Final Project Report

18-758 Wireless Communications

Michael Nye (mnye)

System Design

Modulation

For my final project, I designed a system fairly similar to what was shown in lectures. For coding, I used a rate-1/2 convolution code with 4 states. After codig, the symbol bits are interleaved by a factor of 132 (chosen to be a large integral factor of my packet size). Again, testing of my system has demonstrated that this coding scheme sufficiently decouples error events and allows robust correction given my channel.

I use an 8-PSK modulation scheme. My choice of eight points was motivated by empirical measurements of the channel; I used the highest order constellation I could support without the noise causing symbol errors. I then used a hamming window pulse. The hamming window was chosen as it has better bandwidth properties than other FIR windows, but was easier to implement than a multisymbol pulse such as a raised cosine rolloff pulse. Testing showed that this pulse was sufficient to meet the project specifications.

Finally, I prepend my message with a single pilot sequence to facilitate equalization and timing synchronization. The sequence is created by generating a bit sequence. This sequence is a De Bruijn sequence, which creates a large variety in symbol ordering to create a unique sequence. This sequence is then modulated using a wide BPSK for easy detection.

Constant factors were chosen to be as large as possible while still meeting channel requirements and message size. Since I was fixed to rate-1/2 coding and 3 bits per symbol, this meant to fit my entire message (3036 pixels) in the allowed space, I was able to use at most 4 samples per symbol. I then expanded my pilot sequence to fill most of the remaining space.

Demodulation

My demodulator largely follows the reverse of my modulator. First off, I perform carrier recovery. To do so, I compute a DTFT on my received signal over a range of only low frequencies. I find the maximum amplitude frequency, and then divide by a complex sinusoid of the same frequency.

This is followed by timing recovery and equalization. I find the correlation between my pilot signal and the recovered signal. The maximum lag is the start of my pilot sequence. I can then extract the pilot sequence, and detect the pilot symbols. The channel equalization constant is then found by comparing the detected symbols to the known pilot symbols. Finally, the received signal is divided by this same factor to equalize.

At this point, I can window out only my message. I match filter the message, then sampling the result and perform hard detection of the coded bits. After deinterleaving these detected bits, I find the minimum error path through my coding trellis to correct any bit errors in the coded bits. Finally, the corrected coded bits are decoded into my message bits and returned.

Analysis

My system largely follows the principles demonstrated in lectures. There are three main differences I chose to make to simplify the system.

The first difference was my choice of pilot. I chose to use the same pilot message for carrier recovery, timing recovery and equalization. My testing found that this didn't cause any degradation of performance, and allowed me to minimize the size of my header.

The second difference was my choice of window. While raised cosine rolloff windows are clearly superior to hamming windows in terms of bandwidth properties, I found it difficult to correctly design the window to prevent ISI. A hamming window was much simpler to implement and performed admirably for my purposes.

The final difference was in how I performed carrier recovery. While we were encouraged to use a BPSK pilot sequence, I found this didn't provide sufficient resolution to find the carrier offset frequency. I instead just used the entire received signal, and bandlimited the region I search for the peak in the frequency domain. This proved surprisingly effective, and so I stuck with it.

The other difficulties I ran into were simply implementation details. Until this project, I had never used the MATLAB **'** operator to transpose complex data. I learned that this was a hermitian transpose, but it lingered in dark corners of my code base, and took quite a while to track down the errors. I also spent a lot of time determining exact offsets for my sampling times. This complexity was simply an artifact of using convolution with MATLAB vectors, and the resulting offsets it introduces into my data.

Operational Figures

On the following pages, I have attached sample figures demonstrating my the functioning of my system. I have shown a higher-than-average noise sample demonstrating that even under harder transmit criteria, I receive less than a .3% error rate.

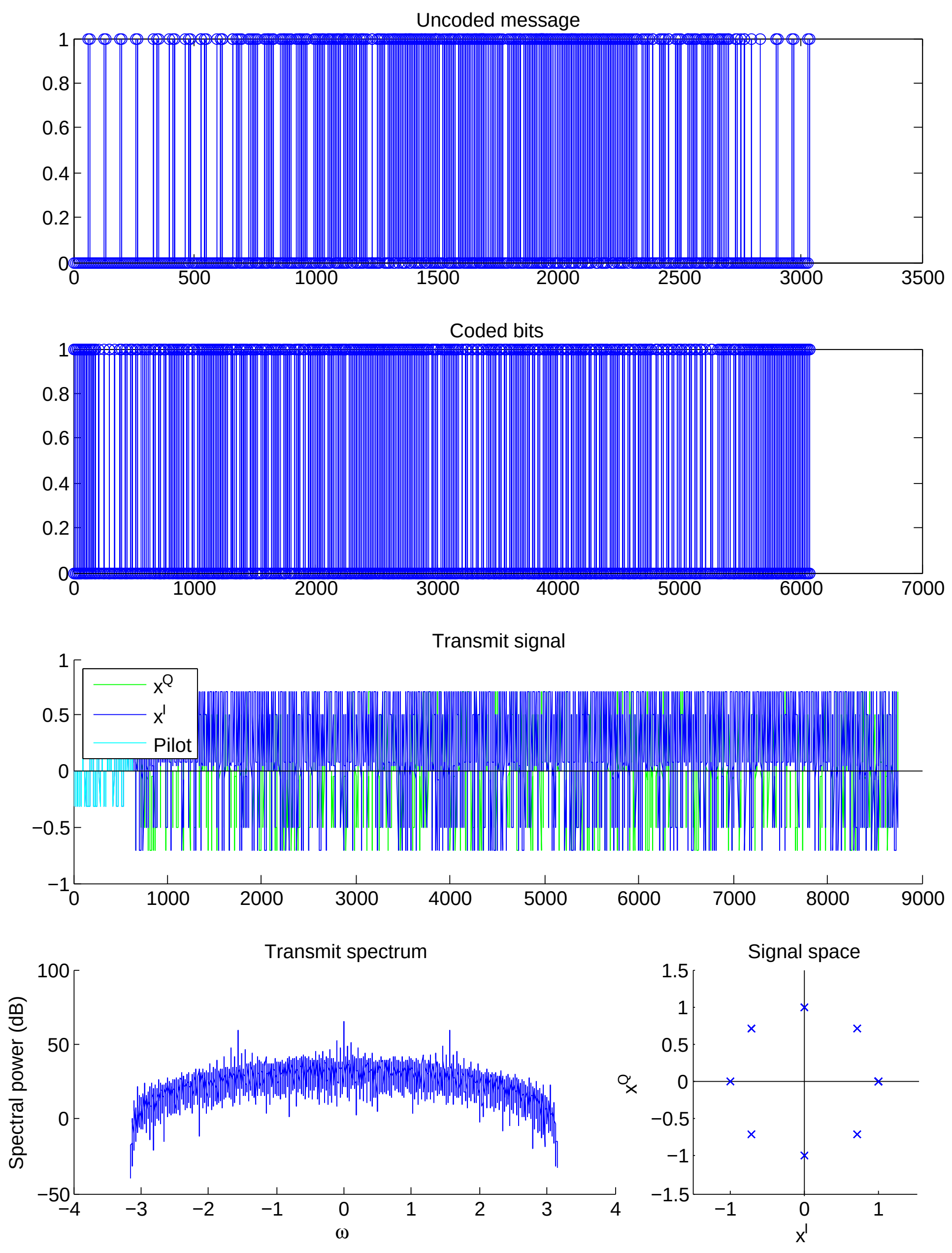


Figure 1: Generated transmit signal

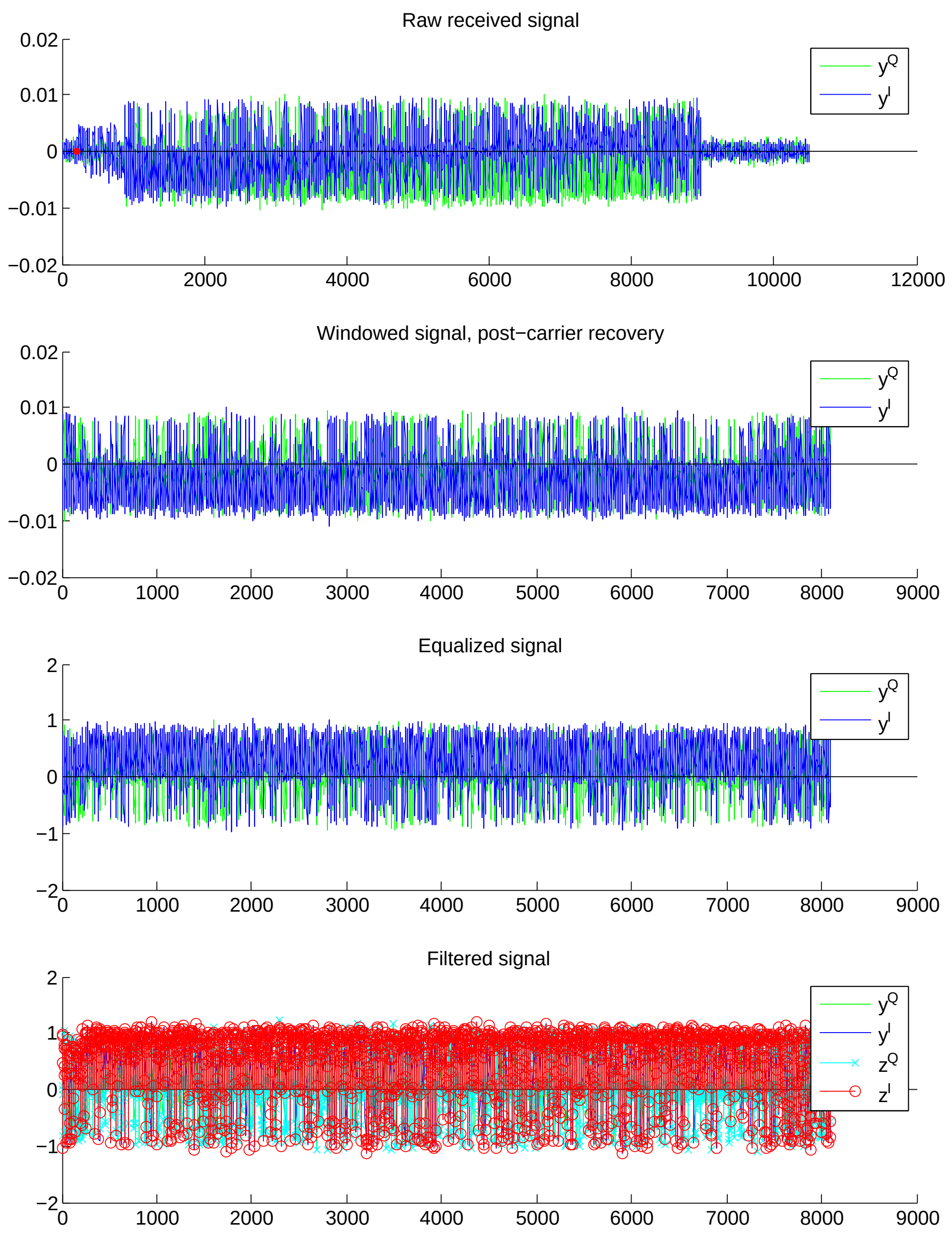
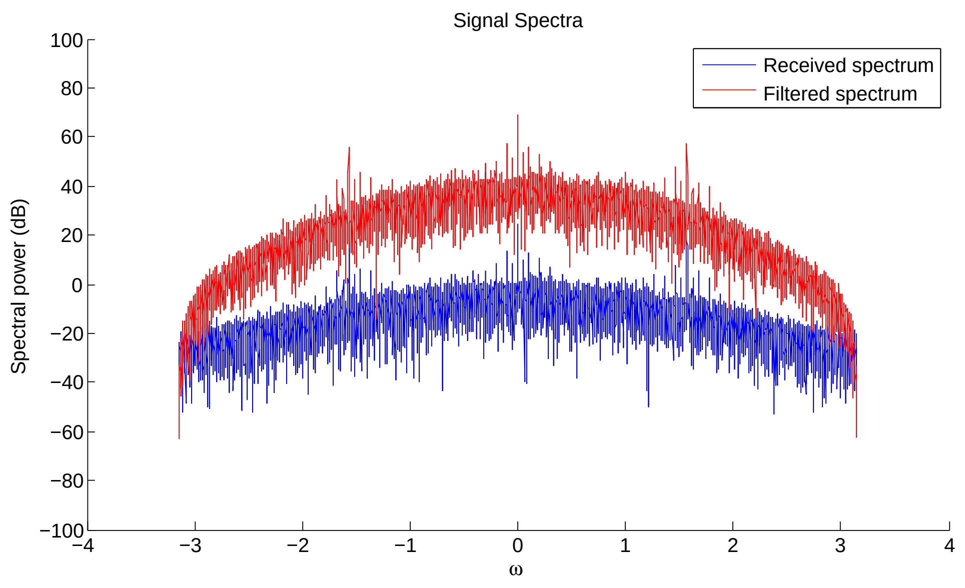


Figure 2: Signal received from radio, and various processing



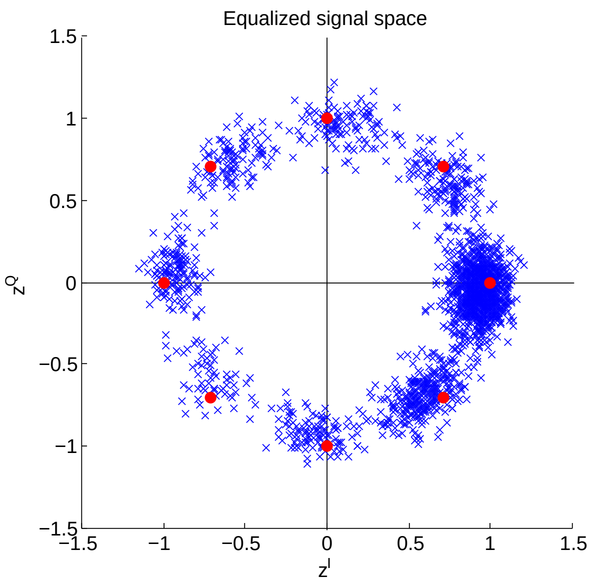


Figure 3: Received signal spectra and signal space

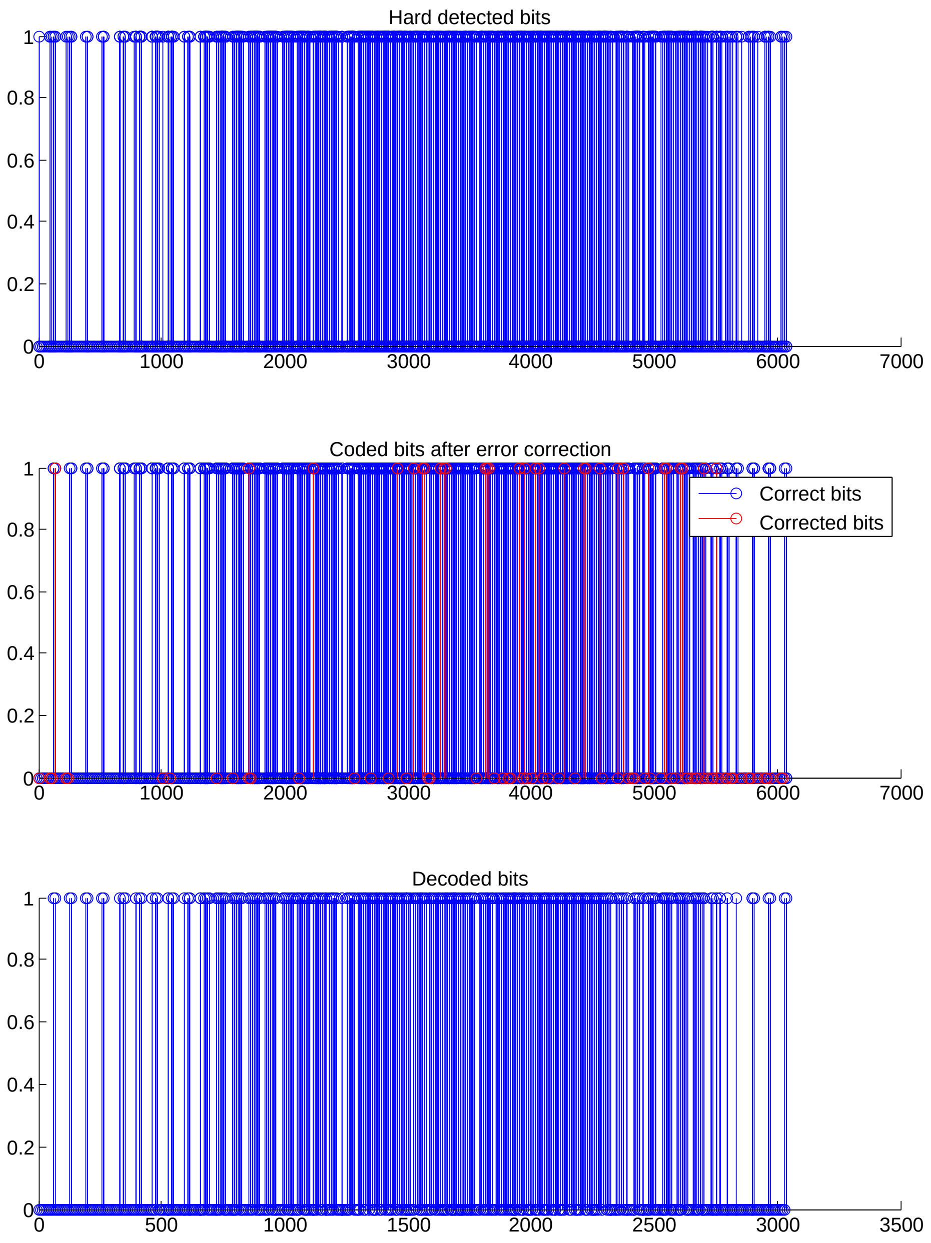


Figure 4: Received signal bits, and decoding

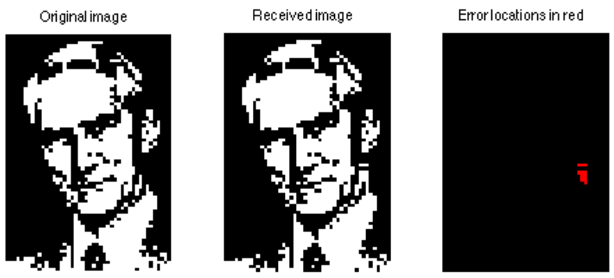


Figure 5: Comparison of transmitted and received images

MATLAB Implementation

On the following pages, I have attached the code used to implement my communications system. There are three files. The first is `constants.m`, and contains just a set of constants referenced in my modulator and demodulator. The second is `generate_transmit_signal.m`, and contains a function that given a bit sequence will return a transmit signal. Finally, `decode_received_signal.m` takes in a received signal and performs recovers the original message in the transmit signal.

```matlab
%
% constants.m
%    Defines the set of constants to be used for the comm system
%

% Constants defined by the radio
Fs = 2e6;     % transmit rate of USRP in Hz
maxL = 10000; % max samples in output signal


% Transmit constants
T = 4;  % samples per symbol
B = 3;  % bits per symbol
L = 3036; % packet size in bits

% Coding constants
coded = true; % enable
R = 2;        % coded bits per data bit
interleaveA = 132; interleaveB = R*23; % Factors of interleaving

if ~coded, R=1; end;


% Hamming pulse for modulation. Normalize to unit energy
pulse = hamming(T)';
pulse = pulse/norm(pulse);


% De Bruijn sequence of order 5 for timing sequence
% Using a BPSK modulation
pilotBits = [0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, ...
             0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1];

pilotT = 20;
pilotPulse = ones(1,pilotT/2); pilotPulse = pilotPulse/norm(pilotPulse);

pilot = upsample(2*pilotBits-1,pilotT);
pilot = conv(pilot,pilotPulse);


% Channel constants for simulation
SNR = 10; % dB
Ex = 1; % expected symbol energy
sigN = Ex / 10.^(SNR/10); % noise variance

maxdelay = 500; % max delay before transmit in samples
atten = [40 60]; % factor of attenuation during transmit
```

```matlab
function x = create_transmit_signal(bits, plots)
    % Load constants
    constants;

    % Fill variables
    if nargin < 2
        plots = false;
    end

    % Pad bits to full length
    if(length(bits) > L)
        error('Provided packet exceeds max packet size');
    end
    N = length(bits);
    bits = [bits zeros(1,L-N)];


    % Display message, pulse and pilot
    if plots
        figure(1); clf(1);
        subplot(4,1,1);
        stem(bits(1:N));
        title('Uncoded message');
    end


    % Generate coded bits from the input
    % Uses a 4-state rate 1/2 convolutional code
    if coded
        codedbits = zeros(1,R*length(bits));
        state = [0 0];
        for ii=1:length(bits)
            % Get the current data bit
            bit = bits(ii);

            % Add the coded bits depending on the state
            if isequal(state, [0 0])
                codedbits(2*ii-1 : 2*ii) = xor(bit, [0 0]);
            elseif isequal(state, [0 1])
                codedbits(2*ii-1 : 2*ii) = xor(bit, [1 0]);
            elseif isequal(state, [1 0])
                codedbits(2*ii-1 : 2*ii) = xor(bit, [1 1]);
            else % isequal(state, [1 1])
                codedbits(2*ii-1 : 2*ii) = xor(bit, [0 1]);
            end

            % Update the state
            state = [state(2) bit];
        end

        % Interleave the coded bits
        codedbits = reshape(codedbits,interleaveA,interleaveB).';
        codedbits = codedbits(:).';
    else
        codedbits = bits;
    end

    if plots
        subplot(4,1,2);
        stem(codedbits(1:R*N));
        title('Coded bits');
    end

    % Map coded bits to symbols
    % Uses 16PSK
    M = 2^B;
    ints = bi2de(reshape(codedbits,B,length(codedbits)/B).').';
    syms = exp(2*pi*1j*ints/M);


    % Expand in time and convolve with pulse sequence
    x = upsample(syms,T);
    x = conv(x,pulse);


    % Place pilot sequence at head
    x = [pilot x];

    if plots
        subplot(4,1,3); hold on;
        plot(imag(x),'g'); plot(real(x));
        plot(pilot,'c');
        title('Transmit signal');
        legend('x^Q', 'x^I','Pilot', 'Location','NorthWest');
        stem(0,'marker','none');

        spec = fftshift(fft(x));
        subplot(4,3,10:11); hold on;
        plot(linspace(-pi,pi,length(spec)),20*log10(abs(spec)+.01));
        title('Transmit spectrum');
        xlabel('\omega'); ylabel('Spectral power (dB)');

        subplot(4,3,12); hold on;
        plot([-2 2],[0 0], 'k');
        plot([0 1e-10], [-2 2], 'k');
        scatter(real(syms), imag(syms), 'bx');
        axis([-1.5 1.5 -1.5 1.5]); axis square;
        title('Signal space');
        xlabel('x^I'); ylabel('x^Q');
    end

    % Normaliae x to maximize power
    x = x/max(abs(x));

    % Error if x is too large
    if(length(x) > maxL)
        error('Computed message exceeds maximum message size');
    end
end
```

```matlab
function bits = decode_received_signal(y, plots, len)
    % Load constant factors and prep to display
    constants;

    % Fill variables
    if nargin < 2
        plots = false;
    end
    if nargin < 3
        len = L; %#ok
    end


    % Display received signal
    if plots
        figure(2); clf(2);
        subplot(4,1,1); hold on;
        plot(imag(y),'g'); plot(real(y));
        legend('y^Q', 'y^I');
        title('Raw received signal');
        stem(0,'marker','none');

        figure(3); clf(3);
        subplot(2,1,1); hold on;
        spec = fftshift(fft(y));
        plot(linspace(-pi,pi,length(spec)),20*log10(abs(spec)));
        title('Signal Spectra');
        xlabel('\omega'); ylabel('Spectral power (dB)');
    end


    % Perform carrier recovery, using DTFT to estimate frequency offset
    ws = linspace(-pi/1000,pi/1000,1500);
    [~,I] = max(abs(dtft(y, ws)));
    y = y .* exp(-1j*(1:length(y))*ws(I));


    % Determine offset
    [corrs,lags] = xcorr(y,pilot, 500);
    [~,I] = max(abs(corrs));
    delta = lags(I);

    if plots; figure(2); scatter(delta,0,'r.'); end


    % Match filter and sample the pilot to equalize
    p = y(delta+1 : delta + length(pilot));
    p = filter(pilotPulse(end:-1:1), 1, p); %#ok
    zs = p(pilotT/2:pilotT:end);
    ps = 2*pilotBits - 1;
    eq = (ps*transpose(zs))/(ps*ps');


    % Drop the offset, pilot and trailing end
    L = min(L,len)*R/B; % number of symbols to read in
    delta = delta + length(pilot);
    y = y(delta+1 : delta + T*L);

    if plots
        subplot(4,1,2); hold on;
        plot(imag(y),'g'); plot(real(y));
        legend('y^Q', 'y^I');
        title('Windowed signal, post-carrier recovery');
        stem(0,'marker','none');
    end


    % Equalize
    y = y/eq;

    if plots
        subplot(4,1,3); hold on;
        plot(imag(y),'g'); plot(real(y));
        title('Equalized signal');
        legend('y^Q', 'y^I');
        stem(0,'marker','none');
    end


    % Match filter, grab inphase and quadrature components, sample
    y = conv(y, pulse(end:-1:1)); %#ok
    yi = real(y);     yq = imag(y);
    zi = yi(T:T:end); zq = yq(T:T:end);

    if plots
        subplot(4,1,4); hold on;
        plot(yq, 'g'); plot(yi);
        stem(T:T:length(y), zq, 'cx');
        stem(T:T:length(y), zi, 'ro');
        title('Filtered signal');
        legend('y^Q', 'y^I', 'z^Q', 'z^I');
        stem(0,'marker','none');

        figure(3);
        subplot(2,1,1); hold on;
        spec = fftshift(fft(y));
        plot(linspace(-pi,pi,length(spec)),20*log10(abs(spec)),'r');
        legend('Received spectrum','Filtered spectrum');

        subplot(2,1,2); hold on;
        plot([-2 2],[0 0], 'k');
        plot([0 1e-10], [-2 2], 'k');
        scatter(zi, zq, 'bx');
        scatter(real(exp(2*pi*1j*(0:(2^B-1))/2^B)), ...
                imag(exp(2*pi*1j*(0:(2^B-1))/2^B)), ...
                'ro', 'MarkerFaceColor','r');
        axis([-1.5 1.5 -1.5 1.5]); axis square;
        title('Equalized signal space');
        xlabel('z^I'); ylabel('z^Q');
    end


    % Hard detect the coded symbols from MPSK
    M = 2^B;
    angs = angle(zi + 1j*zq);
    decs = mod(round(angs*M/(2*pi)),M);
    detectedbits = de2bi(decs,B).';
    detectedbits = detectedbits(:).';


    % Correct the coded bits using viterbi
    if coded
        % Deinterleave the received bits
        detectedbits = reshape(detectedbits,interleaveB,interleaveA).';
        detectedbits = detectedbits(:).';


        % Create empty trelli
        oldtrellis = struct('errors', {0, Inf, Inf, Inf}, ...
                            'bits', {[], [], [], []});
        newtrellis = struct('errors', {0, 0, 0, 0}, ...
                            'bits', {[], [], [], []});

        % Score the whole trellis
        for ii=1:2:length(detectedbits)
            bits = detectedbits(ii:ii+1);

            % State 00
            error1 = oldtrellis(1).errors + sum(bits ~= [0 0]);
            error2 = oldtrellis(3).errors + sum(bits ~= [1 1]);
            if error1 < error2
                newtrellis(1).errors = error1;
                newtrellis(1).bits = [oldtrellis(1).bits 0 0];
            else
                newtrellis(1).errors = error2;
                newtrellis(1).bits = [oldtrellis(3).bits 1 1];
            end

            % State 01
            error1 = oldtrellis(1).errors + sum(bits ~= [1 1]);
            error2 = oldtrellis(3).errors + sum(bits ~= [0 0]);
            if error1 < error2
                newtrellis(2).errors = error1;
                newtrellis(2).bits = [oldtrellis(1).bits 1 1];
            else
                newtrellis(2).errors = error2;
                newtrellis(2).bits = [oldtrellis(3).bits 0 0];
            end

            % State 11
            error1 = oldtrellis(2).errors + sum(bits ~= [1 0]);
            error2 = oldtrellis(4).errors + sum(bits ~= [0 1]);
            if error1 < error2
                newtrellis(3).errors = error1;
                newtrellis(3).bits = [oldtrellis(2).bits 1 0];
            else
                newtrellis(3).errors = error2;
                newtrellis(3).bits = [oldtrellis(4).bits 0 1];
            end

            % State 10
            error1 = oldtrellis(2).errors + sum(bits ~= [0 1]);
            error2 = oldtrellis(4).errors + sum(bits ~= [1 0]);
            if error1 < error2
                newtrellis(4).errors = error1;
                newtrellis(4).bits = [oldtrellis(2).bits 0 1];
            else
                newtrellis(4).errors = error2;
                newtrellis(4).bits = [oldtrellis(4).bits 1 0];
            end

            % Make the new trellis the old trellis and continue
            oldtrellis = newtrellis;
        end

        % Select the lowest error path
        [~,I] = min([oldtrellis.errors]);
        codedbits = oldtrellis(I).bits;


        % Decode the found bits
        bits = zeros(1,length(codedbits)/R);
        state = [0 0];
        for ii=1:length(bits)
            % Get the current coded bits
            cbits = codedbits(2*ii-1 : 2*ii);

            % Find the data bit depending on the state
            if isequal(state, [0 0])
                bits(ii) = isequal(cbits, [1 1]);
            elseif isequal(state, [0 1])
                bits(ii) = isequal(cbits, [0 1]);
            elseif isequal(state, [1 0])
                bits(ii) = isequal(cbits, [0 0]);
            else % isequal(state, [1 1])
                bits(ii) = isequal(cbits, [1 0]);
            end

            % Update the state
            state = [state(2) bits(ii)];
        end


        if plots
            errors = find(codedbits ~= detectedbits);

            figure(4); clf(4);
            subplot(3,1,1); hold on;
            stem(detectedbits);
            title('Hard detected bits');

            subplot(3,1,2); hold on;
            stem(codedbits); stem(errors, codedbits(errors), 'r');
            legend('Correct bits', 'Corrected bits');
            title('Coded bits after error correction');

            subplot(3,1,3); hold on;
            stem(bits);
            title('Decoded bits');
        end
    else
        bits = detectedbits;
    end


    % Return only the requested symbols
    bits = bits(1:len);
end
```