Digital Transmission - Homework 1

Andrea Dittadi, Davide Magrin, Michele Polese

April 9, 2015

MATLAB code

```
%% First spectral analysis
close all
clear
clc

%% Load data
z = load('data for hw1.mat');
z = z.z.'; % make a column vector
z = z - mean(z); % remove average
K = length(z); % signal length

%% Spectral analysis

% AR model: order estimation.
% Compute variance of AR model and plot it to identify the knee.
% It's computed up to K/5 - 1
N_corr = ceil(K/5);
autoc = autocorrelation_biased(z, N_corr);
upp_limit = 30;
sigma_w = zeros(1, upp_limit);
for N = 1:upp_limit
    [~, sigma_w(N)] = arModel(N, autoc);
end
figure, plot(1:upp_limit, 10*log10(sigma_w))
title('sigma_w of the AR model of the whole signal'), grid on
ylabel('sigma_w (dB)'), xlabel('Order of the AR(N) model')

% Plot the spectra of the signal according to various estimators
plot_spectrum(z, 3);
ylim([-10 40])

%% Peaks accumulation
% Find if a peak is present in more than one window of the signal

step = 64; % distance between the first two sample of each window
span = 96; % actual size of the window
overlap = span - step;
max_iter = floor((K-span)/step);

% Initialize
acc_locs_per = zeros(span, 1);
window_span = kaiser(span, 5.65);

for i = 0:max_iter
    z_part = z(i*step + 1: i*step + span);

    % PERIODOGRAM over span samples
    Z = fft(z_part.*window_span);
    periodogr = abs(Z).^2/span;

    % find local maxima
    [peaks, locs_per] = findpeaks(abs(periodogr));
    % accumulate local maxima
    acc_locs_per(locs_per) = acc_locs_per(locs_per) + 1;
```

```
end

% normalize to the number of iterations
acc_locs_per = acc_locs_per/i;

figure
bar((1:span)/span, acc_locs_per)
axis([0, 1, 0, max(acc_locs_per)])
xlabel('Frequency')
title(sprintf('Histogram of periodogram peaks over %d windows', max_iter))

disp(find(acc_locs_per > 0.7)/span);

%% Whitening filter

% Pass the signal through a whitening filter in order to equalize its
% spectrum. The whitening filter will be obtained as the inverse of the AR
% model filter. This is needed to identify peaks. As we do not care about
% the phase in this stage, we can afford to pass it through something that
% is not with linear phase.

% Compute the AR model
[a, sigma_w] = arModel(3, autocorrelation(z, K/5));
[H, omega] = freqz(1, [1; a], K, 'whole');

% Plot the two frequency responses
figure
plot(omega/(2*pi), 10*log10(sigma_w*abs(H).^2));
hold on
plot(omega/(2*pi), 10*log10((1/sigma_w)*abs(1./H).^2));
hold off
legend('AR filter', 'Inverse of AR filter');
title('Frequency response of the AR filter and of its inverse');
axis([0 1 -40 40]);

% Plot the whitened result
figure
equalized_spectrum = abs((fft(z)).*(1./H)*(1/sigma_w));
plot((1:K-1)/K, 10*log10(equalized_spectrum(2:end))); hold on
title('Equalized spectrum of the given signal');
axis([0 1 0 20]);


%% Percentiles

D = 200;
S = 175;
window = kaiser(D, 5.65);
[Pm, PM, Psorted, ~] = findSine(z, window, S);

figure
plot((0:length(Pm)-1)/length(Pm), 10*log10(Pm).'), hold on
plot((0:length(PM)-1)/length(PM), 10*log10(PM).')
title('Minimum and maximum PSD across all windows')
ylabel('PSD (dB)'), ylim([-25 45])
% figure
% plot((0:length(PM)-1)/length(PM), 10*(log10(PM) - log10(Pm)).')
% title('Ratio between min and max PSD across all windows')

% Plot the percentiles
figure
hold on
percentileindices = round([1 30 70 99] * size(Psorted, 2) / 100);
percentileindices = max(percentileindices, ones(size(percentileindices)));
for i = percentileindices
    plot((0:length(Psorted)-1)/length(Psorted), 10*log10(Psorted(:, i)))
end
title('Percentiles of PSD (dB)')
legend('1', '30', '70', '99', 'Location', 'SouthEast')
ylim([-20 40])
```

```
%percentiles zoom
figure
hold on
percentileindices = round([1 10 20 30 70 80 90 99] * size(Psorted, 2) / 100);
percentileindices = max(percentileindices, ones(size(percentileindices)));
for i = percentileindices
    plot((0:length(Psorted)-1)/length(Psorted), 10*log10(Psorted(:, i)))
end
title('Percentiles of PSD (dB)')
legend('1', '10', '20', '30', '70', '80', '90', '99', 'Location', 'SouthEast')
ylim([-20 20]), xlim([.6 .9])
```

```
%% Filter FIR
close all
clear
clc

%% Load data
load('data for hw1.mat');
z = z.'; % make a column vector
load('hp18.mat'); % load an high pass filter
hp18 = hp18.'; % make a column vector
K = length(z); % signal length

%% Complex BPF
% --- Compute the coefficients
f0 = 0.770; % estimated by inspection on the PSD + ampphase_estimation_rls
% Note: cfirpm has a strange behaviour, the center of the band is -(1-f0)*2
% limit of don't care regions, left and right of f0
freq_delimiters = [f0 - 0.02, f0 - 0.002, f0 + 0.002, f0 + 0.02];
matlab_correct_setting = -2*(1-freq_delimiters);
% bandpass filter designed with cfirmpm
bpf = cfirpm(58, [-1, matlab_correct_setting, 1], @bandpass);

% --- Plot frequency response of bandpass filter
% DTFTplot(bpf, 50000);
% ylim([-40 0])
% title('Freq resp of the BPF filter')

%% Filter the signal with HPF + BPF
linesfilter = conv(hp18, bpf);
% normalize linesfilter
linesfilter = linesfilter / max(abs(fft([linesfilter zeros(1, 5000 - length(linesfilter))])));
N_linesfilter = length(linesfilter) - 1; % Order of the filter
z_lines = filter(linesfilter, 1, z);
DTFTplot(linesfilter, 10000); % Plot filter's freq resp
title('Freq response of HPF + BPF (dB)')
ylim([-50 0]), grid on
% Compensate delay (still half transient left)
z_lines = z_lines( (N_linesfilter/2 + 1) : length(z_lines));


%% Compute complementary filter and get "continuous PSD" part

% Compute the complementary of the filter we just used
linesfilter_compl = -linesfilter;
linesfilter_compl(N_linesfilter/2 + 1) = linesfilter_compl(N_linesfilter/2 + 1) + 1;
DTFTplot(linesfilter_compl, 10000);
title('Freq response of complementary filter (dB)')
ylim([-25 5]), grid on

% Filter original signal
z_continuous = filter(linesfilter_compl, 1, z);
% Compensate delay (still half transient left)
z_continuous = z_continuous( (N_linesfilter/2 + 1) : length(z_continuous));


%% Export the two signals

save('split_signal', 'z_continuous', 'z_lines');
```

```
%% See that diff is zero

delayonly = [zeros(N_linesfilter/2, 1); 1];
delayed_z = filter(delayonly, 1, z);
% Compensate delay
delayed_z = delayed_z( (N_linesfilter/2 + 1) : length(delayed_z));
diff = delayed_z - (z_continuous + z_lines);
disp(['Max magnitude of the difference between the original signal and the ', ...
    'sum of its two components (lines and continuous) is ', num2str(max(abs(diff)))])

%% Remove mean and plot spectral analysis (without AR models)

% Remove mean from continuous and lines part
z_continuous = z_continuous - mean(z_continuous);
z_lines = z_lines - mean(z_lines);

% Plot spectral analysis of the signal and its two components (without AR models)
plot_spectrum(z_lines, 0);
axis([0 1 -10 40]), title('Spectral analysis of spectral line part')
plot_spectrum(z_continuous, 0);
axis([0 1 -10 40]), title('Spectral analysis of continuous part')
plot_spectrum(z, 3);
axis([0 1 -10 40]), title('Spectral analysis of original signal')


%% AR model for continuous part

% Find the knee of sigma_w
N_corr = floor(length(z_continuous)/5);
autoc_cont = autocorrelation_biased(z_continuous, N_corr);
upp_limit = 30;
sigma_w = zeros(1, upp_limit);
for N = 1:upp_limit
   [~, sigma_w(N)] = arModel(N, autoc_cont);
end
figure, plot(1:upp_limit, 10*log10(sigma_w))
title('sigma_w of the AR model of the continuous part'), grid on
ylabel('sigma_w (dB)'), xlabel('Order of the AR(N) model')

% Choose order for AR and compute the vector of coefficients a
N = 3;
[a_cont, sigma_w_cont] = arModel(N, autoc_cont);
[H, omega] = freqz(1, [1; a_cont], K, 'whole');
figure, plot(omega/(2*pi), 10*log10(sigma_w_cont*abs(H).^2), 'Color', 'm', 'LineWidth', 1)
axis([0, 1, -10, 40])
title('AR model of the continuous part')
xlabel('Normalized frequency'), ylabel('Magnitude (dB)')

figure, zplane(roots([1;a_cont]))


%% AR model for spectral lines

% Find the knee of sigma_w
N_corr = floor(length(z_lines)/5);
autoc = autocorrelation_biased(z_lines, N_corr);
upp_limit = 30;
sigma_w = zeros(1, upp_limit);
for N = 1:upp_limit
   [~, sigma_w(N)] = arModel(N, autoc);
end
figure, plot(1:upp_limit, 10*log10(sigma_w))
title('sigma_w of the AR model of the spectral lines'), grid on
ylabel('sigma_w (dB)'), xlabel('Order of the AR(N) model')

% Choose order for AR and compute the vector of coefficients a
N = 5;
[a_lines, sigma_w_lines] = arModel(N, autoc);
[H, omega] = freqz(1, [1; a_lines], K, 'whole');
figure, plot(omega/(2*pi), 10*log10(sigma_w_lines*abs(H).^2), 'Color', 'm', 'LineWidth', 1);
axis([0, 1, -35, 15])
```

```
title('AR model of the spectral lines')
xlabel('Normalized frequency'), ylabel('Magnitude (dB)')

figure, zplane(roots([1;a_lines]))
```

```
%% LMS

close all;
clear all;
clc;

%% Load "continuous PSD" signal
load('split_signal.mat', 'z_continuous');
z = z_continuous - mean(z_continuous);
K = length(z); % signal length
autoc_z = autocorrelation(z, round(K/5));

%% AR
% the knee is apparently at N = 3, however LMS doesn't converge for N = 3
% in the required number of iterations
% compute the vector of coefficients a
N = 2;
[a, sigma_w] = arModel(N, autoc_z);
[H, omega] = freqz(1, [1; a], K, 'whole');

%% Initialization and iteration
upper_limit = 399; %MATLAB requires indices from 1 to 401
c = zeros(N, upper_limit + 1); % init c vector, no info -> set to 0
% each column of this matrix is c(k), a vector with coefficients from 1 to
% N (since we are implementing the predictor)!
e = zeros(1, upper_limit);

mu = 0.42/(autoc_z(1)*N); % actually mu must be > 0 and < 2/(N r_z(0))
% watch out, in the predictor y(k) = transp(x(k-1))c(k)
for k = 1:upper_limit
    if (k < N + 1)
        z_k_1 = flipud([zeros(N - k + 1, 1); z(1:k - 1)]); % input vector z_vec_(k-1) of length N
        % for k = 1 z(1:0) is an empty matrix
        y_k = z_k_1.'*c(:, k);
    else
        z_k_1 = flipud(z((k - N):(k-1))); % we need the input from k - 1 to k - N
        y_k = z_k_1.'*c(:, k);
    end
    e_k = z(k) - y_k; % the reference signal d(k) is actually the input at sample k
    e(k) = e_k;
    c(:, k + 1) = c(:, k) + mu*e_k*conj(z_k_1); % update the filter, c(k+1) = c(k) + mu*e(k)*conj(z(k
        -1))
end

% Plot c coefficients
for index = 1:N
    figure
    subplot(2, 1, 1)
    plot(1:upper_limit+1, real(c(index, :)), [1, upper_limit+1], -real(a(index))* [1 1])
    title(['Real part of c' int2str(index)]);
    subplot(2, 1, 2)
    plot(1:upper_limit+1, imag(c(index, :)), [1, upper_limit+1], -imag(a(index))* [1 1])
    title(['Imaginary part of c' int2str(index)]);
end

figure, plot(1:upper_limit, 10*log10(abs(e).^2))
hold on
plot(1:upper_limit, 10*log10(sigma_w)*ones(1, upper_limit))
title('Error function at each iteration');

% Find the value of coefficients at instant k = 350 and the average of e
% over k \in [350 - 10, 350 + 10]
ind = 350;
win_side_len = 10;
win_len = 2*win_side_len + 1;
c_350 = c(:, ind);
```

```
e_350_av = 10*log10(sum(abs(e(ind-win_side_len:ind+win_side_len)).^2)/win_len);
```

```
%% RLS

% We are trying to estimate the vector of coefficients c by an LS method.
% As a ballpark figure, this should converge ~10 times faster than the LMS
% algorithm. This comes at the cost of increased computational complexity.

% For reference, see pages 197, 201-203 of the Benvenuto-Cherubini book.

% Clear stuff
close all;
clear all;
clc;

%% Load "continuous PSD" signal
load('split_signal.mat', 'z_continuous');
z = z_continuous - mean(z_continuous);
K = length(z); % signal length
autoc_z = autocorrelation(z, round(K/5));

% Uncomment to load filtered white noise
% z = randn(5000, 1);
% filtercoeff = [1, 0.2-0.5i, 0.2, 0.2];
% z = filter(1, filtercoeff, z);
% K = length(z); % signal length
% autoc_z = autocorrelation(z, K/10);


%% AR model
% compute the vector of coefficients a
N = 2;
[a, sigma_w] = arModel(N, autoc_z);
[H, omega] = freqz(1, [1; a], K, 'whole');


%% Initialisation
upper_limit = 399; % Number of iterations of the algorithm
lambda = 1; % Forgetting factor. For 1, we do not forget past values
c = zeros(N, upper_limit+1); % Coefficient vector
delta = autoc_z(1)/100; % Value at which to initialise P
% P is a N+1 square matrix. P(n) is achieved by making P a parallelogram
% Access P by using P(row, column, time)
P(:,:,1) = (1/delta) * eye(N);
pi_star = zeros(N, upper_limit+1); % pi_star is a series of column vectors
r = zeros(1,upper_limit+1);   % r is a vector of scalars
k_star = zeros(N, upper_limit+1);
d = z; % The reference signal is the input at time k
epsilon = zeros(1, upper_limit+1); % The a posteriori estimation error
e = zeros(1,upper_limit+1);

%% Begin iterating
% Remember, we are implementing a predictor, so the z(k) of the book is
% actually z(k-1) for us. See page 201 for reference.
%
% NOTE: I _hate_ MATLAB's indexing from 1. All indices are kept just like
% they are in the book, and k simply starts from 2 instead of 1.
for k = 2:upper_limit+1
    % Cut off the x(k-1) for this iteration (this part is stolen from the
    % lms implementation), handling the case in which k < N + 1.
    if (k < N + 1)  % Fill up with zeros
        z_k_1 = flipud([zeros(N - k + 1, 1); z(1:k - 1)]);
    else % Just cut the input vector
        z_k_1 = flipud(z((k - N):(k-1)));
    end
    pi_star(:,k) = P(:,:,k-1) * conj(z_k_1);
    r(k) = 1/(lambda + z_k_1.' * pi_star(:,k));
    k_star(:,k) = r(k) * pi_star(:,k);
    epsilon(k) = d(k) - z_k_1.' * c(:, k-1);
    c(:, k) = c(:, k-1) + epsilon(k) * k_star(:,k);
    e(k) = d(k) - z_k_1.' * c(:,k);
```

```
    P(:,:,k) = 1/lambda * (P(:,:,k-1) - k_star(:,k)*pi_star(:,k)');
end

% End of computation.

% Plot c coefficients
for index = 1:N
    figure
    subplot(2, 1, 1)
    plot(1:upper_limit+1, real(c(index, :)), [1, upper_limit+1], -real(a(index))* [1 1])
    title(['Real part of c' int2str(index)]);
    subplot(2, 1, 2)
    plot(1:upper_limit+1, imag(c(index, :)), [1, upper_limit+1], -imag(a(index))* [1 1])
    title(['Imaginary part of c' int2str(index)]);
end

% Plot the error.
figure, plot(1:upper_limit+1, 10*log10(abs(e).^2), [1, upper_limit+1], 10*log10(sigma_w)*[1 1])
title('Error function at each iteration');

% Find the value of coefficients at instant k = 350 and the average of e
% over k \in [350 - 10, 350 + 10]
ind = 350;
win_side_len = 10;
win_len = 2*win_side_len + 1;
c_350 = c(:, ind);
e_350_av = 10*log10(sum(abs(e(ind-win_side_len:ind+win_side_len)).^2)/win_len);
```

```
%% Amp-Phase Estimation RLS
% For reference, see pages 197, 201-203 of the Benvenuto-Cherubini book.

% Clear stuff
close all
clear all

%% Load data

% Load spectral signal
% load('split_signal.mat');
% z = z_lines;
% K = length(z); % signal length
% autoc_z = autocorrelation(z, floor(K/5));

% Load complete signal
z = load('data for hw1.mat');
z = z.z.'; % make a column vector
K = length(z); % signal length
autoc_z = autocorrelation(z, floor(K/5));

f0 = 0.77; %estimated freq from DFT
w0 = 2*pi*f0;

span = 0.005;
step = 0.0001;

% vectors that will host temporary results
corr_vec = zeros(2*(span/step) + 1, 1);
amp_vec = zeros(2*(span/step) + 1, 1);
phi_vec = zeros(2*(span/step) + 1, 1);
i = 1;

for f1 = (f0 - span):step:(f0 + span)


    % Initialisation
    N = 1; % see the first comment
    upper_limit = length(z)-1;%399; % Number of iterations of the algorithm
    lambda = 1; % Forgetting factor. For 1, we do not forget past values
    c = zeros(N, upper_limit+1); % Coefficient vector
    delta = autoc_z(1)/100; % Value at which to initialise P
    % P is a N+1 square matrix. P(n) is achieved by making P a parallelogram
```

```
    % Access P by using P(row, column, time)
    P(:,:,1) = (1/delta) * eye(N);
    pi_star = zeros(N, upper_limit+1); % pi_star is a series of column vectors
    r = zeros(1,upper_limit+1);   % r is a vector of scalars
    k_star = zeros(N, upper_limit+1);
    d = z; % The reference signal is the input at time k
    epsilon = zeros(1, upper_limit+1); % The a priori estimation error
    e = zeros(1,upper_limit+1);

    % Begin iterating
    % Remember, we are implementing a predictor, so the z(k) of the book is
    % actually z(k-1) for us. See page 201 for reference.
    %
    % NOTE: All indices are kept just like they are in the book, and k
    % simply starts from 2 instead of 1.

    w = 2*pi*f1;
    const = 1;
    x = (const * exp(1i * w * (1 : upper_limit+1))).'; % reference signal

    for k = 2:upper_limit+1
        % Cut off the x(k-1) for this iteration (this part is stolen from the
        % lms implementation), handling the case in which k < N.
        if (k < N)  % Fill up with zeros
            x_k = flipud([zeros(N - k, 1); x(1:k)]);
        else % Just cut the input vector
            x_k = flipud(x((k - N + 1):(k)));
        end
        pi_star(:,k) = P(:,:,k-1) * conj(x_k);
        r(k) = 1/(lambda + x_k.' * pi_star(:,k));
        k_star(:,k) = r(k) * pi_star(:,k);

        % Output y(k) computed with old coefficients c(k-1)
        y = x(k) * (c(1, k-1));
        % Compute a priori estimation error (with old coefficients)
        epsilon(k) = d(k) - y;

        c(:, k) = c(:, k-1) + epsilon(k) * k_star(:,k);

        % Output y(k) computed with new coefficients c(k)
        y = x(k) * (c(1, k));
        % Compute a posteriori estimation error (with new coefficients)
        e(k) = d(k) - y;

        P(:,:,k) = 1/lambda * (P(:,:,k-1) - k_star(:,k)*pi_star(:,k)');
    end

    % End of computation.

    % Find amp and phase

    % Average of coefficients from some iteration on, when hopefully they have converged
    expcoeff = mean(c(:, floor(upper_limit*0.9) : upper_limit), 2);

    estimatedsine = x * (expcoeff(1));
    corr = crosscorrelation(estimatedsine, z, floor(length(z)/5));
    corr_vec(i) = corr(1);
    amp_vec(i) = const*abs(expcoeff(1));
    phi_vec(i) = angle(expcoeff(1));
    i = i+1;
end
[mx, j] = max(abs(corr_vec));
amp_est = amp_vec(j);
phi_est = phi_vec(j);

% Plotting
figure, plot(real(estimatedsine)), hold on, plot(imag(estimatedsine), 'r')
title('Estimated signal - imag and real parts')
legend('Real part', 'Imag part')
figure, plot3(1:30, real(estimatedsine(1:30)), imag(estimatedsine(1:30)))
title('First 30 samples of estimated signal')
```

```
%% LMS TEST

close all
clear all
clc
rng default

upper_limit = 4999; % iterations of lms
ctot = zeros(3, upper_limit + 1);
etot = zeros(1, upper_limit);
filtercoeff = [1, 0.2-0.5i, 0.2, 0.2]; % fixed

iterations = 500;
for i=1:iterations

    %% Load data
    z = randn(5000, 1);
    z = filter(1, filtercoeff, z);
    K = length(z); % signal length
    autoc_z = autocorrelation(z, K/10);

    %% AR
    % the knee is apparently at N = 3
    % compute the vector of coefficients a
    N = 3;
    [a, sigma_w] = arModel(N, autoc_z);
    %[H, omega] = freqz(1, [1; a], K, 'whole');
    %hold on, plot(omega*10000/2/pi, 10*log10(abs(H)))

    %%
    N = 3; % order of the predictor
     %MATLAB requires indices from 1 to 401
    c = zeros(N, upper_limit + 1); % init c vector, no info -> set to 0
    % each column of this matrix is c(k), a vector with coefficients from 1 to N (since we are implementing the predictor)!
    e = zeros(1, upper_limit);

    mu = 0.01/(autoc_z(1)*N); % actually mu must be > 0 and < 2/(N r_z(0))


    % watch out, in the predictor y(k) = transp(x(k-1))c(k)
    for k = 1:upper_limit
        if (k < N + 1)
            z_k_1 = flipud([zeros(N - k + 1, 1); z(1:k - 1)]); % input vector z_vec_(k-1) of length N
            % for k = 1 z(1:0) is an empty matrix
        else
            z_k_1 = flipud(z((k - N):(k-1))); % we need the input from k - 1 to k - N
        end
        y_k = z_k_1.'*c(:, k);
        e_k = z(k) - y_k; % the reference signal d(k) is actually the input at sample k
        e(k) = e_k;
        c(:, k + 1) = c(:, k) + mu*e_k*conj(z_k_1); % update the filter, c(k+1) = c(k) + mu*e(k)*conj(z(k-1))
    end

    % Wikipedia's version
%     for k = 1:upper_limit
%         if (k < N + 1)
%             z_k_1 = flipud([zeros(N - k + 1, 1); z(1:(k-1))]); % input vector z_vec_(k-1) of length N
%             % for k = 1 z(1:0) is an empty matrix
%         else
%             z_k_1 = flipud(z((k - N):(k - 1))); % we need the input from k - 1 to k - N
%         end
%         y_k = z_k_1.' * conj(c(:, k));
%         e_k = z(k) - y_k; % the reference signal d(k) is actually the input at sample k
%         e(k) = e_k;
%         c(:, k + 1) = c(:, k) + mu * conj(e_k) * z_k_1; % update the filter, c(k+1) = c(k) + mu*e(k)*conj(z(k-1))
%     end
```

```
    % moving average of each instance
    ctot = ctot + c / iterations;
    etot = etot + abs(e.^2) / iterations;

    disp(i);
%
%     subplot(2, 1, 1)
%     plot(1:upper_limit+1, real(ctot(1, :)), [1, upper_limit+1], -real(a(1))* [1 1])
%     title('Real part of c1');
%     subplot(2, 1, 2)
%     plot(1:upper_limit+1, imag(ctot(1, :)), 1:upper_limit+1, -imag(a(1)))
%     title('Imaginary part of c1');
%     pause(0.01)
end

figure
subplot(2, 1, 1)
plot(1:upper_limit+1, real(ctot(1, :)), [1, upper_limit+1], -real(filtercoeff(2))* [1 1])
title('Real part of c1');
subplot(2, 1, 2)
plot(1:upper_limit+1, imag(ctot(1, :)), [1, upper_limit+1], -imag(filtercoeff(2))* [1 1])
title('Imaginary part of c1');

figure
subplot(2, 1, 1)
plot(1:upper_limit+1, real(ctot(2, :)), [1, upper_limit+1], -real(filtercoeff(3))* [1 1])
title('Real part of c2');
subplot(2, 1, 2)
plot(1:upper_limit+1, imag(ctot(2, :)), [1, upper_limit+1], -imag(filtercoeff(3))* [1 1])
title('Imaginary part of c2');
figure
subplot(2, 1, 1)
plot(1:upper_limit+1, real(ctot(3, :)), [1, upper_limit+1], -real(filtercoeff(4))* [1 1])
title('Real part of c3');
subplot(2, 1, 2)
plot(1:upper_limit+1, imag(ctot(3, :)), [1, upper_limit+1], -imag(filtercoeff(4))* [1 1])
title('Imaginary part of c3');

figure, plot(1:upper_limit, 10*log10(etot), 1:upper_limit, 10*log10(1) * ones(upper_limit, 1))
title('|e(k)|^2 at each iteration k averaged over different realizations')
ylabel('Mean of |e(k)|^2 (dB)')


return


figure
subplot(2, 1, 1)
plot(1:upper_limit+1, real(c(1, :)), [1, upper_limit+1], -real(a(1))* [1 1])
title('Real part of c1');
subplot(2, 1, 2)
plot(1:upper_limit+1, imag(c(1, :)), 1:upper_limit+1, -imag(a(1)))
title('Imaginary part of c1');

figure
subplot(2, 1, 1)
plot(1:upper_limit+1, real(c(2, :)), 1:upper_limit+1, -real(a(2)))
title('Real part of c2');
subplot(2, 1, 2)
plot(1:upper_limit+1, imag(c(2, :)), 1:upper_limit+1, -imag(a(2)))
title('Imaginary part of c2');
figure
subplot(2, 1, 1)
plot(1:upper_limit+1, real(c(3, :)), 1:upper_limit+1, -real(a(3)))
title('Real part of c3');
subplot(2, 1, 2)
plot(1:upper_limit+1, imag(c(3, :)), 1:upper_limit+1, -imag(a(3)))
title('Imaginary part of c3');
```

```
%% Amp-Phase simulation

%% This is a simulation designed to test how well does the RLS method works
% when it comes to find amplitude, phase of a spectral line, given an
% approximated frequency derived from the observation of the peak in the
% DFT. The idea is to apply the computation for frequencies in an interval
% around the approximated one and pick the one that gives the best
% crosscorrleation(0) between the signal we created and the reconstructed
% signal. We will see that for the freq which gives the best xcorr(0) the
% estimates of given amplitude and phase are correct.

% For reference, see pages 197, 201-203 of the Benvenuto-Cherubini book.

% Show that 1 coefficient is enough
% Ae^(j w0 k + j phi) = Ae^jphi e^jw0k, c = Ae^jphi

% Alternative, long and useless proof
% Ae^j(w0k + phi)
% Acos(w0k + phi) + jAsin(w0k + phi)

% Acos(w0k)cos(phi) - Asin(w0K)sin(phi) + jAsin(w0k)cos(phi) +
% jAcos(w0k)sin(phi)

% cm = Acos(phi), cd = Asin(phi)
% cm^2 + cd^2 = A^2(cos^2 + sin^2) = A^2

%cmcos(w0k) + jcmsin(w0k) + jcdcos(w0k) - cdsin(w0k))

%cm e^(jw0k) + j cd e^(jw0k)

% (cm + j cd) e^(jw0k)


% Clear stuff

close all
clear all

sim_length = 1000;

% Set our parameters
amp_est = zeros(sim_length, 1);
phi_est = zeros(sim_length, 1);
rng('default'); % creates reproducible results
f0 = rand();
r_phi = pi*rand() - pi;
r_amp = 10*rand();

% Simulate sim_length times
for index = 1:sim_length
    z = wgn(1000, 1, 10) + r_amp*exp(1i*2*pi*f0*(1:1000).' + 1i * r_phi);
    K = length(z);
    autoc_z = autocorrelation(z, K/5);
    corr_vec = zeros(6, 1);
    amp_vec = zeros(6, 1);
    phi_vec = zeros(6,1);
    i = 1;
    for f1 = f0-0.02:0.01:f0+0.02

        % Initialisation
        N = 1; % see the first comment
        upper_limit = length(z)-1;%399; % Number of iterations of the algorithm
        lambda = 1; % Forgetting factor. For 1, we do not forget past values
        c = zeros(N, upper_limit+1); % Coefficient vector
        delta = autoc_z(1)/100; % Value at which to initialise P
        % P is a N+1 square matrix. P(n) is achieved by making P a parallelogram
        % Access P by using P(row, column, time)
        P(:,:,1) = (1/delta) * eye(N);
        pi_star = zeros(N, upper_limit+1); % pi_star is a series of column vectors
        r = zeros(1,upper_limit+1);   % r is a vector of scalars
```

```
        k_star = zeros(N, upper_limit+1);
        d = z; % The reference signal is the input at time k
        epsilon = zeros(1, upper_limit+1); % The a priori estimation error
        e = zeros(1,upper_limit+1);

        % Begin iterating
        % Remember, we are implementing a predictor, so the z(k) of the book is
        % actually z(k-1) for us. See page 201 for reference.
        %
        % NOTE: All indices are kept just like they are in the book, and k
        % simply starts from 2 instead of 1.

        w = 2*pi*f1;
        const = 1;
        x = (const * exp(1i * w * (1 : upper_limit+1))).';

        for k = 2:upper_limit+1
            % Cut off the x(k-1) for this iteration (this part is stolen from the
            % lms implementation), handling the case in which k < N.
            if (k < N)  % Fill up with zeros
                x_k = flipud([zeros(N - k, 1); x(1:k)]);
            else % Just cut the input vector
                x_k = flipud(x((k - N + 1):(k)));
            end
            pi_star(:,k) = P(:,:,k-1) * conj(x_k);
            r(k) = 1/(lambda + x_k.' * pi_star(:,k));
            k_star(:,k) = r(k) * pi_star(:,k);

            % Output y(k) computed with old coefficients c(k-1)
            y = x(k) * (c(1, k-1));
            % Compute a priori estimation error (with old coefficients)
            epsilon(k) = d(k) - y;

            c(:, k) = c(:, k-1) + epsilon(k) * k_star(:,k);

            % Output y(k) computed with new coefficients c(k)
            y = x(k) * (c(1, k));
            % Compute a posteriori estimation error (with new coefficients)
            e(k) = d(k) - y;

            P(:,:,k) = 1/lambda * (P(:,:,k-1) - k_star(:,k)*pi_star(:,k)');
        end

        % End of computation.

        % Find amp and phase

        % Average of coefficients from some iteration on, when hopefully they have converged
        expcoeff = mean(c(:, floor(upper_limit*0.9) : upper_limit), 2);

        estimatedsine = x * (expcoeff(1));
        corr = crosscorrelation(estimatedsine, z, length(z)/5);
        corr_vec(i) = corr(1);
        amp_vec(i) = const*abs(expcoeff(1));
        phi_vec(i) = angle(expcoeff(1));
        i = i+1;
    end
    [mx, j] = max(abs(corr_vec));
    amp_est(index) = amp_vec(j);
    phi_est(index) = phi_vec(j);
end

%% Statistical values

mse_amp = sum((amp_est-r_amp).^2)/length(amp_est);
% watch out for the following, if the r_phi is over pi then use (-2*pi+r_phi)
mse_phi = sum((phi_est-r_phi).^2)/length(amp_est);

%% Different approach
% Check if it picks the correct frequency. Freq, amp and phase will be different
% each time.
```

```
% The correct index that should appear in ind_j is span/step + 1 (the
% center of the vector of frequencies passed to RLS, which is actually the
% freq of the input signal)

sim_length = 500;

% Set our parameters
amp_est_2 = zeros(sim_length, 1);
phi_est_2 = zeros(sim_length, 1);
ind_j = zeros(sim_length, 1);
rng('default');
span = 0.005;
step = 0.0001;

% Simulate sim_length times
for index = 1:sim_length
    w0 = rand();
    r_phi = pi*rand() - pi; %[-pi, pi] phase
    r_amp = 10*rand();
    z = wgn(1000, 1, 10) + r_amp*exp(1i*2*pi*w0*(1:1000).' + 1i * r_phi);
    K = length(z);
    autoc_z = autocorrelation(z, K/5);

    corr_vec = zeros(2*(span/step) + 1, 1);
    amp_vec = zeros(2*(span/step) + 1, 1);
    phi_vec = zeros(2*(span/step) + 1, 1);
    i = 1;
    for w1 = w0-span:step:w0+span % Then w0 should be the 6th element of the vector (span/step + 1)

        % Initialisation
        N = 1; % see the first comment
        upper_limit = length(z)-1;%399; % Number of iterations of the algorithm
        lambda = 1; % Forgetting factor. For 1, we do not forget past values
        c = zeros(N, upper_limit+1); % Coefficient vector
        delta = autoc_z(1)/100; % Value at which to initialise P
        % P is a N+1 square matrix. P(n) is achieved by making P a parallelogram
        % Access P by using P(row, column, time)
        P(:,:,1) = (1/delta) * eye(N);
        pi_star = zeros(N, upper_limit+1); % pi_star is a series of column vectors
        r = zeros(1,upper_limit+1);   % r is a vector of scalars
        k_star = zeros(N, upper_limit+1);
        d = z; % The reference signal is the input at time k
        epsilon = zeros(1, upper_limit+1); % The a priori estimation error
        e = zeros(1,upper_limit+1);

        % Begin iterating
        % Remember, we are implementing a predictor, so the z(k) of the book is
        % actually z(k-1) for us. See page 201 for reference.
        %
        % NOTE: I _hate_ MATLAB's indexing from 1. All indices are kept just like
        % they are in the book, and k simply starts from 2 instead of 1.

        %c(:, 1) = 15 + 2i;
        w = 2*pi*w1;
        const = 1;
        x = (const * exp(1i * w * (1 : upper_limit+1))).';

        for k = 2:upper_limit+1
            % Cut off the x(k-1) for this iteration (this part is stolen from the
            % lms implementation), handling the case in which k < N.
            if (k < N)  % Fill up with zeros
                x_k = flipud([zeros(N - k, 1); x(1:k)]);
            else % Just cut the input vector
                x_k = flipud(x((k - N + 1):(k)));
            end
            pi_star(:,k) = P(:,:,k-1) * conj(x_k);
            r(k) = 1/(lambda + x_k.' * pi_star(:,k));
            k_star(:,k) = r(k) * pi_star(:,k);

            % Output y(k) computed with old coefficients c(k-1)
            y = x(k) * (c(1, k-1));
```

```
            % Compute a priori estimation error (with old coefficients)
            epsilon(k) = d(k) - y;

            c(:, k) = c(:, k-1) + epsilon(k) * k_star(:,k);

            % Output y(k) computed with new coefficients c(k)
            y = x(k) * (c(1, k));
            % Compute a posteriori estimation error (with new coefficients)
            e(k) = d(k) - y;

            P(:,:,k) = 1/lambda * (P(:,:,k-1) - k_star(:,k)*pi_star(:,k)');
        end

        % End of computation.

        % Find amp and phase

        % Average of coefficients from some iteration on, when hopefully they have converged
        expcoeff = mean(c(:, floor(upper_limit*0.9) : upper_limit), 2);

        estimatedsine = x * (expcoeff(1));
        corr = crosscorrelation(estimatedsine, z, length(z)/5);
        corr_vec(i) = corr(1);
        amp_vec(i) = const*abs(expcoeff(1));
        phi_vec(i) = angle(expcoeff(1));
        i = i+1;
    end
    [mx, j] = max(abs(corr_vec));
    ind_j(index) = j;
    amp_est_2(index) = amp_vec(j) - r_amp;
    phi_est_2(index) = phi_vec(j) - r_phi;
end

wrong = ind_j(find(ind_j ~= span/step + 1));
```

```
function [ a, sigma_w ] = arModel( N, autoc )
% ARMODEL of order N, given the unbiased/biased estimate of the
% autocorrelation of the signal whose PSD has to be estimated

row1 = conj(autoc);
% create the Toeplitz R matrix
R = toeplitz(row1(1:N));
% create r vector
r = autoc(2:N+1);
% Yule-Walker equations
a = -inv(R)*r;
sigma_w = abs(autoc(1) + r'*a);  % Abs to correct rounding errors

end
```

```
function [ autoc ] = autocorrelation_biased( z1, N_corr )
%AUTOCORRELATION biased estimator pg 83 of Benvenuto Cherubini

K = length(z1);
autoc = zeros(N_corr + 1, 1);
for n = 1:(N_corr + 1)
    d = z1(n:K);
    b = conj(z1(1:(K - n + 1)));
    c = K;
    autoc(n) = d.' * b / c;
end

end
```

```
function [ autoc_complete ] = autocorrelation_complete( z1, N_corr )
% Compute the negative index values of the autocorrelation estimate
autoc = autocorrelation(z1, N_corr);

K = length(z1);
```

```
% consider formulas as more close as possible to the book
% make autocorrelation simmetric
autoc_complete = zeros(K, 1);
autoc_complete(1:N_corr + 1) = autoc;
temp = flipud(conj(autoc));
% it's the same as puttig the conjugate, flipped, at the end of this
% vector, since fft considers a periodic repetition of the signal
autoc_complete((K - N_corr + 1):K) = temp(1:length(temp)-1);

end
```

```
function [ autoc ] = autocorrelation( z1, N_corr )
% Compute the autocorrelation estimate of a signal

K = length(z1);
autoc = zeros(N_corr + 1, 1);
% we should use the unbiased estimator pg 82 1.478
for n = 1:(N_corr + 1)
    d = z1(n:K);
    b = conj(z1(1:(K - n + 1)));
    c = K - n + 1; % check this scaling factor
    autoc(n) = d.' * b / c;
end

end
```

```
function [ correlogram ] = correlogramPsd( z1, window, N_corr )
% Compute the PSD estimate using the correlogram method

K = length(z1);
autoc_complete = autocorrelation_complete(z1, N_corr);
window_complete = zeros(K, 1);
window_complete(1:N_corr + 1) = window(N_corr + 1 : 2*N_corr + 1);
window_complete(K - N_corr + 1 : K) = window(1 : N_corr);

windowed_autoc = autoc_complete .* window_complete;
correlogram = fft(windowed_autoc);

end
```

```
function [ autoc ] = crosscorrelation( z1, z2, N_corr )
% CROSSCORRELATION - z1 and z2 should be of the same length
% This function is used only to find the crosscorrelation between complex
% sinuosoids in the frequency, amplitude and phase detector algorithm. Its
% definition recalls the definition of the xcorr function of MATLAB,
% although this is normalized.
K = length(z1);
autoc = zeros(N_corr + 1, 1);
for n = 1:(N_corr + 1)
    d = z1(n:K);
    b = conj(z2(1:(K - n + 1)));
    c = K - n + 1;
    autoc(n) = d.' * b / c;
end

end
```

```
function [ ] = DTFTplot( x, res )
%DTFTPLOTLOG Logarithmic plot of the DTFT module of the signal

b = x; a = 1;  % Define b and a in z^-1
[Hf, f] = freqz(b, a, res, 1, 'whole');

figure
%subplot(2, 1, 1)
plot(f, 20*log10(abs(Hf)))
%subplot(2, 1, 2), plot(f, angle(Hf))

end
```

```
function plot_spectrum(signal, N_ar)
% This function computes different estimators on a given signal: Periodogram,
% Welch periodogram, Correlogram and if N_ar > 0 also the AR model of order
% N_ar. The parameters and the windows of Welch and Correlogram are hard
% coded in the function and not passed as arguments. It also plots the
% various estimate on the same plot, with frequency normalized in [0,1].
% Use ylim in the main script to set the desired dinamic in the Y axis.

    K = length(signal);

    % PERIODOGRAM pg 84
    Z = fft(signal);
    periodogr = abs(Z).^2/K;

    % compute WELCH estimator pg 85
    D = 200; % window size
    window = kaiser(D, 5.65);
    S = D/2; %common samples
    P_welch = welchPsd(signal, window, S);

    % CORRELOGRAM
    N_corr = ceil(K/5); % N_corr is the order of the autocorrelation estimate
    window_correlogram = kaiser(2*N_corr + 1, 5.65); % window centered around N_corr
    correlogram = correlogramPsd(signal, window_correlogram, N_corr);

    % AR model
    if (N_ar > 0)
        %compute variance of AR model and plot it to identify the knee
        %it's computed up to K/5 - 1
        autoc = autocorrelation_biased(signal, N_corr);

        %compute the vector of coefficients a
        [a, sigma_w] = arModel(N_ar, autoc);
        [H, omega] = freqz(1, [1; a], K, 'whole');
    end


    % Plot PSD estimate
    figure, hold on
    plot((1:K)/K, 10*log10(P_welch), 'Color', 'r', 'LineWidth', 2)
    plot((1:K)/K, 10*log10(abs(correlogram)), 'Color', 'b', 'LineWidth', 1)
    plot((1:K)/K, 10*log10(periodogr), 'c:')
    if (N_ar > 0)
        plot(omega/(2*pi), 10*log10(sigma_w*abs(H).^2), 'Color', 'm', 'LineWidth', 1);
        legend('Welch', 'Correlogram', 'Periodogram', ['AR(' int2str(N_ar) ')'], 'Location', '
              SouthWest')
    else
        legend('Welch', 'Correlogram', 'Periodogram', 'Location', 'SouthWest')
    end
    hold off
    title('Spectral analysis')
    xlabel('Normalized frequency')
    ylabel('Magnitude (dB)')

end
```

```
function [ P_welch ] = welchPsd( z1, window, S )
% WELCHPSD This function computes the Welch estimation of the PSD of a
% random process.
%
%   z1: signal for which to compute the estimate
%   window: an array containing the window to use to frame the signal
%   S: number of overlapping samples

D = length(window);
K = length(z1); % signal length
M_w = 1/D * sum(window.^2); % Power of the window
N_s = floor((K-D)/(D-S) + 1); % number of subsequences
P_per_w = zeros(K, N_s);
for s = 0:(N_s-1)
```

```
  z_s = window .* z1( s*(D-S) + 1 : s*(D-S) + D ); % 1.495 with index + 1
  Z_s = fft(z_s, K);
  P_per_w(:, s + 1) = abs(Z_s).^2/(D*M_w);
end
P_welch = sum(P_per_w, 2)/N_s;

end
```