_b_l(i * dstep / 2 + 1:i * dstep / 2 + dstep / 2) = step(decoderLDPC, bl
end
toc

[Pbit_AWGN(snr_index), errors] = BER(dec_b_l, b_l(1:length(dec_b_l)));
end

clear all; close all; clc;
format long g;

% THIS SCRIPT GENERATES THE SYMBOLS BY USING A PN SEQUENCE
% AND APPLYING LDPC, ENCODING AND INTERLEAVING

b_l = [PN(20); PN(20)];
sstep = 32400;
num_bits = floor(length(b_l) / sstep) * sstep;
b_l = b_l(1:num_bits + 54);

sigma_a = 2;
```

```
% ENCODE VIA LDPC

%create LDPC encoder with the dafault matrix (rate=1/2)
encoderLDPC = comm.LDPCEncoder;
enc_b_l = zeros(2*length(b_l),1);
for i = 0:(floor(length(b_l)/sstep))-1
    %encodes block by block the input bits
    %block length is equal to 32400
    block = b_l(i * sstep + 1:i * sstep + sstep);
    enc_b_l(2 * i * sstep + 1:2 * i * sstep + 2 * sstep) = step(encoderLDPC, bl
end

% INTERLEAVING
interl_b_l = interl(enc_b_l);

% MAP 0 TO -1
interl_b_l = 2 * interl_b_l - 1;

% Generate the channel input response
[q_c, E_qc] = channel_impulse_response();

a = zeros(floor(length(interl_b_l)/2),1);
for i=1:2:(length(interl_b_l) - 1)
   a((i+1)/2) =  interl_b_l(i) + 1i * interl_b_l(i+1);
end

clear pn;

a_prime = upsample(a, 4);

% Filter through the channel
s_c = filter(q_c, 1, a_prime);

SNR_vector = [1.4:0.025:1.75];
Pbit_DFEenc = zeros(length(SNR_vector),1);

for snr_index = 1:length(SNR_vector)
    SNR = SNR_vector(snr_index);
% Add Gaussian Noise
snrlin = 10^(SNR/10);
sigma_w = sigma_a * E_qc / snrlin;
w = wgn(length(s_c), 1, 10 * log10(sigma_w), 'complex');
rcv_bits = s_c + w;

% Reciver structure
g_m = conj(flipud(q_c));

% compute the impulse response h
h = conv(q_c, g_m);
```

```
h = downsample(h,4);
h = h(h ~= 0);

N1 = floor(length(h)/2);
N2 = N1;

r_r = filter(g_m, 1, rcv_bits);

t_0_bar = length(g_m);

x = downsample(r_r(t_0_bar:end), 4);

% Filtering through C and equalization

r_gm = xcorr(g_m);
% r_w = N0 .* downsample(r_gm, 4);
r_w_up = sigma_w / 4 * r_gm;
r_w = downsample(r_w_up, 4);

M1 = 5;
D = 4;
M2 = N2 + M1 - 1 - D;


[c Jmin]= WienerC_DFE(h, r_w, sigma_a, M1, M2, D);

psi = conv(c, h);

M1 = 5; D = 4;
M2 = N2 + M1 - 1 - D;

b = -psi(find(psi==max(psi))+1:end);

y_k = equalization_DFE(x, c, b, D, max(psi));

decisions = zeros(length(y_k),1);
for i = 1:length(y_k)
    decisions(i) = threshold_detector(y_k(i));
end

Jmin_lin = 10^(Jmin/10);
noise_var = (Jmin_lin-sigma_a*abs(1-max(psi))^2)/(abs(max(psi))^2);
% Compute Log Likelihood Ratio
llr = zeros(2*length(y_k),1);
llr(1:2:end, 1) = -2*real(y_k)/(noise_var/2);
llr(2:2:end, 1) = -2*imag(y_k)/(noise_var/2);
new_length = floor(length(llr) / 64800) * 64800;
llr = llr(1:new_length, 1);

% Decode the bits
```

```
llr = deinterleaver(llr); % Deinterleave the loglikelihood ratio first

decoderLDPC = comm.LDPCDecoder;

dstep = 2 * sstep;

tic
for i = 0:(floor(length(llr)/dstep)) - 1
    %decodes block by block the input bits
    %block length is equal to 64800
    block = llr(i * dstep + 1:i * dstep + dstep);
    dec_b_l(i * dstep / 2 + 1:i * dstep / 2 + dstep / 2) = step(decoderLDPC, bl
end
toc

[Pbit_DFEenc(snr_index), errors] = BER(dec_b_l, b_l(1:length(dec_b_l)));

end

clear all; close all; clc;
format long g;

sigma_a = 2;
load('generated_symbols.mat','b_l','uncoded_a')

% Generate the channel input response
[q_c, E_qc] = channel_impulse_response();

a_prime = upsample(uncoded_a, 4);

% Filter through the channel
s_c = filter(q_c, 1, a_prime);

SNR_vector = [4:0.5:14];
Pbit_DFEunc = zeros(length(SNR_vector),1);

for snr_index = 1:length(SNR_vector)
    SNR = SNR_vector(snr_index);
    % Add Gaussian Noise
    snrlin = 10^(SNR/10);
    sigma_w = sigma_a * E_qc / snrlin;
    w = wgn(length(s_c), 1, 10 * log10(sigma_w), 'complex');
    rcv_bits = s_c + w;

    % Reciver structure
    g_m = conj(flipud(q_c));

    % compute the impulse response h
    h = conv(q_c, g_m);
    h = downsample(h,4);
```

```
    h = h(h ~= 0);

    N1 = floor(length(h)/2);
    N2 = N1;

    r_r = filter(g_m, 1, rcv_bits);

    t_0_bar = length(g_m);

    x = downsample(r_r(t_0_bar:end), 4);

    % Filtering through C and equalization
    r_gm = xcorr(g_m);
    % r_w = N0 .* downsample(r_gm, 4);
    r_w_up = sigma_w / 4 * r_gm;
    r_w = downsample(r_w_up, 4);

    M1 = 5;
    D = 4;
    M2 = N2 + M1 - 1 - D;


    [c Jmin]= WienerC_DFE(h, r_w, sigma_a, M1, M2, D);

    psi = conv(c, h);

    M1 = 5; D = 4;
    M2 = N2 + M1 - 1 - D;

    b = -psi(find(psi==max(psi))+1:end);

    y_k = equalization_DFE(x, c, b, D, max(psi));

    decisions = zeros(length(y_k),1);
    for i = 1:length(y_k)
        decisions(i) = threshold_detector(y_k(i));
    end

    detected_bits = IBMAP(decisions);

    [Pbit_DFEunc(snr_index), errors] = BER(detected_bits, b_l(1:length(detected
end

save('DFE_uncoded.mat','Pbit_DFEunc');

 function [Pbit dec_b_l] = OFDM_coded(a, b_l, Npx, t0_bar, SNRlin)

% pad the last symbols with -1-1i to have an integer multiple of 512
M = 512; % number of subchannels
sigma_a = 2;
```

```
a = [a; ones(M - mod(length(a), M), 1) * (-1-1i)];

a_mat = reshape(a, M, []);

A_no_pr = ifft(a_mat);

A = [A_no_pr(M - Npx + 1:M,:); A_no_pr];

s_n = reshape(A, [], 1);

% only to try with ideal channel
% sigma_w_OFDM = ( (sigma_a/M))/ SNRlin;
% w = wgn(length(s_n), 1, 10*log10(sigma_w_OFDM), 'complex');
% x = s_n + w;

%channel construction
ro = 0.0625;
span = 8;
sps = 4;
g_rcos = rcosdesign(ro, span, sps, 'sqrt');
%g_rcos = g_rcos(abs(g_rcos)>=(max(g_rcos)*10^-2));
s_up = upsample(s_n,4);

s_up_rcos = filter(g_rcos, 1, s_up);
%s_up_rcos = s_up_rcos(length(g_rcos):end);

q_c = channel_impulse_response();

s_c = filter(q_c, 1, s_up_rcos);
%s_c = s_c(length(q_c):end);

Eimp = sum(conv(g_rcos,q_c).^2);

sigma_w_OFDM = (sigma_a * Eimp)/ (M * SNRlin);
w = wgn(length(s_c), 1, 10*log10(sigma_w_OFDM), 'complex');
r_c = s_c + w;
% r_c = s_c;

gq = conv(g_rcos,q_c);
q_r_up = conv(gq, g_rcos);
%q_r_up = q_r_up(abs(q_r_up)>=(max(q_r_up)*10^-2));
% t0_bar = find(q_r_up == max(q_r_up));

index = find(q_r_up==max(q_r_up));
start = mod(index,4);
q_r = downsample(q_r_up(start:end),4);
%q_r = q_r(abs(q_r)>=(max(q_r)*10^-2));

x = filter(g_rcos, 1, r_c);
%x = upfirdn(r_c, g_rcos, 2);
```

```
x = downsample(x(t0_bar:end), 4);

K_i = fft(q_r, 512);
K_i = K_i(:);

x = x(1: end - mod(length(x), M+Npx));

r_matrix = reshape(x, M+Npx, []);
r_matrix = r_matrix(Npx + 1:end, :);

K_i_inv = K_i.^(-1);

x_matrix = fft(r_matrix);

y_matrix = x_matrix .* K_i_inv;

sigma_i = 0.5*sigma_w_OFDM * M * abs(K_i_inv).^2;

LLR_real = -2 * real(y_matrix) .* (sigma_i.^(-1));
LLR_imag = -2 * imag(y_matrix) .* (sigma_i.^(-1));
LLR_real_vec = reshape(LLR_real, [], 1);
LLR_imag_vec = reshape(LLR_imag, [], 1);
LLR = zeros(length(LLR_real_vec) + length(LLR_imag_vec), 1);
LLR(1:2:end) = LLR_real_vec;
LLR(2:2:end) = LLR_imag_vec;
LLR = LLR(1:2*length(b_l));

% to try with ideal channel
% y = reshape(x_matrix, [], 1);
% llr = zeros(2*length(y),1);
% sigma_i = 0.5*sigma_w_OFDM*M;
% llr_real = -2*times(real(y),(sigma_i.^(-1)));
% llr_imag = -2*times(imag(y),(sigma_i.^(-1)));
% llr_real_ar = reshape(llr_real, [], 1);
% llr_imag_ar = reshape(llr_imag, [], 1);
% llr(1:2:end) = llr_real_ar;
% llr(2:2:end) = llr_imag_ar;
%
% llr = llr(1:end - mod(length(llr),32400));

LLR = deinterleaver(LLR);

decoderLDPC = comm.LDPCDecoder;

dstep = 64800;

tic
for i = 0:(floor(length(LLR)/dstep)) - 1
    block = LLR(i * dstep + 1:i * dstep + dstep);
    dec_b_l(i * dstep / 2 + 1:i * dstep / 2 + dstep / 2) = step(decoderLDPC, bl
```

```
end
toc

Pbit = BER(b_l, dec_b_l);
end

 function [Pbit b_l_hat] = OFDM_uncoded(a, b_l, Npx, t0_bar, SNRlin)

% pad the last symbols with -1-1i to have an integer multiple of 512
M = 512; % number of subchannels
sigma_a = 2;
a = [a; ones(M - mod(length(a), M), 1) * (-1-1i)];

a_mat = reshape(a, M, []);

%IFFT should be computed at sampling time Tblock=Tofdm*(M+Npx) according to
%the book
A_no_prefix = ifft(a_mat);
%A_no_pr = A_no_pr(1:512,:);

A = [A_no_prefix(M - Npx + 1:M,:); A_no_prefix];

s_n = reshape(A, [], 1);

%channel contruction
ro = 0.0625;
span = 30;
sps = 2;
g_rcos = rcosdesign(ro, span, sps, 'sqrt');

% N    = 30;          % Order
% Fc   = 0.5;         % Cutoff Frequency
% TM   = 'Rolloff';   % Transition Mode
% R    = 0.0625;      % Rolloff
% DT   = 'sqrt';      % Design Type
% Beta = 0.5;         % Window Parameter
%
% % Create the window vector for the design algorithm.
% win = kaiser(N+1, Beta);
%
% % Calculate the coefficients using the FIR1 function.
% g_rcos  = firrcos(N, Fc, R, 2, TM, DT, [], win);
% Hd = dfilt.dffir(g_rcos);

s_up = upsample(s_n,4);

s_up_rcos = filter(g_rcos, 1, s_up);
%s_up_rcos = s_up_rcos(length(g_rcos):end);

q_c = channel_impulse_response();
```

```
s_c = filter(q_c, 1, s_up_rcos);
%s_c = s_c(length(q_c):end);

Eimp = sum(conv(g_rcos,q_c).^2);

sigma_w_OFDM = (sigma_a * Eimp)/ (M * SNRlin);
w = wgn(length(s_c), 1, 10*log10(sigma_w_OFDM), 'complex');
r_c = s_c + w;
% r_c = s_c;

q_r_up = conv(conv(g_rcos,q_c), g_rcos);
q_r_up = q_r_up(abs(q_r_up)>=(max(q_r_up)*10^-2));
%t0_bar = find(q_r_up == max(q_r_up));
%t0_bar = 21;
q_r = downsample(q_r_up(1:end),4);

x = filter(g_rcos, 1, r_c);
x = downsample(x(t0_bar:end), 4);

K_i = fft(q_r, M);
K_i = K_i(:);

x = x(1: end - mod(length(x), M+Npx));

r_matrix = reshape(x, M+Npx, []);
r_matrix = r_matrix(Npx + 1:end, :);

K_i_inv = K_i.^(-1);

x_matrix = fft(r_matrix);

y_matrix = x_matrix .* K_i_inv;

y = reshape(y_matrix, 1, []);

dec_a_k = zeros(length(y),1);
for k=1:length(y)
    dec_a_k(k) = threshold_detector(y(k));
end
b_l_hat = IBMAP(dec_a_k);

[Pbit ~]= BER(b_l, b_l_hat(1:end));
 end
```

Wiener and equalization for DFE

```
function [decisions] = equalization_DFE(x, c, b, D, psiD)
%EQUALIZATION for DFE
M2 = length(b);
```

```
y = conv(x,c);
y = y(1:length(x)+D);
y = y./psiD;
detected = zeros(length(x) + D, 1);
for k=0:length(y)-1
      if (k <= M2)
          a_past = [flipud(detected(1:k)); zeros(M2 - k, 1)];
     else
          a_past = flipud(detected(k - M2 + 1: k));
     end
detected(k + 1) = y(k + 1) + b.' * a_past;
end
%scatterplot(y)
decisions = detected(D + 1:end);
end

 function [c_opt, Jmin] = WienerC_DFE(h, r_w, sigma_a, M1, M2, D)

     N1 = floor(length(h)/2);
     N2 = N1;
     padding = 60;
     hpad = padarray(h, padding);

     % Padding the noise correlation
     r_w_pad = padarray(r_w, padding);

     p  = zeros(M1 ,1);

     for i = 0 : M1-1
          p(i + 1) = sigma_a * conj(hpad(N1 + padding + 1 + D - i));
     end

     R = zeros(M1);
     for row = 0:(M1-1)
          for col = 0:(M1-1)

               fsum = (hpad((padding + 1):(N1 + N2 + padding + 1))).' ...
                    * conj(hpad((padding + 1 - (row - col)):( N1 + N2 + ...
                    padding + 1 - (row - col))));

               if M2==0
                    ssum=0;
               else
                    ssum = (hpad((N1+padding+1+1+D-col): ...
                         (N1+padding+1+M2+D-col))).' * ...
                         conj((hpad((N1+padding+1+1+D-row):...
                         (N1+padding+1+M2+D-row))));
               end

               R(row + 1, col + 1) = sigma_a * (fsum - ssum) +...
```

```
                r_w_pad(padding + 1 + row - col + ...
                (floor(length(r_w) / 2 )));
        end
    end

    c_opt = R \ p;

    temp2 = zeros(M1, 1);

    for l = 0:M1-1
        temp2(l + 1) = c_opt(l + 1) * hpad(N1 + padding + 1 + D - l);
    end

    Jmin = 10*log10(sigma_a * (1 - sum(temp2)));
end
```

Useful scripts (interleaver, BMAP, plots, TD)

```
function [interl_bits] = interl(bits)

    interl_bits = zeros(1,length(bits));

    rows = 43;
    columns = 41;
    for matrix = 0:(length(bits)/(rows*columns) - 1)
        curr_matrix = matrix * rows * columns;
        for col = 0:(columns-1)
            interl_bits(curr_matrix + col * rows + 1 : curr_matrix + col * rows
                bits(curr_matrix + col + 1 : columns : curr_matrix + col + colu
        end
    end
end

function [deinterleaved_bits] = deinterleaver(bits)
    % This function receives a sequence of bits and unscrambles it
    % INPUT:
    % bits: the bits to interleave
    % OUTPUT:
    % deinterleaved_bits: the deinterleaved bits

    % Input should be a multiple of 14061600 = lcm(rows*columns, 64800) bits
    if (mod(length(bits), 32400) ~= 0)
        disp('Length of the input vector should be a multiple of 14061600');
        return;
    end

    deinterleaved_bits = zeros(1,length(bits));

    % The deinterleaver is just an interleaver with rows and cols switched
    rows = 41;
```

```
    columns = 43;

    % We work with a rowsxcolumns matrix
    for matrix = 0:(length(bits)/(rows*columns) - 1)
        curr_matrix = matrix * rows * columns;
        for col = 0:(columns-1)
            deinterleaved_bits(curr_matrix + col * rows + 1 : curr_matrix + col
                bits(curr_matrix + col + 1 : columns : curr_matrix + col + colu
        end
    end
end

function [Pbit, count_errors] = BER(sent, detected)
% Computes the symbol-error rate, it accepts QPSK symbols
count_errors = 0;
for i=1:length(sent)
    if sent(i) ~= detected(i)
        count_errors = count_errors + 1;
    end
end
Pbit = count_errors/length(sent);
end

function [symbols] = BMAP(bits)

% bits are given as a row vector because of the interleaver function output
L = length(bits);
symbols = zeros(L,1);
    % gray coding of the input bits for QPSK symbols
    for k = 1:2:L-1
        if (isequal(bits(k:k+1), [0 0] ))
            symbols(k) = -1-1i;
        elseif (isequal(bits(k:k+1), [1 0] ))
            symbols(k) = 1-1i;
        elseif (isequal(bits(k:k+1), [0 1] ))
            symbols(k) = -1+1i;
        elseif (isequal(bits(k:k+1), [1 1] ))
            symbols(k) = +1+1i;
        end
    end
    symbols = symbols(1:2:end);
end

function [b_i] = IBMAP(a_k)
    % Check if the input array has even length
    L = length(a_k);

    b_i = zeros(2*L,1);

    % Map each couple of values to the corresponding symbol
```

```
    % The real part gives the bit
    for k = 1:2:length(b_i)-1
        symbol = a_k((k+1)/2);
        if (real(symbol) == 1)
            b2k = 1;
        else
            b2k = 0;
        end

        if (imag(symbol) == 1)
             b2k1 = 1;
        else
             b2k1 = 0;
        end

        b_i(k)= b2k;
        b_i(k+1)= b2k1;
    end
end

clear all; close all; clc;

load('AWGN_coded.mat', 'Pbit_AWGN');
load('OFDM_uncoded.mat','Pbit_OFDM_uncoded')
load('DFE_uncoded.mat','Pbit_DFEunc');
load('DFE_coded1.mat','Pbit_DFEenc');
load('OFDM_coded3.mat','Pbit_OFDM_coded');

SNR_dB = [4:0.5:14];
SNR_lin = 10.^(SNR_dB./10);
sigma_a = 2;
awgn_bound = qfunc(sqrt(SNR_lin));

figure,
semilogy(SNR_dB, Pbit_DFEunc,'g-o')
ylim([10^-5 10^-1])
grid on;
hold on,
semilogy(SNR_dB, Pbit_OFDM_uncoded,'b-<')
hold on,
semilogy(SNR_dB, awgn_bound,'k')
xlabel('SNR [dB]')
ylabel('P_{bit}')
title('Uncoded')
legend({'DFE','OFDM','AWGN'})

SNR_dB_encDFE = [1.4:0.025:1.75];
SNR_dB_encOFDM = [0.9:0.05:1.6];
SNR_dB_awgn =[0.4:0.05:1];
figure,
```

```
semilogy(SNR_dB_encDFE, Pbit_DFEenc,'g-o')
hold on,
ylim([10^-5 10^-1])
xlim([0 3]);
semilogy(SNR_dB_encOFDM, Pbit_OFDM_coded,'b-<')
grid on;
hold on,
semilogy(SNR_dB_awgn, Pbit_AWGN,'k')
xlabel('SNR [dB]')
ylabel('P_{bit}')
title('Coded')
legend({'DFE','OFDM','AWGN'})

clear all; close all; clc;

load('generated_symbols.mat','a','enc_b_l','b_l','uncoded_a')

%parpool;

%% CODED
% SNR_dB = [0.9:0.05:1.6];
% parfor i=1:length(SNR_dB)
%     [Pbit_OFDM_coded(i) b_l_hat] = OFDM_coded(a, b_l, 17, 91, 10^(SNR_dB(i)/1
% end

% save('OFDM_coded.mat','Pbit_OFDM_coded')

SNR_dB = 1.3;
[Pbit_OFDM_coded b_l_hat] = OFDM_coded(a, b_l, 16, 3, 10^(SNR_dB/10));

% SNR_dB = 1.1;
% parfor i=1:60
%     [Pbit_coded(i) b_l_hat] = OFDM_coded(a, b_l, 16, i, 10^(SNR_dB./10));
% end
% save('OFDM_coded.mat','Pbit_OFDM_coded')

% SNR_dB = 1.1 ;
% parfor i=10:30
%     [Pbit_coded(i) b_l_hat] = OFDM_coded(a, b_l, i, 3, 10^(SNR_dB./10));
% end

%% UNCODED
% SNR_dB = 10;
% for i=10:100
% [Pbit_OFDM_uncoded(i) b_l_hat] = OFDM_uncoded(uncoded_a, b_l, 11, i, 10^(SNR_
% end


% SNR_dB = [4:0.5:14];
% parfor i=1:length(SNR_dB)
```

```
% [Pbit_OFDM_uncoded(i) b_l_hat] = OFDM_uncoded(uncoded_a, b_l, 11, 91, 10^(SNR
% end
%save('OFDM_uncoded.mat','SNR_dB','Pbit_OFDM_uncoded1');
%

% [Pbit_uncoded b_l_hat] = OFDM_uncoded(uncoded_a, b_l, 8, 21, 10.^(SNR_dB/10))
```

```
function [a_hat_kD] = threshold_detector(y_k)

if (real(y_k) > 0)
    if (imag(y_k) > 0)
        a_hat_kD = 1+1i;
    else
        a_hat_kD= 1-1i;
    end
else
    if (imag(y_k) > 0)
        a_hat_kD = -1+1i;
    else
        a_hat_kD = -1-1i;
    end
end

end
```

Signal/channel generators

```
function [pn] = PN(r)

L = pow2(r) - 1;
pn = zeros(L,1);

pn(1:r) = ones(1,r).';

for l=r+1:L
    switch r
        case 1
            pn(l) = pn(l-1);
        case 2
            pn(l) = xor(pn(l-1), pn(l-2));
        case 3
            pn(l) = xor(pn(l-2), pn(l-3));
        case 4
            pn(l) = xor(pn(l-3), pn(l-4));
        case 5
            pn(l) = xor(pn(l-3), pn(l-5));
        case 6
            pn(l) = xor(pn(l-5), pn(l-6));
        case 7
            pn(l) = xor(pn(l-6), pn(l-7));
```

```
        case 8
            pn(l) = xor(xor(pn(l-2), pn(l-3)), xor(pn(l-4), pn(l-8)));
        case 9
            pn(l) = xor(pn(l-5), pn(l-9));
        case 10
            pn(l) = xor(pn(l-7), pn(l-10));
        case 11
            pn(l) = xor(pn(l-9), pn(l-11));
        case 12
            pn(l) = xor(xor(pn(l-2), pn(l-10)), xor(pn(l-11), pn(l-12)));
        case 13
            pn(l) = xor(xor(pn(l-1), pn(l-11)), xor(pn(l-12), pn(l-13)));
        case 14
            pn(l) = xor(xor(pn(l-2), pn(l-12)), xor(pn(l-13), pn(l-14)));
        case 15
            pn(l) = xor(pn(l-14), pn(l-15));
        case 16
            pn(l) = xor(xor(pn(l-11), pn(l-13)), xor(pn(l-14), pn(l-16)));
        case 17
            pn(l) = xor(pn(l-14), pn(l-17));
        case 18
            pn(l) = xor(pn(l-11), pn(l-18));
        case 19
            pn(l) = xor(xor(pn(l-14), pn(l-17)), xor(pn(l-19), pn(l-18)));
        case 20
            pn(l) = xor(pn(l-17), pn(l-20));
    end
end
end

clear all; close all; clc;

%% THIS SCRIPT GENERATES THE SYMBOLS BY USING A PN SEQUENCE
%% AND APPLYING LDP ENCODING AND INTERLEAVING

b_l = [PN(20); PN(20)];
sstep = 32400;
num_bits = floor(length(b_l) / sstep) * sstep;
b_l = b_l(1:num_bits);

sigma_a = 2;

%% ENCODE VIA LDPC

%create LDPC encoder with the dafault matrix (rate=2)
encoderLDPC = comm.LDPCEncoder;
enc_b_l = zeros(2*length(b_l),1);
for i = 0:(ceil(length(b_l)/sstep))-1
    %encodes block by block the input bits
    %block length is equal to 32400
```

```
    block = b_l(i * sstep + 1:i * sstep + sstep);
    enc_b_l(2 * i * sstep + 1:2 * i * sstep + 2 * sstep) = step(encoderLDPC, bl
end

%% INTERLEAVING

interl_b_l = interl(enc_b_l);

%% BITMAP

a = BMAP(interl_b_l);
uncoded_a = BMAP(b_l.');

save('generated_symbols.mat','a','enc_b_l','b_l','uncoded_a')

function [q_c, E_qc] = channel_impulse_response()
q_c_num   = [0 0 0 0 0 0.7424];
q_c_denom = [1 -0.67];

q_c = impz(q_c_num, q_c_denom);

% cut the impulse response when too small
q_c = [0; 0; 0; 0; 0; q_c(q_c >= max(q_c)*10^(-2) )];
E_qc = sum(q_c.^2);

end

clc; clear all; close all;

%% Configuration parameters
if ~exist("Noise.mat", 'file')
    noise_seq;
end
load('Noise','w');
load('generated_symbols.mat','a');
verbose = false;
plot_figure = true;

r = 20;
SNR_dB = [0:14];
SNR_lin = 10.^(SNR_dB./10);
sigma_a = 2;

T = 1;
q_c_num   = [0 0 0 0 0 0.7424];
q_c_denom = [1 -0.67];
q_c = impz(q_c_num, q_c_denom);

% cut the impulse response when too small
q_c = [0; 0; 0; 0; 0; q_c( q_c >= max(q_c)*10^(-2) )];
```

```
E_qc = sum(q_c.^2);

sigma_w = (sigma_a * E_qc) ./ SNR_lin;
N0 = sigma_w./4;

%% Filtering through the channel

a_prime = upsample(a, 4);

s_c = filter(q_c_num, q_c_denom, a_prime);

%% Add noise

r_c = zeros(length(s_c), length(SNR_dB));

for i = 1:length(SNR_dB)
    r_c(:,i) = s_c + w(1:length(s_c))*sqrt(sigma_w(i));
end

clear w;
%% Save the workspace

save("common.mat");
```