Digital Communications and Laboratory
Fourth Homework

Faccin Dario, Santi Giovanni

MATLAB code

```
clc; close all; clear global; clearvars;

load('Useful.mat', 'qc');
load('Input_symbols.mat');

SNR_vect = 1.4:0.05:1.8;
sigma_a = 2;                            % Input variance
M = 4;                          % Constellation cardinality

gm = conj(qc(end:-1:1));    % Matched filter: complex conjugate of qc
h = conv(qc,gm);                        % Impulse response

t0_bar = find(h == max(h));             % Timing phase: peak of h
h = h(h>max(h)/100);
h = h(3:end-2);
h_T = downsample(h,4);

r_gm = xcorr(gm,gm);                    % Matched filter autocorrelation
Pbit_DFE_code = zeros(length(SNR_vect),1);
N2 = floor(length(h_T)/2);
N1 = N2;
M1 = 5;
D = 4;
M2 = N2 + M1 - 1 - D;
tic
parfor i=1:length(SNR_vect)
        snr_db = SNR_vect(i);
        snr_lin = 10^(snr_db/10);
        % Single carrier channel simulation
        [r_c, sigma_w, qc] = channel_sim(symbols_ak, snr_db, sigma_a);
        % Additive complex-Gaussian noise
        w = wgn(length(r_c),1, 10*log10(sigma_w), 'complex');
        r_c = r_c + w;
        r_c_prime = filter(gm,1,r_c);
        r_c_prime = r_c_prime(t0_bar:end);
        % Signal at the input of the DFE
        x_aa = downsample(r_c_prime,4);
        rw_tilde = sigma_w/4 .* downsample(r_gm, 4);
        % DFE
        [c_opt, Jmin] = Adaptive_DFE(h_T, rw_tilde, sigma_a, M1, M2, D);
        psi = conv(c_opt, h_T);
        b = - psi(end - M2 + 1:end);
        y_hat = x_aa/max(psi);
        % Equalized signal
        detected = equalization_DFE(y_hat, c_opt, b, M1, M2, D);
        % Decoder as LLR
        llr = zeros(2*length(detected),1);
        Jmin_lin = 10^(Jmin/10);
        noise_var = (Jmin_lin-sigma_a*abs(1-max(psi))^2)/(abs(max(psi))^2);
        llr(1:2:end) = -2*real(detected)/(noise_var/2);
        llr(2:2:end) = -2*imag(detected)/(noise_var/2);
        % Deinterleave and decode bits
        llr_deinter = deinterleaver(llr);
        decoded = LDPC_decoder(llr_deinter).';
```

```
        Pbit_DFE_code(i) = length(find(x(1:length(decoded))~=decoded))/length(decoded);
end
toc
```

```
clc; close all; clear global; clearvars;

load('Useful.mat', 'qc');
load('Input_symbols.mat');
symbols = bitmap(x).';
SNR_vect = 0:14;
sigma_a = 2;                    % Input variance
M = 4;                          % Constellation cardinality
gm = conj(qc(end:-1:1));                % Matched filter: complex conjugate of qc
h = conv(qc,gm);                % Impulse response
t0_bar = find(h == max(h));             % Timing phase: peak of h
h = h(h>max(h)/100);
h = h(3:end-2);
h_T = downsample(h,4);
r_gm = xcorr(gm,gm);                    % Matched filter autocorrelation
Pbit_DFE_uncode = zeros(length(SNR_vect),1);
N2 = floor(length(h_T)/2);
N1 = N2;
M1 = 5;
D = 4;
M2 = N2 + M1 - 1 - D;
tic
parfor i=1:length(SNR_vect)
        snr_db = SNR_vect(i);
        snr_lin = 10^(snr_db/10);
        % Single carrier channel simulation
        [r_c, sigma_w, qc] = channel_sim(symbols, snr_db, sigma_a);
        % Additive complex-Gaussian noise
        w = wgn(length(r_c),1, 10*log10(sigma_w), 'complex');
        r_c = r_c + w;
        r_c_prime = filter(gm,1,r_c);
        r_c_prime = r_c_prime(t0_bar:end);
        % Signal at the input of the DFE
        x_aa = downsample(r_c_prime,4);
        rw_tilde = sigma_w/4 .* downsample(r_gm, 4);
        % DFE
        [c_opt, Jmin] = Adaptive_DFE(h_T, rw_tilde, sigma_a, M1, M2, D);
        psi = conv(c_opt, h_T);
        b = - psi(end - M2 + 1:end);
        y_hat = x_aa/max(psi);
        % Equalized signal
        detected = equalization_DFE(y_hat, c_opt, b, M1, M2, D);
        hard_d = zeros(length(detected),1);
        for k=1:length(hard_d)
                hard_d(k) = QPSK_detector(detected(k));
        end
        hard_bits = ibmap(hard_d);
        Pbit_DFE_uncode(i) = length(find(x(1:length(hard_bits))~=hard_bits))/length(hard_bits);
end
toc
```

```
clc; close all; clear global; clearvars;

load('Input_symbols.mat');
symbols = symbols_ak;
SNR_vect = 0.35:0.05:1;
sigma_a = 2;
Pbit_AWGN_code = zeros(length(SNR_vect),1);

tic
parfor i=1:length(SNR_vect)
        snr_db = SNR_vect(i);
        snr_lin = 10^(snr_db/10);
    % noise variance
        sigma_w = sigma_a / snr_lin;
        w = wgn(length(symbols_ak),1, 10*log10(sigma_w), 'complex');
```

```
    % received symbols
        r_c = symbols_ak + w;
        r_c_prime = r_c;
        detected = r_c_prime;
    % LLR
        llr = zeros(2*length(detected),1);
        llr(1:2:end) = -2*real(detected)/(sigma_w/2);
        llr(2:2:end) = -2*imag(detected)/(sigma_w/2);
        llr_deinter = deinterleaver(llr);
    % LDPC decoder
        decoded = LDPC_decoder(llr_deinter).';
        Pbit_AWGN_code(i) = length(find(x(1:length(decoded))~=decoded))/length(decoded);
end
toc
```

```
clc; close all; clear global; clearvars;

load('Input_symbols.mat');
symbols = bitmap(x).';
SNR_vect = 0:14;
sigma_a = 2;
Pbit_AWGN_uncode = zeros(length(SNR_vect),1);

tic
parfor i=1:length(SNR_vect)
        snr_db = SNR_vect(i);
        snr_lin = 10^(snr_db/10);
    % noise variance
        sigma_w = sigma_a / snr_lin;
        w = wgn(length(symbols),1, 10*log10(sigma_w), 'complex');
    % received signal
        r_c = symbols + w;
        r_c_prime = r_c;
        detected = r_c_prime;
    % QPSK detector
        hard_d = zeros(length(detected),1);
        for u=1:length(detected)
                hard_d(u) = QPSK_detector(detected(u));
        end
        hard_bits = ibmap(hard_d);
        Pbit_AWGN_uncode(i) = length(find(x(1:length(hard_bits))~=hard_bits))/length(hard_bits);
end
toc
```

```
clc; close all; clear global; clearvars;

load('Useful.mat', 'qc');
load('Input_symbols.mat');

SNR_vect = 0.8:0.05:1.4;
sigma_a = 2;    % Input variance

Pbit_OFDM_code = zeros(length(SNR_vect),1);
M = 512;                % Sub-channels
Npx = 18;               % Prefix length
tic
parfor i=1:length(SNR_vect)
        snr_db = SNR_vect(i);
        snr_lin = 10^(snr_db/10);
    % simulate transmission over the OFDM channel
        [r_c, sigma_w, g_srrc, g, t0] = channel_OFDM(symbols_ak, snr_db, sigma_a, Npx);
        G = fft(g,512).';
        a_matrix = reshape(r_c(1:end-mod(length(r_c),M+Npx)), M+Npx, []);
        rn = a_matrix(Npx+1:end,:);
    % DFT
        x_k = fft(rn);
        K_i = 1./G;
    % output of the channel
        y_matrix = x_k.*K_i;
        sigma_i = 0.5*sigma_w*M*abs(K_i).^2;
```

```
    % LLR
        llr_real = -2*real(y_matrix).*sigma_i.^(-1);
        llr_imag = -2*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;
        llr = deinterleaver(llr);
    % LDPC decoder
        dec_bits = LDPC_decoder(llr).';
        nerr = length(find(x(1:length(dec_bits))~=dec_bits));
        Pbit_OFDM_code(i) = nerr/length(dec_bits);
end
toc
```

```
clc; close all; clear global; clearvars;

load('Input_symbols.mat');

symbols = bitmap(x).';
SNR_vect = 0:14;
sigma_a = 2;    % Input variance
Pbit_OFDM_uncode = zeros(length(SNR_vect),1);
M = 512;                % Sub-channels
Npx = 18;               % Prefix length
tic
parfor k=1:length(SNR_vect)
        snr_db = SNR_vect(k);
        snr_lin = 10^(snr_db/10);
        [r_c, sigma_w, g_srrc, g, t0] = channel_OFDM(symbols, snr_db, sigma_a, Npx);
        G = fft(g,512).';
        a_matrix = reshape(r_c(1:end-mod(length(r_c),M+Npx)), M+Npx, []);
        rn = a_matrix(Npx+1:end,:);
        x_k = fft(rn);
        K_i = 1./G;
        y_matrix = x_k.*K_i;
        y = reshape(y_matrix,1,[]);
        hard_d = zeros(length(y),1);
        for i=1:length(hard_d)
                hard_d(i) = QPSK_detector(y(i));
        end
        hard_bits = ibmap(hard_d);
        nerr = length(find(x(1:length(hard_bits))~=hard_bits));
        Pbit_OFDM_uncode(k) = nerr/length(hard_bits);
end
toc
```

```
clc; close all; clear global; clearvars;
set(0,'defaultTextInterpreter','latex')    % latex format

%% Uncoded OFDM + DFE + AWGN
SNR_vect_uncoded = 0:14;
load('Pbit_DFE_uncoded.mat');
load('Pbit_OFDM_uncoded.mat');
load('Pbit_AWGN_uncoded.mat');
figure();
semilogy(SNR_vect_uncoded, Pbit_DFE_uncode,'g');
hold on; grid on;
semilogy(SNR_vect_uncoded, Pbit_OFDM_uncode, 'b');
semilogy(SNR_vect_uncoded, Pbit_AWGN_uncode, 'k');
ylim([10^-5 10^-1]); xlim([4 14]);
xlabel('SNR'); ylabel('$P_{bit}$');
legend('Uncoded DFE','Uncoded OFDM','Uncoded AWGN');
set(legend,'Interpreter','latex');

%% Coded OFDM + DFE + AWGN
load('Pbit_DFE_coded.mat');
load('Pbit_OFDM_coded.mat');
load('Pbit_AWGN_coded.mat');
```

```
figure();
semilogy(1.4:0.05:1.8, Pbit_DFE_code,'g');
hold on; grid on;
semilogy(0.8:0.05:1.4, Pbit_OFDM_code, 'b');
semilogy(0.35:0.05:1, Pbit_AWGN_code, 'k');
ylim([10^-5 10^-1]); xlim([0 2]);
xlabel('SNR'); ylabel('$P_{bit}$');
legend('Coded DFE','Coded OFDM', 'Coded AWGN');
set(legend,'Interpreter','latex');
```

```
function [outsym] = QPSK_detector(insym)

if (real(insym)>0)
    if (imag(insym)>0)
        outsym = 1+1i;
    else
        outsym = 1-1i;
    end
else
    if (imag(insym)>0)
        outsym = -1+1i;
    else
        outsym = -1-1i;
    end
end

end
```

```
function [c_opt, Jmin] = Adaptive_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
```

```
function [decisions] = equalization_DFE(x, c, b, M1, M2, D)
%EQUALIZATION for DFE

y = zeros(length(x) + D , 1); % output of ff filter
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.'*xconv;
    detected(k+1) = y(k+1) + b.'*a_old;

end
% scatterplot(y)
decisions = detected(D + 1:end);
end
```

```
function [pn] = PNSeq(L)

r = log2(L+1);
pn = zeros(L,1);

% Initial conditions (set to one, arbitrary)
% Must not be ALL zeros
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 15
            pn(l) = xor(pn(l-14), pn(l-15));
        case 20
```

```
            pn(l) = xor(pn(l-17), pn(l-20));
    end
end
end
```

```
function [coded] = LDPC_encoder(input, h, N)

packets_num = N;
packet_length = 32400;
coded = zeros(length(input)*2,1);
for i=0:packets_num-1
    uncoded = input(i*packet_length+1:i*packet_length+packet_length);
    coded(2 * i * packet_length + 1:2 * i * packet_length + 2 * packet_length) = step(h, uncoded);
end

coded = reshape(coded,length(input)*2,1);

end
```

```
function [decoded] = LDPC_decoder(deinterleaved)

H = comm.LDPCDecoder('DecisionMethod','Hard decision');

numInfoBits = 32400;
decoded_bits = zeros(1,length(deinterleaved)/2);

% Iterate over the input info bits and decode them
for idx = 0:(length(deinterleaved)/(2*numInfoBits))-1
    current_bits = deinterleaved(2*idx*numInfoBits + 1 : 2*idx*numInfoBits + 2*numInfoBits);
    decoded_bits(idx*numInfoBits+1:idx*numInfoBits + numInfoBits) = step(H,current_bits.');
end
decoded = decoded_bits;
end
```

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

output = zeros(1,L);

% 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

output = output(1:2:end);
end
```

```
function [output] = ibmap(input)
    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
```

```
function [output, sigma_w, qc] = channel_sim(x, snr, sigma_a)

T = 1;
Tc = T/4;
Q = T/Tc;

alpha = 0.67;
beta = 0.7424;

snr_db = snr;
snr_lin = 10^(snr_db/10);

qc_num = [0 0 0 0 0 beta];
qc_denom = [1 -alpha];
qc = impz(qc_num, qc_denom);
qc = [0; 0; 0; 0; 0; qc(qc >=max(qc)*10^(-2))];
E_qc = sum(qc.^2);

sigma_w = sigma_a * E_qc / snr_lin;

a_prime = upsample(x,Q);

s_c = filter(qc_num, qc_denom, a_prime);

% to be added later
% noise = wgn(length(s_c),1,sigma_w,'complex');

r_c = s_c;
output = r_c;
```

```
function[output, sigma_w, g_srrc, tot_ds, t0, tot] = channel_OFDM(input, snr, sigma_a, Npx)

snr_db = snr;
snr_lin = 10^(snr_db/10);
% number of subchannels
M = 512;
% add padding bits to avoid length errors
a_pad = [input; ones(M - mod(length(input), M), 1) * (1+1i)];
a_matrix = reshape(a_pad, M, []); % Should mantain columnwise order
% IDFT
A_matrix = ifft(a_matrix);
A_matrix = [A_matrix(M-Npx+1:M, :); A_matrix];

% Start transmission in the channel
r = reshape(A_matrix, [], 1);
in_upsampled = upsample(r, 4);

% Square-root raised cosine
ro = 0.0625;
span = 30;
sps = 4;
g_srrc = rcosdesign(ro, span, sps, 'sqrt');
% signal at the output of the first filter
in_after_srrc = filter(g_srrc,1,in_upsampled);
% qc from hw3
alpha = 0.67;
beta = 0.7424;
qc_num = [0 0 0 0 0 beta];
qc_denom = [1 -alpha];
qc = impz(qc_num, qc_denom);
```

```
qc = [0; 0; 0; 0; 0; qc(qc >=max(qc)*10^(-2))];
% signal at the output of the second filter
in_after_qc = filter(qc,1,in_after_srrc);
all_ch = conv(g_srrc, conv(g_srrc,qc));
E_tot = sum(conv(g_srrc,qc).^2);
% add noise
sigma_w = sigma_a/M * E_tot / snr_lin;
in_after_qc = in_after_qc + wgn(length(in_after_qc),1,10*log10(sigma_w),'complex');
% overall impulse response at Tc
tot = all_ch(abs(all_ch)>=(max(abs(all_ch))*1e-2));
% downsampler
tot_ds = downsample(tot, 4);
% signal at the output of the last filter
in_after_srrc = filter(g_srrc, 1, in_after_qc);
% timing phase
t0 = find(in_after_srrc==max(in_after_srrc));
% h at kT_ofdm starting from t0
in_after_srrc = in_after_srrc(t0:end);
in_after_srrc = downsample(in_after_srrc,4);

output = in_after_srrc;
end
```

```
clc; close all; clear global; clearvars;

% Initial bits
L = 2^20-1;
x = [PNSeq(L); PNSeq(L)];
% Matlab LDPC Encoder
H = comm.LDPCEncoder();          % parity check matrix

% encode block of 32400 bits
sstep = 32400;
% avoid final block to have less bit then required
numbits = floor(length(x) / sstep) * sstep;
x = x(1:numbits + 54);
% number of encoded packets
N = floor(length(x) / sstep);
% LDPC encoder
encoded = LDPC_encoder(x,H,N);
% interleaver
interleaved = interleaver(encoded);
% QPSK modulation with gray coding
symbols_ak = bitmap(interleaved.').';

save('Input_symbols.mat', 'symbols_ak', 'x', 'H', 'sstep');
```

```
function [interleaved_bits] = interleaver(input)

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

rows = 42;
columns = 42;

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

```
function [deinterleaved_bits] = deinterleaver(bits)
% This function receives a sequence of bits and unscrambles it
deinterleaved_bits = zeros(1,length(bits));
```

```
% The deinterleaver is just an interleaver with rows and cols switched
rows = 42;
columns = 42;

% 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
```