Digital Transmission - Homework 4

Andrea Dittadi, Davide Magrin, Michele Polese

June 7, 2015

MATLAB code

problem_1

```
% This script solves the first problem.
clear
close all
clc

rng default

%parpool(15);

% data
sigma_a = 2;

%% Get optimal number of bits
desired_bits = 2^24;
% Compute the closest number of bits that both interleaver and encoder will like
search_step = 32400;
bit_number = ceil(desired_bits / search_step) * search_step;

%% Estimate Pbit for ideal channel without encoding
snr_vec = 0:0.5:14;
Pbit_noenc = zeros(1,length(snr_vec));

parfor curr_snr = 1:length(snr_vec)
   disp(curr_snr);
   snrdb = snr_vec(curr_snr);
   % Simulate Pbit for the ideal channel, no encoding
   bits = randi([0 1], 1, bit_number);
   symbols = bitmap(bits.');

   % Send data through the ideal channel
   snrlin = 10^(snrdb/10);
   Eh = 1;  % Energy of the ideal channel ir
   sigma_w = sigma_a*Eh/snrlin;
   w = wgn(length(symbols), 1, 10*log10(sigma_w), 'complex');
   rcv_bits = symbols + w;

   % Threshold the bits
   decided_symbols = zeros(1, length(rcv_bits));
   for idx = 1:length(rcv_bits)
      decided_symbols(idx) = qpsk_td(rcv_bits(idx));
   end

   decided_bits = ibmap(decided_symbols);

   % Estimate pbit
   Pbit_noenc(curr_snr) = sum(xor(decided_bits.', bits))/length(bits);

end

%% Estimate Pbit for ideal channel with encoding
snr_vec_enc = 0:0.02:1;
Pbit_enc = zeros(1,length(snr_vec_enc));
```

```
parfor curr_snr = 1:length(snr_vec_enc)
   disp(snr_vec_enc(curr_snr));
   snrdb = snr_vec_enc(curr_snr);
   bits = randi([0 1], 1, bit_number);

   enc_bits = encodeBits(bits);
   int_enc_bits = interleaver(enc_bits);  % Interleave the encoded bits

   symbols = bitmap(int_enc_bits.');

   % Send the data through the ideal channel
   snrlin = 10^(snrdb/10);
   Eh = 1;
   sigma_w = sigma_a*Eh/snrlin;
   w = wgn(length(symbols), 1, 10*log10(sigma_w), 'complex');
   rcv_bits = symbols + w;

   % Compute Log Likelihood Ratio
   llr = zeros(2*length(symbols),1);
   llr(1:2:end) = -2*real(rcv_bits)/(sigma_w/2);
   llr(2:2:end) = -2*imag(rcv_bits)/(sigma_w/2);

   % Decode the bits
   llr = deinterleaver(llr); % Deinterleave the loglikelihood ratio first
   dec_bits = decodeBits(llr).';

   Pbit_enc(curr_snr) = sum(xor(dec_bits, bits))/length(bits);

end

%% Save results
save('Problem1', 'snr_vec', 'snr_vec_enc', 'bit_number', 'Pbit_noenc', 'Pbit_enc');

%% Plot results
load ('Problem1');
semilogy(snr_vec, Pbit_noenc), hold on,
semilogy(snr_vec_enc, Pbit_enc)
ylim([10^-5, 10^-1]), xlim([0, 14]), grid on
xlabel('\Gamma (dB)')
ylabel('Pbit')
legend('Uncoded QPSK', 'Coded QPSK')
title('Bit Error Rate for uncoded and coded QPSK');
```

MLsequence

```
function [ p ] = MLsequence( L )
% Generate a Maximum Length Pseudo Noise sequence, using shift and xor
% operators. L is the desired length of the resulting PN sequence.

% Extract r from the given L
r = log2(L+1);
p = zeros(L,1);
p(1:r) = ones(r,1); % Set arbitrary initial condition
for l = r+1:(L) % Skip the initial condition for the cycle
    switch L
        case 3
            p(l) = xor(p(l-1), p(l-2));
        case 7
            p(l) = xor(p(l-2), p(l-3));
        case 15
            p(l) = xor(p(l-3), p(l-4));
        case 31
            p(l) = xor(p(l-3), p(l-5));
        case 63
            p(l) = xor(p(l-5), p(l-6));
        case 127
            p(l) = xor(p(l-6), p(l-7));
        case 2^10 -1
```

```
            p(l) = xor(p(l-7), p(l-10));
        case 2^11 -1
            p(l) = xor(p(l-9), p(l-11));
        case 2^12 -1
            p(l) = xor(xor(xor(p(l-2), p(l-10)), p(l-11)), p(l-12));
        case 2^13 -1
            p(l) = xor(xor(xor(p(l-1), p(l-11)), p(l-12)), p(l-13));
        case 2^14 -1
            p(l) = xor(xor(xor(p(l-2), p(l-12)), p(l-13)), p(l-14));
        case 2^15 -1
            p(l) = xor(p(l-14), p(l-15));
        case 2^16 -1
            p(l) = xor(xor(xor(p(l-11), p(l-13)), p(l-14)), p(l-16));
        case 2^18 -1
            p(l) = xor(p(l-11), p(l-18));
        case 2^19 -1
            p(l) = xor(xor(xor(p(l-14), p(l-17)), p(l-18)), p(l-19));
        case 2^20 -1
            p(l) = xor(p(l-17), p(l-20));
        otherwise
            p = [];
            disp('MLsequence: length not supported');
    end
end
end
```

bitmap

```
function [output] = bitmap(input)
    % Check if the input array has even length
    L = length(input);
    if (mod(L, 2) ~= 0)
        disp('Must input an even length array');
        return;
    end

    output = zeros(L,1);

    % Map each couple of values to the corresponding symbol
    for idx = 1:2:L-1
        if (isequal(input(idx:idx+1), [0; 0] ))
            output(idx) = -1-1i;
        elseif (isequal(input(idx:idx+1), [1; 0] ))
            output(idx) = 1-1i;
        elseif (isequal(input(idx:idx+1), [0; 1] ))
            output(idx) = -1+1i;
        elseif (isequal(input(idx:idx+1), [1; 1] ))
            output(idx) = +1+1i;
        end
    end

    % Finally, only keep the useful values of the output
    output = output(1:2:end);

end
```

ibmap

```
function [output] = ibmap(input)
    % Check if the input array has even length
    L = length(input);

    output = zeros(2*L,1);

    % Map each couple of values to the corresponding symbol
    % The real part gives the bit
    for k = 1:2:length(output)-1
        symbol = input((k+1)/2);
        if (real(symbol) == 1)
           b2k = 1;
        else
           b2k = 0;
        end

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

        output(k)= b2k;
        output(k+1)= b2k1;
    end
end
```

channel_output

```
function [ r, sigma_w, h ] = channel_output( x, snr, OFDM)
% CHANNEL_OUTPUT Generates channel output (that is the desired signal)
% with an hard coded non varying channel.
% Returns the channel output r given the input parameters
% x is a column vector for consistency
% snr must be linear

h = [0,0,0,0,0,0.7*exp(-1i*2.57), 0.24*exp(1i*1.34), 0.15*exp(-1i*2.66), ...
    0.58*exp(-1i*1.51), 0.4*exp(-1i*1.63), 0, 0, 0];
M = 512;

E_h = sum(abs(h).^2);
if (OFDM == true) % different definition of SNR Msigma_a E_h / sigma_w
    sigma_a = 2/M;
else
    sigma_a = 2; % for a QPSK
end
sigma_w = sigma_a*E_h/snr;

w = wgn(1, length(x) + length(h) - 1, 10*log10(sigma_w), 'complex');

r = conv(h, x) + w.';
end
```

encodeBits

```
function [encoded_bits] = encodeBits(bits)
    % Create the encoder
    warning('off', 'all');
    enc = fec.ldpcenc;
    numInfoBits = enc.NumInfoBits; % Length of the info words

    if (mod(length(bits), numInfoBits) ~= 0)
        disp('Length of the input vector should be a multiple of 32400');
        return;
    end

    encoded_bits = zeros(2*length(bits),1);
    % Iterate over the input info bits and encode them
    for idx = 0:(ceil(length(bits)/numInfoBits))-1
        current_bits = bits(idx*numInfoBits+1:idx*numInfoBits + numInfoBits);
        encoded_bits(2*idx*numInfoBits+1:2*idx*numInfoBits + 2*numInfoBits) = encode(enc, current_bits
              );
    end
end
```

decodeBits

```
function [decoded_bits] = decodeBits(bits)
    % Create the encoder
    warning('off', 'all');

    dec = fec.ldpcdec;
    dec.DecisionType = 'Hard Decision';
    dec.OutputFormat = 'Information Part';
    dec.NumIterations = 50;
    dec.DoParityChecks = 'Yes';

    numInfoBits = dec.NumInfoBits; % Length of the info words

    if (mod(length(bits), numInfoBits) ~= 0)
        disp('Length of the input vector should be a multiple of 64800');
        return;
    end

    decoded_bits = zeros(length(bits)/2,1);
    % Iterate over the input info bits and encode them
    for idx = 0:(length(bits)/(2*numInfoBits))-1
        current_bits = bits(2*idx*numInfoBits + 1 : 2*idx*numInfoBits + 2*numInfoBits);
        decoded_bits(idx*numInfoBits+1:idx*numInfoBits + numInfoBits) = decode(dec, current_bits);
    end
end
```

interleaver

```
function [interleaved_bits] = interleaver(bits)
    % This function receives a sequence of bits and scrambles it
    % INPUT:
    % bits: the bits to interleave
    % OUTPUT:
    % interleaved_bits: the interleaved 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

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

    rows = 30;
    columns = 36;

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

deinterleaver

```
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 = 36;
    columns = 30;

    % 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 * rows + rows) = ...
                bits(curr_matrix + col + 1 : columns : curr_matrix + col + columns * rows);
        end
    end
end
```

qpsk_td

```
function [ out ] = qpsk_td( in )
% Threshold detector for QPSK
if (real(in) > 0)
    if (imag(in) > 0)
        out = 1+1i;
    else
        out= 1-1i;
    end
else
    if (imag(in) > 0)
        out = -1+1i;
    else
        out = -1-1i;
    end
end

end
```

problem_2

```
% This script solves the second problem.
clear, clc, close all
rng default

% Initialize parameters based on the assigned channel
t0 = 5;
N1 = 0;
N2 = 4;
M1_dfe = 15;
D_dfe = M1_dfe - 1;
M2_dfe = N2 + M1_dfe - 1 - D_dfe;

parpool(15);

%% Known channel, DFE, uncoded data (HW3)

snr_vec_knownch_uncoded = 0:14;
seq_lengths_knownch_uncoded = 2.^[13 13 13 13 13 13 13 15 18 18 20 20 22 23 23];
Pbit_knownch_uncoded = zeros(length(snr_vec_knownch_uncoded),1);

parfor snr_idx = 1:length(snr_vec_knownch_uncoded)
    curr_snr = snr_vec_knownch_uncoded(snr_idx);
    fprintf('Known channel, uncoded, snr = %.2f\n', curr_snr);

    % Generate the current needed sequence
    bits = randi([0 1], 1, seq_lengths_knownch_uncoded(snr_idx));
    symbols = bitmap(bits.');

    % Send through the channel
    [rcv_symb, sigma_w, h] = channel_output(symbols, 10^(curr_snr/10), false);
    rcv_symb = rcv_symb(t0+1 : end-7)/h(t0+1);
    hi = h(t0+1-N1:t0+1+N2)/h(t0+1);

    % Receiver: filter with DFE
    [~, rcv_symb] = DFE_filter(rcv_symb, hi.', N1, N2, sigma_w, D_dfe, M1_dfe, M2_dfe, false, false);
    rcv_bits = ibmap(rcv_symb);

    % Compute the Pbit and store it
    Pbit_knownch_uncoded(snr_idx) = sum(xor(rcv_bits.', bits))/length(bits);

end

% Save current results
save('Problem2_knownch_uncoded', 'snr_vec_knownch_uncoded', ...
    'seq_lengths_knownch_uncoded', 'Pbit_knownch_uncoded');

%% Known channel, DFE, coded data

% Get optimal number of bits
desired_bits = 2^22;
% Compute the closest number of bits that both interleaver and encoder will like
search_step = 32400;
bit_number = ceil(desired_bits / search_step) * search_step;

snr_vec_knownch_coded = [1, 1.5, 2:0.02:2.4];  % Pbit falls at 2.2 dBs
seq_lengths_knownch_coded = bit_number*ones(1, length(snr_vec_knownch_coded));
Pbit_knownch_coded = zeros(length(snr_vec_knownch_coded),1);

parfor snr_idx = 1:length(snr_vec_knownch_coded)
    curr_snr = snr_vec_knownch_coded(snr_idx);
    fprintf('Known channel, coded, snr = %.2f\n', curr_snr);

    % Generate the current needed sequence
    bits = randi([0 1], 1, seq_lengths_knownch_coded(snr_idx));

    enc_bits = encodeBits(bits);
    int_enc_bits = interleaver(enc_bits);  % Interleave the encoded bits
    symbols = bitmap(int_enc_bits.');
    % Send through the channel
    [rcv_symb, sigma_w, h] = channel_output(symbols, 10^(curr_snr/10), false);
    rcv_symb = rcv_symb(t0+1 : end-7)/h(t0+1);
    hi = h(t0+1-N1:t0+1+N2)/h(t0+1);

    % Receiver: filter with DFE
    [Jmin, psi, rcv_symb] = DFE_filter(rcv_symb, hi, N1, N2, sigma_w, D_dfe, M1_dfe, M2_dfe, true,
          false);

    noise_var = (Jmin-sigma_a*abs(1-psi(D_dfe+1))^2)/abs(psi(D_dfe+1))^2; % This includes white noise
          and isi

    % Compute Log Likelihood Ratio
    llr = zeros(2*length(packet),1);
    llr(1:2:end) = -2*real(rcv_symb(L+Nseq+1:end))/(noise_var/2);
    llr(2:2:end) = -2*imag(rcv_symb(L+Nseq+1:end))/(noise_var/2);

    % Decode the bits
    llr = deinterleaver(llr); % Deinterleave the loglikelihood ratio first
    dec_bits = decodeBits(llr).';

    % Compute the Pbit and store it
    Pbit_knownch_coded(snr_idx) = sum(xor(dec_bits, bits))/length(bits);

end

% Save current results
save('Problem2_knownch_coded', 'snr_vec_knownch_coded', ...
    'seq_lengths_knownch_coded', 'Pbit_knownch_coded');


%% Estimated channel, DFE, uncoded data
L = 31;
Nseq = 7;

snr_vec_estch_uncoded = 0:14;
seq_lengths_estch_uncoded = 2.^[13 13 13 13 13 13 13 13 13 14 15 15 18 18 18] -1;
Pbit_estch_uncoded = zeros(length(snr_vec_estch_uncoded),1);

parfor snr_idx = 1:length(snr_vec_estch_uncoded)
    curr_snr = snr_vec_estch_uncoded(snr_idx);
    fprintf('Estimated channel, uncoded, snr = %.2f\n', curr_snr);

    % Send through the channel
    [packet, rcv_symb, sigma_w] = txrc(seq_lengths_estch_uncoded(snr_idx), curr_snr);

    % Perform estimation
    [h, est_sigma_w] = get_channel_info(rcv_symb(t0+1:t0+L+Nseq), N1, N2);

    rcv_symb = rcv_symb(t0+1:end-7)/h(N1+1);
    hi = h / h(N1+1);

    % Receiver: filter with DFE
    [~, rcv_symb] = DFE_filter(rcv_symb, hi, N1, N2, est_sigma_w, D_dfe, M1_dfe, M2_dfe, false, false)
          ;
    rcv_bits = ibmap(rcv_symb);
    packet = ibmap(packet);

    % Compute the Pbit and store it
    Pbit_estch_uncoded(snr_idx) = sum(xor(rcv_bits, packet))/length(packet);

end

% Save current results
save('Problem2_estch_uncoded', 'snr_vec_estch_uncoded', ...
    'seq_lengths_estch_uncoded', 'Pbit_estch_uncoded');

%% Estimated channel, DFE, coded data

L = 31;
Nseq = 7;
```

```
% Get optimal number of bits
desired_bits = 2^22;
% Compute the closest number of bits that both interleaver and encoder will like
search_step = 32400;
bit_number = ceil(desired_bits / search_step) * search_step;

numsim = 10;

snr_vec_estch_coded = [1, 2, 3, 3.2:0.02:3.6];  % Pbit falls at 3.5 dB
seq_lengths_estch_coded = bit_number*ones(1, length(snr_vec_estch_coded));
Pbit_estch_coded = zeros(length(snr_vec_estch_coded),numsim);

for sim = 1:numsim
    parfor snr_idx = 1:length(snr_vec_estch_coded)
        curr_snr = snr_vec_estch_coded(snr_idx);
        fprintf('Estimated channel, coded, snr = %.2f\n', curr_snr);

        % Generate the current needed sequence
        packet = randi([0 1], 1, seq_lengths_estch_coded(snr_idx));

        enc_packet = encodeBits(packet);
        int_enc_packet = interleaver(enc_packet);  % Interleave the encoded bits

        symbols = [ts_generation(L, Nseq); bitmap(int_enc_packet.')];

        % Send through the channel
        [rcv_symb, sigma_w, ~] = channel_output(symbols, 10^(curr_snr/10), false);

        % Perform estimation
        [h, est_sigma_w] = get_channel_info(rcv_symb(t0+1:t0+L+Nseq), N1, N2);

        rcv_symb = rcv_symb(t0+1:end-7)/h(N1+1);
        hi = h / h(N1+1);

        % Receiver: filter with DFE
        [Jmin, psi, rcv_symb] = DFE_filter(rcv_symb, hi, N1, N2, est_sigma_w, D_dfe, M1_dfe, M2_dfe,
              true, false);

        noise_var = (Jmin-sigma_a*abs(1-psi(D_dfe+1))^2)/abs(psi(D_dfe+1))^2; % This includes white
              noise and isi

        % Compute Log Likelihood Ratio
        llr = zeros(2*length(packet),1);
        llr(1:2:end) = -2*real(rcv_symb(L+Nseq+1:end))/(noise_var/2);
        llr(2:2:end) = -2*imag(rcv_symb(L+Nseq+1:end))/(noise_var/2);

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

        dec_packet = decodeBits(llr).';

        % Compute the Pbit and store it
        Pbit_estch_coded(snr_idx, sim) = sum(xor(dec_packet, packet))/length(packet);

    end
end
% Save current results
save('Problem2_estch_coded', 'snr_vec_estch_coded', ...
    'seq_lengths_estch_coded', 'Pbit_estch_coded');

%% Plot BER graphs

load('Problem2_estch_uncoded.mat');
load('Problem2_estch_coded.mat');
load('Problem2_knownch_uncoded.mat');
load('Problem2_knownch_coded.mat');

figure, semilogy(snr_vec_knownch_uncoded, Pbit_knownch_uncoded), hold on
semilogy(snr_vec_knownch_coded, Pbit_knownch_coded), hold on
semilogy(snr_vec_estch_uncoded, Pbit_estch_uncoded, '--')
semilogy(snr_vec_estch_coded, Pbit_estch_coded, '--')
```

```
xlabel('\Gamma [dB]'), ylabel('Pbit'), grid on
ylim([10^-5, 10^-1])
xlim([0, 14])
legend('Known channel, uncoded', 'Known channel, coded', ...
    'Estimated channel, uncoded', 'Estimated channel, coded');
title('Bit Error Rate for a DFE receiver')

%% Clean parpool
delete(gcp)
```

DFE_filter

```
function [ Jmin, psi, r ] = DFE_filter(x, hi, N1, N2, est_sigmaw, D, M1, M2, coding, verb)
% Function that performs DFE filtering. It needs
% packet is the sequence of sent symbols
% x is the received samples vector in T, normalized by h0
% h is the estimated impulse response in T, normalized by h0
% N1 is the estimated number of precursors
% N2 is the estimated number of postcursors
% est_sigmaw is the estimated variance of the noise
% t0 timing phase
% D is the delay of the FF filter
% M1 is the number of coefficients of FF filter
% M2 is the number of coefficients of FB filter, set to 0 to get LE
% if verb is true then plots will be plotted

% Power of the input sequence
sigma_a = 2;

% Zero padding of the i.r.
nb0 = 60;
nf0 = 60;
hi = [zeros(nb0,1); hi; zeros(nf0,1)];

% Get the Weiner-Hopf solution
p = zeros(M1, 1);
for i = 0:(M1 - 1)
    p(i+1) = sigma_a * conj(hi(N1+nb0+1+D-i));
end

R = zeros(M1);
for row = 0:(M1-1)
    for col = 0:(M1-1)
        first_sum = (hi((nb0+1):(N1+N2+nb0+1))).' * ...
            conj(hi((nb0+1-(row-col)):(N1+N2+nb0+1-(row-col))));
        second_sum = (hi((N1+nb0+1+1+D-col):(N1+nb0+1+M2+D-col))).' * ...
            conj((hi((N1+nb0+1+1+D-row):(N1+nb0+1+M2+D-row))));
        r_w = (row == col) * est_sigmaw; % This is a delta only if there is no g_M.

        R(row+1, col+1) = sigma_a * (first_sum - second_sum) + r_w;

    end
end

c_opt = R \ p;

Jmin = sigma_a*(1-c_opt.'* flipud(hi(N1+nb0+1+D-M1+1:N1+nb0+1+D)));

b = zeros(M2,1);
for i = 1:M2
    b(i) = - (fliplr(c_opt.')*hi((i+D+N1+nb0+1-M1+1):(i+D+N1+nb0+1)));
end

psi = conv(hi(nb0+1:nb0+1+N1+N2),c_opt);

if (verb == true)
    % Plot hhat, c, psi=conv(h,c), b and get a sense of what is happening
    figure
    subplot(4,1,1)
    stem(-N1:N2, abs(hi(nb0+1:nb0+1+N1+N2))), title('|h_{hat} normalized|'), xlim([-5 8]), ylim([0 1])
    subplot(4,1,2)
    stem(abs(c_opt)), title('|c|'), xlim([1 14]), ylim([0 1])
    subplot(4,1,3)
    stem(-N1:-N1+length(psi)-1, abs(psi)), title('|psi|'), xlim([-N1 -N1+length(psi)]), ylim([0 1])
    subplot(4,1,4)
    stem(abs(b)), title('|b|'), xlim([1 14]), ylim([0 1])
end

y = zeros(length(x) + D , 1); % output of ff filter
r = zeros(length(x) + D, 1);
detected = zeros(length(x) + D, 1); % output of td
for k = 0:length(x) - 1 + D
    if (k < M1 - 1)
        xconv = [flipud(x(1:k+1)); zeros(M1 - k - 1, 1)];
    elseif k > length(x)-1 && k < length(x) - 1 + M1
        xconv = [zeros(k-length(x)+1, 1); flipud(x(end - M1 + 1 + k - length(x) + 1:end))];
    elseif k >= length(x) - 1 + M1 % just in case D is greater than M1
        xconv = zeros(M1, 1);
    else
        xconv = flipud(x(k-M1+1 + 1:k + 1));
    end

    if (k <= M2)
        a_old = [flipud(detected(1:k)); zeros(M2 - k, 1)];
    else
        a_old = flipud(detected(k-M2+1:k));
    end

    y(k+1) = c_opt.'*xconv;

    y(k+1) = y(k+1) / psi(D+1); % normalize y

    r(k+1) = y(k+1) + b.'*a_old;
    detected(k+1) = qpsk_td(r(k+1));
end

if (coding == true)
   r = r(D+1:end);
else % if coding is not used return the decisions
   r = detected(D+1:end);
end

end
```

LE_DFE_param_evaluator

```
%% This script performs parametric sweep on the parameters for DFE, for
% different SNR and the assigned channel

clear
close all
clc
rng default
snr_vec = [0, 14]; % dB

% From the assignment
t0 = 6;
N1 = 0;
N2 = 4;
N = N1 + N2 + 1;

%% DFE (parametric evaluation without channel coding)

printmsg_delete = ''; % Just to display progress updates

M1_max = N+30;
D_max = M1_max-1;
JminDFE = zeros(length(snr_vec), M1_max, D_max);
```

```
L_data = 128; % useless
bits = randi([0 1], 1, L_data);
symbols = bitmap(bits);

for snr_i = 1:length(snr_vec)
    snr_ch = snr_vec(snr_i);

    % Create, send and receive data with the given channel
    snrlin = 10^(snr_ch/10);
    [rcv_symb, sigma_w, h] = channel_output(symbols, snrlin, false);

    % Normalization!
    rcv_symb = rcv_symb(t0:end-7)/h(t0);
    hi = h(t0-N1:t0+N2)/h(t0);

    %% Receiver: compute Jmin

    for M1_dfe = 1:M1_max;   % FF filter: equal to the span of h
        for D_dfe = 1:D_max
            % Print progress update
            printmsg = sprintf('for DFE, snr = %d, M1 = %d, D = %d\n', snr_ch, M1_dfe, D_dfe);
            fprintf([printmsg_delete, printmsg]);
            printmsg_delete = repmat(sprintf('\b'), 1, length(printmsg));

            M2_dfe = N2 + M1_dfe - 1 - D_dfe;  % FB filter: one less than the FF filter
            [JminDFE(snr_i, M1_dfe, D_dfe), ~] = DFE_filter(rcv_symb, hi.', N1, N2, sigma_w, D_dfe,
                  M1_dfe, M2_dfe, false, false);
        end
    end
end

save('jmin_DFE', 'JminDFE')

for i = 1:length(snr_vec)
    figure, mesh(1:D_max, 1:M1_max, 10*log10(reshape(abs(JminDFE(i, :, :)), size(JminDFE(i, :, :), 2),
          size(JminDFE(i, :, :), 3))))
    title(strcat('Jmin for DFE, snr= ', num2str(snr_vec(i))))
    xlabel('D'), ylabel('M1'), zlabel('Jmin [dB]')
end
```

get_channel_info

```
function [ h_i, est_sigmaw ] = get_channel_info( r, N1, N2 )
%GET_CHANNEL_INFO

L = 31;
Nseq = 7;
trainingsymbols = ts_generation(L, Nseq);


% --- Estimate impulse response h @T and compute estimated noise power
N = N1+N2+1;
x_for_ls = trainingsymbols(end - (L+N-1) + 1 : end);
% r is in T
d_for_ls = r(end - (L+N-1) + 1 - N1 : end - N1 );
[h_i, r_hat] = h_estimation_onebranch(x_for_ls, d_for_ls, L, N);
d_no_trans = d_for_ls(N : N+L-1);
est_sigmaw = sum(abs(r_hat - d_no_trans).^2)/length(r_hat);

h_i = h_i.'; % for convenience

end
```

h_estimation_onebranch

```
function [ h_hat, d_hat ] = h_estimation_onebranch( x, d, L, N )
% This function performs the estimation of the h coefficients, given the
% input sequence, the output of the channel and the number of coefficients
% to estimate. Additionally, the function outputs d_hat, i.e. the output of
% a channel that would have the estimated coefficients as impulse response.

% In this case the estimation cannot be performed.
if N > L
    h_hat = [];
    d_hat = [];
    return
end

%% Estimate h


% Using the data matrix (page 246), easier implementation
h_hat = zeros(1,N);
I = zeros(L,N);
for column = 1:N
    I(:,column) = x(N-column+1:(N+L-column));
end
o = d(N:N + L - 1);

% Compute the Phi matrix and the theta vector
Phi = I'*I;
theta = I'*o;

h_hat(1, 1:N) = Phi \ theta;

%% Compute d_hat

d_hat = conv(x, h_hat);
d_hat = d_hat(N : N+L-1);

end
```

ts_generation

```
function [ trainingsymbols ] = ts_generation( L, Nseq )
% Training sequence generation
% This function outputs a partially repeated ML sequence in which the
% symbols are 1+j and -1-j (two orthogonal symbols from the QPSK alphabet).

mlseq = MLsequence(L); % Get the 0-1 ML sequence

% Replace every 0 with two 0s and every 1 with two 1s
mlseqdouble = zeros(2*L,1);
for i = 1:L
    switch mlseq(i)
        case 0
            mlseqdouble(2*i-1) = 0;
            mlseqdouble(2*i) = 0;
        case 1
            mlseqdouble(2*i-1) = 1;
            mlseqdouble(2*i) = 1;
    end
end

% Repeat the sequence and bitmap it to get the symbols
trainingseq = [mlseqdouble; mlseqdouble(1:2*Nseq)];
trainingsymbols = bitmap(trainingseq);

end
```

txrc

```
function [packet, r, sigma_w] = txrc(L_data, snr)

% This script produces an output given by the 50 bit of ML training sequence and
% L_data bit computed with an MLsequence (therefore L_data has to be 2^smth
% - 1)

%% Packet generation w/ ts, data
L = 31;
Nseq = 7;

trainingsymbols = ts_generation(L, Nseq);

MAX_ML = 2^20 - 1;
if (L_data > MAX_ML)
    dataseq = MLsequence(MAX_ML);
    dataseq = repmat(dataseq, ceil(L_data / MAX_ML), 1);
    dataseq = dataseq(1:L_data);
else
    dataseq = MLsequence(L_data);
end
datasymbols = bitmap(dataseq(1 : end - mod(L_data, 2)));
% QPSK requires an even number of bits

packet = [trainingsymbols; datasymbols];
%% Generate the channel output
snr_lin = 10^(snr/10);
[r, sigma_w, ~] = channel_output(packet, snr_lin, false);

end
```

OFDM_choose_N2

```
%% Channel ESTIMATION for OFDM
% Send one block of data with symbols spaced of 16 channels

clear, close all
OFDM = true;
M = 512;
allowed_symb = 32;
spacing = M/allowed_symb;
Npx = 7;
t0 = 5;

block = ones(M, 1)*(-1-1i);

ts = ts_generation(allowed_symb-1, 1) * sqrt(2);
init_step = 1; % < 16
indices = init_step : spacing : init_step + spacing*(allowed_symb-1);
block(indices) = ts;

% Compute IDFT, add prefix, P/S
A = ifft(block);
A_pref = [A(end-Npx + 1:end); A];
s = reshape(A_pref, [], 1);

%% Channel output

snr = 6; %dB
snr_lin = 10^(snr/10);
% Send over the noisy channel
[r, sigma_w, g] = channel_output(s, snr_lin, OFDM);
g = g(1+t0 : end);   % Take t0 into account (just to plot stuff)
G = fft(g, 512);
G = G(:);


%% Process at the receiver
```

```
r = r(1+t0 : end - mod(length(r), M+Npx) + t0);

% Perform the DFT
r_matrix = reshape(r, M+Npx, []);
r_matrix = r_matrix(Npx + 1:end, :);
x_matrix = fft(r_matrix);

% Select useful samples
x_rcv = x_matrix(indices, 1);
x_known = diag(ts);

% Compute G_est by dividing the received symbol by the transmitted one
G_est = x_rcv ./ ts;

% Solve LS for F*g=G_est where g is an 8x1 vector: do it for different values of N2
F_complete = dftmtx(M);

for N2 = 1:Npx
   F = F_complete(indices, 1:N2+1);
   g_hat = (F' * F) \ (F' * G_est);
   g_est = ifft(G_est);
   G_hat = fft(g_hat, M);

   % Noise estimation
   xhat = x_known * G_hat(indices);
   E = sum(abs(xhat - x_rcv).^2)/length(xhat);
   est_sigma_w(N2) = E/M; %#ok<SAGROW>
end

%% Plot

figure, hold on
plot(1:Npx, 10*log10(est_sigma_w))
plot([1 Npx], 10*log10(sigma_w)*[1 1])
title('Error functional for channel estimation varying N2')
xlabel('N2'), ylabel('Error functional (dB)')
grid on, box on
```

OFDM_channel_estimation

```
function [G_hat, est_sigma_w] = OFDM_channel_estimation(snr, Npx, N2, t0)

%% Channel ESTIMATION for OFDM
% Send one block of data with symbols spaced of 16 channels

OFDM = true;
M = 512;
allowed_symb = 32;
spacing = M/allowed_symb;

block = ones(M, 1)*(-1-1i);

% Remember: for the symbols on which the estimation is performed, for a
% given snr (computed with the usual sigma_a^2 = 2), the power of the
% "estimation symbols" is doubled (-> better snr)
% Note that the variance of the noise at the receiver, after the DFT, is
% multplied by M, therefore it could be high
ts = ts_generation(allowed_symb-1, 1) * sqrt(2);
init_step = 1; % < 16
indices = init_step : spacing : init_step + spacing*(allowed_symb-1);
% Scale in order to double the power of tx symbols
block(indices) = ts;

% Compute IDFT, add prefix, P/S
A = ifft(block);
A_pref = [A(end-Npx + 1:end); A];
s = reshape(A_pref, [], 1);
```

```
%% Transmission and reception of the training sequence

snr_lin = 10^(snr/10);
[r, ~, ~] = channel_output(s, snr_lin, OFDM);


%% Process at the receiver

r = r(1+t0 : end - mod(length(r), M+Npx) + t0);

% Perform the DFT
r_matrix = reshape(r, M+Npx, []);
r_matrix = r_matrix(Npx + 1:end, :);
x_matrix = fft(r_matrix);

% Select useful samples
x_rcv = x_matrix(init_step:spacing:end, 1);
X_known = diag(ts)*sqrt(2);

% Compute G_est by dividing the received symbol by the transmitted one
G_est = x_rcv ./ ts;

% Solve LS for F*g=G_est where g is an 8x1 vector
F = dftmtx(M);
F = F(indices, 1:N2+1);
g_hat = (F' * F) \ (F' * G_est);
G_hat = fft(g_hat, M);

% Noise estimation
xhat = X_known * G_hat(init_step : spacing : end);
E = sum(abs(xhat - x_rcv).^2)/length(xhat);
est_sigma_w = E/M;


end
```

OFDM_channel_estimation_2

```
function [G_hat, est_sigma_w] = OFDM_channel_estimation_2(snr, Npx, N2, t0)

%% Channel ESTIMATION for OFDM (second method)
% Send one block of data using 8 equally spaced groups of 4
% adjacent subchannels.

OFDM = true;
M = 512;
allowed_symb = 32;

block = ones(M, 1)*(-1-1i);
% Scale in order to double the power of tx symbols
ts = ts_generation(allowed_symb-1, 1) * sqrt(2);
sigma_ts = 4;


nsamples = 8;
symbpersegment = allowed_symb / nsamples;
indices = reshape(1:M, M/nsamples, nsamples);
indices = reshape(indices(1:symbpersegment, :), size(indices, 2)*symbpersegment, 1); % Second way
block(indices) = ts;

% Compute IDFT, add prefix, P/S
A = ifft(block);
A_pref = [A(end-Npx + 1:end); A];
s = reshape(A_pref, [], 1);

%% Send and receive
snr_lin = 10^(snr/10);
% Send over the noisy channel
[r, ~, ~] = channel_output(s, snr_lin, OFDM);
```

```
% --- Process at the receiver

r = r(1+t0 : end - mod(length(r), M+Npx) + t0);

% Perform the DFT
r_matrix = reshape(r, M+Npx, []);
r_matrix = r_matrix(Npx + 1:end, :);
x_matrix = fft(r_matrix);

% Select useful samples
x_rcv = x_matrix(indices, 1);

% Compute G_est by dividing the received symbol by the transmitted one
G_est = x_rcv ./ ts;

% Solve LS for F*g=G_est where g is an (N2+1)x1 vector
F = dftmtx(M);
F = F(indices, 1:N2+1);
g_hat = (F' * F) \ (F' * G_est);
G_hat = fft(g_hat, M);

% Noise estimation new method
est_sigma_w = 0;
for j=0:nsamples-1
   tempGest = G_est((j*symbpersegment + 1) : (j*symbpersegment + 1)+3);
   est_sigma_w = est_sigma_w + var(tempGest);
end
est_sigma_w = est_sigma_w / nsamples / M * sigma_ts;


end
```

OFDM_channel_estimation_comparison

```
%% Channel ESTIMATION for OFDM (second method)
% Send one block of data using 8 equally spaced groups of 4
% adjacent subchannels.

%#ok<*SAGROW>

clear, close all
numsim = 1000;
OFDM = true;
M = 512;
allowed_symb = 32;
Npx = 7;
N2 = 4;
t0 = 5;

snr_vec = 0:2:24;

block = ones(M, 1)*(-1-1i);
% Scale in order to double the power of tx symbols
ts = ts_generation(allowed_symb-1, 1) * sqrt(2);
sigma_ts = 4;


%% First method

indices = 1 : M/allowed_symb : M;
block(indices) = ts;

% Compute IDFT, add prefix, P/S
A = ifft(block);
A_pref = [A(end-Npx + 1:end); A];
s = reshape(A_pref, [], 1);
```

```
for snr_i = 1:length(snr_vec)
   for sim=1:numsim

      snr = snr_vec(snr_i); %dB
      snr_lin = 10^(snr/10);

      % --- Send over the noisy channel
      [r, sigma_w(snr_i), g] = channel_output(s, snr_lin, OFDM);
      g = g(1+t0 : end);   % Take t0 into account (just to plot stuff)
      G = fft(g, 512);
      G = G(:);


      % --- Process at the receiver

      r = r(1+t0 : end - mod(length(r), M+Npx) + t0);

      % Perform the DFT
      r_matrix = reshape(r, M+Npx, []);
      r_matrix = r_matrix(Npx + 1:end, :);
      x_matrix = fft(r_matrix);

      % Select useful samples
      x_rcv = x_matrix(indices, 1);
      x_known = diag(ts);

      % Compute G_est by dividing the received symbol by the transmitted one
      G_est = x_rcv ./ ts;

      % Solve LS for F*g=G_est where g is an 8x1 vector
      F = dftmtx(M);
      F = F(indices, 1:N2+1);
      g_hat = (F' * F) \ (F' * G_est);
      g_est = ifft(G_est);
      G_hat = fft(g_hat, M);

      % Noise estimation original
      xhat = x_known * G_hat(indices);
      E = sum(abs(xhat - x_rcv).^2)/length(xhat);
      est_sigma_w(sim, snr_i) = E/M;

      % Error on the estimate of G
      est_err(sim, snr_i) = sum(abs(G_hat - G).^2) / M;

   end

end

% Save simulation results
meanest1 = mean(est_sigma_w);
ciest1 = 1.96 * std(est_sigma_w) / sqrt(numsim);
meanesterr = mean(est_err);
ciesterr = 1.96 * std(est_err) / sqrt(numsim);


%% Second method

nsamples = 8;
symbpersegment = allowed_symb / nsamples;
indices = reshape(1:M, M/nsamples, nsamples);
indices = reshape(indices(1:symbpersegment, :), size(indices, 2)*symbpersegment, 1); % Second way
block(indices) = ts;

% Compute IDFT, add prefix, P/S
A = ifft(block);
A_pref = [A(end-Npx + 1:end); A];
s = reshape(A_pref, [], 1);

for snr_i = 1:length(snr_vec)
   for sim=1:numsim
```

```
      snr = snr_vec(snr_i); %dB
      snr_lin = 10^(snr/10);
      % Send over the noisy channel
      [r, sigma_w(snr_i), g] = channel_output(s, snr_lin, OFDM);
      g = g(1+t0 : end);   % Take t0 into account (just to plot stuff)
      G = fft(g, 512);
      G = G(:);

      % --- Process at the receiver

      r = r(1+t0 : end - mod(length(r), M+Npx) + t0);

      % Perform the DFT
      r_matrix = reshape(r, M+Npx, []);
      r_matrix = r_matrix(Npx + 1:end, :);
      x_matrix = fft(r_matrix);

      % Select useful samples
      x_rcv = x_matrix(indices, 1);
      x_known = diag(ts);

      % Compute G_est by dividing the received symbol by the transmitted one
      G_est = x_rcv ./ ts;

      % Solve LS for F*g=G_est where g is an 8x1 vector
      F = dftmtx(M);
      F = F(indices, 1:N2+1);
      g_hat = (F' * F) \ (F' * G_est);
      g_est = ifft(G_est);
      G_hat = fft(g_hat, M);

      % Noise estimation new method
      est_sigma_w(sim, snr_i) = 0;
      for j=0:nsamples-1
         tempGest = G_est((j*symbpersegment + 1) : (j*symbpersegment + 1)+3);
         est_sigma_w(sim, snr_i) = est_sigma_w(sim, snr_i) + var(tempGest);
      end
      est_sigma_w(sim, snr_i) = est_sigma_w(sim, snr_i) / nsamples / M * sigma_ts;


      % Error on the estimate of G
      est_err(sim, snr_i) = sum(abs(G_hat - G).^2) / M;
   end
end

% Save simulation results
meanest2 = mean(est_sigma_w);
ciest2 = 1.96 * std(est_sigma_w) / sqrt(numsim);
meanesterr2 = mean(est_err);
ciesterr2 = 1.96 * std(est_err) / sqrt(numsim);


%% Plots

figure, hold on
errorbar(snr_vec, meanest1, ciest1)
errorbar(snr_vec, meanest2, ciest2)
plot(snr_vec, sigma_w)
hold off
grid on, box on, set(gca, 'yscale', 'log')
legend('First method', 'Second method', 'Actual \sigma_w')
xlim([snr_vec(1), snr_vec(end)])
ax = gca; ax.XTick = snr_vec;
xlabel('SNR (dB)')
ylabel('Estimated \sigma_w^2')
title('Comparison of estimates of \sigma_w^2')

figure, hold on
errorbar(snr_vec, meanesterr, ciesterr)
errorbar(snr_vec, meanesterr2, ciesterr2)
hold off
```

```
grid on, box on, set(gca, 'yscale', 'log')
legend('First method', 'Second method')
xlim([snr_vec(1), snr_vec(end)])
ax = gca; ax.XTick = snr_vec;
xlabel('SNR (dB)')
ylabel('Estimation error')
title('Estimation error on G')


% Plots for second method

figure, hold on
stem(0:Npx, abs(g))
stem(0:N2, abs(g_hat), 'x')
stem(0:15, abs(g_est(1:16)), '^')
legend('Actual g', 'g_hat', 'IDFT of G_est')

figure,
subplot 211
plot(real(G)), hold on
plot(real(G_hat))
plot(indices, real(G_est), '^')
title(strcat('Comparison between estimated - LS+interpol - and real at ', num2str(snr), ' dB'))
legend('real(G)', 'real(G_hat)', 'real(G_{est})'), xlabel('i - subchannels'), ylabel('Real(G)'),
grid on, xlim([1, M])

subplot 212
plot(imag(G)), hold on
plot(imag(G_hat))
plot(indices, imag(G_est), '^')
title(strcat('Comparison between estimated - LS+interpol - and real at ', num2str(snr), ' dB'))
legend('imag(G)', 'imag(G_hat)', 'imag(G_{est})'), xlabel('i - subchannels'), ylabel('imag(G)'),
grid on, xlim([1, M])
```

OFDM_pbit_estimator

```
%% ODFM script, it calls the OFDM function and performs different BER estimates
clear
close all
clc
rng default

%% Data
M = 512;
Npx = 7;
desired_bits = 2^22;
estMethod = 2;

%% BER with coding, known channel

isKnown = true;
coding = true;
snr_vec_coding_known = 0:0.05:4; % no more is needed
BER_coding_known = zeros(length(snr_vec_coding_known), 1);
for snr_i = 1:length(snr_vec_coding_known)
    snr_c = snr_vec_coding_known(snr_i);
    fprintf('Coded, snr = %.2f\n', snr_c);
    [BER_coding_known(snr_i), ~] = OFDM_BER(M, Npx, desired_bits, snr_c, coding, isKnown);
end

save('OFDM_coded_known', 'BER_coding_known', 'snr_vec_coding_known', 'desired_bits');

%% BER without coding, known channel

coding = false;
snr_vec_nocoding_known = 0:15;
BER_nocoding_known = zeros(length(snr_vec_nocoding_known), 1);
for snr_i = 1:length(snr_vec_nocoding_known)
    snr_nc = snr_vec_nocoding_known(snr_i);
    fprintf('Uncoded, snr = %.2f\n', snr_nc);
    [BER_nocoding_known(snr_i), ~] = OFDM_BER(M, Npx, desired_bits, snr_nc, coding, isKnown);
end

save('OFDM_uncoded_known', 'BER_nocoding_known', 'snr_vec_nocoding_known', 'desired_bits');

%% BER with coding, estimated channel

isKnown = false;
coding = true;
snr_vec_coding_estimated = [0, 1, 2:0.05:3];
BER_coding_estimated = zeros(length(snr_vec_coding_estimated), 1);
for snr_i = 1:length(snr_vec_coding_estimated)
    snr_c = snr_vec_coding_estimated(snr_i);
    fprintf('Coded, snr = %.2f\n', snr_c);
    [BER_coding_estimated(snr_i), ~] = OFDM_BER(M, Npx, N2, t0, desired_bits, snr_c, coding, isKnown,
          estMethod);

end

save('OFDM_coded_estimated', 'BER_coding_estimated', 'snr_vec_coding_estimated', 'desired_bits');

%% BER without coding, estimated channel

coding = false;
snr_vec_nocoding_estimated = 0:15;
BER_nocoding_estimated = zeros(length(snr_vec_nocoding_estimated), 1);
for snr_i = 1:length(snr_vec_nocoding_estimated)
    snr_nc = snr_vec_nocoding_estimated(snr_i);
    fprintf('Uncoded, snr = %.2f\n', snr_nc);
    [BER_nocoding_estimated(snr_i), ~] = OFDM_BER(M, Npx, N2, t0, desired_bits, snr_nc, coding,
          isKnown, estMethod);
end

save('OFDM_uncoded_estimated', 'BER_nocoding_estimated', 'snr_vec_nocoding_estimated', 'desired_bits')
      ;

%% BER with coding, estimated channel, first method

estMethod = 1;
isKnown = false;
coding = true;
snr_vec_coding_estimated = [0, 1, 2:0.05:3];
BER_coding_estimated = zeros(length(snr_vec_coding_estimated), 1);
for snr_i = 1:length(snr_vec_coding_estimated)
    snr_c = snr_vec_coding_estimated(snr_i);
    fprintf('Coded, snr = %.2f\n', snr_c);
    [BER_coding_estimated(snr_i), ~] = OFDM_BER(M, Npx, N2, t0, desired_bits, snr_c, coding, isKnown,
          estMethod);

end

save('OFDM_coded_estimated_1', 'BER_coding_estimated', 'snr_vec_coding_estimated', 'desired_bits');

%% BER without coding, estimated channel, first method

coding = false;
snr_vec_nocoding_estimated = 0:15;
BER_nocoding_estimated = zeros(length(snr_vec_nocoding_estimated), 1);
for snr_i = 1:length(snr_vec_nocoding_estimated)
    snr_nc = snr_vec_nocoding_estimated(snr_i);
    fprintf('Uncoded, snr = %.2f\n', snr_nc);
    [BER_nocoding_estimated(snr_i), ~] = OFDM_BER(M, Npx, N2, t0, desired_bits, snr_nc, coding,
          isKnown, estMethod);
end

save('OFDM_uncoded_estimated_1', 'BER_nocoding_estimated', 'snr_vec_nocoding_estimated', 'desired_bits
      ');
```

OFDM_BER

```
function [ BER, G ] = OFDM_BER( M, Npx, N2, t0, desired_bits, snr, coding, chIsKnown, varargin )
%This function performs the transmission and reception of bits with ODFM,
%with or without encoding
%   It needs
%   - M block length
%   - Npx prefix length
%   - the number of desired bits
%   - the snr
%   - the option coding to be set either true or false
%   - the estimation method (1 standard, 2 8 points+noise) - optional, if
%   not indicated the classic is used
%   It returns
%   - the BER
%   - the channel frequency response
%   This implementation doesn't rescale the power of sent symbols after the
%   IFFT operation, therefore sigma_s^2 = sigma_a^/M

if (length(varargin) == 1)
    if (varargin{1} == 1)
        fprintf('Use classic estimation method \n')
        estMethod = 1;
    elseif (varargin{1} == 2)
        fprintf('Use new estimation method \n')
        estMethod = 2;
    end
else
    fprintf('Use classic estimation method \n')
    estMethod = 1;
end

warning('off', 'all');
OFDM = true;

% Compute the optimal number of bits
fprintf('Start transmission...\n');
if (coding == true)
    % Compute the closest number of bits that both interleaver and encoder will like
    search_step = 32400;
    bit_number = ceil(desired_bits / search_step) * search_step;
else
    bit_number = desired_bits;
end

% Generate and encode bits
bits = randi([0 1], 1, bit_number).';

if (coding == true)
    enc_bits = encodeBits(bits.');
    int_enc_bits = interleaver(enc_bits);  % Interleave the encoded bits
    a = bitmap(int_enc_bits.');
else
    a = bitmap(bits);
end

% Create data blocks

% perform a padding of the last symbols in order to have blocks of 512 symbols: we use
% -1-j to perform the padding
a_pad = [a; ones(M - mod(length(a), M), 1) * (-1-1i)];
a_matrix = reshape(a_pad, M, []); % it should mantain columnwise order

% compute the ifft of blocks of 512 symbols
% http://it.mathworks.com/matlabcentral/newsreader/view_thread/17104
% it should be a columnwise operation!
A_matrix = ifft(a_matrix);

% add the preamble to each column
A_matrix = [A_matrix(M-Npx+1:M, :); A_matrix]; % very powerful operation
```

```
% serialize in order to call channel output
s = reshape(A_matrix, [], 1);

fprintf('Symbols are pushed into the channel...\n');
% Send over the noisy channel
snr_lin = 10^(snr/10);
[r, sigma_w, g] = channel_output(s, snr_lin, OFDM);
if (chIsKnown)
    g = g(1+t0 : end);
    G = fft(g, 512);
    G = G(:);
else
    if(estMethod == 1)
        [G, sigma_w] = OFDM_channel_estimation(snr, Npx, N2, t0);
    elseif(estMethod == 2)
        [G, sigma_w] = OFDM_channel_estimation_2(snr, Npx, N2, t0);
    end
end

% Process at the receiver
% consider the effect of the convolution at the end should be easy,
% since the resulting data should have a length which is a multiple of M +
% Npx
fprintf('Symbols received, processing begins...\n');
r = r(1+t0 : end - mod(length(r), M+Npx) + t0);

% perform the DFT
r_matrix = reshape(r, M+Npx, []);
r_matrix = r_matrix(Npx + 1:end, :);

G_inv = G.^(-1);
x_matrix = fft(r_matrix);

y_matrix = bsxfun(@times, x_matrix, G_inv);

% Detect and compute BER
if (coding == true)
    % sigma_i after the DFT and the scaling by G_i of each branch
    sigma_i = 0.5*sigma_w*M*abs(G_inv).^2;
    % Compute Log Likelihood Ratio
    % It is different for each branch
    llr_real = -2*bsxfun(@times, real(y_matrix), sigma_i.^(-1));
    llr_imag = -2*bsxfun(@times, imag(y_matrix), sigma_i.^(-1));
    llr_real_ar = reshape(llr_real, [], 1);
    llr_imag_ar = reshape(llr_imag, [], 1);
    llr = zeros(numel(llr_real) + numel(llr_imag), 1);
    llr(1:2:end) = llr_real_ar;
    llr(2:2:end) = llr_imag_ar;
    % Drop the zero padding
    llr = llr(1:length(enc_bits));
    % Decode the bits
    llr = deinterleaver(llr); % Deinterleave the loglikelihood ratio first
    dec_bits = decodeBits(llr).';
else
    y = reshape(y_matrix, [], 1);
    % drop the zero padding
    y = y(1:length(a));
    decision = zeros(length(y), 1);
    for k = 1:length(y)
        decision(k) = qpsk_td(y(k));
    end
    dec_bits = ibmap(decision);
end

% make dec_bits a column even if it is already
dec_bits = dec_bits(:);

BER = sum(xor(dec_bits, bits))/length(bits);
fprintf('End, the BER is %d \n', BER);

% my_llr_linear = 10.^(llr(1:100)/10);
% my_dec_bits = dec_bits(1:100);
% my_p = 1 ./ (1 + my_llr_linear);
% figure, stem(my_p), hold on, stem(my_dec_bits)

end
```