EEL4514 Final Project:

Design of a digital communication system

Jonathan Helm

May 3rd, 2006
# Table of Contents

Part I: Introduction............................................................................................................................................. p.3

Part II: Design and Solutions........................................................................................................................................ p.4
  A. A/D conversion........................................................................................................................................ p.4
  B. Digital modulation................................................................................................................................. p.5
  C. Digital demodulation............................................................................................................................. p.6
  D. D/A conversion......................................................................................................................................... p.8
  E. Sending the picture in packets......................................................................................................... p.10
  F. Fine tuning the system......................................................................................................................... p.10

Part III: Comments and Conclusions................................................................................................................. p.14

Part IV: References........................................................................................................................................... p.15

Part V: Appendix............................................................................................................................................... p.16
  A. Final project function........................................................................................................................  p.16
  B. 'send' function.................................................................................................................................. p.18
  C. 'separate' function............................................................................................................................. p.21
  D. 'a2d' function.................................................................................................................................... p.22
  E. 'polarmrz' function............................................................................................................................. p.23
  F. 'moddsbsc' function............................................................................................................................. p.24
  G. 'channel' function............................................................................................................................. p.25
  H. 'demod' function.................................................................................................................................... p.26
  I. 'binconvert' function............................................................................................................................. p.27
  J. 'd2a' function..................................................................................................................................... p.28
  K. 'correct' function................................................................................................................................... p.29
  L. 'combine' function................................................................................................................................ p.30
Part I: Introduction

Digital communication systems are becoming common place in the world of electronics. In the past 20 years, many forms of communication have gone from analog to digital. These systems include the plain old telephone system, television, and the cellular phone system. There is even contemplation of changing radio from analog to digital. Digital communication systems are becoming more popular because of their strength against noise, the ability to use repeater stations, and because the digital signal doesn’t change due to quality standards in components.

For this project we were asked to design a digital communication system that would send an image of Darth Vader across a specified channel similar to a phone line. In doing so we would be able to use any method that we could find on the Internet, scientific journals, or textbooks. The design requires the digital communication systems to have the lowest power to noise ratio possible while transferring the signal over the channel in the least amount of time. We were to design an A/D converter, a digital modulator, a digital demodulator, a D/A converter, and any other components we would require.

The communication system I designed has a data transfer rate of 450kbps per stream of data. There are six streams of data that are transferred over the channel simultaneously on different carrier frequencies to make a total data transfer rate of 2.7Mbps. The optimal power to noise ratio is 1.875. There are also several elements to reduce the effect of noise while minimizing the data transfer time. First there is 5 bit A/D converter to compress the image from 256 to 32 colors per data stream. This will produce 32,768 different color combinations per pixel rather than 16.777 million. Secondly, there is an added parity bit that is transferred along with each five bit sample. The parity is used to detect an error, if an error is detected it is corrected by the D/A converter. Total transfer time for the entire picture is 339.8ms.
Part II: Design and Solutions

There were many things to take into consideration in the design of my digital communication system. In the A/D conversion one has to consider sample rate, quantization levels, and the addition of a parity bit (if necessary). The design of the digital modulator must take into consideration things such as data transfer rate, line coding, carrier frequency, and transmission of multiple streams of data. Digital demodulation design must focus on separation of signals getting the original line code back, noise minimization, and filter design as well. Lastly the D/A conversion has to combine the serial digital signals to reproduce the original picture, and attempt to correct any errors that have been detected. Finally due the size of the picture and in an attempt to not cause buffer overflow in the receiver, the picture must be broken up and sent in packets.

A. A/D conversion

As mentioned earlier, A/D conversion must take into consideration things such as sample rate, quantization levels, and the addition of parity. Code for the A/D conversion can be found on pages 21 and 22.

The first thing I tried to figure out how to do in the A/D conversion was how to take the picture and separate it into the three colors matrices that make it up. This is similar to a digital camera that has three color filters and a bunch of photo-comparators that detect the intensity of each each color(red, green, and blue). The three color matrices were reshaped to become three vectors with the same numbers. Then 8 bit quantization was used to create three matrices with the same number of columns as the vectors, but with eight rows. There would be a binary 1 or 0 in each place in the matrix. Once again, the matrices were reshaped again to form three binary vectors. The three vectors were combined to create one very long vector that would contain the red, green, and blue binary vectors in series.

This was sufficient but later analysis would show the channel could hold many data streams simultaneously. In order to decrease the data transfer time, I decided to split the picture up into six different data streams. There would be two data streams for each color, one for the upper half the picture, and one for the lower half of the picture. The implementation of this is idea is similar to sending one long vector. The image is divided up into its three color matrices. Then the three color matrices are further split in half, where the upper half is in on matrix and the lower half is in the second matrix. This creates a total of six different color matrices, or two matrices for each color. The six matrices are then reshaped to form six vectors of numbers. The vectors of numbers are then converted into eight bit binary numbers, that are stored six matrices with eight rows and same number of columns as with the vectors. Once again a digital 1 or 0 is placed in each spot in the matrices, and matrices are then reshaped to for six vectors each each with length equal to eight times the original number vector length. This solution is nice because the data transfer time is decreased by a factor of six. Instead of sending one giant vector of binary numbers, six smaller vectors are sent simultaneously.

The next thing that I considered in the A/D conversion was the addition of a parity bit. This parity could be used at the receiver to detect any errors that were found. Once the errors are detected, attempts to correct for the errors could be implemented by the communication system. The type of parity that I decided to use was even parity, although the type of parity (even or odd) used really is arbitrary. This was implemented in the A/D converter right before the binary matrices were reshaped to form a single binary vector. The columns in each binary vector are summed to form a vector. The odd numbers in this vector are found by subtracting two times each vector element divided by two rounded down from the original vector element. This creates a vector of ones and zeros where zeros mark the columns where the sum is even, and ones mark the column where the sum is odd. This “odd” vector is attached the bottom row of the original binary matrix. Now all of the columns in the matrix sum up to only even numbers.

Then the question was asked if the user would be able to notice a difference in few quantization levels were used. Instead of using 256 quantization levels for each color (red, green, or blue) producing 16 million different color combinations, would it be possible to use fewer and if so how few? Some modifications were made on the A/D converter (and the D/A converter). Each number was quantized to 2^n levels, where n is number of bits used, by dividing each original number by 2^(8-n). Then the A/D continued it job using n bits rather than eight. By asking several people to look at different pictures reconstructed using four, five, six, seven, eight bit quantization I found that the nobody could detect a difference between the original picture and a picture
using five bit quantization. The detection threshold was at four bit quantization. At four bit quantization, people noticed the a quality difference in the received picture. Therefore I chose to use five bit quantization in my design.

The total picture size is 278 by 185 pixels. With five bit quantization, there are 15 bits per pixels that are transmitted over the channel. When the even parity bits are added, this number increases to 18 bits per pixel. The total number of bits to be transferred over the channel is 925,740. The total number of bits per stream of data sent is this number divided by six or 154,290. This is much lower than the 1.234 million bits that would be transmitted over a single stream as in the first iteration of the design. A flow chart of the final A/D conversion can be viewed below.

Analog Signal

---

B. Digital Modulation

The digital modulator must take a digital signal process it so that it can be transmitted across the channel. The digital signal must first be line coded. Then it must be modulated at an appropriate carrier frequency. If multiple signals are being transmitted over the channel, then these signals must be summed together using the digital modulator. The .m script for my digital modulator can be found on pages 23 and 24.

The first thing that must be done in the digital modulator is to determine what type of line coding to use. There are many types of line coding to choose from, the ones I considered were Polar NRZ, Unipolar NRZ, Bipolar RZ, and Manchester NRZ. The type of line coding must be easy to implement, and have a long pulse width to minimize bandwidth and enable longer averaging times. A small DC component is also desirable, to reduce DC bias damage in RF electronics. Manchester NRZ and Bipolar RZ both have shorter shorter pulse widths than Polar NRZ or Unipolar NRZ, so they would require more bandwidth to use. Both Manchester NRZ and Bipolar RZ are also difficult to implement in MATLAB. Polar NRZ has a smaller DC component than Unipolar NRZ so I chose to use Polar NRZ to line code my signal.

It was also necessary to determine a good pulse width for the line code. The longer the pulse width, the more resilient the signal would be to noise and transients in the filters. However as the pulse width increases, data transfer time also increases. Therefore a balance must be struck between a short transfer time and resilience to noise and filter transients. During initial setup I chose a pulse width of 10us (100kbps). This would fine tuned after the rest of the communication system was working. Implementing the Polar NRZ at 100kbps in MATLAB was pretty straight forward. A vector or zeros was allocated for the Polar NRZ signal. The number of
elements for each pulse was determined by dividing the sampling frequency by the data transfer rate (in this case, 10MHz/100kHz). If the corresponding digital element was a zero, a group of '-1's would be placed in the Polar NRZ vector. If the elements was a one, a group of '+1's would be placed in the Polar NRZ vector.

The line coded signal would then be multiplied by a cosine carrier before being transmitted over the channel. The only thing that needs to be considered is the impulse response of the channel. This can be found by inputting an impulse signal into the channel and plotting the PSD of the response.

![Impulse Response of Channel](image)

It can be seen than that sending a message through at any frequency from about 1.0MHz to 3.0MHz would be adequate. It is also possible to send data thought at about 4.0MHz. Since there would be multiple streams of data being sent through the channel, it would be a good idea to maximize the amount of frequency space between different different streams of data. In order to do this I chose to have modulating frequencies at 1.0MHz, 1.5MHz, 2.0MHz, 2.5MHz, 3.0MHz, and 4.0MHz. All of these frequencies readily pass through the channel. The implementation of this is simply multiplying the Polar NRZ signal by its cosine carrier. I also chose each carrier amplitude to be equal to 1 originally, but this could be modified later in the fine tuning stage.

The last thing that needed to be done in the design of the digital modulator was to sum up the six different streams of data before transmission over the channel. In practice this could be done with a series of op-amp adding circuits. The MATLAB implementation of this is to add each of the corresponding elements in each vector up, to create the transmitted signal. A block diagram of the digital modulator is shown below.

![Digital Modulator Block Diagram](image)

C. Digital Demodulation

Digital demodulation can be considered one of the most critical aspects of in a digital communication system. The demodulator must take the received signal, pass it through a series of bandpass filters. Those signal
must then be multiplied by a local carrier reference. The resulting signal must be passed through a low pass filter to eliminate the double frequency component. That signal must then be passed though an averaging circuit. The averaged signals are then converted by the digital demodulator to digital digital. MATLAB code for the digital demodulator can be found on page 26 and 27.

The received signal must be passed through a bandpass filter before it can be demodulated. There are two considerations for filter. The first consideration is what type of filter to use. The second is where the cutoff frequencies should be. In this project I tested two types of filters, a Chebyshev filter and a Finite Impulse Response filter. I found that the Finite Impulse response filter had a shorter transient response time than the Chebyshev filter. Even though the Chebyshev filter has a higher Q than the Finite Impulse Response filter, I chose the use the FIR filter. This was because having a filter with a high Q wasn't necessary because there is more than enough bandwidth on the channel to allow for low Q an FIR filter has. Having a quicker transient response time on the other hand enabled me to increase the transfer rate during the fine tuning stage.

The cutoff frequencies of the FIR bandpass filters could be determined by looking at the PSD of the received signal as well. By observing the PSD of the received signal, it can be determined that the cut off frequencies should be at the carrier frequency ± 200kHz.

Each signal would then be multiplied by a local carrier reference. Each local carrier reference would need to have the same frequency and be exactly in phase the with carrier frequency. This would be a challenge if it weren't for a Costas Loop. Although the Costas Loop was not implemented in MATLAB, its effect can be simulated by adding the delay from the channel convolution and FIR filters to a locally generated cosine.

The resulting signals must then be passed though another lowpass FIR filter to get rid of the double frequency components. The cutoff frequency of the lowpass FIR filter must be as close as possible to the essential bandwidth of the baseband signal to minimize the effect of noise. Of course then one also has to consider what essential bandwidth is being specified. Instead of doing a lot of mathematical calculations I decided to look at the PSD of the received signal again. It can be seen that most of the signal energy is below 200kHz. Therefore a good cutoff frequency for the lowpass filter is 200kHz.

After the baseband signal is recovered it must be averaged. Averaging the signal over the duration of the pulse width will minimize the effect of noise. This is different from simply sampling the recovered signal. If sampling was used, and there is a possibility that the amount of noise at that point is greater than the recovered signal, in this case the wrong value would be sampled. On other hand the signal was averaged over time, then the average of the added noise must have a greater magnitude that then average of the signal in order for an error to be caused. Implementation of the averaging circuit in MATLAB involves cutting off the delay. Then the signal is reshaped to have a n rows, and m columns, where n is number of samples per pulse, and m is number of pulses. Each row is summed together, and then divided by n.

The averaged signal is a recovered polar NRZ signal. Although the values of the polar NRZ signal will
not be $\pm 1$, the digital signal can still be recovered by sampling the averaged signal and comparing to 0. If the averaged signal sample is greater than 0, then a digital '1' is received. If the averaged signal sample is less than 0, a digital '0' is received. It doesn't really matter where the averaged signal is sampled so long as it is not sampled close to a rising or falling edge. If the signal is sampled too close to an edge, the sampler might put out the wrong value. I chose the middle of each pulse to fulfill this requirement. A block diagram of the digital demodulator is shown below.

---

**D. D/A Conversion**

The last component that needed to be designed in the digital communication system was the D/A converter. This converter will take the digital stream of data, and output an analog signal that can be viewed by the user on the receiving end of the communication system. The D/A that I designed takes received stream of numbers, looks for detected errors by using parity, attempts to correct the errors. After the errors have been corrected, the bits are then converted in the analog values. The six vectors or analog values are recombined to form a received picture. MATLAB code for the D/A Conversion can be found on pages 28 through 30.

Each stream of digital information coming out of the digital demodulator is reshaped, to form a matrix with six rows, and total number of bits divided by six columns. This action is similar to a serial to parallel digital buffer in electronics. The sixth bit is the parity bit and is used to determine which sequence of numbers is in error. That is, if the sum of every row does not add up to an even number, then there is an error in that group of bits. Every error that is detected using parity, is corrected by taking the average of the two adjacent groups. For example consider the stream, ‘111111’, ‘101100’, ‘111010’. The last bit in each group of six bits is the parity bit. The decimal of this binary stream is ‘31’, ‘24’, ‘29’. It can be seen that the middle number has an error because its bits do not add up to an even number. The middle number then becomes the average of the two adjacent numbers(‘31’, and ‘29’) which is equal to ‘30’. This is implemented in MATLAB by first detecting which groups of six have errors and which don’t. The errors are marked by a vector of ‘1’s and ‘0’s, where a ‘1’ denotes a corresponding error. The error is corrected later in the D/A after the analog output is generated. This is done by...
taking the previous next elements in the analog vector and averaging them.

Although this averaging idea sounds promising in theory, in practice it only helps out a little bit. There are still things that can go wrong. One thing that can go wrong is if the parity bit gets corrupted. This would normally be a benign issue if the adjacent bit groups were not corrupted. Another thing that could go wrong is if even number of bits were corrupted in a bit group. In this case, the corrupted bit groups isn't detected and therefore would not be corrected. Another thing that could go wrong is if there were more than two corrupted groups next to each other. In this case the corrupted bit groups are not not fully corrected. But the averaging still helps a little bit. The worst case scenario would be if there were was a undetected corrupt bit group adjacent to a bit group with a corrupted parity bit. In this case, the bit group with two corrupted bits remains unchanged. Unfortunately the bit group next to it, which only had a corrupted parity bit, becomes the average of the corrupted bit group and another adjacent bit. However it can be shown experimentally that if the power to noise ratio is low enough, the averaging-of-adjacent-bit-groups method does help a little bit. Shown below are three pictures, the original image, the received image without average correction, and the received image with average correction. The bits were transferred at 450kbps, with 24 bits per pixel, and a power to noise ratio of 1.2.

The last thing that needed to be done in the D/A converter was to generate the analog values for each digital value. If no quantization compression is used, this is simply done by ignoring the parity bit, and using a binary to decimal routine. This is what was done originally when I didn't use any quantization compression. Each digital stream of bits was brought in, and reshaped to form a matrix with eight rows, and as many columns as there were pixels. For each column the sum of 2^(8-n) times the binary value in each row is taken; where n is the row number. The analog vectors were reshaped to form have the same number of rows and columns as the input picture. Lastly the six analog matrices are recombined to form the final reconstructed picture. The two 'red' matrices are grouped for from one larger red matrix, with twice as many rows. The same is done for the 'green' and 'blue' matrices. The final picture matrix is a three dimensional matrix; the 'red', 'green' and 'blue' matrices are placed in first, second, and third layers of the three dimensional matrix respectively.

If fewer quantization levels are used then this process remains the same, except for the digital to analog vector conversion. In this case the number of bits per color sample is used rather than eight, in reshaping the binary digital stream. So if five bits were used in the A/D quantization process, the binary stream is reshaped to have five rows and as many columns as there are pixels sent. For each column the sum of 2^(b-n) times the binary
value in each row is taken; where $b$ is the number of bits per sample and $n$ and is the row number. This vector is scaled by $2^{8-b}$; where $b$ is the number bits used in the A/D quantization. For example if 5 bits were used in quantization, then the vector will be scaled by 8. A final block diagram of the D/A converter is shown below.

![Block Diagram of D/A Converter](image)

E. Sending the picture in packets.

In order to prevent buffer overflow from occurring at the receiver. The original picture was divided up into many smaller pictures. This was originally done because MATLAB would run out of memory if I sent the entire picture at once. I chose to divide the picture up into 100 smaller pictures and send each smaller picture through the channel one at a time. This can be implemented using the 'imread' function and some parameters that specify what region of the picture is being sent in each packet. At the receiving end, the same can be done using 'imwrite'.

F. Fine tuning the System

It was required that the communication system achieve the lowest power to noise ratio and have the shortest possible transfer time as possible. It works out that as the transfer time is decreased (by increasing the bit rate), tolerance to noise is also decreased and one must increase the power to noise ratio. A series of tests were done to optimize power to noise ratio and the total transfer time. Also a series of test were conducted to determine the fewest number of quantization levels required before a there was a noticeable degradation in picture quality.

The first series I conducted were to find out what the maximum transfer rate was without any noise. These test were conducted using 8 bit quantization, with transfer rates at 400kbps, 500kbps, 600kbps and 550kbps. This was done until transient errors began occurring. There is a limit of 1Mbps, at the point the transfer rate would be equal to the lowest carrier frequency and the message could not be modulated using a 1 MHz carrier.

<table>
<thead>
<tr>
<th>Bit Rate</th>
<th>Error Count</th>
<th>Transfer Time</th>
<th>Qualitative View</th>
</tr>
</thead>
<tbody>
<tr>
<td>400kbps</td>
<td>0 bit errors</td>
<td>0.5789s</td>
<td>No errors</td>
</tr>
<tr>
<td>500kbps</td>
<td>12 bit errors</td>
<td>0.4632s</td>
<td>No errors can be detected</td>
</tr>
<tr>
<td>550kbps</td>
<td>5776 bit errors</td>
<td>0.4169s</td>
<td>Some bad pixels are noticed</td>
</tr>
<tr>
<td>600kbps</td>
<td>10653 bit errors</td>
<td>0.3938s</td>
<td>Many bad pixels are noticed</td>
</tr>
</tbody>
</table>
After it was determined that maximum data transfer rate I should use without taking noise into consideration was 500kbps. This speed would need to be changed once noise was taken into consideration. The next series of tests were conducted to see what a kind of power to noise ratios could be achieved using different transfer rates. Power was kept at a value of 3. Data transfer rates of 400kbps, 450kbps, and 500kbps were all tested. In each case, the power to noise ratio lowered until a detection threshold was reached.

<table>
<thead>
<tr>
<th>Bit Rate</th>
<th>Transfer Time</th>
<th>Minimum P/N0</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>400kbps</td>
<td>0.5789s</td>
<td>1.3633</td>
<td>1351</td>
</tr>
<tr>
<td>450kbps</td>
<td>0.5095</td>
<td>1.875</td>
<td>1320</td>
</tr>
<tr>
<td>500kbps</td>
<td>0.4632</td>
<td>6.000</td>
<td>1084</td>
</tr>
</tbody>
</table>

The optimum power to noise ratios for data transfer at 400kbps and 450kbps are similar, and very low. This is much better than the best power to noise ratio when data is transferred at 500kbps. I chose to have a data transfer rate of 450kbps because I felt that waiting the extra 69.4ms to have a slightly better power to noise...
ratio wasn't worth it. Therefore data will be transferred in my communication system at 450kbps, with signal power equal to 3. The maximum rated power to noise ratio is 1.875. Below are the sent and received pictures at the optimum power to noise ratio.

Original Picture

Received Picture
With Average Correction
400kbps, P/NO = 1.9633, 24 bits/pix

Original Picture

Received Picture
With Average Correction
450kbps, P/NO = 1.875, 24 bits/pix

Original Picture

Received Picture
With Average Correction
500kbps, P/NO = 6, 24 bits/pix
The final thing that had to be done in the fine tuning stage of my project was to determine how many quantization levels should be used. That is, how many bits should be used to quantize the image. To determine this I did a qualitative analysis by comparing the received image when using 4-bit, 5-bit, 6-bit, 7-bit, and 8-bit quantization. The analysis was done with a power to noise ratio of 1.875, with signal power at 3, data transfer rate of 450kbps, and averaging of errors on. Below is a table illustrating the decrease in transfer time, and a qualitative analysis of the received picture.

<table>
<thead>
<tr>
<th>Bits per Sample</th>
<th>Transfer Time</th>
<th>Quality of Received Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>0.5095s</td>
<td>Same as original picture</td>
</tr>
<tr>
<td>7 bits</td>
<td>0.4529s</td>
<td>No noticeable change</td>
</tr>
<tr>
<td>6 bits</td>
<td>0.3963s</td>
<td>No noticeable change</td>
</tr>
<tr>
<td>5 bits</td>
<td>0.3398s</td>
<td>Not really noticeable</td>
</tr>
<tr>
<td>4 bits</td>
<td>0.2832s</td>
<td>Noticeable degradation of picture</td>
</tr>
</tbody>
</table>

Five bit quantization was determined to be smallest quantization that could be used without a noticeable change in picture quality. Therefore I decided to use five bit quantization for each color, or a total of 15 bits per pixel. The final transfer time of the image with 5 bit quantization is 0.3398s.
Part III: Comments and Conclusions

The digital communication system that I designed can send the image of Darth Vader in 339.8ms. This short transmission time was achieved by using a number of both conventional and novel approaches. The A/D converter used 5 bit quantization rather than 8 bit quantization. Although this produced few colors, the user will not notice the change in picture quality. A parity bit was also added to aid in marking the location of a corrupted bit. The A/D also divides the picture into six digital stream. The digital demodulator takes each digital stream, and converts it to polar NRZ. The six polar NRZ signals are then modulated to six different carrier frequencies using DSB-SC, summed together and transmitted through the channel. On the receiving end the digital demodulator takes transmitted signal, and passes it through a series of bandpass filters tuned in each carrier frequency ± 200kHz. These six filtered signals are then multiplied by a local carrier reference that divides the signal up into a baseband component and an double frequency component. The double frequency component is filtered out by a lowpass filter tuned to 200kHz. The resulting signal is averaged over the time duration of a single pulse width. That averaged signal is then compared to 0 to determine whether the pulse is a digital '1' or '0'. The D/A takes the digital stream and makes the conversion to analog by using a standard binary to decimal conversion and scaling the result by a factor of eight. The errors are also detected by the D/A by checking for even parity. These errors are corrected by averaging the two adjacent pixels.

The communication system that I designed transmits a signal with an average power of 3. The best power to noise ratio at this power is 1.875. Data is transferred at a rate of 450kbps per digital signal. The six digital signals combined make a total transfer rate of 2.7Mbps. There are 5 bits per color per pixel being transferred over the channel or 15 bits per pixel. There is also 1 parity bit transferred per pixel per color. The total number of bits transferred is 925740. The total transmission time over the channel, with the channel delay included, is 339.8ms. Other things that I would have liked to implement for this project include QAM, more sophisticated error correction coding, and some variant of DCPM.

Overall I I found the project to be enjoyable. Although I did find that the 75% of the work involved in this project was trying to make some of the concepts to work in MATLAB. I would hope to see more cool MATLAB projects in future classes I take.
Part IV: References


Part V: Appendix

A. Final project function

```matlab
function[] = project() 

simufreq = 10e6; 
simuperiod = 1/simufreq; 
bitrate = 450e3
amplitude = 1;
N0 = 1.6;
pixel_bit = 5;

error_count = 0;
total_time = 0;
tot_power = 0;
txpicture = imread('vader.tif');
figure
subplot(1, 2, 1)
imshow(txpicture)
title({'Original Picture'; ''})
drawnow

rxpicture1 = zeros(size(txpicture));

for m = 0:8
    for k = 0:8
        txpart1 = imread('vader.tif', 'PixelRegion',{[28*k+1, 28*k+28], [19*m+1, 19*m+19]});
        [rxpart1, errors, time, power] = send(txpart1, simufreq, simuperiod, bitrate, amplitude, N0, pixel_bit);
        error_count = error_count + errors;
        total_time = total_time + time;
        tot_power = power + tot_power;
        rxpicture1(28*k+1:28*k+28, 19*m+1:19*m+19, :) = rxpicture1;
        rxpicture1 = uint8(rxpicture1);
        subplot(1, 2, 2)
imshow(rxpicture1)
drawnow
    end
    txpart1 = imread('vader.tif', 'PixelRegion',{[28*9+1, 28*9+26], [19*m+1, 19*m+19]});
    [rxpart1, errors, time, power] = send(txpart1, simufreq, simuperiod, bitrate, amplitude, N0, pixel_bit);
    error_count = error_count + errors;
    total_time = total_time + time;
    tot_power = power + tot_power;
    rxpicture1(28*9+1:28*9+26, 19*m+1:19*m+19, :) = rxpicture1;
    rxpicture1 = uint8(rxpicture1);
    subplot(1, 2, 2)
imshow(rxpicture1)
drawnow
end

m = 9;
for k = 0:8
    txpart1 = imread('vader.tif', 'PixelRegion',{[28*k+1, 28*k+28], [19*m+1, 19*m+14]});
    [rxpart1, errors, time, power] = send(txpart1, simufreq, simuperiod, bitrate, amplitude, N0, pixel_bit);
    error_count = error_count + errors;
    total_time = total_time + time;
    tot_power = power + tot_power;
end
```

p. 16
rxpicture1(28*k+1:28*k +28, 19*m+1: 19*m+14, :) = rxpatic1;
rxpicture1 = uint8(rxpatic1);
subplot(1, 2, 2)
imshow(rxpatic1)
drawnow
end
txpatic1 = imread('vader.tif', 'PixelRegion', {[28*9+1, 28*9+26], [19*m+1, 19*m+14]});
[rxpatic1, errors, time, power] = send(txpatic1, simufreq, simuperiod, bitrate, amplitude, N0, pixel_bit);
error_count = error_count + errors;
total_time = total_time+ time;
tot_power = power + tot_power;
rxpicture1(28*9+1 : 28*9+26, 19*m+1: 19*m+14, :) = rxpatic1;
rxpicture1 = uint8(rxpatic1);

imwrite(rxpatic1, 'rxpicture.tif')
error_count
total_time
power = tot_power/100

subplot(1, 2, 2)
imshow(rxpatic1)
title({'Received Picture';'With Average Correction'; num2str(bitrate/1000), 'kbps, P/N0 = ', num2str(power/N0), ', ', num2str(3*pixel_bit), ' bits/pix'})
drawnow
B. 'send' function

```matlab
function [rxanasig1, error_count, time, power] = communications(txanasig, simufreq, simuperiod, bitrate, amplitude, N0, pixel_bit)

[row, col, dep] = size(txanasig);
delay = 0;
error_count = 0;
power = 0;

% Carrier frequencies
fcr1 = 1.000e6;
fcr2 = 1.500e6;
fcr2 = 2.000e6;
fcr2 = 2.500e6;
fcr2 = 3.000e6;
fcr2 = 4.000e6;

pulsewidth = single(int16(simufreq/bitrate));

% Separate Picture in 6 Streams
[antr1, antg1, antb1, antr2, antg2, antb2] = separate(txanasig);

% Converts analog streams into unsigned binary code
[txbinr1, color_div] = a2d(antr1, pixel_bit);
[txbinr2, color_div] = a2d(antr2, pixel_bit);
[txbing1, color_div] = a2d(antg1, pixel_bit);
[txbing2, color_div] = a2d(antg2, pixel_bit);
[txbinb1, color_div] = a2d(antb1, pixel_bit);
[txbinb2, color_div] = a2d(antb2, pixel_bit);

% Convert digital streams to polar NRZ
txmesr1 = polarnrz(txbinr1, pulsewidth);
 txmesr2 = polarnrz(txbinr2, pulsewidth);
 txmsg1 = polarnrz(txbing1, pulsewidth);
 txmsg2 = polarnrz(txbing2, pulsewidth);
 txmsgb1 = polarnrz(txbinb1, pulsewidth);
 txmsgb2 = polarnrz(txbinb2, pulsewidth);

% Modulate polar NRZ signals to carrier frequencies
txsigr1 = moddsbsc(txmesr1, fcr1, simuperiod, amplitude);
 txsigr2 = moddsbsc(txmesr2, fcr2, simuperiod, amplitude);
 txsigg1 = moddsbsc(txmsg1, fcg1, simuperiod, amplitude);
 txsigg2 = moddsbsc(txmsg2, fcg2, simuperiod, amplitude);
 txsigb1 = moddsbsc(txmsgb1, fcb1, simuperiod, amplitude);
 txsigb2 = moddsbsc(txmsgb2, fcb2, simuperiod, amplitude);
```

% Sum All DSB-SC Signals together prior to transmission
txsignal = txsigr1;

%--------------------------------------------------------------
%Transmit Message over channel
rxsignal = channel(txsignal, N0);
time = max(size(rxsignal))/simufreq;
power = sum(txsignal.^2)/max(size(txsignal));
delay1 = (max(size(rxsignal)) - max(size(txsignal)))/2;

%--------------------------------------------------------------
%Demodulate Message

[rxmesr1, delay2] = demod(rxsignal, fcr1, delay1, simuperiod, simufreq);
[rxmesr2, delay2] = demod(rxsignal, fcr2, delay1, simuperiod, simufreq);
[rxtsg1, delay2] = demod(rxsignal, fcg1, delay1, simuperiod, simufreq);
[rxtsg2, delay2] = demod(rxsignal, fcg2, delay1, simuperiod, simufreq);
[rxtsb1, delay2] = demod(rxsignal, fcb1, delay1, simuperiod, simufreq);
[rxtsb2, delay2] = demod(rxsignal, fcb2, delay1, simuperiod, simufreq);

%--------------------------------------------------------------
%Take average and convert recovered average into unsigned binary code

rxbinr1 = binconvert(rxmesr1, pulsewidth, delay2);
rxbinr2 = binconvert(rxmesr2, pulsewidth, delay2);
rxbing1 = binconvert(rxtsg1, pulsewidth, delay2);
rxbing2 = binconvert(rxtsg2, pulsewidth, delay2);
rxbinb1 = binconvert(rxtsb1, pulsewidth, delay2);
rxbinb2 = binconvert(rxtsb2, pulsewidth, delay2);
error_count = error_count + sum(abs(rxbinr1 - txbinr1));
error_count = error_count + sum(abs(rxbinr2 - txbinr2));
error_count = error_count + sum(abs(rxbing1 - txbing1));
error_count = error_count + sum(abs(rxbing2 - txbing2));
error_count = error_count + sum(abs(rxbinb1 - txbinb1));
error_count = error_count + sum(abs(rxbinb2 - txbinb2));
% Convert binary analog streams
[anrr11] = d2a(rxbinr1, pixel_bit, color_div);
[anrr21] = d2a(rxbinr2, pixel_bit, color_div);
[anrg11] = d2a(rxbing1, pixel_bit, color_div);
[anrg21] = d2a(rxbing2, pixel_bit, color_div);
[anrb11] = d2a(rxbinb1, pixel_bit, color_div);
[anrb21] = d2a(rxbinb2, pixel_bit, color_div);

%--------------------------------------------------------------
% Recombine all streams to form a sign analog signal
rxanasig1 = combine(anrr11, anrg11, anrb11, anrr21, anrg21, anrb21, row, col, dep);
rxanasig1 = uint8(rxanasig1);
C. 'separate' function

function [ sigr1, sigg1, sigb1, sigr2, sigg2, sigb2 ] = separate(or_sig)

sigr = or_sig(:,:,1);
sigg = or_sig(:,:,2);
sigb = or_sig(:,:,3);

sigr1 = sigr( 1: int16(size(sigr, 1)/2), : );
sigr2 = sigr( 1+ int16(size(sigr, 1)/2) : size(sigr, 1), : );
sigg1 = sigg( 1: int16(size(sigg, 1)/2), : );
sigg2 = sigg( 1+ int16(size(sigg, 1)/2) : size(sigg, 1), : );
sigb1 = sigb( 1: int16(size(sigb, 1)/2), : );
sigb2 = sigb( 1+ int16(size(sigb, 1)/2) : size(sigb, 1), : );

sigr1 = reshape( sigr1, 1, prod(size(sigr1))); sigr2 = reshape( sigr2, 1, prod(size(sigr2)));
sigg1 = reshape( sigg1, 1, prod(size(sigg1))); sigg2 = reshape( sigg2, 1, prod(size(sigg2)));
sigb1 = reshape( sigb1, 1, prod(size(sigb1))); sigb2 = reshape( sigb2, 1, prod(size(sigb2)));

D. 'a2d' function

function [ out, color_div ] = a2d( stream, n )

color_div = 2^(8-n);
stream = single(uint8(stream/color_div));

for k = 1:n
    a = stream + 1 - 2^(n-k);
    a = (a + abs(a))/2;
    b(k,:) = single( and(a, stream) );
    g = single(b*2^(n-k));
    stream = stream - g(k,:);
end

%parity vector creation
sumbits = sum(b, 1);
fix(sumbits/2);
odds = sumbits - 2*fix(sumbits/2);
b(n+1,:) = odds;

out = b;
out = reshape(out, 1, (1 + n)*max(size(stream)));
E. 'polarnrz' function

function message = polarnrz(bits, pulsewidth)

length = max(size(bits));

message = ones(pulsewidth, length);

for k = 1:pulsewidth
    message(k,:) = 2*bits(1,:)-1;
end

message = reshape(message, 1, pulsewidth*length);
F. 'moddsbsc' function

function [out] = moddsbsc(message, fc, simuperiod, A)

    time = simuperiod * max(size(message));
    t = zeros(1, int16(time/simuperiod));
    t = [0 : simuperiod : time - simuperiod];

    out = zeros(1, int16(time/simuperiod));
    out = A*cos(2*pi*fc*t);
    clear t
    out = message.*out;
G. 'channel' function

% Channel function
% assume sampling rate = 10MHz
% x: transmitted signal vector (row vector)
% N0: noise spectral density
% needs signal processing toolbox to run

function v=channel(x,N0);

a=fir2(100,[0 0.2 0.4 0.65 0.70 0.8 1],[0 0.9 1 0.8 1e-2 1 0]);
b=fir1(128,[0.15 0.85]);
h=conv(a,b);
h=h(99:131);
h=h/norm(h);

n=length(x)+33-1;
v=conv(x,h)+sqrt(N0/2)*randn(1,n);
H. 'demod' function

function[out, delay2] = demod(signal, fc, delay1, simuperiod, simufreq);
orig_sig_size = max(size(signal));
carrier_period = 1/fc * simufreq;

passwidth = 400e3; %285.714e3
pass = [(fc -passwidth/2)/simufreq, (fc + passwidth/2)/simufreq];
[b] = fir1(6, pass, 'bandpass');
signal = conv(b, signal);
delay = (max(size(signal)) - orig_sig_size)/2 +delay1;

t = zeros(1, max(size(signal)));
t = 0:simuperiod: simuperiod * (max(size(signal)) - 1);
loccarrier = zeros(1, max(size(signal)));
loccarrier = cos(2*pi*fc*t - 2*pi*delay/carrier_period);
clear t

signal = loccarrier.*signal;
clear loccarrier

[b] = fir1(12, 5e3*2/simufreq, 'low');
out = conv(b, signal);
delay2 = delay1 + (max(size(out)) - orig_sig_size)/2;
clear signal
I. 'binconvert' function

function [output] = binconvert(input, pulsewidth, delay)

input1 = zeros(1, max(size(input)) -2*delay);
input1 = input( delay+1 :(max(size(input))) -delay);
length = max(size(input1))/ pulsewidth;
pulsewidth = int16(fix(pulsewidth));
length = int16(fix(length));

input1 = reshape(input1, pulsewidth, length);
average = ones(pulsewidth, length);

for k = 1:pulsewidth
    pulsewidth = single(pulsewidth);
    average(k,:) = sum( input1, 1 )/pulsewidth ;
end

average = average(int8(pulsewidth/2),:);
output = and((average + abs(average))/ 2, average);
clear average
output = single(output);
J. 'd2a' function

function[out] = d2a(bits, bitsperbyte, color_div)

    temp = reshape(bits, bitsperbyte +1, max(size(bits))/(bitsperbyte+1));
    temp = flipud(temp);

    out = zeros(1, max(size(temp)));
    for k = 2:bitsperbyte+1
        out = out + temp(k, :)\*2^(k-2);
    end
    out = color_div\*out;
    bittot = sum(temp, 1);
    error = bittot - 2*fix(bittot/2);
    out = correct(out, error);
K. 'correct' function

function [out1] = correct(in, error, col)

    out1 = in;
    for k = 1:10
        for n = 2 : max(size(in))-1
            if error(n) == 1
                out1(n) = (in(n-1) + in(n+1))/2 ;
                n = 2.1;
            end
        end
    end
L. ‘combine’ function

function [rxanasig] = combine(anar1, anag1, anab1, anar2, anag2, anab2, row, col, dep)

    anar1 = reshape(anar1, row/2, col);
    anar2 = reshape(anar2, row/2, col);
    anag1 = reshape(anag1, row/2, col);
    anag2 = reshape(anag2, row/2, col);
    anab1 = reshape(anab1, row/2, col);
    anab2 = reshape(anab2, row/2, col);

    rxanasigr = [anar1; anar2];
    rxanasigg = [anag1; anag2];
    rxanasigb = [anab1; anab2];

    %rxanasigr = reshape(rxanasigr, row, col);
    %rxanasigg = reshape(rxanasigg, row, col);
    %rxanasigb = reshape(rxanasigb, row, col);

    rxanasig(:,:,1) = rxanasigr;
    rxanasig(:,:,2) = rxanasigg;
    rxanasig(:,:,3) = rxanasigb;