18-758 Project Report

Jean-Baptiste Cordonnier - Thomas Mullins

December 11, 2014

1 Pulse

We use a square-root raised cosine pulse with a rolloff factor of $\alpha = 0.25$. The pulse was chosen because it is a Nyquist pulse of unit energy, which minimizes ISI, and because it conveniently falls off quickly in both the time domain and in the frequency domain.

$$\mathrm{SRRC}(t) = \begin{cases} \dfrac{1-\alpha+\frac{4}{\pi}\alpha}{\sqrt{T_s}} & t = 0 \\ \dfrac{\alpha\left(\left(1+\frac{2}{\pi}\right)\sin\frac{\pi}{4\alpha}+\left(1-\frac{2}{\pi}\right)\cos\frac{\pi}{4\alpha}\right)}{\sqrt{2T_s}} & |t| = \frac{T_s}{4\alpha} \\ \dfrac{\sin\left((1-\alpha)\frac{\pi}{T_s}t\right)+\frac{4\alpha}{T_s}t\cos\left((1+\alpha)\frac{\pi}{T_s}t\right)}{\frac{\pi}{\sqrt{T_s}}t\left(1-\left(\frac{4\alpha}{T_s}t\right)^2\right)} & otherwise \end{cases}$$

$T_s = 8\mu s$ is the symbol period for our transmitted signal. When sampling the received signal, we use a matched filter to optimally boost the signal while suppressing noise.

2 Frequency sync

We do not do frequency synchronization in the final version of our project. We found that the periodic channel re-estimation is sufficient to handle the gradual phase drift.

Initially we had a string of 40 ones at the start of the packet, and then the receiver would search for a peak in the discrete Fourier transform of the starting sequence and apply a correction to the whole received signal. However, this was ineffective because of the wildly fluctuating phase at the start of the signal, which can be seen in figure 1. Next we tried throwing out the first 250µs or so of the received signal and putting the frequency synchronization sequence after that, where the phase has a linear drift. This worked better, but it was then too hard to fit the entire message into 800µs, so we removed frequency synchronization altogether.

3 Timing sync

We use a simplified version of the timing synchronization we have seen in class. We use a random sequence of symbols that is highly uncorrelated, such that we see a high peak during the correlation of the message and the timing synchronization (figure 2). Then we get $\hat{\tau}$ and know where the signal starts exactly.

In order to get the best results from the timing synchronization, we upsample the received signal by a factor of four. The precision of the timing synchronization is limited by the sampling frequency, so upsampling allows the timing synchronization to be more effective. This also conveniently made the received sample rate the same as the transmitted sample rate.

In a previous version of the project, we were also reestimating the symbol period by taking the highest correlation peak with $T \in \{7, 8, 9\}$. But as we have seen that for all our trial, the symbol period for the receiver was always 8 we removed this part and took for granted that $T_{rx} = T_{tx} = 8$ samples.

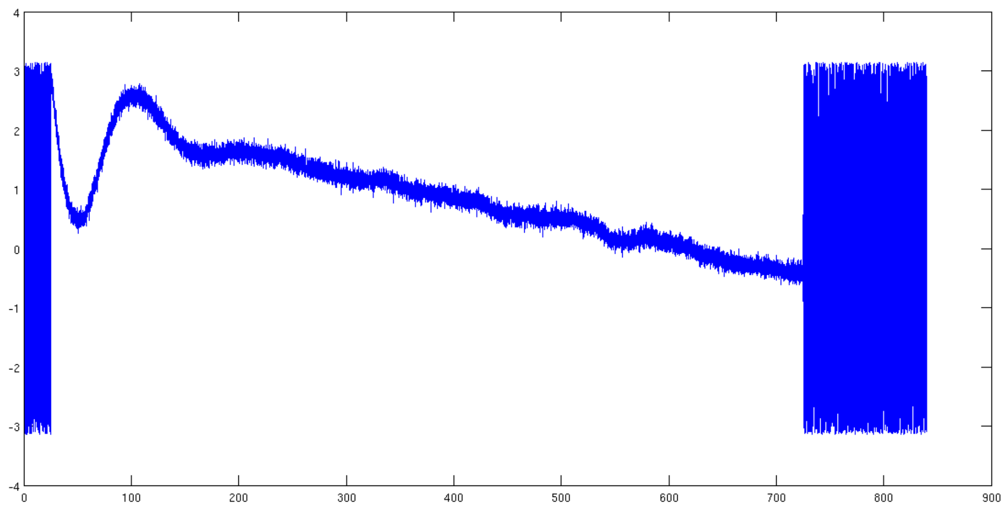

Figure 1: Received phase when sending all ones

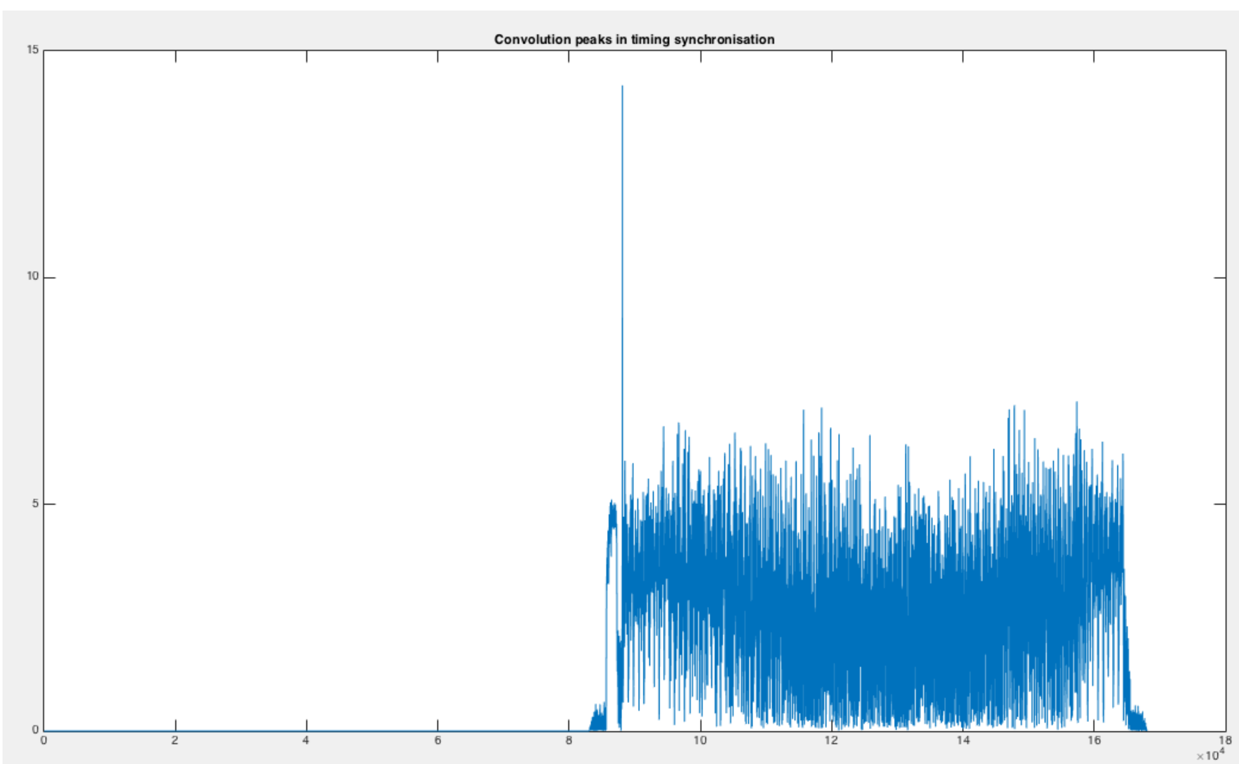

Figure 2: Cross-correlation of the signal and the timing synchronization preamble

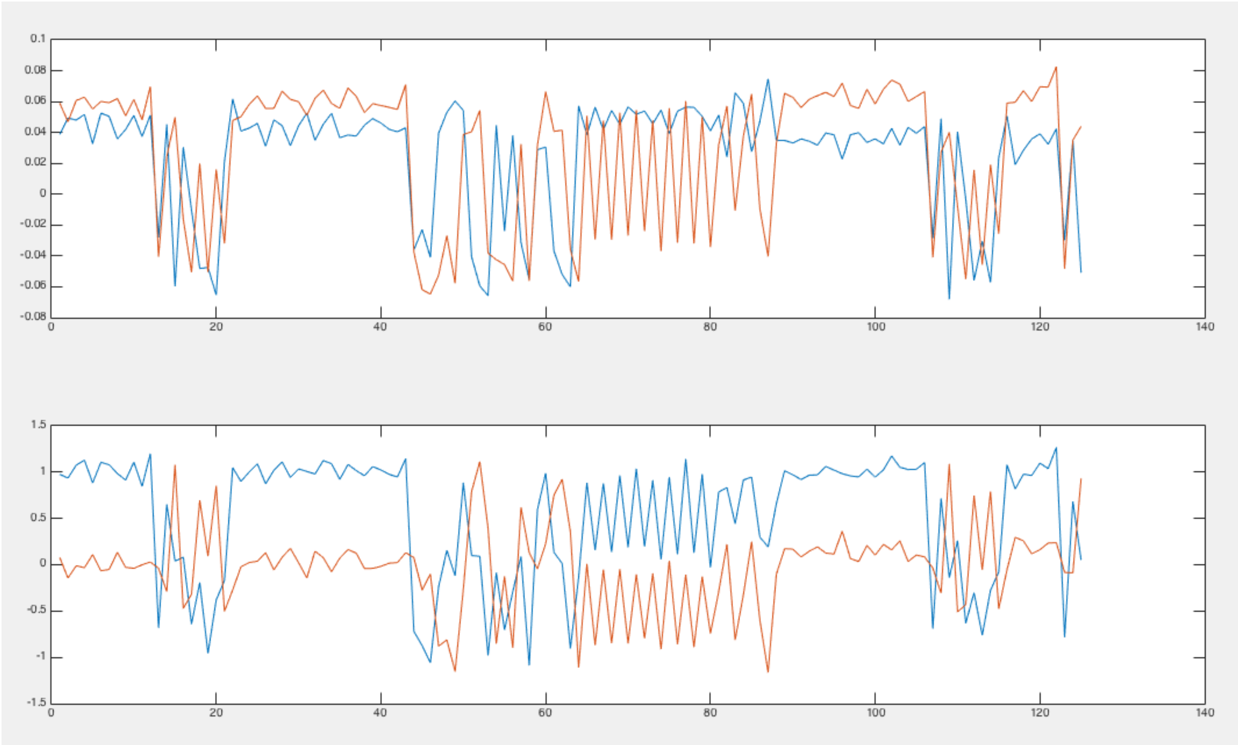

Figure 3: One message segment before and after equalization

4 One-tap equalizer

We are using a one-tap equalizer for each segment of the message. We send a 5-length symbols pilot (all one) every 120 symbols of information. The one tap equalizer uses the following formula :

$$h_0 = \frac{\text{txPilot} \cdot \text{rxPilot}}{\text{txPilot}^2} \qquad\qquad \text{eqMessage} = \frac{\text{message}}{h_0}$$

The equalization is very efficient as we can see in figure 3. Even if the pilot sequence is very short, the h_0 is still precise and equalizes well the received segment of the message. It corrects the phase drift and the gain of the signal.

5 Constellation

We are using a 4-PSK constellation with gray encoding to limit the number of bits affected by a transmission error. We have tried to use 8-PSK but with our other parameters, it wasn't efficient enough. As you can in figure 4, the equalization is not good enough to cluster the received points around the constellation points, then the decoding was generating a big BER. That's why we decided to stay with a 4-PSK constellation, which as you can see in figure 5 is quite efficient and has a good BER.

6 Channel Coding

We are using a rate $\frac{3}{4}$, 64 state convolutional code. The matrix which describes the trellis was taken from the lecture handouts:

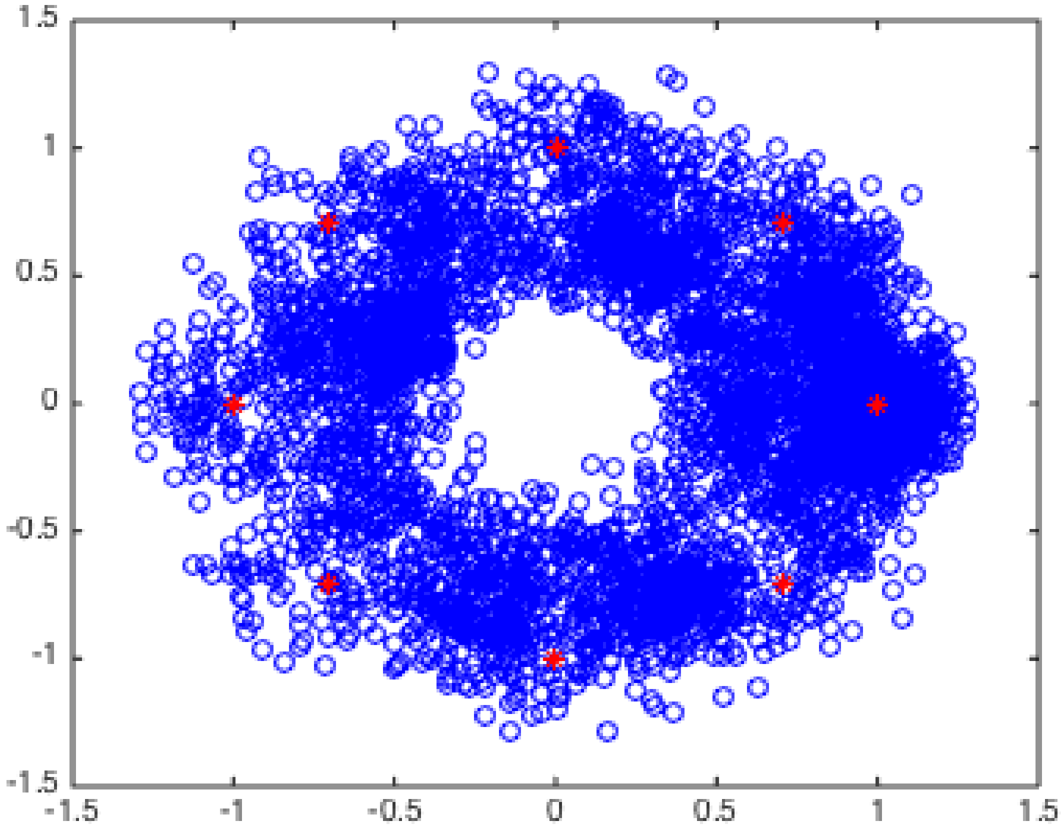

Figure 4: Received symbols when using 8-PSK

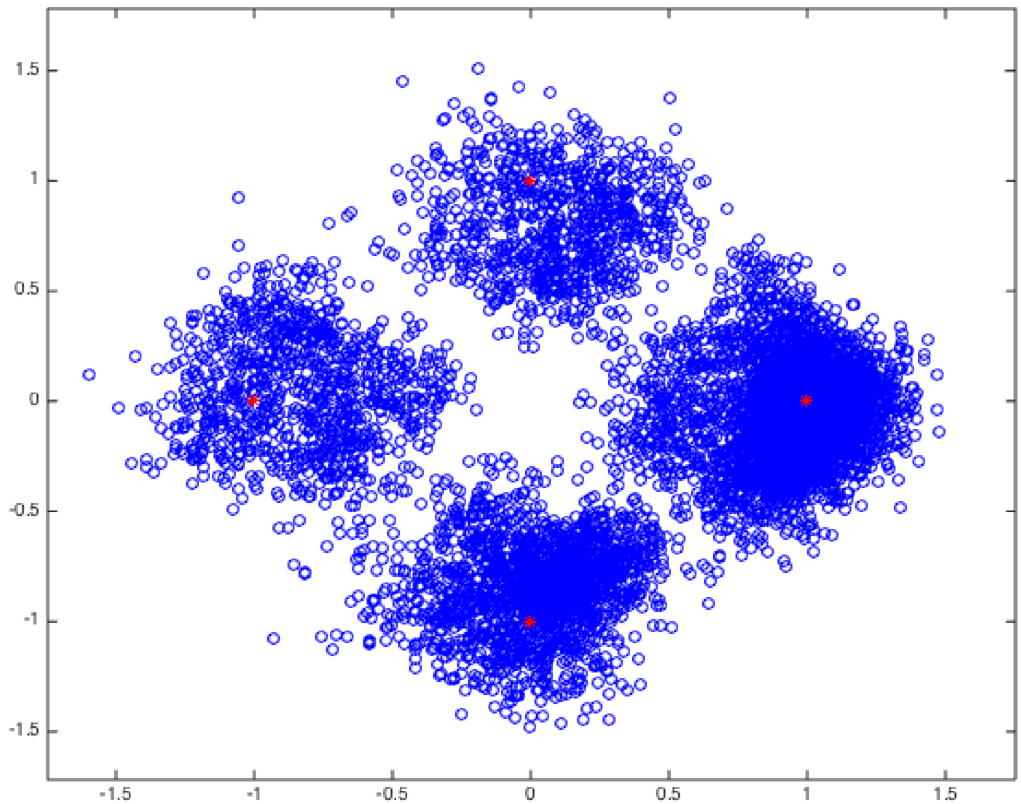

Figure 5: Received symbols when using 4-PSK

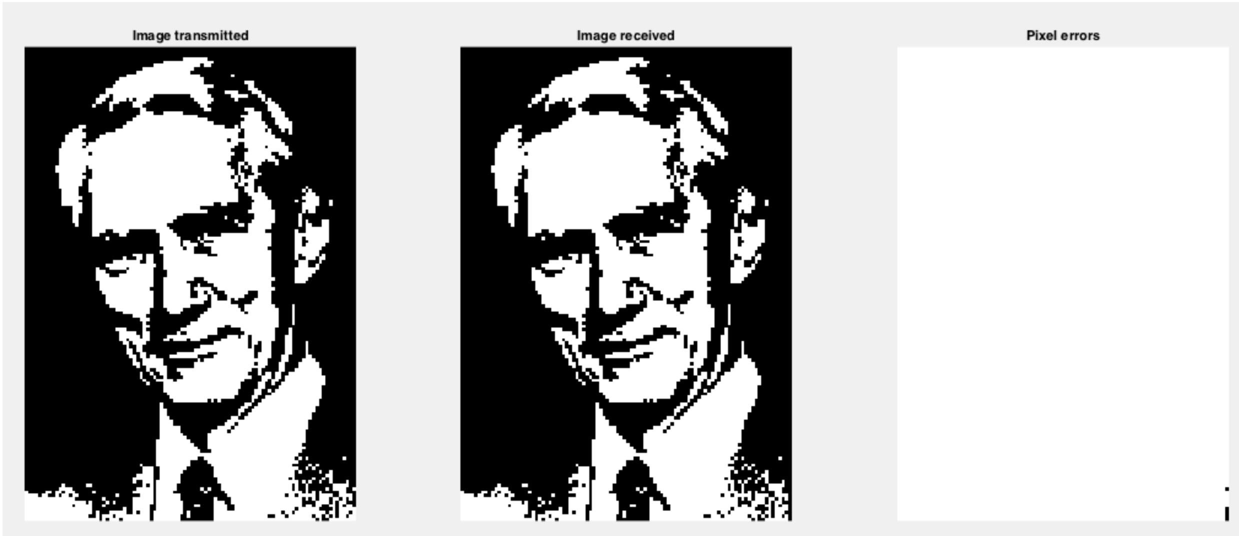

Figure 6: Transmitted and Received Images

$$g = \begin{pmatrix} 6 & 1 & 4 & 3 \\ 3 & 4 & 0 & 7 \\ 2 & 6 & 7 & 1 \end{pmatrix}$$

In this matrix g, each row corresponds to an information bit, each column corresponds to a coded bit, and each element is an octal number whose binary representation describes which bits of history are exclusive-or'd together. Each row has elements up to three bits in size for a total of nine input bits. Of these, six are state bits and three are new information bits. This means there are a total of 64 states in the trellis.

Before encoding, $(\nu+1)\cdot 3 = 21$ zeros are appended to the signal so that it always ends in a known state, which improves the error rate of the last few information bits. If this is not done, the last few bits do not have the full benefit of the convolutional code.

The receiver uses hard decoding on received symbols to recover the coded bits, and then uses the Viterbi algorithm to recover the information bits. Hard decoding is done using a standard minimum distance detector. Viterbi decoding involves first walking forward through the trellis, calculating the minimum cumulative bit error for each branch entering a state and keeping track of which branch that minimum is from. Then the decoder walks backwards through the trellis, following the path of minimum bit error, and recording the information bits which produce that path.

The channel coding is effective, but could much more effective if we used a lower rate convolutional code. With a couple bit errors very near each other, the rate $\frac{3}{4}$ code can amplify the error and cause a larger string of errors, which is occasionally a problem. Also, as seen in figure 6 there is always a string of errors near the end. This could be some noise that always appears near the end of the image, such as due to the drifting phase of the signal. This could also be due to a bug in the Viterbi decoding. Interleaving may help alleviate some of these problems.

A params.m

```
% signal timing
txSamplingFrequency =  100 * 10^6; % Hz
rxSamplingFrequency =   25 * 10^6; % Hz
symbolRate          = 12.5 * 10^6; % Hz

% signal parameters
M = 4;         % size of constelation
constelation = zeros(1,M);
for i = 0:M-1
    constelation(i+1) = M_PSK_encode(de2bi(i, nextpow2(M)),M,1);
end
alpha = 0.25; % SRRC coefficient
txPad = 32;   % extra 0 symbols to transmit on either side of message


pulseCenter = 500;

% receive paramters
rxUpsample = 4;

% transmit signal sync sequences
timingSync = [1 1 1 -0.4352 - 0.4398i  -0.2497 + 0.5485i  -0.0417 - 0.1385i  -0.4626 + ...
    0.6763i  -0.1335 + 0.4968i   0.1663 - 0.1779i 0.5440 + 0.4108i  -0.0505 + 0.4528i   ...
    0.5746 - 0.1020i  -0.2389 + 0.1394i   0.5797 + 0.2914i  -0.1767 + 0.3388i 0.6549 + ...
    0.0617i   0.0578 - 0.2724i  -0.6183 - 0.4586i  -0.5870 - 0.0527i  -0.7076 + 0.5985i   ...
    0.2059 - 0.7190i -0.6773 - 0.4204i  -0.0649 - 0.5375i  -0.7086 + 0.3275i  -0.2104 + ...
    0.4044i  -0.0914 - 0.0915i  -0.6501 - 0.6495i -0.5897 + 0.1356i  -0.3734 + 0.4923i   ...
    0.5152 + 0.6686i  -0.0160 - 0.4034i  -0.3948 + 0.0531i   0.3780 - 0.2198i -0.0559 + ...
    0.2009i   0.6019 - 0.4881i   0.3110 + 0.1121i  -0.0962 + 0.5541i  -0.1542 - 0.4630i   ...
    0.1923 + 0.1788i -0.2481 + 0.4369i   0.7203 + 0.6936i  -0.5379 - 0.3861i  -0.6870 + ...
    0.1549i  -0.5613 - 0.1335i   0.5539 + 0.0694i -0.1889 - 0.4206i  -0.0852 + 0.6579i  ...
    -0.5422 - 0.0422i   0.5147 - 0.6585i   0.2764 + 0.6908i  -0.3126 - 0.5281i 0.2672 + ...
    0.5905i   0.1599 + 0.5768i  -0.4421 + 0.3669i  -0.2217 - 0.1174i  -0.4965 + 0.4600i   ...
    0.1802 + 0.3440i 0.4400 - 0.6241i   0.6501 - 0.0035i   0.3680 + 0.3496i   0.4775 - ...
    0.4954i  -0.0616 + 0.1703i   0.6233 + 0.4832i 0.5703 + 0.1190i   0.1193 + 0.5119i  ...
    -0.6708 + 0.5558i  -0.1331 - 0.6686i   0.3550 - 0.4978i  -0.5135 + 0.1528i -0.3541 - ...
    0.2536i  -0.1416 - 0.1350i  -0.1641 + 0.1584i  -0.4804 - 0.4498i  -0.5846 - 0.2550i   ...
    0.3888 - 0.3834i 0.3466 + 0.2781i   0.4674 + 0.4730i  -0.2980 - 0.2749i   0.0332 - ...
    0.2519i   0.4786 + 0.4475i   0.0822 - 0.3418i 0.2604 - 0.3841i  -0.0628 - 0.1665i   ...
    0.0557 + 0.7091i   0.3681 + 0.6929i  -0.3825 + 0.0412i  -0.6469 + 0.3705i 0.1471 + ...
    0.5151i   0.7042 + 0.6194i  -0.1305 - 0.7206i   0.0590 - 0.4215i  -0.4048 - 0.2512i  ...
    -0.5827 + 0.3570i 0.3584 + 0.0624i  -0.2334 + 0.4793i   0.0758 + 0.6598i   0.5665 - ...
    0.2069i   0.0669 - 0.2211i   0.1771 + 0.4278i 0.3546 - 0.5400i   0.4649 - 0.6848i  ...
    -0.1234 + 0.3337i   0.4058 - 0.1914i ones(1,10) zeros(1,10)];
pilot = ones(1, 5);

imageDimension = [140 98];
imageSize = imageDimension(1) * imageDimension(2);
imageFile = strcat('images/shannon', int2str(imageSize), '.bmp');
txImage = imread(imageFile);
txMessageBits = reshape(txImage, [1 imageSize]);
txMessageBits = [ txMessageBits 0 0];

% rate 3/4
g = oct2dec([6,1,4,3;3,4,0,7;2,6,7,1]);
nu = 6;

txCodedBits = channelEncode(txMessageBits, g, nu);

messageSizeBits = length(txCodedBits);
messageSizeSymb = messageSizeBits / nextpow2(M);
packetSizeInfo = 120; % information symbols
packetSizeTot = length(pilot) + packetSizeInfo;
```

B receiver.m

```
params;
Fs = rxSamplingFrequency * rxUpsample; % Hz
n = Fs / symbolRate; % samples per symbol

load('receivedsignal.mat');

receivedsignal = resample(receivedsignal, rxUpsample, 1);
plotSignal(receivedsignal, Fs);

% start = find(abs(receivedsignal) > 0.075, 1);
% receivedsignal = receivedsignal(start:length(receivedsignal));

% Time recovery to determine tau hat
tau_hat = doTimingSync(receivedsignal, timingSync, n, alpha);

% sampling
nSample = length(timingSync) + packetSizeTot * ceil(messageSizeSymb / packetSizeInfo);
samples = doSampling(receivedsignal, nSample, txSamplingFrequency / symbolRate, tau_hat);
cutSamples = samples((length(timingSync) + 2) : length(samples));

figure;
t = 1:length(cutSamples);
plot(t, real(cutSamples), 'b', t, imag(cutSamples), 'r')
title('cutSamples')

% Equalization and pilot removal
nSegments = floor(messageSizeSymb / packetSizeInfo);
messageSymbols = zeros(1, messageSizeSymb);
samp = ((1:nSegments+1)-1) * packetSizeTot + 1;
mess = ((1:nSegments+1)-1) * packetSizeInfo + 1;
for i = 1:nSegments
    s = cutSamples(samp(i) : samp(i)+packetSizeTot-1);
    eqSamples = equalize(pilot, s);
    messageSymbols(mess(i):mess(i) + packetSizeInfo - 1) = eqSamples;
end

remainingSamples = mod(messageSizeSymb, packetSizeInfo);
if remainingSamples ≠ 0
    samples = samples(samp(nSegments + 1) : samp(nSegments + 1) + length(pilot) + ...
        remainingSamples - 1);
    eqSamples = equalize(pilot, samples);
    messageSymbols(mess(nSegments + 1):mess(nSegments + 1) + remainingSamples - 1) = ...
        eqSamples;
end

% Constelation before and after equalization plots
figure;
subplot(1,2,1)
endMessage = find(abs(samples) > 0.075, 5, 'last');
plot(real(samples(1:endMessage)), imag(samples(1:endMessage)), 'bo', real(constelation), ...
    imag(constelation), 'r*');
title('Samples before equalization')
subplot(1,2,2)
plot(real(messageSymbols), imag(messageSymbols), 'bo', real(constelation), ...
    imag(constelation), 'r*');
title('Samples after equalization')

% symobols to bit decoding
rxCodedBits = M_PSK_decode(messageSymbols, M);
rxMessageBits = channelDecode(rxCodedBits, g, nu);

codedBER = sum(rxCodedBits ≠ txCodedBits) / length(rxCodedBits);
BER = sum(rxMessageBits ≠ txMessageBits) / length(rxMessageBits);
fprintf('Coded BER = %f\n', codedBER);
```

```
fprintf('BER = %f\n', BER);

% show image
rxMessageBits = rxMessageBits(1:length(rxMessageBits)-2);
rxImage = reshape(rxMessageBits, imageDimension);
figure
subplot(1,3,1)
imshow(txImage)
title('Image transmitted')
subplot(1,3,2)
imshow(rxImage)
title('Image received')
subplot(1,3,3)
imshow(1-abs(rxImage-txImage))
title('Pixel errors')
```

C transmitter.m

```
params;
n = txSamplingFrequency / symbolRate;

nSegments = ceil(messageSizeSymb / packetSizeInfo);
messageSymb = zeros(1, nSegments * length(pilot) + messageSizeSymb);
mb = ((1:nSegments) - 1) * packetSizeInfo * nextpow2(M) + 1;
ms = ((1:nSegments) - 1) * packetSizeTot + 1;
for i = 1:nSegments
   symbols = M_PSK_encode(txCodedBits(mb(i):min(length(txCodedBits), mb(i) + ...
       packetSizeInfo * nextpow2(M) - 1)), M, 1);
   messageSymb(ms(i) : min(length(messageSymb), ms(i) + packetSizeTot - 1))  = [ pilot ...
       symbols ];
end

symbols = [ones(1, 200) timingSync messageSymb];

fprintf('Transmitting message of %d bits in %d us\n', messageSizeBits, ...
    ceil(length(symbols) / symbolRate * 10^6));
if length(transmitsignal) > 800 * 10^-6 * txSamplingFrequency
    error('Signal is too long (%d samples)\n', ...
        length(transmitsignal));
end

pulseTx = srrc(-pulseCenter:pulseCenter, alpha, n);
padding = zeros(1, txPad);
X = applyPulse([padding symbols padding], pulseTx, pulseCenterTx, n);
X = 0.9 * X / max(abs(X));

plotSignal(X, txSamplingFrequency);

transmitsignal = X;
save('transmitsignal.mat', 'transmitsignal');
```

D doTimingSync.m

```
function tau_hat = doTimingSync(sign, timingSync, T, alpha)

% Create the centered pulse
pulseRx = srrc(-pulseCenter:pulseCenter, alpha, T);

% calculate expected timing frame
```

```
timingSync = applyPulse(timingSync, pulseRx, pulseCenterRx, T);

% correlate that with entire signal
[C, lag] = xcorr(sign, timingSync);

% plot it
figure;
plot(abs(C));
title('Convolution peaks in timing synchronisation');

% find the maximum correlation
[¬, i_max] = max(C);
tau_hat = lag(i_max);

end
```

E equalize.m

```
function [message, h0] = equalize(pilot, signal)
% equalize the signal using the pilot sequence at the begining
% and return the remaining equalized message

    receivedPilot = signal(1:length(pilot));
    h0 = (pilot .* receivedPilot) / (pilot .* pilot);
    message = signal(length(pilot)+1:length(signal)) / h0;

    % Plot of the equlalization before, after
    if 0
        figure
        subplot(2,1,1)
        plot(1:length(signal), real(signal), 1:length(signal), imag(signal))
        title('Signal before equalization')
        subplot(2,1,2)
        plot(1:length(signal), real(signal / h0), 1:length(signal), imag(signal / h0))
        title('Signal after equalization')
        pause
    end
end
```

F doSampling.m

```
function samples = doSampling(signal, nSamples, T_hat, tau_hat)

    signal = signal(tau_hat:tau_hat+nSamples*T_hat-1);
    preSamples = reshape(signal, T_hat, nSamples);
    samples = preSamples(1,:);

end
```

G srrc.m

```
function X = srrc(t, alpha, Ts)
%SRRC Returns a square-root raised cosine pulse

    X = (sin((1 - alpha) * pi / Ts * t) ...
```

```
        + 4 * alpha / Ts * t .* cos((1 + alpha) * pi / Ts * t)) ...
        ./ (pi / sqrt(Ts) * t .* (1 - (4 * alpha / Ts * t) .^ 2));

    X(t == 0) = (1 - alpha + 4 * alpha / pi) / sqrt(Ts);

    X(abs(t) == Ts / 4 / alpha) = alpha / sqrt(2 * Ts) * ...
        ((1 + 2 / pi) * sin(pi / 4 / alpha) + ...
        (1 - 2 / pi) * cos(pi / 4 / alpha));

end
```

H applyPulse.m

```
function X = applyPulse(symbols, pulse, pulseCenter, n)

    nSamples = n * (length(symbols) + 1);
    X = zeros(1, nSamples);

    paddedPulse = [ zeros(1, nSamples) pulse zeros(1, nSamples) ];
    for i = 1:length(symbols)
        t = n*i;
        start = nSamples + pulseCenter - t;
        pulseI = paddedPulse(start:start+nSamples-1);
        X = X + symbols(i) * pulseI;
    end

end
```

I M_PSK_encode.m

```
function symbolSequence = M_PSK_encode(bitSequence, M, r)

    b = nextpow2(M);
    if 2^b ≠ M || mod(length(bitSequence), b) ≠ 0
        error('error during constellation encoding')
    end

    symbolSequence = zeros(1, length(bitSequence) / b);
    reshaped = reshape(bitSequence, [length(bitSequence) / b, b]);
    for i = 1:length(bitSequence) / b
        n = bin2gray(bi2de(reshaped(i,:)), 'psk', M);
        symbolSequence(i) = r * exp(1j*2*pi*n/M);
    end
end
```

J M_PSK_decode.m

```
function bitSequence = M_PSK_decode(symbolSequence, M)

    b = nextpow2(M);
    bitSequence = zeros(1, length(symbolSequence) * b);

    values = mod(round(angle(symbolSequence) * (M / (2*pi))), M);
    values = gray2bin(values, 'psk', M);

```

```
    for i = 1:length(symbolSequence)
        bits = de2bi(values(i), b);
        index = (i-1) * b + 1;
        bitSequence(index:index+b-1) = bits;
    end
end
```

K channelEncode.m

```
function Y = channelEncode(X, g, nu)
% X The information bits
% Y The coded bits
% g  The table describing the trellis
% nu The number of bits of past state

    k = size(g, 1);
    n = size(g, 2);

    nPastBits = ceil(nu/k);

    % convert g to a 3-dimensional array:
    %  dim 1: k
    %  dim 2: nPastBits+1, one for each bit in octal number
    %  dim 3: n
    G = [];
    for i = 1:n
        G = cat(3, G, de2bi(g(:, i), nPastBits+1));
    end

    % ensure we have k rows to group bits by stage
    X = reshape(X, k, []);

    % add some zero padding so we finish in the first state
    X = [X zeros(k, nu + 1)];

    Y = zeros(1, size(X, 2)*n);
    Yi = 1;

    % for each column (size k), emit n coded bits
    state = zeros(k, nPastBits+1);
    for info = X
        % push info into front of state
        state = [info state(:,1:nPastBits)];
        % for each output bit, do "xor" computation
        for i = 1:n
            andResult = G(:, :, i) .* state; % "and"
            Y(Yi) = mod(sum(sum(andResult)), 2); % "xor"
            Yi = Yi + 1;
        end
    end

end
```

L channelDecode.m

```
function Y = channelDecode(X, g, nu)
% X  The coded bits
% Y  The information bits
% g  The table describing the trellis
```

```
% nu The number of bits of past state

    k = size(g, 1);
    n = size(g, 2);

    nStates = 2^nu;
    nInputs = 2^k;
    nPastBits = ceil(nu/k);

    % convert g to a 3-dimensional array:
    %  dim 1: k
    %  dim 2: nPastBits+1, one for each bit in octal number
    %  dim 3: n
    G = [];
    for i = 1:n
        G = cat(3, G, de2bi(g(:, i), nPastBits+1));
    end

    % ensure we have n rows to group bits by stage
    X = reshape(X, n, []);
    Xlen = size(X, 2);

    % generate table of state transitions and codes ahead of time
    nextStateTable = zeros(nStates, nInputs);
    codedTable = zeros(n, nStates, nInputs);
    for state = 0:nStates-1
        % zero fill binState to ensure it can be reshaped into k rows
        binState = [de2bi(state, nu) zeros(1, mod(nu, k))];
        binState = reshape(binState, k, []);
        for input = 0:nInputs-1
            binInput = de2bi(input, k);
            for i = 1:n
                andResult = G(:, :, i) .* [binInput' binState]; % "and"
                codedTable(i, state+1, input+1) = ...
                    mod(sum(sum(andResult)), 2); % "xor"
            end
            nextState = [binInput' binState(:,1:nPastBits-1)];
            nextState = reshape(nextState, 1, []);
            nextStateTable(state+1, input+1) = bi2de(nextState(1:nu));
        end
    end

    dist = zeros(nStates, Xlen+1) + Inf;
    dist(1, 1) = 0;
    prevState = zeros(nStates, Xlen+1);
    prevInput = zeros(nStates, Xlen+1);

    % move forward through stages, filling dist, prevState, and prevInput
    for Xi = 1:Xlen
        % get a column vector of received bits, repeated for each state
        coded = repmat(X(:, Xi), 1, nStates);
        for input = 0:nInputs-1
            % get table of (coded bits, states)
            idealCoded = codedTable(:, :, input+1);
            % get row vector of next states
            nextStates = nextStateTable(:, input+1)';
            % compute difference for each state
            diff = sum(abs(idealCoded - coded))';
            totalDists = dist(:, Xi) + diff;
            for state = 1:nStates
                nextState = nextStates(state)+1;
                if totalDists(state) < dist(nextState, Xi+1)
                    dist(nextState, Xi+1) = totalDists(state);
                    prevState(nextState, Xi+1) = state-1;
                    prevInput(nextState, Xi+1) = input;
                end
            end
        end
```

```
    end

    % move backward through stages, tracking min distance
    state = 0;
    Y = zeros(1, Xlen*k);
    for Xi = Xlen:-1:1
        input = prevInput(state+1, Xi+1);
        Y((Xi-1)*k + 1 : Xi*k) = de2bi(input, k);
        state = prevState(state+1, Xi+1);
    end

    % remove 0 padding from encoding
    Y = Y(1:length(Y)-(nu+1)*k);

end
```

M plotSignal.m

```
function plotSignal(X, Fs)
%PLOTSIGNAL Plots a complex signal in time and frequency domains
%   X is the signal. Fs is the sample frequency in Hz.

    [T, F, Y] = DTFT(X, Fs / 10^6, 0);

    figure();

    subplot(2, 1, 1);
    plot(T, real(X), T, imag(X));
    xlabel('Time (us)');
    pan xon;
    zoom xon;

    rgnA = abs(F) > 11.25;
    rgnB = abs(F) > 12.5;
    rgnC = abs(F) > 35;
    valA = (abs(F) - 11.25) * -40 / 1.25;
    valB = (abs(F) - 12.5) * -30 / 22.5 - 40;
    envelope = zeros(length(F), 1);
    envelope(rgnA) = valA(rgnA);
    envelope(rgnB) = valB(rgnB);
    envelope(rgnC) = -70;

    subplot(2, 1, 2);
    plot(F, 10*log10(abs(Y)), F, envelope);
    xlabel('Frequency (MHz)');
    pan xon;
    zoom xon;
end
```

N DTFT.m

```
function [t, f, Z] = DTFT(z, Fs, n)
%DTFT Computes the DTFT of a signal with proper units

    L = length(z);
    L2 = pow2(n + nextpow2(L));
    t = (0:L-1) / Fs;
    f = (-L2/2:L2/2-1) * (Fs / L2);
    Z = fftshift(fft(z, L2)) * 2 / L;
```

```

end
```