

SSY130 - Applied Signal Processing



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

$$BW = \frac{f_s}{R} = 2 \text{ kHz}. \quad (1)$$

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.

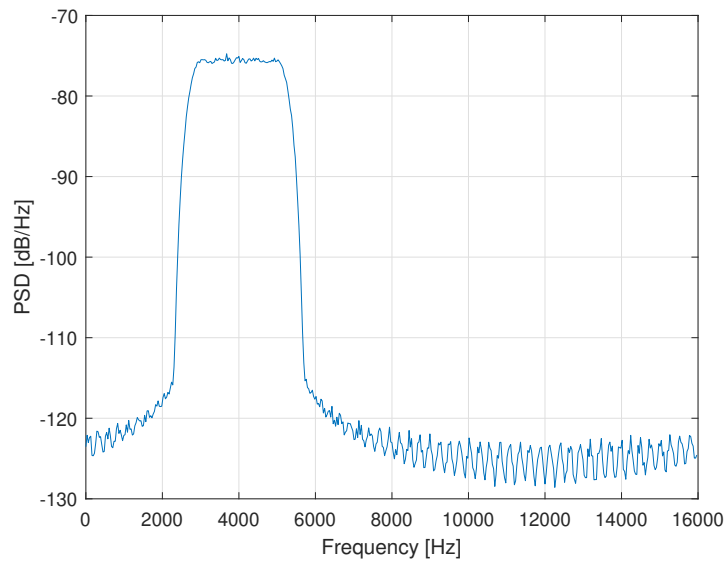


Figure 1: Power-spectral density of the transmitted signal around the carrier frequency.

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.

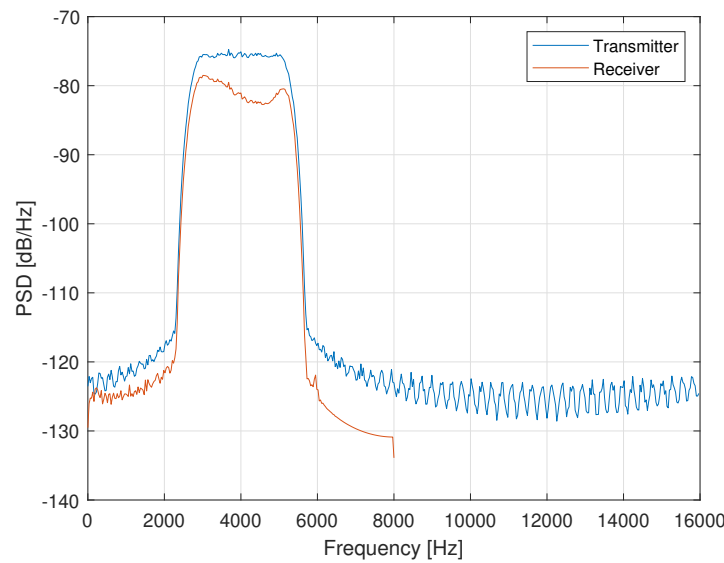


Figure 2: Power-spectral density of the transmitted and received signal around the carrier frequency.

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

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

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

$$y_r = z_r \cdot \cos(2\pi f_c), \quad (3)$$

$$y_i = z_r \cdot \sin(2\pi f_c), \quad (4)$$

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

$$\hat{H} = \bar{T} \cdot R, \quad (5)$$

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

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

and a phase response of

$$\angle \hat{H} = \angle \bar{T} + \angle R = \angle R - \angle T, \quad (7)$$

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

$$|H| = \left| \frac{R}{T} \right|, \quad (8)$$

and the phase response is given by

$$\angle H = \angle \frac{R}{T} = \angle R - \angle T. \quad (9)$$

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

$$R_{eq} = R \cdot \hat{\hat{H}}, \quad (10)$$

where $\hat{\hat{H}}$ 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

$$\angle R_{eq} = \angle R + \angle \hat{\hat{H}}. \quad (11)$$

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

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

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.

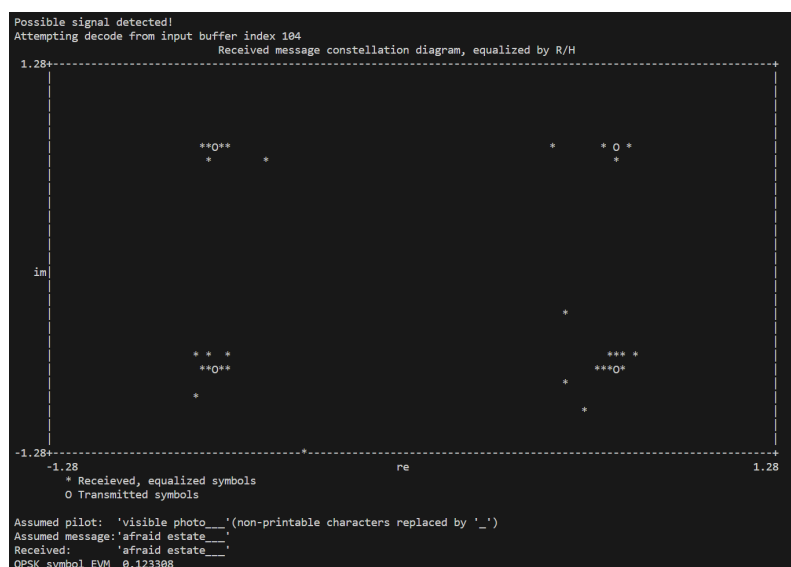


Figure 3: Constellation diagram for high signal amplitude.



Figure 4: Constellation diagram for low signal amplitude.

10 Appendix

The MATLAB code is given below:

```

1 %NO_FILE
2 function [funs, student_id] = student_sols()
3 %STUDENT_SOLS Contains all student solutions to problems.
4
5 % -----
6 %               STEP 1
7 % -----
8 % Set to your birthdate / the birthdate of one member in the group.
9 % Should a numeric value of format YYYYMMDD, e.g.
10 % student_id = 19900101;
11 % This value must be correct in order to generate a valid secret key.
12 student_id = 19990919;
13
14 % -----
15 %               STEP 2
16 % -----
17 % Your task is to implement the following skeleton functions.
18 % You are free to use any of the utility functions located in the same
19 % directory as this file as well as any of the standard matlab functions.
20
21
22
23 function z = add_cyclic_prefix(x,Ncp) %#ok<*INUSD>
24 % Adds (prepends) a Ncp long cyclic prefix to the ofdm block x.
25 x = x(:); %#ok<*NASGU> % Ensure x is a column vector
26 z = [x(end-Ncp+1:1:end).', x.'].';
27 end
28
29 function x = remove_cyclic_prefix(z,Ncp)
30 % Removes a Ncp long cyclic prefix from the ofdm package z
31 z = z(:); % Ensure z is a column vector
32 x = z(Ncp+1:1:end);
33 end
34
35 function symb = bits2qpsk(bits)
36 % Encode bits as qpsk symbols
37 % ARGUMENTS:
38 % bits = array of bits. Numerical values converted as:
39 %   zero -> zero
40 %   nonzero -> one
41 % Must be of even length!
42 % OUTPUT:
43 % x = complex array of qpsk symbols encoding the bits. Will contain
44 % length(bits)/2 elements. Valid output symbols are
45 % 1/sqrt(2)*(+/-1 +/- i). Symbols grouped by pairs of bits, where
46 % the first corresponds the real part of the symbol while the
47 % second corresponds to the imaginary part of the symbol. A zero
48 % bit should be converted to a negative symbol component, while a
49 % nonzero bit should be converted to a positive symbol component.
50
51 % Convert bits vector of +/- 1
52 bits = double(bits);
53 bits = bits(:);
54
55 for i = 1:1:length(bits)
56     if bits(i) == 0
57         bits(i) = -1;
58     else
59         bits(i) = 1;
60     end
61 end
62
63 if rem(length(bits),2) == 1
64     error('bits must be of even length');

```

```

65     end
66
67     for k = 1:1:length(bits)/2
68         symb(k) = sqrt(1/2)*(bits(2*k-1)+1i*bits(2*k));
69     end
70
71     symb = symb.';
72 end
73
74 function bits = qpsk2bits(x)
75     % Convert qpsk symbols to bits.
76     % Output will be a vector twice as long as the input x, with values
77     % 0 or 1.
78     x = x(:);
79     bits = false(2*length(x),1);
80     % Note: you only need to check which quadrant of the complex plane
81     % the symbol lies in in order to map it to a pair of bits. The
82     % first bit corresponds to the real part of the symbol while the
83     % second bit corresponds to the imaginary part of the symbol.
84
85     bits_double = zeros(length(x),2);
86
87     for k = 1:1:length(x)
88         if real(x(k)) > 0 && imag(x(k)) > 0
89             bits_double(k,:) = [1, 1];
90         elseif real(x(k)) < 0 && imag(x(k)) > 0
91             bits_double(k,:) = [-1, 1];
92         elseif real(x(k)) > 0 && imag(x(k)) < 0
93             bits_double(k,:) = [1, -1];
94         elseif real(x(k)) < 0 && imag(x(k)) < 0
95             bits_double(k,:) = [-1, -1];
96         end
97     end
98
99     for l = 1:1:length(bits_double)
100         if l == 1
101             bits_org = [bits_double(l,:)];
102         else
103             bits_org = [bits_org, bits_double(l,:)];
104         end
105     end
106
107     bits = bits_org.';
108
109     bits(bits ~= -1) = 1;
110     bits(bits == -1) = 0;
111
112     % Ensure output is of correct type
113     % zero value -> logical zero
114     % nonzero value -> logical one
115     bits = logical(bits);
116 end
117
118 function [rx, evm, ber, syms] = sim_ofdm_known_channel(tx, h, N_cp, snr,
119 sync_err)
119     % Simulate OFDM signal transmission/reception over a known channel.
120     %
121     % -----
122     % NOTE: THIS FUNCTION WILL NOT BE SELF-TESTED!
123     % It will be up to you to study the output from this function and
124     % determine if the results are correct or not.
125     % -----
126     %
127     % Arguments:
128     % tx          Bits to transmit [-]
129     % h           Channel impulse response [-]
130     % N_cp        Cyclic prefix length [samples]

```

```

131 % snr          Channel signal/noise ration to apply [dB]
132 % sync_err    Reciever synchronization error [samples]
133 % Outputs:
134 % rx          Recieved bits [-]
135 % evm         Error vector magnitude (see below) [-]
136 % ber         Bit error rate (see below) [-]
137 % syms        Structure containing fields:
138 %             .tx          Transmitted symbols
139 %             .rx_pe       Recieved symbols, pre-equalization
140 %             .rx_e        Recieved symbols, post-equalization
141 %
142 % In this function, you will now fully implement a simulated
143 % base-band OFDM communication scheme. The relevant steps in this
144 % are:
145 %   - Get a sequence of bits to transmit
146 %   - Convert the bits to OFDM symbols
147 %   - Create an OFDM block from the OFDM symbols
148 %   - Add a cyclic prefix
149 %   - Simulate the transmission and reception over the channel using the
150 %     simulate_baseband_channel function.
151 %   - Remove the cyclic prefix from the recieved message.
152 %   - Equalize the recieved symbols by the channel gain
153 %   - Convert the equalized symbols back to bits
154 %   - Compare the recieved bits/symbols to the transmitted bits/symbols.
155 %
156 % If you have implemented the skeleton functions earlier in this
157 % file then this function will be very simple as you can call your
158 % functions to perform the needed tasks.
159
160 warning('Note that this function is _not_ self-tested. It is up to you to study
161         the output any verify that it is correct! You can remove this warning if you
162         wish. ');
163
164 % Ensure inputs are column vectors
165 tx = tx(:);
166 h = h(:);
167
168 tx_alt = double(tx);
169
170 for i = 1:length(tx_alt)
171     if tx_alt(i) == 0
172         tx_alt(i) = -1;
173     else
174         tx_alt(i) = 1;
175     end
176 end
177
178 if rem(length(tx_alt),2) == 1
179     error('bits must be of even length');
180 end
181
182 % Convert bits to QPSK symbols
183 for k = 1:length(tx_alt)/2
184     x(k) = sqrt(1/2)*(tx_alt(2*k-1)+1i*tx_alt(2*k));
185 end
186
187 x = x.';
188
189 syms.tx = x; % Store transmitted symbols for later
190
191 % Number of symbols in message
192 N = length(x);
193
194 % Create OFDM time-domain block using IDFT
195 for n = 1:1:N
196     for k = 1:1:N
197         if k == 1

```

```

196         z(n) = (1/N).*x(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
197     else
198         z(n) = z(n) + (1/N).*x(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
199     end
200 end
201 end
202
203 z = z.';
204
205 % Add cyclic prefix to create OFDM package
206 zcp = [z(end-N_cp+1:1:end).', z.'].';
207
208 % Send package over channel
209 ycp = simulate_baseband_channel(zcp, h, snr, sync_err);
210 % Only keep the first N+Ncp recieved samples. Consider why ycp is longer
211 % than zcp, and why we only need to save the first N+Ncp samples. This is
212 % important to understand.
213 ycp = ycp(1:N+N_cp);
214
215 % Remove cyclic prefix
216 y = ycp(N_cp+1:1:end);
217
218 % Convert to frequency domain using DFT
219 for k = 1:1:N
220     for n = 1:1:N
221         if n == 1
222             r(k) = y(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
223         else
224             r(k) = r(k) + y(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
225         end
226     end
227 end
228
229 r = r.';
230
231 syms.rx_pe = r; % Store symbols for later
232
233 % Remove effect of channel by equalization. Here, we can do this by
234 % dividing r (which is in the frequency domain) by the channel gain (also
235 % in the frequency domain).
236 Nh = length(h);
237
238 for k = 1:1:N
239     for n = 1:1:Nh
240         if n == 1
241             H(k) = h(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
242         else
243             H(k) = H(k) + h(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
244         end
245     end
246 end
247
248 H = H.';
249
250 r_eq = r./H;
251
252 syms.rx_e = r_eq; %Store symbols for later
253
254 % Calculate the quality of the received symbols.
255 % The error vector magnitude (EVM) is one useful metric.
256 evm = norm(x - r_eq)/sqrt(N);
257
258 % Convert the recieved symsbols to bits
259 rx = false(2*length(r_eq),1);
260
261 bits_double = zeros(length(r_eq),2);
262

```

```

263     for k = 1:length(r_eq)
264         if real(r_eq(k)) > 0 && imag(r_eq(k)) > 0
265             bits_double(k,:) = [1, 1];
266         elseif real(r_eq(k)) < 0 && imag(r_eq(k)) > 0
267             bits_double(k,:) = [-1, 1];
268         elseif real(r_eq(k)) > 0 && imag(r_eq(k)) < 0
269             bits_double(k,:) = [1, -1];
270         elseif real(r_eq(k)) < 0 && imag(r_eq(k)) < 0
271             bits_double(k,:) = [-1, -1];
272         end
273     end
274
275     for l = 1:length(bits_double)
276         if l == 1
277             bits_org = [bits_double(l,:)];
278         else
279             bits_org = [bits_org, bits_double(l,:)];
280         end
281     end
282
283     rx = bits_org.';
284
285     rx(rx ~= -1) = 1;
286     rx(rx == -1) = 0;
287
288     rx = logical(rx);
289
290     % Calculate the bit error rate (BER).
291     % This indicates the relative number of bit errors.
292     % Typically this will vary from 0 (no bit errors) to 0.5 (half of all
293     % received bits are different, which is the number we'd expect if we
294     % compare two random bit sequences).
295     ber = 1-sum(rx == tx)/length(rx);
296 end
297
298 function txFrame = concat_packages(txPilot,txData)
299     % Concatenate two ofdm blocks of equal size into a frame
300     txPilot = txPilot(:);
301     txData = txData(:);
302     if(length(txData) ~= length(txPilot))
303         error('Pilot and data are not of the same length!');
304     end
305     txFrame = [txPilot.', txData.'].';
306 end
307
308 function [rxPilot, rxData] = split_frame(rxFrame)
309     % Split an ofdm frame into 2 equal ofdm packages
310     rxFrame = rxFrame(:);
311     if rem(length(rxFrame),2) > 0
312         error('Vector z must have an even number of elements');
313     end
314     N = length(rxFrame);
315     rxPilot = rxFrame(1:1:N/2);
316     rxData = rxFrame(N/2+1:1:end);
317 end
318
319 function [rx, evm, ber, syms] = sim_ofdm_unknown_channel(tx, h, N_cp, snr,
sync_err)
320     % Simulate OFDM signal transmission/reception over an unknown
321     % channel.
322     %
323     % -----
324     % NOTE: THIS FUNCTION WILL NOT BE SELF-TESTED!
325     % It will be up to you to study the output from this function and
326     % determine if the results are correct or not.
327     % -----
328     %

```

```

329 % Arguments:
330 % tx      Structure with fields:
331 % .p      Pilot bits to transmit
332 % .d      Data bits to transmit
333 % h      Channel impulse response [-]
334 % N_cp    Cyclic prefix length [samples]
335 % snr     Channel signal/noise ration to apply [dB]
336 % sync_err Reciever synchronization error [samples]
337 % Outputs:
338 % rx      Recieved bits [-]
339 % evm     Error vector magnitude (see below) [-]
340 % ber     Bit error rate (see below) [-]
341 % syms    Structure containing fields:
342 % .tx      Transmitted symbols
343 % .rx_pe   Recieved symbols, pre-equalization
344 % .rx_e    Recieved symbols, post-equalization
345 %
346 %
347 % This function is similar to the known-channel problem, but with
348 % the added complexity of requiring to estimate the channel
349 % response. The relevant steps to perform here are:
350 % - Get a sequence of pilot and data bits to transmit
351 % - Convert the pilot and data bits to OFDM symbols
352 % - Create an OFDM block from the OFDM symbols for the pilot and
353 % data
354 % - Add a cyclic prefix to the pilot and data
355 % - Concatenate the pilot and data blocks to create an entire
356 % OFDM frame
357 % - Simulate the transmission and reception over the channel using the
358 % simulate_baseband_channe function.
359 % - Split the recieved message into a recieved pilot and data
360 % segment
361 % - Remove the cyclic prefixes from the recieved messages
362 % - Estimate the channel gain from the pilot block
363 % - Equalize the recieved data symbols by the channel gain
364 % - Convert the equalized symbols back to bits
365 % - Compare the recieved bits/symbols to the transmitted bits/symbols.
366
367 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.');
```

```

368
369 % Ensure inputs are column vectors
370 tx.d = tx.d(:);
371 tx.p = tx.p(:);
372 h = h(:);
373
374 tx_alt.d = double(tx.d);
375
376 for i = 1:length(tx_alt.d)
377     if tx_alt.d(i) == 0
378         tx_alt.d(i) = -1;
379     else
380         tx_alt.d(i) = 1;
381     end
382 end
383
384 if rem(length(tx_alt.d),2) == 1
385     error('bits must be of even length');
386 end
387
388 tx_alt.p = double(tx.p);
389
390 for i = 1:length(tx_alt.p)
391     if tx_alt.p(i) == 0
392         tx_alt.p(i) = -1;
393     else
```

```

394         tx_alt.p(i) = 1;
395     end
396 end
397
398 if rem(length(tx_alt.p),2) == 1
399     error('bits must be of even length');
400 end
401
402 % Convert bits to QPSK symbols
403 for k = 1:1:length(tx_alt.d)/2
404     x.d(k) = sqrt(1/2)*(tx_alt.d(2*k-1)+1i*tx_alt.d(2*k));
405 end
406
407 x.d = x.d.';
408
409 for k = 1:1:length(tx_alt.p)/2
410     x.p(k) = sqrt(1/2)*(tx_alt.p(2*k-1)+1i*tx_alt.p(2*k));
411 end
412
413 x.p = x.p.';
414
415 syms.tx = x.d; % Store transmitted data symbols for later
416
417 % Number of symbols in message
418 N = length(x.d);
419 if length(x.d) ~= length(x.p)
420     error('Pilot and data messages must be of equal length');
421 end
422
423 % Create OFDM time-domain block using IDFT
424 for n = 1:1:N
425     for k = 1:1:N
426         if k == 1
427             z.d(n) = (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
428         else
429             z.d(n) = z.d(n) + (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
430         end
431     end
432 end
433
434 z.d = z.d.';
435
436 for n = 1:1:N
437     for k = 1:1:N
438         if k == 1
439             z.p(n) = (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
440         else
441             z.p(n) = z.p(n) + (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
442         end
443     end
444 end
445
446 z.p = z.p.';
447
448 % Add cyclic prefix to create OFDM package
449 zcp.d = [z.d(end-N_cp+1:1:end).', z.d.'].';
450 zcp.p = [z.p(end-N_cp+1:1:end).', z.p.'].';
451
452 % Concatenate the messages
453 tx_frame = [zcp.p.', zcp.d.'].';
454
455 % Send package over channel
456 rx_frame = simulate_baseband_channel(tx_frame, h, snr, sync_err);
457 % As before, only keep the first samples
458 rx_frame = rx_frame(1:2*(N+N_cp));
459
460 % Split frame into packages

```



```

461 ycp = struct();
462 ycp.d = rx_frame(length(rx_frame)/2+1:1:end);
463 ycp.p = rx_frame(1:1:length(rx_frame)/2);
464
465 % Remove cyclic prefix
466 y.d = ycp.d(N_cp+1:1:end);
467 y.p = ycp.p(N_cp+1:1:end);
468
469 % Convert to frequency domain using DFT
470 for k = 1:1:N
471     for n = 1:1:N
472         if n == 1
473             r.d(k) = y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
474         else
475             r.d(k) = r.d(k) + y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
476         end
477     end
478 end
479
480 r.d = r.d.';
481
482 for k = 1:1:N
483     for n = 1:1:N
484         if n == 1
485             r.p(k) = y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
486         else
487             r.p(k) = r.p(k) + y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
488         end
489     end
490 end
491
492 r.p = r.p.';
493
494 syms.rx_pe = r.d; % Store symbols for later
495
496 % Estimate channel
497 H = r.p./x.p;
498
499 % Remove effect of channel on the data package by equalization.
500 r_eq = r.d./H;
501
502 syms.rx_e = r_eq; %Store symbols for later
503
504 % Calculate the quality of the received symbols.
505 % The error vector magnitude (EVM) is one useful metric.
506 evm = norm(x.d - r_eq)/sqrt(N);
507
508 % Convert the recieved symsbols to bits
509 rx = false(2*length(r_eq),1);
510
511 bits_double = zeros(length(r_eq),2);
512
513 for k = 1:length(r_eq)
514     if real(r_eq(k)) > 0 && imag(r_eq(k)) > 0
515         bits_double(k,:) = [1, 1];
516     elseif real(r_eq(k)) < 0 && imag(r_eq(k)) > 0
517         bits_double(k,:) = [-1, 1];
518     elseif real(r_eq(k)) > 0 && imag(r_eq(k)) < 0
519         bits_double(k,:) = [1, -1];
520     elseif real(r_eq(k)) < 0 && imag(r_eq(k)) < 0
521         bits_double(k,:) = [-1, -1];
522     end
523 end
524
525 for l = 1:length(bits_double)
526     if l == 1
527         bits_org = [bits_double(l,:)];

```

```

528         else
529             bits_org = [bits_org , bits_double(1,:)];
530         end
531     end
532
533     rx = bits_org.';
534
535     rx(rx ~= -1) = 1;
536     rx(rx == -1) = 0;
537
538     rx = logical(rx);
539
540     % Calculate the bit error rate (BER).
541     % This indicates the relative number of bit errors.
542     % Typically this will vary from 0 (no bit errors) to 0.5 (half of all
543     % received bits are different, which is the number we'd expect if we
544     % compare two random bit sequences).
545     ber = 1-sum(rx == tx.d)/length(rx);
546 end
547
548 function z = frame_interpolate(x,L,hlp)
549     % Interpolate (upsample) a signal x by factor L, with an optionally
550     % configurable lowpass filter.
551     % Arguments:
552     % x    Signal to interpolate, length N
553     % L    Upsampling factor
554     % hlp  FIR filter coefficients for lowpass filter, length Nh
555     %      If not supplied, a default filter will be used with length
556     %      62.
557     % Returns:
558     % z    Interpolated signal of length N*L + Nh-1
559     %
560
561     if nargin < 3        % Default filter design
562         SBscale = 1.7; % Factor for stop band position
563         Nfir = 61;      % The filter length if Nfir + 1
564         hlp = firpm(Nfir, [0 1/L 1/L*SBscale 1], [1 1 0 0]);
565     end
566
567     % Make x, hlp column vectors
568     x = x(:);
569     hlp = hlp(:);
570
571     % Get the length of the input signal
572     N = length(x);
573
574     % Preallocate vector for upsampled, unfiltered, signal
575     zup = zeros((N)*L,1);
576
577     % Upsample by a factor L, i.e. insert L-1 zeros after each original
578     % sample
579     for l = 1:L:length(zup)
580         if l == 1
581             zup(l,:) = x(1,:);
582         elseif mod(l, L) == 1
583             zup(l,:) = x((l-1)/L+1,:);
584         elseif mod(l, L) ~= 1
585             zup(l,:) = 0;
586         end
587     end
588
589     % Apply the LP filter to the upsampled (unfiltered) signal.
590     z = conv(hlp, zup);
591 end
592
593 function z = frame_decimate(x,L,hlp)
594     % Decimate (downsample) a signal x by factor L, with an optionally

```

```

595 % configurable lowpass filter .
596 % Arguments:
597 %   x   Signal to decimate, length N
598 %   L   Downsampling factor
599 %   hlp  FIR filter coefficients for lowpass filter , length Nh
600 %       If not supplied, a default filter will be used with length
601 %       61.
602 % Returns:
603 %   z   Interpolated signal of length N*L + Nh-1
604
605 if nargin < 3      % Default filter design
606     SBscale = 1.7; % Factor for stop band position
607     Nfir = 61;    % The filter length if Nfir + 1
608     hlp = firpm(Nfir, [0 1/L 1/L*SBscale 1], [1 1 0 0]);
609 end
610
611 % Make x, hlp column vectors
612 x = x(:);
613 hlp = hlp(:);
614
615 % Apply the lowpass filter to avoid aliasing when decimating
616 xf = conv(hlp, x);
617
618 % Downsample by keeping samples [1, 1+L, 1+2*L, ...]
619 for l = 1:length(xf)
620     if l == 1
621         z(l,:) = xf(l,:);
622     elseif mod(l, L) == 1
623         z((l-1)/L+1,:) = xf(l,:);
624     end
625 end
626 end
627
628 function z = frame_modulate(x, theta)
629 % Modulates a signal of length N with a modulation frequency theta.
630 % Arguments:
631 %   x       Signal to modulate of length N
632 %   theta   Normalized modulation frequency
633 % Outputs:
634 %   z       Modulated signal
635
636 % Make x a column vector
637 x = x(:);
638
639 N = length(x);
640
641 % Generate vector of sample indices
642 n = (0:N-1);
643 n = n(:);
644
645 % Modulate x by multiplying the samples with the complex exponential
646 % exp(i * 2 * pi * theta * n)
647 z = x.*exp(1i*2*pi*theta*n);
648 end
649
650 function [rx, evm, ber, syms] = sim_ofdm_audio_channel(tx, N_cp, snr, sync_err
, f_s, f_c, L)
651 % Simulate modulated OFDM signal transmission/reception over an
652 % audio channel. This fairly accurately simulates the physical
653 % channel of audio between a loudspeaker and a microphone.
654 %
655 %
656 % NOTE: THIS FUNCTION WILL NOT BE SELF-TESTED!
657 % It will be up to you to study the output from this function and
658 % determine if the results are correct or not.
659 %
660 %

```

```

661 % Arguments:
662 %   tx          Structure with fields:
663 %   .p          Pilot bits to transmit
664 %   .d          Data bits to transmit
665 %   N_cp        Cyclic prefix length [samples]
666 %   snr         Channel signal/noise ration to apply [dB]
667 %   f_s         The up-sampled sampling frequency [Hz]
668 %   f_c         The modulation carrier frequency [Hz]
669 %   L           The upsampling/downsampling factor [-]
670 % Outputs:
671 %   rx          Recieved bits [-]
672 %   evm         Error vector magnitude (see below) [-]
673 %   ber         Bit error rate (see below) [-]
674 %   syms        Structure containing fields:
675 %   .tx         Transmitted symbols
676 %   .rx_pe      Recieved symbols, pre-equalization
677 %   .rx_e       Recieved symbols, post-equalization
678 %
679 %
680 % This function is similar to the unknown-channel problem, but with
681 % the added complexity of requiring to interpolate and modulate the
682 % signal before transmission, followed by demodulation and
683 % decimation on reception. The relevant steps to perform here are:
684 % - Get a sequence of pilot and data bits to transmit
685 % - Convert the pilot and data bits to OFDM symbols
686 % - Create an OFDM block from the OFDM symbols for the pilot and
687 %   data
688 % - Add a cyclic prefix to the pilot and data
689 % - Concatenate the pilot and data blocks to create an entire
690 % OFDM frame
691 % - Interpolate the signal to a higher sample-rate
692 % - Modulate the signal, thereby moving it from the base-band to
693 %   being centered about the modulation frequency.
694 % - Simulate the transmission and reception over the channel using the
695 %   simulate_baseband_channel function.
696 % - Demodulate the signal, moving the recieved signal back to the
697 %   base-band
698 % - Decimate the signal, reducing the sample-rate back to the
699 %   original rate.
700 % - Split the recieved message into a recieved pilot and data
701 %   segment
702 % - Remove the cyclic prefixes from the recieved messages
703 % - Estimate the channel gain from the pilot block
704 % - Equalize the recieved data symbols by the channel gain
705 % - Convert the equalized symbols back to bits
706 % - Compare the recieved bits/symbols to the transmitted bits/symbols.
707
708 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.');
```

```

709
710 % Ensure input is a column vector
711 tx.d = tx.d(:);
712 tx.p = tx.p(:);
713
714 tx_alt.d = double(tx.d);
715
716 for i = 1:length(tx_alt.d)
717     if tx_alt.d(i) == 0
718         tx_alt.d(i) = -1;
719     else
720         tx_alt.d(i) = 1;
721     end
722 end
723
724 if rem(length(tx_alt.d),2) == 1
725     error('bits must be of even length');
```

```

726     end
727
728     tx_alt.p = double(tx.p);
729
730     for i = 1:1:length(tx_alt.p)
731         if tx_alt.p(i) == 0
732             tx_alt.p(i) = -1;
733         else
734             tx_alt.p(i) = 1;
735         end
736     end
737
738     if rem(length(tx_alt.p),2) == 1
739         error('bits must be of even length');
740     end
741
742     % Convert bits to QPSK symbols
743     for k = 1:1:length(tx_alt.d)/2
744         x.d(k) = sqrt(1/2)*(tx_alt.d(2*k-1)+1i*tx_alt.d(2*k));
745     end
746
747     x.d = x.d.';
748
749     for k = 1:1:length(tx_alt.p)/2
750         x.p(k) = sqrt(1/2)*(tx_alt.p(2*k-1)+1i*tx_alt.p(2*k));
751     end
752
753     x.p = x.p.';
754
755     syms.tx = x.d;    % Store transmitted data symbols for later
756
757     % Number of symbols in message
758     N = length(x.d);
759     if length(x.d) ~= length(x.p)
760         error('Pilot and data messages must be of equal length');
761     end
762
763     % Create OFDM time-domain block using IDFT
764     for n = 1:1:N
765         for k = 1:1:N
766             if k == 1
767                 z.d(n) = (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
768             else
769                 z.d(n) = z.d(n) + (1/N).*x.d(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
770             end
771         end
772     end
773
774     z.d = z.d.';
775
776     for n = 1:1:N
777         for k = 1:1:N
778             if k == 1
779                 z.p(n) = (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
780             else
781                 z.p(n) = z.p(n) + (1/N).*x.p(k).*exp(1i*2*pi*(k-1)*(n-1)/N);
782             end
783         end
784     end
785
786     z.p = z.p.';
787
788     % Add cyclic prefix to create OFDM package
789     zcp.d = [z.d(end-N_cp+1:end).', z.d.'].';
790     zcp.p = [z.p(end-N_cp+1:end).', z.p.'].';
791
792     % Concatenate the messages

```

```

793 tx_frame = [zcp.p.', zcp.d.'].';
794
795 % Increase the sample rate by interpolation
796 SBscale = 1.7;
797 Nfir = 61;
798 hlp = firpm(Nfir, [0 1/L 1/L*SBscale 1], [1 1 0 0]);
799
800 hlp = hlp(:);
801
802 N = length(tx_frame);
803
804 tx_frame_up = zeros((N)*L,1);
805
806 for l = 1:L:length(tx_frame_up)
807     if l == 1
808         tx_frame_up(l,:) = tx_frame(1,:);
809     elseif mod(l, L) == 1
810         tx_frame_up(l,:) = tx_frame((l-1)/L+1,:);
811     elseif mod(l, L) ~= 1
812         tx_frame_up(l,:) = 0;
813     end
814 end
815
816 tx_frame_us = conv(hlp, tx_frame_up);
817
818 % Modulate the upsampled signal
819 N = length(tx_frame_us);
820
821 n = (0:N-1);
822 n = n(:);
823
824 tx_frame_mod = tx_frame_us.*exp(1i*2*pi*n*(f_c/f_s));
825
826 % Discard the imaginary part of the signal for transmission over a
827 % scalar channel (simulation of audio over air)
828 tx_frame_final = real(tx_frame_mod);
829
830 [PSD_Tx, freq_Tx] = pwelch(tx_frame_mod, 500, 300, 500, f_s);
831 figure(4);
832 plot(freq_Tx, 10*log10(PSD_Tx));
833 xlabel('Frequency [Hz]');
834 ylabel('PSD [dB/Hz]');
835 grid on
836
837 % Send package over channel
838 [rx_frame_raw, rx_idx] = simulate_audio_channel(tx_frame_final, f_s, snr,
839 sync_err);
840
841 % Discard data before/after package
842 rx_frame_raw = rx_frame_raw(rx_idx:rx_idx + length(tx_frame_final));
843
844 [PSD_Rx, freq_Rx] = pwelch(rx_frame_raw, 500, 300, 500, f_s);
845 figure(5);
846 plot(freq_Tx, 10*log10(PSD_Tx));
847 hold on
848 plot(freq_Rx, 10*log10(PSD_Rx));
849 xlabel('Frequency [Hz]');
850 ylabel('PSD [dB/Hz]');
851 grid on
852 legend({'Transmitter', 'Receiver'}, 'Location', 'NorthEast');
853
854 % Demodulate to bring the signal back to the baseband
855 N = length(rx_frame_raw);
856
857 n = (0:N-1);
858 n = n(:);

```

```

859 rx_frame_us = rx_frame_raw.*exp(-1i*2*pi*n*(f_c/f_s));
860
861 % Decimate the signal to bring the sample rate back to the original
862 rx_frame_us_filt = conv(hlp, rx_frame_us);
863
864 % Downsample by keeping samples [1, 1+L, 1+2*L, ...]
865 for l = 1:1:length(rx_frame_us_filt)
866     if l == 1
867         rx_frame(l,:) = rx_frame_us_filt(l,:);
868     elseif mod(l, L) == 1
869         rx_frame((l-1)/L+1,:) = rx_frame_us_filt(l,:);
870     end
871 end
872
873 N = length(x.d);
874
875 % Discard samples beyond OFDM frame
876 rx_frame = rx_frame(1:2*(N+N_cp));
877
878 % Split frame into packages
879 ycp = struct();
880 ycp.d = rx_frame(length(rx_frame)/2+1:end);
881 ycp.p = rx_frame(1:1:length(rx_frame)/2);
882
883 % Remove cyclic prefix
884 y.d = ycp.d(N_cp+1:end);
885 y.p = ycp.p(N_cp+1:end);
886
887 % Convert to frequency domain using DFT
888 for k = 1:1:N
889     for n = 1:1:N
890         if n == 1
891             r.d(k) = y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
892         else
893             r.d(k) = r.d(k) + y.d(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
894         end
895     end
896 end
897
898 r.d = r.d.';
899
900 for k = 1:1:N
901     for n = 1:1:N
902         if n == 1
903             r.p(k) = y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
904         else
905             r.p(k) = r.p(k) + y.p(n).*exp(-1i*2*pi*(k-1)*(n-1)/N);
906         end
907     end
908 end
909
910 r.p = r.p.';
911
912 syms.rx_pe = r.d; % Store symbols for later
913
914 % Estimate channel
915 H = r.p./x.p;
916
917 % Remove effect of channel on the data package by equalization.
918 r_eq = r.d./H;
919
920 syms.rx_e = r_eq; %Store symbols for later
921
922 % Calculate the quality of the received symbols.
923 % The error vector magnitude (EVM) is one useful metric.
924 evm = norm(x.d - r_eq)/sqrt(N);
925

```

```

926 % Convert the recieved symsbols to bits
927 rx = false(2*length(r_eq),1);
928
929 bits_double = zeros(length(r_eq),2);
930
931 for k = 1:length(r_eq)
932     if real(r_eq(k)) > 0 && imag(r_eq(k)) > 0
933         bits_double(k,:) = [1, 1];
934     elseif real(r_eq(k)) < 0 && imag(r_eq(k)) > 0
935         bits_double(k,:) = [-1, 1];
936     elseif real(r_eq(k)) > 0 && imag(r_eq(k)) < 0
937         bits_double(k,:) = [1, -1];
938     elseif real(r_eq(k)) < 0 && imag(r_eq(k)) < 0
939         bits_double(k,:) = [-1, -1];
940     end
941 end
942
943 for l = 1:length(bits_double)
944     if l == 1
945         bits_org = [bits_double(l,:)];
946     else
947         bits_org = [bits_org, bits_double(l,:)];
948     end
949 end
950
951 rx = bits_org.';
952
953 rx(rx ~= -1) = 1;
954 rx(rx == -1) = 0;
955
956 rx = logical(rx);
957
958 % Calculate the bit error rate (BER).
959 % This indicates the relative number of bit errors.
960 % Typically this will vary from 0 (no bit errors) to 0.5 (half of all
961 % recieved bits are different, which is the number we'd expect if we
962 % compare two random bit sequences).
963 ber = 1-sum(rx == tx.d)/length(rx);
964 end
965
966
967 % Generate structure with handles to functions
968 funs.add_cyclic_prefix = @add_cyclic_prefix;
969 funs.remove_cyclic_prefix = @remove_cyclic_prefix;
970 funs.bits2qpsk = @bits2qpsk;
971 funs.qpsk2bits = @qpsk2bits;
972 funs.sim_ofdm_known_channel = @sim_ofdm_known_channel;
973 funs.concat_packages = @concat_packages;
974 funs.split_frame = @split_frame;
975 funs.sim_ofdm_unknown_channel = @sim_ofdm_unknown_channel;
976
977 funs.frame_interpolate = @frame_interpolate;
978 funs.frame_decimate = @frame_decimate;
979 funs.frame_modulate = @frame_modulate;
980 funs.sim_ofdm_audio_channel = @sim_ofdm_audio_channel;
981
982
983 % This file will return a structure with handles to the functions you have
984 % implemented. You can call them if you wish, for example:
985 % funs = student_sols();
986 % some_output = funs.some_function(some_input);
987
988 end

```

The C code is given below:

```

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

```



```

2  /*
3  * Demodulate a real signal (pSrc) into a complex signal (pRe and pIm)
4  * with modulation center frequency f and length 'length'.
5  *
6  * Note: to avoid getting a false-fail in the self-test routine from
7  * successive rounding errors, be sure to implement this function using
8  * something like (using matlab notation):
9  *
10 * omega = 0;
11 * inc = - 2*pi*f;
12 * for i=1:length
13 *     z(i) = pSrc(i) * exp(1j*omega);
14 *     omega = omega + inc;
15 * end
16 * pRe = real(z);
17 * pIm = imag(z);
18 *
19 * Be sure *NOT* to do something like:
20 *
21 * for i=1:length
22 *     omega = -2*pi*f*(i-1);
23 *     z(i) = pSrc(i) * exp(1j*omega);
24 * end
25 * pRe = real(z);
26 * pIm = imag(z);
27 *
28 * The reason for this is that the term 'inc' will be rounded to the nearest
29 * floating-point number and be kept constant for all iterations of the for
30 * loop. This is in contrast to the second case where the modulation frequency
31 * might effectively jitter between different values with the lowest rounding
32 * error. Though the latter gives a more accurate average frequency, we are
33 * fairly sensitive to frequency jitter as this is equivalent to a change in
34 * phase.
35 *
36 */
37 #ifndef MASTERMODE
38 #include "../secret_sauce.h"
39 DO_OFDM_DEMODULATE();
40 #else
41 /* TODO: Add code from here... */
42
43 /* ...to here */
44 #endif
45 }
46
47 void cnvt_re_im_2_cmplx( float * pRe, float * pIm, float * pCmplx, int length ){
48 /*
49 * Converts a complex signal represented as pRe + sqrt(-1) * pIm into an
50 * interleaved real-valued vector of size 2*length
51 * I.e. pCmplx = [pRe[0], pIm[0], pRe[1], pIm[1], ... , pRe[length-1], pIm[length
52 * -1]]
53 */
54 #ifndef MASTERMODE
55 #include "../secret_sauce.h"
56 DO_OFDM_RE_IM_2_CMPLX();
57 #else
58 /* TODO: Add code from here... */
59
60 /* ...to here */
61 #endif
62 }
63
64 void ofdm_conj_equalize(float * prxMes, float * prxPilot,
65 float * ptxPilot, float * pEqualized, float * hhat_conj, int length){
66 /* Generate estimate of the channel and equalize recieved message by
67 * multiplying by the conjugate of the channel.
68 */

```

```
68 * In this function the channel is estimated by (using matlab notation)
69 * hhat_conj = conj(conj(ptxPilot) .* prxPilot)
70 * and the recieved symbols are equalized by
71 * pEqualized = prxMes .* hhat_conj
72 * as described in the project PM.
73 *
74 * INP:
75 * prxMes[] – complex vector with received data message in frequency domain (FD)
76 * prxPilot[] – complex vector with received pilot in FD
77 * ptxPilot[] – complex vector with transmitted pilot in FD
78 * length – number of complex OFDM symbols
79 * OUT:
80 * pEqualized[] – complex vector with equalized data message (Note: only phase
81 * is equalized)
82 * hhat_conj[] – complex vector with estimated conjugated channel gain
83 */
84 //Temporary storage array for general-purpose use
85 float pTmp[2*length];
86 #ifdef MASTERMODE
87 #include "../secret_sauce.h"
88 DO_OFDM_CONJ_EQUALIZE();
89 #else
90 /* You can use a combination of the functions
91 * arm_cmplx_conj_f32()
92 * arm_cmplx_mult_cmplx_f32()
93 * The reference page for these DSP functions can be found here:
94 * http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html
95 * The array pTmp may be freely used and is long enough to store any complex
96 * vector of up to length elements. */
97
98 /* TODO: Add code from here...*/
99
100 /* ...to here */
101 #endif
102 }
```