SSY130 - Applied Signal Processing

CHALMERS
UNIVERSITY OF TECHNOLOGY

Project 1b: Group 19

Ana Crkvenjas
Jahanvi Bhadrashetty Dinesh
Lisa Mårtensson
Rob Theodoor Wilhem Anton Vissers

Birthdate: 19990919
Passphrase: Cubchoo

29th November 2023

1 Introduction

This document will describe project alternative 1b regarding 'Acoustic Communication - Interpolation, Modulation, Demodulation and Decimation'. It will provide answers to the exercises, as well as a graphical overview of the workings of the code.

2 Exercise 1

For an interpolation factor, R, of 8, sampling frequency, f_s, of 16 kHz, and a modulation frequency, f_c, of 4 kHz, the bandwidth of the signal is computed as

$$BW = \frac{f_s}{R} = 2 \text{ kHz}. \tag{1}$$

Moreover, the modulation frequency is the carrier frequency to which the baseband signal is upconverted. This will yield that the transmission band lies between 3 kHz and 5 kHz, considering the bandwidth of the signal around the carrier frequency, as presented in Fig. 1. Hence, the carrier frequency is 4 kHz, and the transmitted signal band will lie between [-1, 1] kHz of the carrier, so the band lies between 3 kHz and 5 kHz.

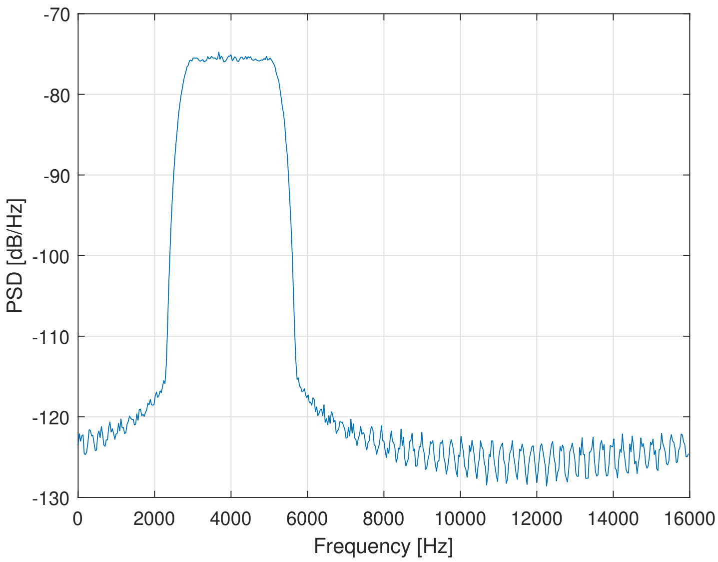

Figure 1: Power-spectral density of the transmitted signal around the carrier frequency.

3 Exercise 2

Compared to project 1a, there is always a non-zero Error Vector Magnitude (EVM), which can be attributed to potential aliasing after decimation of the signal. On top of that, the filter used in project 1b is a non-ideal Low-Pass Filter (LPF), which affects the received signal for its higher frequency passband, as shown in Fig. 2. Since we applied a non-ideal LPF to remove high-frequency components, it also introduced distortion due to the filter its transition band and stopband. The transition band, which is the region between the passband and the stopband, allows some high-frequency components to pass through, contributing to EVM. Similarly, the stopband, which is the region beyond the passband, does not perfectly attenuate all high-frequency components, allowing some leakage into the passband, further increasing EVM.

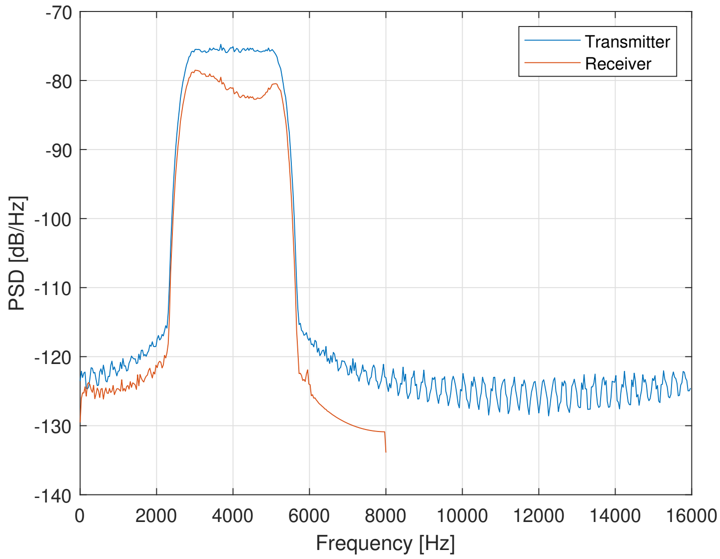

Figure 2: Power-spectral density of the transmitted and received signal around the carrier frequency.

4 Exercise 3

The channel, H, is influenced by the different blocks of the system. Each block will be shortly discussed to show how it might affect the estimated channel response.

Firstly, interpolation is considered, where the input signal is upsampled by a factor R and then passed through a LPF. High frequency spectral components arise from the fact that the upsampling creates $R-1$ copies of the input signal, which the LPF has to remove. Any non-idealities in the LPF might affect the filtered response of the pilot symbols in the interpolation step.

Secondly, the baseband signal is modulated around its carrier frequency, generating a passband signal. This step is performed by multiplying the baseband signal with a complex exponential at the carrier frequency, which will shift the pilot symbols in the frequency domain.

Thirdly, only the real part of the signal is transmitted over the channel, since all information of the message can be retrieved by multiplying by a cosine and sine at the carrier frequency.

Next up, in the demodulation step, the passband signal is multiplied by the conjugate of the exponential around the carrier frequency, bringing the signal back to baseband. This step should only present a frequency shift.

Finally, when performing the decimation step, first an LPF is applied, followed by down-sampling by a factor $D = R$. This will remove the $D-1$ copies of the signal generated in the interpolation step. Hence, in total, the pilot symbols should be retrieved fully correct if the interpolation and decimation step are equal.

5 Exercise 4

Only the real part of the signal, z_r, is transmitted, which yields

$$z_r = \frac{1}{2}(z + \bar{z}), \tag{2}$$

where $\bar{z}$ is the complex conjugate of the complete signal, z. It is sufficient to only transmit the real part, since the real and imaginary data can be fully reconstructed by multiplying with a cosine and sine term as

$$y_r = z_r \cdot \cos(2\pi f_c), \tag{3}$$
$$y_i = z_r \cdot \sin(2\pi f_c), \tag{4}$$

where y_r is the real part of the received signal and y_i is the imaginary part of the received signal.

6 Exercise 5

Different LPF properties are less or more crucial when doing the interpolation or decimation step. Some of the properties will now be discussed in more detail. To get closer to an ideal filter when doing interpolation or decimation, the ripple is desired to be small.

First of all, the passband ripple, δ_p defines the fluctuations around the passband of the filter, and is preferably as small as possible. The passband ripple can be reduced in practice by having a higher-order filter.

Secondly, the stopband attenuation is preferably as large as possible, with a minimized ripple. This is to ensure that the filter operates as close to an ideal filter as possible. Also, when the stopband attenuation is very high, the transition bandwidth will decrease significantly. Hence, when interpolating or decimating, it should not be too steep to avoid spectral regrowth.

Thirdly, the transition bandwidth is defined by the frequency region from the maximum passband frequency to the minimum stopband frequency. Generally, the filter has a smooth roll-off factor instead of a very steep roll-off factor to reduce unwanted spectral content. A higher-order filter can narrow down the transition bandwidth. This property is most important when doing interpolation and decimation, since the LPF should not generate the unwanted spectral components.

Lastly, the phase linearity is least important, as it is only needed for signal shaping. For both the interpolation and decimation step, it can be mainly disregarded due to filter non-idealities.

7 Exercise 6

The Fourier transform of the channel estimate, $\hat{H}$, is given by

$$\hat{H} = \bar{T} \cdot R, \tag{5}$$

where $\bar{T}$ is the complex conjugate of the Fourier transform of the transmitted signal and R is the Fourier transform of the received signal. This will yield a magnitude response of

$$|\hat{H}| = |\bar{T} \cdot R| = |H| \cdot |T|^2, \tag{6}$$

and a phase response of

$$\angle\hat{H} = \angle\bar{T} + \angle R = \angle R - \angle T, \tag{7}$$

following from the complex value $\bar{T}$. For the actual value of the channel, H, the magnitude is given by

$$|H| = \left|\frac{R}{T}\right|, \tag{8}$$

and the phase response is given by

$$\angle H = \angle\frac{R}{T} = \angle R - \angle T. \tag{9}$$

Hence, the angle is equal and the magnitude is scaled by a factor of $|T|^2$, which is in turn eliminated by the modulation format scaling of $\sqrt{\frac{1}{2}}$, giving the same magnitude response as a result.

8 Exercise 7

The received symbols are equalized as R_{eq} according to

$$R_{eq} = R \cdot \bar{\hat{H}}, \tag{10}$$

where $\bar{\hat{H}}$ is the complex conjugate of the Fourier transform of the channel estimate and R is the Fourier transform of the received signal. Equivalently, this has a phase response of

$$\angle R_{eq} = \angle R + \angle \bar{\hat{H}}. \tag{11}$$

According to Eq. 7 the phase response can be rewritten as

$$\angle R_{eq} = \angle R - (\angle R - \angle T) = \angle T, \tag{12}$$

which shows that the equalized phase is the same as the phase of the transmitted signal.

9 Exercise 8

When the speakers and microphones are placed close together and the highest volume amplitude is used, the EVM is very small, around 0.1, and there are almost no bit errors (see Fig. 3). However, as the amplitude of the transmitted signal is decreased, the EVM increases and the symbols become more spread out or less precise in relation to the correct symbol position in the constellation graph (see Fig. 4).This is because reducing the amplitude of the signal while keeping the background noise level constant effectively reduces the Signal-to-Noise Ratio (SNR). As a result, the received signal contains noise with a large energy compared to the energy of the transmitted message. This is similar to what was observed in project 1a, where decreasing the SNR from 30 dB to 5 dB significantly increased the noise clutter around the desired constellation points.

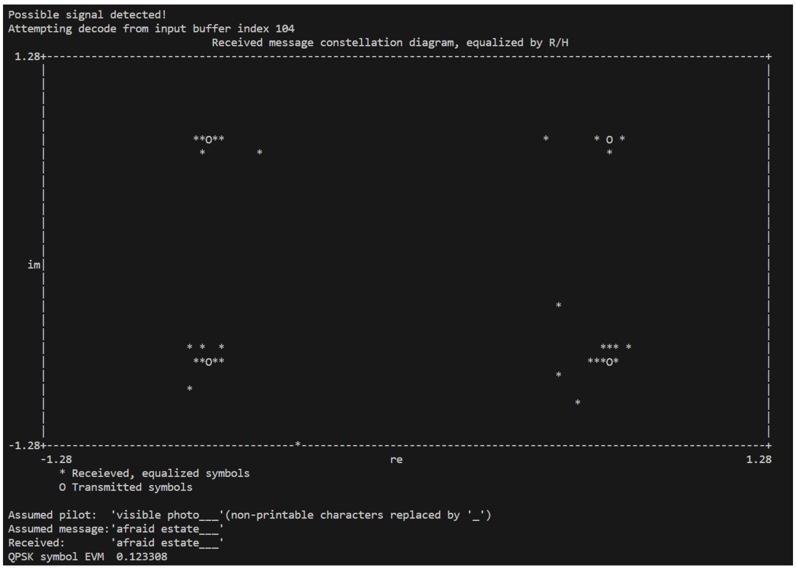

Figure 3: Constellation diagram for high signal amplitude.

Figure 4: Constellation diagram for low signal amplitude.

10 Appendix

The MATLAB code is given below:

```
%NO_PFILE
function [funs, student_id] = student_sols()
%STUDENT_SOLS Contains all student solutions to problems.

% ----------------------------------------
%               STEP 1
% ----------------------------------------
% Set to your birthdate / the birthdate of one member in the group.
% Should a numeric value of format YYYYMMDD, e.g.
% student_id = 19900101;
% This value must be correct in order to generate a valid secret key.
student_id = 19990919;


% ----------------------------------------
%               STEP 2
% ----------------------------------------
% Your task is to implement the following skeleton functions.
% You are free to use any of the utility functions located in the same
% directory as this file as well as any of the standard matlab functions.


 function z = add_cyclic_prefix(x,Ncp)  %#ok<*INUSD>
  % Adds (prepends) a Ncp long cyclic prefix to the ofdm block x.
  x = x(:);   %#ok<*NASGU> % Ensure x is a column vector
  z = [x(end-Ncp+1:1:end).', x.'].';
    end

    function x = remove_cyclic_prefix(z,Ncp)
        % Removes a Ncp long cyclic prefix from the ofdm package z
        z = z(:);   % Ensure z is a column vector
        x = z(Ncp+1:1:end);
    end

    function symb = bits2qpsk(bits)
        % Encode bits as qpsk symbols
        % ARGUMENTS:
        % bits = array of bits. Numerical values converted as:
        %   zero -> zero
        %   nonzero -> one
        % Must be of even length!
        % OUTPUT:
        % x = complex array of qpsk symbols encoding the bits. Will contain
        % length(bits)/2 elements. Valid output symbols are
        % 1/sqrt(2)*(+/-1 +/- i). Symbols grouped by pairs of bits, where
        % the first corresponds the real part of the symbol while the
        % second corresponds to the imaginary part of the symbol. A zero
        % bit should be converted to a negative symbol component, while a
        % nonzero bit should be converted to a positive symbol component.

        % Convert bits vector of +/- 1
        bits = double(bits);
        bits = bits(:);

        for i = 1:1:length(bits)
            if bits(i) == 0
                bits(i) = -1;
            else
                bits(i) = 1;
            end
        end

        if rem(length(bits),2) == 1
            error('bits must be of even length');
```

```
        end

        for k = 1:1:length(bits)/2
            symb(k) = sqrt(1/2)*(bits(2*k-1)+1i*bits(2*k));
        end

        symb = symb.';
    end

    function bits = qpsk2bits(x)
        % Convert qpsk symbols to bits.
        % Output will be a vector twice as long as the input x, with values
        % 0 or 1.
        x = x(:);
        bits = false(2*length(x),1);
        % Note: you only need to check which quadrant of the complex plane
        % the symbol lies in in order to map it to a pair of bits. The
        % first bit corresponds to the real part of the symbol while the
        % second bit corresponds to the imaginary part of the symbol.

        bits_double = zeros(length(x),2);

        for k = 1:1:length(x)
            if real(x(k)) > 0 && imag(x(k)) > 0
                bits_double(k,:) = [1, 1];
            elseif real(x(k)) < 0 && imag(x(k)) > 0
                bits_double(k,:) = [-1, 1];
            elseif real(x(k)) > 0 && imag(x(k)) < 0
                bits_double(k,:) = [1, -1];
            elseif real(x(k)) < 0 && imag(x(k)) < 0
                bits_double(k,:) = [-1, -1];
            end
        end

        for l = 1:1:length(bits_double)
            if l == 1
                bits_org = [bits_double(l,:)];
            else
                bits_org = [bits_org, bits_double(l,:)];
            end
        end

        bits = bits_org.';

        bits(bits ~= -1) = 1;
        bits(bits == -1) = 0;

        % Ensure output is of correct type
        % zero value -> logical zero
        % nonzero value -> logical one
        bits = logical(bits);
    end

    function [rx, evm, ber, symbs] = sim_ofdm_known_channel(tx, h, N_cp, snr, sync_err)
        % Simulate OFDM signal transmission/reception over a known channel.
        %
        % ---------------------------------------------------------------------
        % NOTE: THIS FUNCTION WILL NOT BE SELF-TESTED!
        % It will be up to you to study the output from this function and
        % determine if the results are correct or not.
        % ---------------------------------------------------------------------
        %
        % Arguments:
        %   tx          Bits to transmit [-]
        %   h           Channel impulse response [-]
        %   N_cp        Cyclic prefix length [samples]
```

```
    %   snr         Channel signal/noise ration to apply [dB]
    %   sync_err    Reciever synchronization error [samples]
    % Outputs:
    %   rx          Recieved bits [-]
    %   evm         Error vector magnitude (see below) [-]
    %   ber         Bit error rate (see below) [-]
    %   symbs       Structure containing fields:
    %       .tx         Transmitted symbols
    %       .rx_pe      Recieved symbols, pre-equalization
    %       .rx_e       Recieved symbols, post-equalization
    %
    % In this function, you will now fully implement a simulated
    % base-band OFDM communication scheme. The relevant steps in this
    % are:
    %   - Get a sequence of bits to transmit
    %   - Convert the bits to OFDM symbols
    %   - Create an OFDM block from the OFDM symbols
    %   - Add a cyclic prefix
    %   - Simulate the transmission and reception over the channel using the
    %   simulate_baseband_channe function.
    %   - Remove the cyclic prefix from the recieved message.
    %   - Equalize the recieved symbols by the channel gain
    %   - Convert the equalized symbols back to bits
    %   - Compare the recieved bits/symbols to the transmitted bits/symbols.
    %
    % If you have implemented the skeleton functions earlier in this
    % file then this function will be very simple as you can call your
    % functions to perform the needed tasks.

warning('Note that this function is _not_ self-tested. It is up to you to study
  the output any verify that it is correct! You can remove this warning if you
  wish.');

    % Ensure inputs are column vectors
    tx = tx(:);
    h = h(:);

    tx_alt = double(tx);

    for i = 1:1:length(tx_alt)
        if tx_alt(i) == 0
            tx_alt(i) = -1;
        else
            tx_alt(i) = 1;
        end
    end

    if rem(length(tx_alt),2) == 1
        error('bits must be of even length');
    end

    % Convert bits to QPSK symbols
    for k = 1:1:length(tx_alt)/2
        x(k) = sqrt(1/2)*(tx_alt(2*k-1)+1i*tx_alt(2*k));
    end

    x = x.';

    symbs.tx = x;   % Store transmitted symbols for later

    % Number of symbols in message
    N = length(x);

    % Create OFDM time-domain block using IDFT
    for n = 1:1:N
        for k = 1:1:N
            if k == 1
```

```
                z(n) = (1/N).*x(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
            else
                z(n) = z(n) + (1/N).*x(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
            end
        end
    end

    z = z.';

    % Add cyclic prefix to create OFDM package
    zcp = [z(end-N_cp+1:1:end).', z.'].';

    % Send package over channel
    ycp = simulate_baseband_channel(zcp, h, snr, sync_err);
    % Only keep the first N+Ncp recieved samples. Consider why ycp is longer
    % than zcp, and why we only need to save the first N+Ncp samples. This is
    % important to understand.
    ycp = ycp(1:N+N_cp);

    % Remove cyclic prefix
    y = ycp(N_cp+1:1:end);

    % Convert to frequency domain using DFT
    for k = 1:1:N
        for n = 1:1:N
            if n == 1
                r(k) = y(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
            else
                r(k) = r(k) + y(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
            end
        end
    end

    r = r.';

    symbs.rx_pe = r; % Store symbols for later

    % Remove effect of channel by equalization. Here, we can do this by
    % dividing r (which is in the frequency domain) by the channel gain (also
    % in the frequency domain).
    Nh = length(h);

    for k = 1:1:N
        for n = 1:1:Nh
            if n == 1
                H(k) = h(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
            else
                H(k) = H(k) + h(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
            end
        end
    end

    H = H.';

    r_eq = r./H;

    symbs.rx_e = r_eq; %Store symbols for later

    % Calculate the quality of the received symbols.
    % The error vector magnitude (EVM) is one useful metric.
    evm = norm(x - r_eq)/sqrt(N);

    % Convert the recieved symsbols to bits
    rx = false(2*length(r_eq),1);

    bits_double = zeros(length(r_eq),2);

```

```
    for k = 1:1:length(r_eq)
        if real(r_eq(k)) > 0 && imag(r_eq(k)) > 0
            bits_double(k,:) = [1, 1];
        elseif real(r_eq(k)) < 0 && imag(r_eq(k)) > 0
            bits_double(k,:) = [-1, 1];
        elseif real(r_eq(k)) > 0 && imag(r_eq(k)) < 0
            bits_double(k,:) = [1, -1];
        elseif real(r_eq(k)) < 0 && imag(r_eq(k)) < 0
            bits_double(k,:) = [-1, -1];
        end
    end

    for l = 1:1:length(bits_double)
        if l == 1
            bits_org = [bits_double(l,:)];
        else
            bits_org = [bits_org, bits_double(l,:)];
        end
    end

    rx = bits_org.';

    rx(rx ~= -1) = 1;
    rx(rx == -1) = 0;

    rx = logical(rx);

    % Calculate the bit error rate (BER).
    % This indicates the relative number of bit errors.
    % Typically this will vary from 0 (no bit errors) to 0.5 (half of all
    % receieved bits are different, which is the number we'd expect if we
    % compare two random bit sequences).
    ber = 1-sum(rx == tx)/length(rx);
end

function txFrame = concat_packages(txPilot,txData)
    % Concatenate two ofdm blocks of equal size into a frame
    txPilot = txPilot(:);
    txData = txData(:);
    if(length(txData) ~= length(txPilot))
        error('Pilot and data are not of the same length!');
    end
    txFrame = [txPilot.', txData.'].';
end

function [rxPilot, rxData] = split_frame(rxFrame)
    % Split an ofdm frame into 2 equal ofdm packages
    rxFrame = rxFrame(:);
    if rem(length(rxFrame),2) > 0
        error('Vector z must have an even number of elements');
    end
    N = length(rxFrame);
    rxPilot = rxFrame(1:1:N/2);
    rxData = rxFrame(N/2+1:1:end);
end

function [rx, evm, ber, symbs] = sim_ofdm_unknown_channel(tx, h, N_cp, snr,
sync_err)
    % Simulate OFDM signal transmission/reception over an unknown
    % channel.
    %
    % ----------------------------------------------------------------
    % NOTE: THIS FUNCTION WILL NOT BE SELF-TESTED!
    % It will be up to you to study the output from this function and
    % determine if the results are correct or not.
    % ----------------------------------------------------------------
    %
```

```
    % Arguments:
    %   tx          Structure with fields:
    %     .p        Pilot bits to transmit
    %     .d        Data bits to transmit
    %   h           Channel impulse response [-]
    %   N_cp        Cyclic prefix length [samples]
    %   snr         Channel signal/noise ration to apply [dB]
    %   sync_err    Reciever synchronization error [samples]
    % Outputs:
    %   rx          Recieved bits [-]
    %   evm         Error vector magnitude (see below) [-]
    %   ber         Bit error rate (see below) [-]
    %   symbs       Structure containing fields:
    %       .tx         Transmitted symbols
    %       .rx_pe      Recieved symbols, pre-equalization
    %       .rx_e       Recieved symbols, post-equalization
    %
    %
    % This function is similar to the known-channel problem, but with
    % the added complexity of requiring to estimate the channel
    % response. The relevant steps to perform here are:
    %   - Get a sequence of pilot and data bits to transmit
    %   - Convert the pilot and data bits to OFDM symbols
    %   - Create an OFDM block from the OFDM symbols for the pilot and
    %   data
    %   - Add a cyclic prefix to the pilot and data
    %   - Concatenate the pilot and data blocks to create an entire
    %   OFDM frame
    %   - Simulate the transmission and reception over the channel using the
    %   simulate_baseband_channe function.
    %   - Split the recieved message into a recieved pilot and data
    %   segment
    %   - Remove the cyclic prefixes from the recieved messages
    %   - Estimate the channel gain from the pilot block
    %   - Equalize the recieved data symbols by the channel gain
    %   - Convert the equalized symbols back to bits
    %   - Compare the recieved bits/symbols to the transmitted bits/symbols.

warning('Note that this function is _not_ self-tested. It is up to you to study the output any verify that it is correct! You can remove this warning if you wish.');

    % Ensure inputs are column vectors
    tx.d = tx.d(:);
    tx.p = tx.p(:);
    h = h(:);

    tx_alt.d = double(tx.d);

    for i = 1:1:length(tx_alt.d)
        if tx_alt.d(i) == 0
            tx_alt.d(i) = -1;
        else
            tx_alt.d(i) = 1;
        end
    end

    if rem(length(tx_alt.d),2) == 1
        error('bits must be of even length');
    end

    tx_alt.p = double(tx.p);

    for i = 1:1:length(tx_alt.p)
        if tx_alt.p(i) == 0
            tx_alt.p(i) = -1;
        else
```

```
                tx_alt.p(i) = 1;
            end
        end

        if rem(length(tx_alt.p),2) == 1
            error('bits must be of even length');
        end

        % Convert bits to QPSK symbols
        for k = 1:1:length(tx_alt.d)/2
            x.d(k) = sqrt(1/2)*(tx_alt.d(2*k-1)+1i*tx_alt.d(2*k));
        end

        x.d = x.d.';

        for k = 1:1:length(tx_alt.p)/2
            x.p(k) = sqrt(1/2)*(tx_alt.p(2*k-1)+1i*tx_alt.p(2*k));
        end

        x.p = x.p.';

        symbs.tx = x.d;    % Store transmitted data symbols for later

        % Number of symbols in message
        N = length(x.d);
        if length(x.d) ~= length(x.p)
           error('Pilot and data messages must be of equal length');
        end

        % Create OFDM time-domain block using IDFT
        for n = 1:1:N
            for k = 1:1:N
                if k == 1
                    z.d(n) = (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                else
                    z.d(n) = z.d(n) + (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        z.d = z.d.';

        for n = 1:1:N
            for k = 1:1:N
                if k == 1
                    z.p(n) = (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                else
                    z.p(n) = z.p(n) + (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        z.p = z.p.';

        % Add cyclic prefix to create OFDM package
        zcp.d = [z.d(end-N_cp+1:1:end).', z.d.'].';
        zcp.p = [z.p(end-N_cp+1:1:end).', z.p.'].';

        % Concatenate the messages
        tx_frame = [zcp.p.', zcp.d.'].';

        % Send package over channel
        rx_frame = simulate_baseband_channel(tx_frame, h, snr, sync_err);
        % As before, only keep the first samples
        rx_frame = rx_frame(1:2*(N+N_cp));

        % Split frame into packages
```

```
        ycp = struct();
        ycp.d = rx_frame(length(rx_frame)/2+1:1:end);
        ycp.p = rx_frame(1:1:length(rx_frame)/2);

        % Remove cyclic prefix
        y.d = ycp.d(N_cp+1:1:end);
        y.p = ycp.p(N_cp+1:1:end);

        % Convert to frequency domain using DFT
        for k = 1:1:N
            for n = 1:1:N
                if n == 1
                    r.d(k) = y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                else
                    r.d(k) = r.d(k) + y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        r.d = r.d.';

        for k = 1:1:N
            for n = 1:1:N
                if n == 1
                    r.p(k) = y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                else
                    r.p(k) = r.p(k) + y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        r.p = r.p.';

        symbs.rx_pe = r.d; % Store symbols for later

        % Esimate channel
        H = r.p./x.p;

        % Remove effect of channel on the data package by equalization.
        r_eq = r.d./H;

        symbs.rx_e = r_eq; %Store symbols for later

        % Calculate the quality of the received symbols.
        % The error vector magnitude (EVM) is one useful metric.
        evm = norm(x.d - r_eq)/sqrt(N);

        % Convert the recieved symsbols to bits
        rx = false(2*length(r_eq),1);

        bits_double = zeros(length(r_eq),2);

        for k = 1:1:length(r_eq)
            if real(r_eq(k)) > 0 && imag(r_eq(k)) > 0
                bits_double(k,:) = [1, 1];
            elseif real(r_eq(k)) < 0 && imag(r_eq(k)) > 0
                bits_double(k,:) = [-1, 1];
            elseif real(r_eq(k)) > 0 && imag(r_eq(k)) < 0
                bits_double(k,:) = [1, -1];
            elseif real(r_eq(k)) < 0 && imag(r_eq(k)) < 0
                bits_double(k,:) = [-1, -1];
            end
        end

        for l = 1:1:length(bits_double)
            if l == 1
                bits_org = [bits_double(l,:)];
```

```
            else
                bits_org = [bits_org, bits_double(l,:)];
            end
        end

        rx = bits_org.';

        rx(rx ~= -1) = 1;
        rx(rx == -1) = 0;

        rx = logical(rx);

        % Calculate the bit error rate (BER).
        % This indicates the relative number of bit errors.
        % Typically this will vary from 0 (no bit errors) to 0.5 (half of all
        % receieved bits are different, which is the number we'd expect if we
        % compare two random bit sequences).
        ber = 1-sum(rx == tx.d)/length(rx);
    end

    function z = frame_interpolate(x,L,hlp)
        % Interpolate (upsample) a signal x by factor L, with an optionally
        % configurable lowpass filter.
        % Arguments:
        %   x    Signal to interpolate, length N
        %   L    Upsampling factor
        %   hlp FIR filter coefficents for lowpass filter, length Nh
        %        If not supplied, a default filter will be used with length
        %        62.
        % Returns:
        %   z    Interpolated signal of length N*L + Nh-1
        %

        if nargin < 3       % Default filter design
            SBscale = 1.7;  % Factor for stop band position
            Nfir = 61;      % The filter length if Nfir + 1
            hlp = firpm(Nfir, [0 1/L 1/L*SBscale 1], [1 1 0 0]);
        end

        % Make x, hlp column vectors
        x = x(:);
        hlp = hlp(:);

        % Get the length of the input signal
        N = length(x);

        % Preallocate vector for upsampled, unfiltered, signal
        zup = zeros((N)*L,1);

        % Upsample by a factor L, i.e. insert L-1 zeros after each original
        % sample
        for l = 1:1:length(zup)
            if l == 1
                zup(l,:) = x(l,:);
            elseif mod(l, L) == 1
                zup(l,:) = x((l-1)/L+1,:);
            elseif mod(l, L) ~= 1
                zup(l,:) = 0;
            end
        end

        % Apply the LP filter to the upsampled (unfiltered) signal.
        z = conv(hlp, zup);
    end

    function z = frame_decimate(x,L,hlp)
        % Decimate (downsample) a signal x by factor L, with an optionally
```

```
        % configurable lowpass filter.
        % Arguments:
        %   x    Signal to decimate, length N
        %   L    Downsampling factor
        %   hlp FIR filter coefficents for lowpass filter, length Nh
        %        If not supplied, a default filter will be used with length
        %        61.
        % Returns:
        %   z    Interpolated signal of length N*L + Nh-1

        if nargin < 3         % Default filter design
            SBscale = 1.7;    % Factor for stop band position
            Nfir = 61;        % The filter length if Nfir + 1
            hlp = firpm(Nfir, [0 1/L 1/L*SBscale 1], [1 1 0 0]);
        end

        % Make x, hlp column vectors
        x = x(:);
        hlp = hlp(:);

        % Apply the lowpass filter to avoid aliasing when decimating
        xf = conv(hlp, x);

        % Downsample by keeping samples [1, 1+L, 1+2*L, ...]
        for l = 1:1:length(xf)
            if l == 1
                z(l,:) = xf(l,:);
            elseif mod(l, L) == 1
                z((l-1)/L+1,:) = xf(l,:);
            end
        end
    end

    function z = frame_modulate(x, theta)
       % Modulates a signal of length N with a modulation frequency theta.
       % Arguments:
       %    x        Signal to modulate of length N
       %    theta    Normalized modulation frequency
       % Outputs:
       %    z        Modulated signal

       % Make x a column vector
       x = x(:);

       N = length(x);

       % Generate vector of sample indices
       n = (0:N-1);
       n = n(:);

       % Modulate x by multiplying the samples with the complex exponential
       % exp(i * 2 * pi * theta * n)
       z = x.*exp(1i*2*pi*theta*n);
    end

    function [rx, evm, ber, symbs] = sim_ofdm_audio_channel(tx, N_cp, snr, sync_err
, f_s, f_c, L)
        % Simulate modulated OFDM signal transmission/reception over an
        % audio channel. This fairly accurately simulates the physical
        % channel of audio between a loudspeaker and a microphone.
        %
        % ---------------------------------------------------------------
        % NOTE: THIS FUNCTION WILL NOT BE SELF-TESTED!
        % It will be up to you to study the output from this function and
        % determine if the results are correct or not.
        % ---------------------------------------------------------------
        %
```

```
        % Arguments:
        %   tx          Structure with fields:
        %     .p        Pilot bits to transmit
        %     .d        Data bits to transmit
        %   N_cp        Cyclic prefix length [samples]
        %   snr         Channel signal/noise ration to apply [dB]
        %   f_s         The up-sampled sampling frequency [Hz]
        %   f_c         The modulation carrier frequency [Hz]
        %   L           The upsampling/downsampling factor [-]
        % Outputs:
        %   rx          Recieved bits [-]
        %   evm         Error vector magnitude (see below) [-]
        %   ber         Bit error rate (see below) [-]
        %   symbs       Structure containing fields:
        %       .tx         Transmitted symbols
        %       .rx_pe      Recieved symbols, pre-equalization
        %       .rx_e       Recieved symbols, post-equalization
        %
        %
        % This function is similar to the unknown-channel problem, but with
        % the added complexity of requiring to interpolate and modulate the
        % signal before transmission, followed by demodulation and
        % decimation on reception. The relevant steps to perform here are:
        %   - Get a sequence of pilot and data bits to transmit
        %   - Convert the pilot and data bits to OFDM symbols
        %   - Create an OFDM block from the OFDM symbols for the pilot and
        %   data
        %   - Add a cyclic prefix to the pilot and data
        %   - Concatenate the pilot and data blocks to create an entire
        %   OFDM frame
        %   - Interpolate the signal to a higher sample-rate
        %   - Modulate the signal, thereby moving it from the base-band to
        %   being centered about the modulation frequency.
        %   - Simulate the transmission and reception over the channel using the
        %   simulate_baseband_channe function.
        %   - Demodulate the signal, moving the recieved signal back to the
        %   base-band
        %   - Decimate the signal, reducing the sample-rate back to the
        %   original rate.
        %   - Split the recieved message into a recieved pilot and data
        %   segment
        %   - Remove the cyclic prefixes from the recieved messages
        %   - Estimate the channel gain from the pilot block
        %   - Equalize the recieved data symbols by the channel gain
        %   - Convert the equalized symbols back to bits
        %   - Compare the recieved bits/symbols to the transmitted bits/symbols.

    warning('Note that this function is _not_ self-tested. It is up to you to study
      the output any verify that it is correct! You can remove this warning if you
      wish.');

        % Ensure input is a column vector
        tx.d = tx.d(:);
        tx.p = tx.p(:);

        tx_alt.d = double(tx.d);

        for i = 1:1:length(tx_alt.d)
            if tx_alt.d(i) == 0
                tx_alt.d(i) = -1;
            else
                tx_alt.d(i) = 1;
            end
        end

        if rem(length(tx_alt.d),2) == 1
            error('bits must be of even length');
```

```
        end

        tx_alt.p = double(tx.p);

        for i = 1:1:length(tx_alt.p)
            if tx_alt.p(i) == 0
                tx_alt.p(i) = -1;
            else
                tx_alt.p(i) = 1;
            end
        end

        if rem(length(tx_alt.p),2) == 1
            error('bits must be of even length');
        end

        % Convert bits to QPSK symbols
        for k = 1:1:length(tx_alt.d)/2
            x.d(k) = sqrt(1/2)*(tx_alt.d(2*k-1)+1i*tx_alt.d(2*k));
        end

        x.d = x.d.';

        for k = 1:1:length(tx_alt.p)/2
            x.p(k) = sqrt(1/2)*(tx_alt.p(2*k-1)+1i*tx_alt.p(2*k));
        end

        x.p = x.p.';

        symbs.tx = x.d;   % Store transmitted data symbols for later

        % Number of symbols in message
        N = length(x.d);
        if length(x.d) ~= length(x.p)
           error('Pilot and data messages must be of equal length');
        end

        % Create OFDM time-domain block using IDFT
        for n = 1:1:N
            for k = 1:1:N
                if k == 1
                    z.d(n) = (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                else
                    z.d(n) = z.d(n) + (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        z.d = z.d.';

        for n = 1:1:N
            for k = 1:1:N
                if k == 1
                    z.p(n) = (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                else
                    z.p(n) = z.p(n) + (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        z.p = z.p.';

        % Add cyclic prefix to create OFDM package
        zcp.d = [z.d(end-N_cp+1:1:end).', z.d.'].';
        zcp.p = [z.p(end-N_cp+1:1:end).', z.p.'].';

        % Concatenate the messages
```

```
        tx_frame = [zcp.p.', zcp.d.'].';

        % Increase the sample rate by interpolation
        SBscale = 1.7;
        Nfir = 61;
        hlp = firpm(Nfir, [0 1/L 1/L*SBscale 1], [1 1 0 0]);

        hlp = hlp(:);

        N = length(tx_frame);

        tx_frame_up = zeros((N)*L,1);

        for l = 1:1:length(tx_frame_up)
            if l == 1
                tx_frame_up(l,:) = tx_frame(l,:);
            elseif mod(l, L) == 1
                tx_frame_up(l,:) = tx_frame((l-1)/L+1,:);
            elseif mod(l, L) ~= 1
                tx_frame_up(l,:) = 0;
            end
        end

        tx_frame_us = conv(hlp, tx_frame_up);

        % Modulate the upsampled signal
        N = length(tx_frame_us);

        n = (0:N-1);
        n = n(:);

        tx_frame_mod = tx_frame_us.*exp(1i*2*pi*n*(f_c/f_s));

        % Discard the imaginary part of the signal for transmission over a
        % scalar channel (simulation of audio over air)
        tx_frame_final = real(tx_frame_mod);

        [PSD_Tx, freq_Tx] = pwelch(tx_frame_mod, 500, 300, 500, f_s);
        figure(4);
        plot(freq_Tx, 10*log10(PSD_Tx));
        xlabel('Frequency [Hz]');
        ylabel('PSD [dB/Hz]');
        grid on

        % Send package over channel
        [rx_frame_raw, rx_idx] = simulate_audio_channel(tx_frame_final, f_s, snr,
    sync_err);

        % Discard data before/after package
        rx_frame_raw = rx_frame_raw(rx_idx:rx_idx + length(tx_frame_final));

        [PSD_Rx, freq_Rx] = pwelch(rx_frame_raw, 500, 300, 500, f_s);
        figure(5);
        plot(freq_Tx, 10*log10(PSD_Tx));
        hold on
        plot(freq_Rx, 10*log10(PSD_Rx));
        xlabel('Frequency [Hz]');
        ylabel('PSD [dB/Hz]');
        grid on
        legend({'Transmitter', 'Receiver'}, 'Location', 'NorthEast');

        % Demodulate to bring the signal back to the baseband
        N = length(rx_frame_raw);

        n = (0:N-1);
        n = n(:);

```

```
        rx_frame_us = rx_frame_raw.*exp(-1i*2*pi*n*(f_c/f_s));

        % Decimate the signal to bring the sample rate back to the original
        rx_frame_us_filt = conv(hlp, rx_frame_us);

        % Downsample by keeping samples [1, 1+L, 1+2*L, ...]
        for l = 1:1:length(rx_frame_us_filt)
            if l == 1
                rx_frame(l,:) = rx_frame_us_filt(l,:);
            elseif mod(l, L) == 1
                rx_frame((l-1)/L+1,:) = rx_frame_us_filt(l,:);
            end
        end

        N = length(x.d);

        % Discard samples beyond OFDM frame
        rx_frame = rx_frame(1:2*(N+N_cp));

        % Split frame into packages
        ycp = struct();
        ycp.d = rx_frame(length(rx_frame)/2+1:1:end);
        ycp.p = rx_frame(1:1:length(rx_frame)/2);

        % Remove cyclic prefix
        y.d = ycp.d(N_cp+1:1:end);
        y.p = ycp.p(N_cp+1:1:end);

        % Convert to frequency domain using DFT
        for k = 1:1:N
            for n = 1:1:N
                if n == 1
                    r.d(k) = y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                else
                    r.d(k) = r.d(k) + y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        r.d = r.d.';

        for k = 1:1:N
            for n = 1:1:N
                if n == 1
                    r.p(k) = y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                else
                    r.p(k) = r.p(k) + y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
                end
            end
        end

        r.p = r.p.';

        symbs.rx_pe = r.d; % Store symbols for later

        % Esimate channel
        H = r.p./x.p;

        % Remove effect of channel on the data package by equalization.
        r_eq = r.d./H;

        symbs.rx_e = r_eq; %Store symbols for later

        % Calculate the quality of the received symbols.
        % The error vector magnitude (EVM) is one useful metric.
        evm = norm(x.d - r_eq)/sqrt(N);
```

```
        % Convert the recieved symsbols to bits
        rx = false(2*length(r_eq),1);

        bits_double = zeros(length(r_eq),2);

        for k = 1:1:length(r_eq)
            if real(r_eq(k)) > 0 && imag(r_eq(k)) > 0
                bits_double(k,:) = [1, 1];
            elseif real(r_eq(k)) < 0 && imag(r_eq(k)) > 0
                bits_double(k,:) = [-1, 1];
            elseif real(r_eq(k)) > 0 && imag(r_eq(k)) < 0
                bits_double(k,:) = [1, -1];
            elseif real(r_eq(k)) < 0 && imag(r_eq(k)) < 0
                bits_double(k,:) = [-1, -1];
            end
        end

        for l = 1:1:length(bits_double)
            if l == 1
                bits_org = [bits_double(l,:)];
            else
                bits_org = [bits_org, bits_double(l,:)];
            end
        end

        rx = bits_org.';

        rx(rx ~= -1) = 1;
        rx(rx == -1) = 0;

        rx = logical(rx);

        % Calculate the bit error rate (BER).
        % This indicates the relative number of bit errors.
        % Typically this will vary from 0 (no bit errors) to 0.5 (half of all
        % receieved bits are different, which is the number we'd expect if we
        % compare two random bit sequences).
        ber = 1-sum(rx == tx.d)/length(rx);
    end



% Generate structure with handles to functions
funs.add_cyclic_prefix = @add_cyclic_prefix;
funs.remove_cyclic_prefix = @remove_cyclic_prefix;
funs.bits2qpsk = @bits2qpsk;
funs.qpsk2bits = @qpsk2bits;
funs.sim_ofdm_known_channel = @sim_ofdm_known_channel;
funs.concat_packages = @concat_packages;
funs.split_frame = @split_frame;
funs.sim_ofdm_unknown_channel = @sim_ofdm_unknown_channel;

funs.frame_interpolate = @frame_interpolate;
funs.frame_decimate = @frame_decimate;
funs.frame_modulate = @frame_modulate;
funs.sim_ofdm_audio_channel = @sim_ofdm_audio_channel;


% This file will return a structure with handles to the functions you have
% implemented. You can call them if you wish, for example:
% funs = student_sols();
% some_output = funs.some_function(some_input);
end
```

The C code is given below:

```
    void ofdm_demodulate(float * pSrc, float * pRe, float * pIm,  float f, int
    length ){
```

```
/*
* Demodulate a real signal (pSrc) into a complex signal (pRe and pPim)
* with modulation center frequency f and length 'length'.
  *
* Note: to avoid getting a false-fail in the self-test routine from
* successive rounding errors, be sure to implement this function using
* something like (using matlab notation):
*
* omega = 0;
* inc = - 2*pi*f;
* for i=1:length
*   z(i) = pSrc(i) * exp(1j*omega);
* omega = omega + inc;
* end
* pRe = real(z);
* pIm = imag(z);
*
* Be sure *NOT* to do something like:
*
* for i=1:length
* omega = -2*pi*f*(i-1);
*   z(i) = pSrc(i) * exp(1j*omega);
* end
* pRe = real(z);
* pIm = imag(z);
*
* The reason for this is that the term 'inc' will be rounded to the nearest
* floating-point number and be kept constant for all interations of the for
* loop. This is in contrast to the second case where the modulation frequency
* might effectively jitter between different values with the lowest rounding
* error. Though the latter gives a more accurate average frequency, we are
* fairly sensistive to frequency jitter as this is equivalent to a change in
* phase.
*
*/
#ifdef MASTER_MODE
#include "../../secret_sauce.h"
DO_OFDM_DEMODULATE();
#else
/* TODO: Add code from here... */

/* ...to here */
#endif
}

void cnvt_re_im_2_cmplx( float * pRe, float * pIm, float * pCmplx, int length ){
/*
* Converts a complex signal represented as pRe + sqrt(-1) * pIm into an
* interleaved real-valued vector of size 2*length
* I.e. pCmplx = [pRe[0], pIm[0], pRe[1], pIm[1], ... , pRe[length-1], pIm[length
    -1]]
*/
#ifdef MASTER_MODE
#include "../../secret_sauce.h"
  DO_OFDM_RE_IM_2_CMPLX();
#else
/* TODO: Add code from here... */

/* ...to here */
#endif
}

void ofdm_conj_equalize(float * prxMes, float * prxPilot,
  float * ptxPilot, float * pEqualized, float * hhat_conj, int length){
/* Generate estimate of the channel and equalize recieved message by
* multiplying by the conjugate of the channel.
*
```

```
 * In this function the channel is estimated by (using matlab notation)
 *  hhat_conj = conj(conj(ptxPilot) .* prxPilot)
 * and the recieved symbols are equalized by
 *  pEqualized = prxMes .* hhat_conj
 * as described in the project PM.
 *
 * INP:
 *  prxMes[] - complex vector with received data message in frequency domain (FD)
 *  prxPilot[] - complex vector with received pilot in FD
 *  ptxPilot[] - complex vector with transmitted pilot in FD
 *  length  - number of complex OFDM symbols
 * OUT:
 *  pEqualized[] - complex vector with equalized data message (Note: only phase
 *  is equalized)
 *  hhat_conj[] -  complex vector with estimated conjugated channel gain
 */
 //Temporary storage array for general-purpose use
 float pTmp[2*length];
#ifdef MASTER_MODE
#include "../../secret_sauce.h"
 DO_OFDM_CONJ_EQUALIZE();
#else
 /* You can use a combination of the functions
 *  arm_cmplx_conj_f32()
 *  arm_cmplx_mult_cmplx_f32()
 * The reference page for these DSP functions can be found here:
 * http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html
 * The array pTmp may be freely used and is long enough to store any complex
 * vector of up to length elements. */

 /* TODO: Add code from here...*/

 /* ...to here */
#endif
}
```