Digital Transmission - Homework 2

Andrea Dittadi, Davide Magrin, Michele Polese

April 29, 2015

MATLAB code

channel_generator

```
%% Data and variables initialization

data_init;

% PDP (aleatory part)
N_h = 3;
tau = 0:Tc:N_h-1;
M_iTc = 1/tau_rms * exp(-tau/tau_rms);
C = sqrt(K/(K+1));
% normalize pdp: it must be sum(E[|htilde_i|^2]) = 1 - C^2
M_iTc = M_iTc.*(1-C^2)/sum(M_iTc);

%% Impulse responses generation
% Some will be dropped because of transient, since enough time, memory and
% computational power are available.
h_samples_needed = 8000000 + ceil(Tp/Tc*length(h_dopp));
w_samples_needed = ceil(h_samples_needed / Tp);
% The filter is IIR, from Anastasopoulos and Chugg (1997) it appears that
% the effect of the transient is present in about 2000 samples for an
% interpolation of factor Q = 100. This model uses Q = 80. Since memory and
% computational power are not an issue, in order to be conservative it
% drops 80*length(h_dopp) samples.
transient = ceil(Tp/Tc*length(h_dopp));

h_mat = zeros(N_h, h_samples_needed - transient);
for ray = 1:N_h
    % Complex-valued Gaussian white noise with zero mean and unit variance
    w = wgn(w_samples_needed,1,0,'complex');

    hprime = filter(b_dopp, a_dopp, w);

    % Interpolation
    t = 1:length(hprime);
    t_fine = Tq/Tp:Tq/Tp:length(hprime);
    h_fine = interp1(t, hprime, t_fine, 'spline');

    % Drop the transient
    h_mat(ray, :) = h_fine(transient+1:end);
end

% Energy scaling
for k = 1:N_h
    h_mat(k, :) = h_mat(k, :)*sqrt(M_iTc(k));
end

% Only for LOS component, add deterministic component
h_mat(1, :) = h_mat(1, :) + C;

clear tau tau_rms h_fine hprime pdp_gauss t_dopp t_fine ...
    h_samples_needed w_samples_needed M_d k ray
```

channel_output

```
function [ d, h_mean ] = channel_output( x, T, Tc, sigma_w, N_h, h_mat )
% CHANNEL_OUTPUT Generates channel output (that is the desired signal) via a
% polyphase implementation (for Nh<=4). Returns the channel output d given
% the input parameters, and a vector with the average coefficients of the
% actual impulse response during the considered window.

d = zeros(T * length(x), 1);
h_used_coeff = zeros(4, length(x));
for k = 0 : length(x)-1
    % Generate white noise
    w = wgn(4, 1, 10*log10(sigma_w), 'complex');
    for i = 0:3 % Branch index
        if (i < N_h)
            d(k*T + i*Tc + 1) = h_mat(i+1,k*T + i*Tc+1) * x(k+1) + w(i+1);
            % store the coefficient actually used, it will be useful later on
            h_used_coeff(i + 1, k + 1) = h_mat(i+1,k*T + i*Tc+1);
        else
            d(k*T + i*Tc + 1) = w(i+1); % No ray, just the noise
            h_used_coeff(i + 1, k + 1) = 0;
        end
    end
end

% No need to drop the coefficients of the transient, since the transient is zero when Nh <= 4.

% Compute the mean coefficient of impulse response of each ray over the interval of
% interest of L samples in order to compare them with the estimated impulse response.
h_mean = mean(h_used_coeff, 2);

end
```

channelModel

```
%% Clean up and initialize useful quantities
clear
close all
clc
rng default

%% Data initialization

% The data_init script initializes all the input parameters that don't
% change across the simulation
data_init;

% Plot residual energy of IIR filter in order to determine transient length
[h_dopp, ~] = impz(b_dopp, a_dopp);
residual_nrg = sum(h_dopp.^2) - cumsum(h_dopp.^2);
plot(0:length(residual_nrg)-1, 10*log10(residual_nrg)),
xlim([0 100]), grid on, box on, title('Residual energy of I.R. of the IIR filter')
xlabel('Transient length (samples)'), ylabel('Residual energy (dB)')
% Doppler filter shape
[Hf, f] = freqz(b_dopp, a_dopp, 1000, 'whole');
figure, plot(f/(2*pi), 20*log10(abs(Hf)))
ylim([0, 12])
title('Frequency response of the Doppler filter')

%% Criterion for N_h
% We consider the error that is introduced by dropping the tail of the of
% the sampled exp function that describes the aleatory part of the PDP. The
% entire function is normalized to sum to 1-C^2.

M_complete = 1/tau_rms*exp(-(0:894)*Tc/tau_rms);
M_complete = M_complete.*(1-C^2)/sum(M_complete);

for N_h = 1:10
    % Only consider the tail
```

```
    deltaM = M_complete(N_h+1:end);
    lambda_n(N_h) = 1/(snr_lin * sum(abs(deltaM)));
end

figure
plot(1:length(lambda_n), 10*log10(lambda_n), 'd-')
grid on, title('\Lambda_n')
xlabel('N_h'), ylabel('\Lambda_n [dB]')
axis([1 4 -2 10]), ax = gca; ax.XTick = 1:5;

%% Display the PDP for the channel

% The PDP is the sampling of a continuous time exponential PDP with
% tau_rms / T = 0.3, so tau_rms / Tc = 1.2
% (See 4.224 for reference)
N_h = 3; % As determined before
tau = 0:Tc:N_h-1;
M_iTc = 1/tau_rms * exp(-tau/tau_rms); %
% normalize pdp: it must be sum(E[|htilde_i|^2]) = 1 - C^2
M_iTc = M_iTc.*(1-C^2)/sum(M_iTc);
M_d = sum(M_iTc);

pdp = M_iTc;

% add LOS component power
pdp(1) = pdp(1) + C^2;
pdp_log = 10*log10(pdp);

% Plot the normalised PDP
figure, stem(tau, pdp_log), title('PDP'), xlabel('iTc'), ylabel('E[|h_i(nTc)|^2]');
grid on;
axis([-0.25 2.25 -15 0]), ax = gca; ax.XTick = 0:2;
legend('PDP', 'Location', 'SouthWest')

%% Generation of impulse responses
% The code to generate the impulse responses is externalized in
% channel_generator script, which will be invoked both in the first and
% second exercise
channel_generator;

%% Show the behavior of |h_1(nTc)| for n = 0:1999, dropping the transient

figure, hold on
plot(0:1999, abs(h_mat(2, 1:2000).'))
grid on, box on, xlabel('nT_c'), ylabel('|h_1(nT_c)|')
title('|h_1(nT_C)|')

%% Show all |h_i|'s
% Just for debugging purposes

figure, hold on
plot(0:9999, abs(h_mat(:, 1:10000).'))
grid on, box on, xlabel('Time samples'), ylabel('|h_i(nT_C)|_{dB}')
legend('h_0', 'h_1', 'h_2')
title('|h_i|')

%% Histogram of h_1

% Plot of the required histogram
figure, histogram(abs(h_mat(2, 1:1000)).'/sqrt(M_iTc(2)), 20, ...
    'Normalization','pdf', 'DisplayStyle', 'stairs');
title('Experimental PDF from 1000 samples of hbar_1')
xlabel('hbar_1');

% This plot can be used to explain that because of the correlation we can't
% get a nice pdf (too little samples, correlation in peaks)
figure,
subplot 121
histogram(abs(h_mat(2, 1:1000).')/sqrt(M_iTc(2)), 20, ...
    'Normalization','pdf', 'DisplayStyle', 'stairs');
camroll(90)
```

```
title('1000 samples of hbar_1')
subplot 122
plot(0:999, abs(h_mat(2, 1:1000).')/sqrt(M_iTc(2)));
title('Realization of hbar_1 over which the histogram is computed');
xlabel('Samples');
ylabel('hbar_1');
grid on

% Here we show that the experimental PDF gets better with more samples
figure
histogram(abs(h_mat(2, 1: 100000).')/sqrt(M_iTc(2)), 20, ...
    'Normalization','pdf', 'DisplayStyle', 'stairs')
title('100000 samples of hbar_1 vs Rayleigh pdf')
hold on
a = 0:0.01:3;
plot(a, 2.*a.*exp(-a.^2), 'LineWidth', 1.5);  % Theoretical PDF (page 308, BC)
hold off
legend('hbar_1', 'Rayleigh pdf');
ylabel('p_{hbar_1(kT_C)}(a)')
xlabel('a');

%% Simulation in order to compute the histogram of |h1(151Tc)|/sqrt(E(|h1(151Tc)|^2))
% This simulation repeats for numexp times, indipendently, the generation
% of the impulse response for ray 1. The task is the same as in the more
% general channel_generator. However, in order to speed up the simulation
% and since only one ray is involved, we don't invoke channel_generator
% script as before but generate the required impulse response only.
% Moreover, since we are interested in the 151th sample after the transient
% we can generate shorter impulse responses at each iteration.
numsim = 1000;
h_samples_needed = 200000 + ceil(Tp/Tc*length(h_dopp));
% Some will be dropped because of transient, since
% enough time, memory and computational power are available
w_samples_needed = ceil(h_samples_needed / Tp);
transient = ceil(Tp/Tc*length(h_dopp));

h_1 = zeros(numsim, 1);
for k = 1:numsim
    disp(k)
    w = wgn(w_samples_needed,1,0,'complex');
    hprime = filter(b_dopp, a_dopp, w);
    % Interpolation
    t = 1:length(hprime);
    t_fine = Tq/Tp:Tq/Tp:length(hprime);
    h_fine = interp1(t, hprime, t_fine, 'spline');
    % Drop the transient and energy scaling
    h_notrans = h_fine(50000:end)*sqrt(M_iTc(2));
    % Energy scaling
    h_1(k) = h_notrans(152);
end

figure,
histogram(abs(h_1)/sqrt(sum(abs(h_1).^2)/length(h_1)), 20, ...
    'Normalization','pdf', 'DisplayStyle', 'stairs')
title('hbar_1(151T_C) over 1000 realizations vs Rayleigh pdf')
hold on
a = 0:0.01:3;
plot(a, 2.*a.*exp(-a.^2), 'LineWidth', 1.5);  % Theoretical PDF (page 308, BC)
hold off
legend('hbar_1', 'Rayleigh pdf');
xlabel('a');
ylabel('p_{hbar_1(151T_C)}(a)');
```

data_init

```
% This script initializes all system specifications, that will always
% remain constant throughout the execution of all the scripts.

% Sampling times
Tc = 1;     % This is the smallest time interval we want to simulate
T = 4*Tc;   % Time sampling interval of the input of the channel
Tq = Tc;    % Fundamental sampling time. This is the same as Tc
fd = 5*10^-3/T; % Doppler spread
Tp = 1/10 * (1/fd) * Tq; % Sampling time used for filtering the white noise,
% in order to apply Anastasopoulos and Chugg (1997) filter it must be
% Tp = 0.1

Kdb = 3; % 3 dB, given
K = 10^(Kdb/10); % Linear K
C = sqrt(K/(K+1)); % This holds if PDP sums to 1

tau_rms = 0.3*T;

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

% Filter for doppler spectrum. The filter will have the
% classical Doppler spectrum in the frequency domain, with f_d * T = 5*10^-3
% By using the approach suggested in Anastasopoulos and Chugg (1997) we use an iir filter
% with known coefficients which is the convolution of a Cheby lowpass and a shaping filter.
a_dopp = [1, -4.4153, 8.6283, -9.4592, 6.1051, -1.3542, -3.3622, 7.2390, ...
    -7.9361, 5.1221, -1.8401, 2.8706e-1];
b_dopp = [1.3651e-4, 8.1905e-4, 2.0476e-3, 2.7302e-3, 2.0476e-3, 9.0939e-4, ...
    6.7852e-4, 1.3550e-3, 1.8076e-3, 1.3550e-3, 5.3726e-4, 6.1818e-5, -7.1294e-5, ...
    -9.5058e-5, -7.1294e-5, -2.5505e-5, 1.3321e-5, 4.5186e-5, 6.0248e-5, 4.5186e-5, ...
    1.8074e-5, 3.0124e-6];
% The energy needs to be normalized to 1
[h_dopp, ~] = impz(b_dopp, a_dopp);
hds_nrg = sum(h_dopp.^2);
b_dopp = b_dopp / sqrt(hds_nrg);
```

h_estimation

```
function [ h_hat, d_hat ] = h_estimation( x, d, L, N_i )
% 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 max(N_i) > L
    h_hat = [];
    d_hat = [];
    return
end

%% Estimate h

% Create four different d_i vectors, by sampling with step 4 the complete
% vector d. Each of them is the output of the branch that has "lag" iTc.
d_poly = zeros(length(d)/4, 4); % each column is a d_i
for idx  = 1:4
    d_poly(:, idx) = d(idx:4:end);
end

% Using the data matrix (page 246), easier implementation
h_hat = zeros(4,max(N_i));
% We're estimating 4 branches of the polyphase representation.
% This matrix has the maximum number of coefficients for each of the four
% branches. The unused (i.e. unestimated) ones will be left zero.
for idx = 1:4
    if N_i(idx) > 0
```

```
        I = zeros(L,N_i(idx));
        for column = 1:N_i(idx)
            I(:,column) = x(N_i(idx)-column+1:(N_i(idx)+L-column));
        end
        o = d_poly(N_i(idx):N_i(idx) + L - 1, idx);

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

        h_hat(idx, 1:N_i(idx)) = Phi \ theta;
    end % if N_branch is 0 don't estimate and leave hhat to 0
end

%% Compute d_hat

x = [0; x];
x_toep = toeplitz(x);
x_toep = x_toep(1:max(N_i), end-L:end);
% d_hat with the final part of the transient or with some useless zeros.
d_hat = h_hat * x_toep;
% Get d_hat in a line and then discard samples.
d_hat = reshape(d_hat, numel(d_hat), 1);
d_hat_discard_num = max(0, sum(N_i)-4)-4*ceil(sum(N_i)/4)+8;
d_hat = d_hat(d_hat_discard_num + 1 : end);
d_no_trans = d(end-length(d_hat)+1 : end);

end
```

impulseResponse

```
% Impulse response estimation
% We are at the receiver, we know what the sender is sending and we try to
% estimate it with the LS method (for reference, see page 244).
% Note: As the receiver, we do _not_ know neither N_h nor sigma_w

clear, clc, close all
rng default

%% Generate time-variant i.r. of the channel and initialize everything
channel_generator;
sigma_w = 1/(T/Tc*snr); % the PN sequence has power 1

%% Loop to determine suitable values of N, L

printmsg_delete = ''; % Just to display progress updates
maxN = 10;
% Note that with maxN<13 we don't have problems with the condition N<=L.

% Time counter that allows the output d to be computed with a different
% impulse response at every iteration, as it would happen in reality.
time = 1;
L_vec = [3, 7, 15, 31, 63, 127];
numsim = 100; % It seems to converge even with small values of numsim
error_func = zeros(length(L_vec), maxN, numsim);
for L_index = 1:length(L_vec)
    L = L_vec(L_index);

    % --- Generate training sequence
    % The x sequence must be a partially repeated M-L sequence of length L. We
    % need it to have size L+N-1. To observe L samples, we need to send L+N-1
    % samples of the training sequence {x(0), ..., x((N-1)+(L-1))}.
    p = MLsequence(L);
    x = [p; p(1:ceil(maxN/4)-1)]; % create a seq which is long enough for the maximum N
    x(x == 0) = -1;

    % --- Estimation of h and d multiple times
    for k =1:numsim
```

```
        % Print progress update
        printmsg = sprintf('L = %d, simulation number %d\n', L, k);
        fprintf([printmsg_delete, printmsg]);
        printmsg_delete = repmat(sprintf('\b'), 1, length(printmsg));

        % Transmit only one time and estimate h for different N
        [d, ~] = channel_output(x, T, Tc, sigma_w, N_h, h_mat(:, time:end));
        time = time + 50*(L+maxN)*T/Tc; % the time windows are sufficiently spaced apart
        for N = 1:maxN % N is the supposed length of the impulse response of the channel
            % Compute the supposed length of each branch
            n_short = mod(4-N, 4); % Num branches with a shorter filter than others
            % N_i is the number of coefficients of the filter of the i-th branch.
            N_i(1:4-n_short) = ceil(N/4);
            N_i(4-n_short + 1 : 4) = ceil(N/4) - 1;
            [h_hat, d_hat] = h_estimation( x(end-(L+max(N_i)-1) + 1 : end), ...
                        d(end - 4*(L+max(N_i)-1) + 1: end), L, N_i);
            d_no_trans = d(end-length(d_hat)+1 : end);
            error_func(L_index, N, k) = sum(abs(d_hat - d_no_trans).^2)/length(d_hat);
        end
    end
end
error_func = mean(error_func, 3);

% Plot the empirical error functional for different pairs (L, N)
figure, hold on
for i = 1:length(L_vec)
    plot(10*log10(error_func(i, :)), 'DisplayName', strcat('L=', num2str(L_vec(i))))
    legend('-DynamicLegend')
end
xlabel('N'), ylabel('\epsilon [dB]'), title('Error function')
grid on, box on, ylim([-20, -10])

%% Estimate E(|h-hhat|^2) by repeating the estimate 1000 times and assuming
% h known

printmsg_delete = '';

% time counter that let the desired output d to be computed with a
% different impulse response at every iteration, as it would happen in
% reality.
time = 1;
L_vec = [3, 7, 15, 31];
numsim = 1000;
deltah_square = zeros(length(L_vec), numsim, maxN);
for L_index = 1:length(L_vec)
    L = L_vec(L_index);

    % --- Generate training sequence
    % The x sequence must be a partially repeated M-L sequence of length L. We
    % need it to have size L+N-1. To observe L samples, we need to send L+N-1
    % samples of the training sequence {x(0), ..., x((N-1)+(L-1))}.
    p = MLsequence(L);
    x = [p; p(1:ceil(maxN/4)-1)]; % create a seq which is long enough for the maximum N
    x(x == 0) = -1;

    % --- Estimation of h multiple times
    for k=1:numsim
        printmsg = sprintf('L = %d, simulation number = %d\n', L, k);
        fprintf([printmsg_delete, printmsg]);
        printmsg_delete = repmat(sprintf('\b'), 1, length(printmsg));

        % We transmit only one time and then estimate h for different N
        [d, h_mean] = channel_output(x, T, Tc, sigma_w, N_h, h_mat(:, time:end));
        time = time + 50*(L+maxN)*T/Tc; % the time windows are sufficiently spaced apart

        for N = 1:maxN % N is the supposed length of the impulse response of the channel
            n_short = mod(4-N, 4); % Num branches with a shorter filter than others
            % N_i is the number of coefficients of the filter of the i-th branch.
            N_i(1:4-n_short) = ceil(N/4);
            N_i(4-n_short + 1 : 4) = ceil(N/4) - 1;
```

```
            % LS estimation of h
            [h_hat, ~] = h_estimation(x, d, L, N_i);

            % Compute delta_h squared
            h_hat_array = reshape(h_hat, 4*max(N_i), 1);
            h_hat_array = h_hat_array(1:N);
            h_mean_array = h_mean(1:N_h);
            % The vector to which we compare the estimate has length N_h,
            % the estimated h_hat has length N. Now we make them the same length.
            if N < N_h
                h_hat_array = [h_hat_array; zeros(N_h - N, 1)];
            elseif N > N_h
                h_mean_array = [h_mean_array; zeros(N - N_h, 1)];
            end % if N=N_h already ok
            deltah_square(L_index, N, k) = sum(abs(h_hat_array - h_mean_array).^2);
        end
    end
end
deltah_square = mean(deltah_square, 3);

%% Compare with theoretical

deltah_square_theor = zeros(length(L_vec), maxN);
for L_index = 1:length(L_vec)
    L = L_vec(L_index);
    for N = 1:maxN
        if(ceil(N/4) > L)   % The estimate cannot be performed (see report)
            deltah_square_theor(L_index, N) = NaN;
            break
        end

        n_short = mod(4-N, 4); % Num branches with a shorter filter than others
        % N_i is the number of coefficients of the filter of the i-th branch.
        N_i(1:4-n_short) = ceil(N/4);
        N_i(4-n_short + 1 : 4) = ceil(N/4) - 1;

        deltah_square_theor(L_index, N) = ...
                sigma_w / (L+1) * sum(N_i .* (L+2-N_i) ./ (L+1-N_i));
    end
end

% Plot results
figure, hold on
for L_index = 1:4
    plot(10*log10(deltah_square(L_index, :)), ...
            'DisplayName', sprintf('L=%d experimental', L_vec(L_index)))
    plot(10*log10(deltah_square_theor(L_index, :)),'-.', ...
            'DisplayName', sprintf('L=%d theoretical', L_vec(L_index)))
    legend('-DynamicLegend')
end
xlabel('N that tracks the real N_h'), ylabel('Estimate of E(|h - hhat|^2) [dB]')
title('Estimate of E(|h - hhat|^2) across 1000 realizations')
ax = gca; ax.XTick = 1:maxN;
ylim([-30 -5]), grid on, box on
```

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. The
% script can handle L = 3, 7, 15, 31, 63 and 127.

% Extract r from the given L
r = log2(L+1);
p = zeros(L,1);
p(1:r) = ones(1,r).'; % 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));
    end
end
end
```