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.