Lab 5

Clint Olsen

ECEN 4532: DSP Lab

April 20, 2018

University of Colorado
Boulder

Contents

Introduction **2**

Assignment 1 **2**

Assignment 2 **3**

Assignment 3 **4**

Assignment 4 **6**

Assignment 5 **10**

Assignment 6 **12**

Assignment 7 **12**

Assignment 8 **13**

Code Appendix **16**

Introduction

As an extension from the previous lab, this lab covers a standard compression technique for yet another popular form of media stored on computers: images. JPEG is an important and well defined image compression algorithm that has many variants, but a simple version of the algorithm will be explored in this lab in two main parts: the encoder and decoder. Besides understanding the algorithm itself, the other important analysis piece of this lab is to understand and witness the effects of various quantization methods. From there, the effects of the compression can be seen visually and a metric can be created for how "good" the image needs to be in terms of quality to be acceptable. This is dependent on many factors which is really dependent on the application, all the way from personal preference on personal images to necessary high quality in medical imaging. This lab explores this algorithm and analysis using MATLAB and various example images for visual feedback on the techniques used.

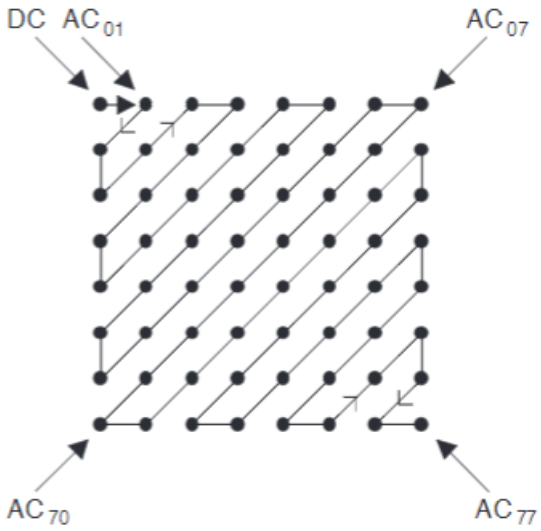
Assignment 1

The first half of the JPEG algorithm is the encoder step which, on a basic level, prepares the image for quantization and compression later in the algorithm. The images used in this lab are all of size 512x512 and to further speed up the processing of the images, 8x8 blocks will be processed at a time. This processing dimension is from the JPEG standard itself and is based off of the statistical and arbitrary choice that pixels within an 8x8 block will on average be similar enough to represent that area of the image. From here, the image can be sent block by block through the first part of the processing, which is the discrete cosine transform. This transformation is the standard method for the original versions of JPEG, where newer versions of the standard, such as JPEG 2000, use the discrete wavelet transform. The purpose of the transform is to extract frequency information regarding the image, but the DCT gives the results as real coefficients that represent the energy well of highly correlated data, such as that of nearby pixels in an image. The DCT is defined below:

$$F_{u,v} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}, \quad u, v = 0, \ldots 7$$

In order to simplify the programming and utilize the functionality of MATLAB, the basis vectors of size 8x8 for the DCT can be stored into a matrix which simplifies the computation for the DCT. The simple result is the multiplication of 3 matrices which MATLAB makes easy. These three matrices are the DCT basis vectors, the 8x8 block from the original image, and the transpose of the DCT basis vectors AXA^T. Having both the normal and transpose of the DCT vectors results in the 8x8 block being multiplied by all the rows and columns demanded by the definition of the transform. The result is a 8x8 block of transform values which is then fed into the next portion of the processing. From here, the 8x8 block of transformed values is ran through a zig zag scan algorithm which is an

interesting data organization which transforms the block into a single vector with the lower frequency components of the block near the beginning of the vector and the higher frequency components near the end. This method is shown below:

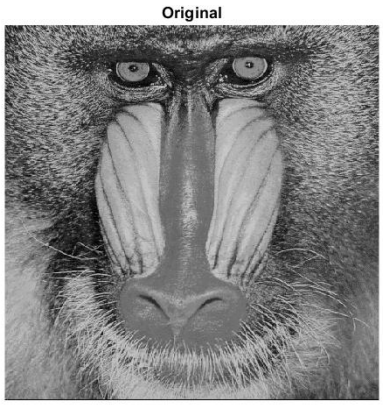


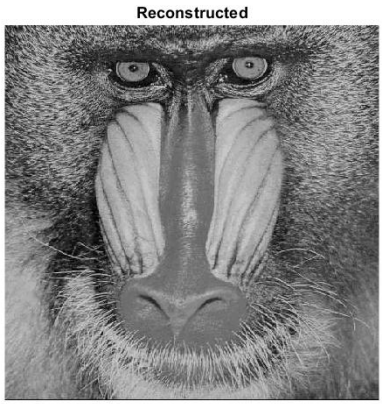
This ordering is convenient for the fact that on average images will have most of the energy contained in the lower frequencies, so this makes the access to them in an array quick and simple. After this processing has been done on every 8x8 block in the image from top to bottom, left to right, the result is a matrix of transform coefficients where each column is a length 64 vector of the coefficients of a 8x8 block formatted in the zig zag scan fashion. The final step in the setup for quantization and compression is performing encoding on the DC (zero frequency) components of each block, which once the coefficient matrix is created, is simply encoding the first row in the matrix. This is a differential encoding that is performed between neighboring DC coefficients to provide a basic way to encode how different neighboring DC parts are. All of this functionality is encompassed in the *dctmgr*() function which can be seen the the Assignment 1 Code Appendix.

Assignment 2

At this point in the lab, it was important to test the ability to undo the processing that was performed in the previous step before adding the complications of the quantization and arithmetic encoding. Therefore, the inverse decode step was implemented next. This follows a very similar path to the previous assignment, but in previous order. The first step to reversing this process is to undo the DC differential encoding on the first row of coefficients in the coefficients matrix, which consists of just adding back the value subtracted in the forward method. From here, the coefficient matrix is processed column by column (block by block). First, the column is passed through and inverse zig zag scan to convert the vector back into the correctly ordered 8x8 block that was originally used in the forward transform. This block is then passed into the inverse DCT, which using MATLAB is another simple matrix

multiply $A^T XA$ which is the opposite operation as before which projects the transformed values back onto the original set of basis vectors. This is performed on each column of the coefficients matrix until the entire image has been reconstructed. Below shows the original and reconstructed image for the *mandril.pgm* image, which is not interesting at this point visually because it is the same image, but verifies that the transform works both directions:





Similarly to the previous example, this functionality is wrapped into a function called *idctmgr*() which can be seen in the Assignment 2 Code Appendix.

Assignment 3

The next step is where the interesting visual impacts begin to occur within JPEG, which is the quantization and inverse quantization. The purpose of quantization is to map certain values within a domain of numbers to be represented with a smaller subset of numbers. This is really the compression is action because the idea later will be to represent close to the same images with less bits and therefore less memory. The quantization step is very important in any compression algorithm and if values are handled incorrectly, it can result in unnecessary damage to the signal. For the image processing, the following quantization table will be used:

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

and in the forward direction the following quantization scheme will be used:

$$F^q_{u,v} = \text{floor}\left(\frac{F_{u,v}}{\texttt{loss-factor} \times Q_{u,v}} + 0.5\right)$$

After looking at this equation, it can be seen why the quantization table is selected as it is. Since the transformed values are divided by the table values, the lower frequency components in the upper left of the table have less of an effect because as stated before, they hold more energy in the signal. The converse is true with high values in the bottom right corner for high frequencies. The other factor at play here is the loss factor which determines how much the image is compressed by scaling the reduction of the transformed values through multiplying the loss factor by the table. In turn, a higher loss factor results in more visual damage and quantization of the image. A subtle note is that the DC coefficients of the signal are not quantized due to the fact they were already encoded in a different manner with the DC differential encoding in Assignment 1. It should be also noted that this step occurs immediately after the DCT.

When reversing the transform, the inverse quantization needs to occur to recover the image properly. This step occurs immediately before the inverse DCT as is described by the following:

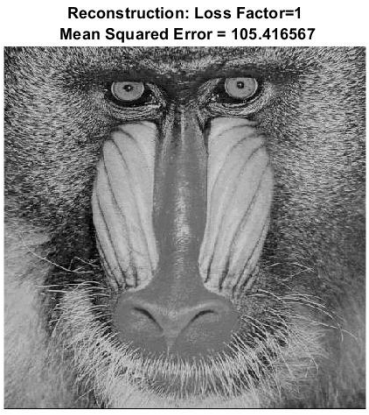
$$\widetilde{F}_{u,v} = F^q_{u,v} \times \texttt{loss-factor} \times Q_{u,v}$$

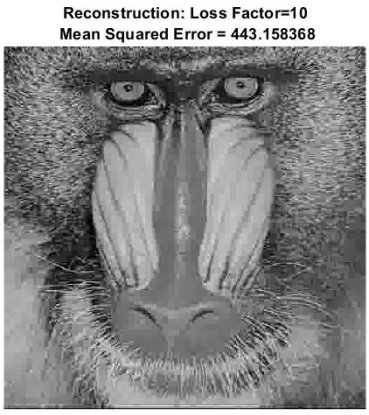
This equation makes sense because it re-multiplies the table and loss factor that were divided in previous step. The integration of this procedure into MATLAB can be seen in the

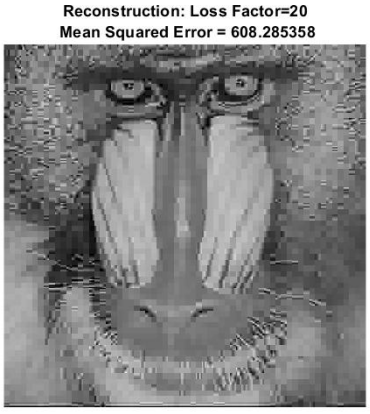
Assignment 3 Code Appendix.

Assignment 4

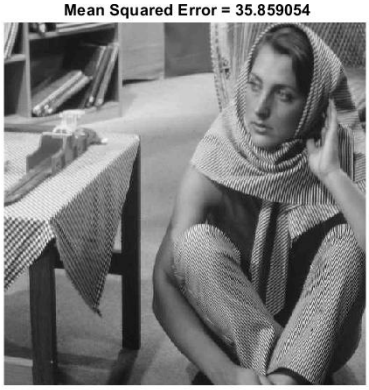
From this point the transform and reconstruction systems can be used to analyze their effect of various images. The quantization is also included with various loss factors where, based on the characteristics of the image itself, can be visualized and quantified. Below is the original and reconstruction for all 6 of the provided images. In the title of the reconstructed image is the mean squared error between the original and reconstructed error. This is a good approximation of the error due to the fact that this gives a rough estimate of average error across the whole image, which is important due to the fact the viewer of the image will recognize blurriness wherever it may occur:







Reconstruction: Loss Factor=1
Mean Squared Error = 35.859054



Reconstruction: Loss Factor=10
Mean Squared Error = 270.174007

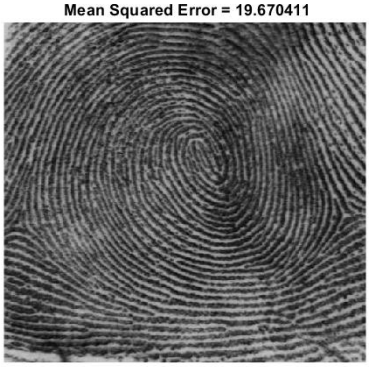
Reconstruction: Loss Factor=20
Mean Squared Error = 355.366893

Reconstruction: Loss Factor=1
Mean Squared Error = 14.547785

Reconstruction: Loss Factor=10
Mean Squared Error = 111.835199

Reconstruction: Loss Factor=20
Mean Squared Error = 203.666680

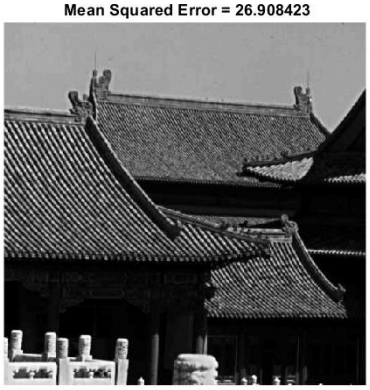
Reconstruction: Loss Factor=1
Mean Squared Error = 19.670411



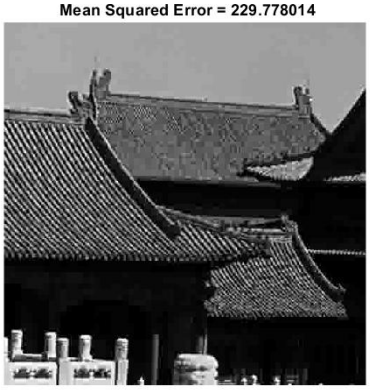
Reconstruction: Loss Factor=10
Mean Squared Error = 202.987351

Reconstruction: Loss Factor=20
Mean Squared Error = 427.542937

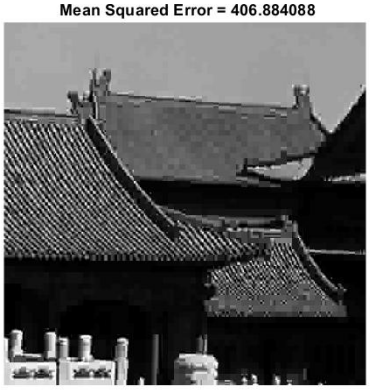
Reconstruction: Loss Factor=1
Mean Squared Error = 26.908423



Reconstruction: Loss Factor=10
Mean Squared Error = 229.778014



Reconstruction: Loss Factor=20
Mean Squared Error = 406.884088



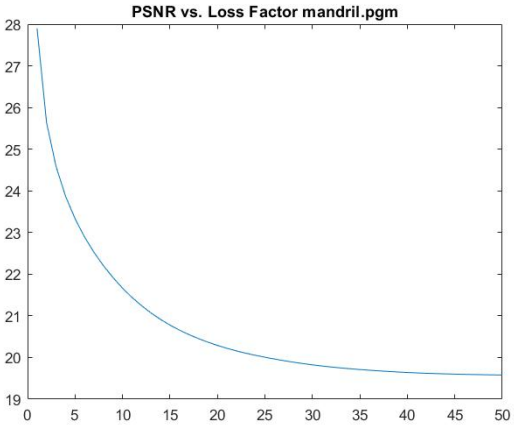
Reconstruction: Loss Factor=1
Mean Squared Error = 176.603679

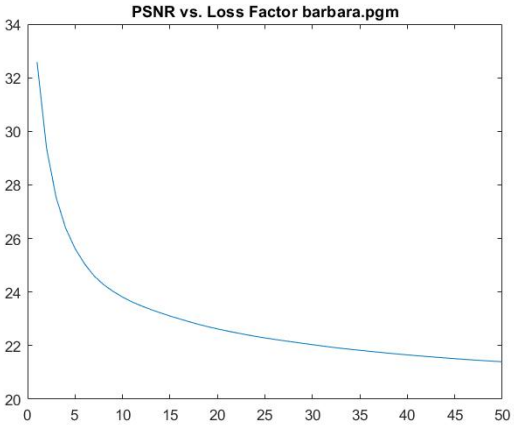
Reconstruction: Loss Factor=10
Mean Squared Error = 1161.502805

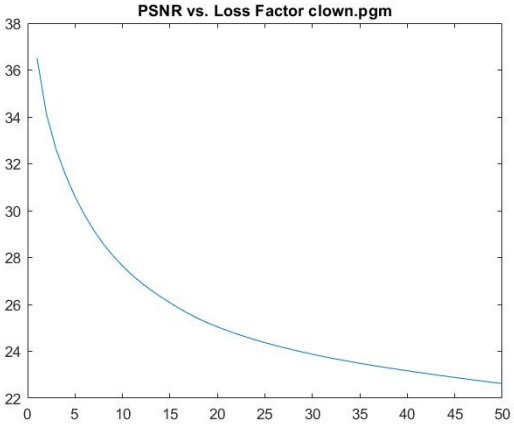
A couple of immediate observations that can be made in visually and numerically analyzing these images is that as the loss factor increases, the blurriness of the image and the mean squared error increases. This is to be expected since the quantization table is increased as the loss factor increases, so when in is used as the divisor, it more greatly affects the image. This is true for all of the images, but in different patterns based on the characteristics of the images. For example, in looking at the *straw.pgm* image, it can bee seen that the error and blurriness quickly increase to a greater magnitude than the other images relatively. This is due to the fact that there are many edges and transitions between the pieces of straw in the image, so when the quantization greatly reduces the high frequencies, this characteristic is lost. This is also true in the fine detail of the hair on the mandril which creates many high frequency transitions. In contrast, images such as the clown and Barbara show a small mean squared error, but at the higher loss factor, are still visually displeasing. These images have more solid colors which create localized areas of similarity, so the error in these areas is small. At a smaller loss factor, the damage from the quantization is also a bit less obvious but still noticeable. Overall this shows that the amount of quantization is really dependent on the application the image will be used in. For personal images that are quickly shown to a friend, the quality may not be completely paramount, especially in a large library, but for a professional photographer this is likely not acceptable at all. The code for plotting the images can be found in the Assignment 2 and 3 section of the Lab 5 Driver Script Code Appendix.

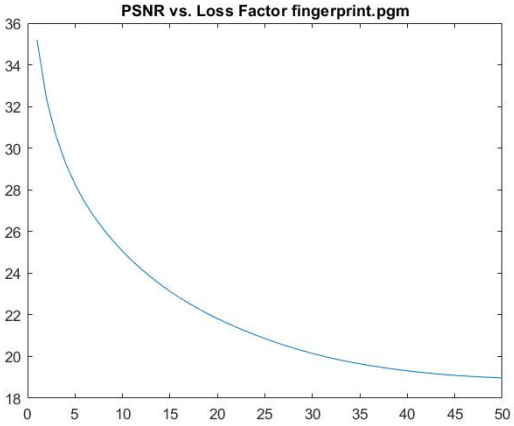
Assignment 5

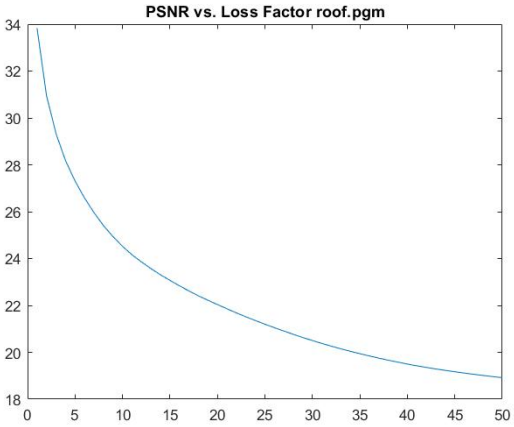
In order to further graphically and numerically describe the arguments regarding quality in the section above, the peak signal to noise ration can be plotted for each of the images. This gives a good representation of how prominent the intended signal in within the reconstructed image over various loss factors. The x axis in these plots is loss factor from 1 to 50, and as expected, the prominence of the signal diminishes as the loss factor increases for all of the signals:

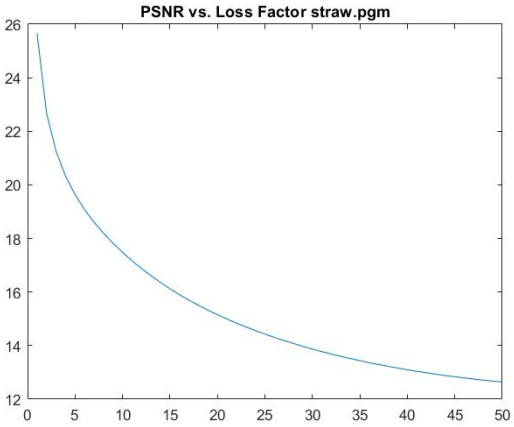

PSNR vs. Loss Factor mandril.pgm
28
27
26
25
24
23
22
21
20
19
0 5 10 15 20 25 30 35 40 45 50


PSNR vs. Loss Factor barbara.pgm
34
32
30
28
26
24
22
20
0 5 10 15 20 25 30 35 40 45 50


PSNR vs. Loss Factor clown.pgm
38
36
34
32
30
28
26
24
22
0 5 10 15 20 25 30 35 40 45 50


PSNR vs. Loss Factor fingerprint.pgm
36
34
32
30
28
26
24
22
20
18
0 5 10 15 20 25 30 35 40 45 50


PSNR vs. Loss Factor roof.pgm
34
32
30
28
26
24
22
20
18
0 5 10 15 20 25 30 35 40 45 50


PSNR vs. Loss Factor straw.pgm
26
24
22
20
18
16
14
12
0 5 10 15 20 25 30 35 40 45 50

First, it can be seen that the assumption of the shape of the curve was correct in that the PSNR decreased as a function of loss factor. In further looking at the graphs, there are many observations that can be made. The first is that for the images described in the previous section as having less immediate effect due to the quantization can be quantified here. For example the clown and Barbara photos begin and end with a larger magnitude SNR in comparison to the photos such as straw. This is again due to the prominence of low frequencies in these photos, with little edges and transitions. This can also be seen visually. The signals greatly affected by the quantization can also be quantified with these graphs, such as the straw and mandril photos which overall have a lower magnitude SNR in comparison due to the granularity of detail within the photos. This comes from the mandril's hair and the many pieces of straw close together. The code for this section can be found in the Assignment 5 section of the Lab 5 Driver Script Code Appendix.

Assignment 6

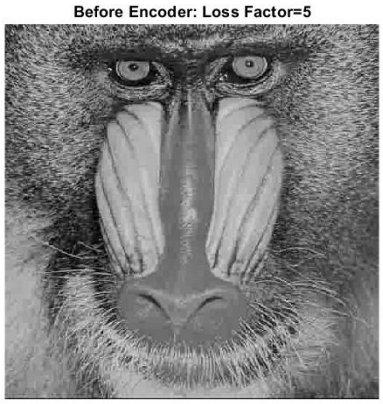
Now that the data has been verified to operate as intended through the system in both directions, further operations can be performed. The next step is to prepare the transformed data after the forward transform and quantization for encoding using an arithmetic encoding. This is done by condensing the representation of the data into a triplet form where the format is [precedingZeros numberOfBits coeffValue]. Therefore, when encoding the data, sequential zeros are ignored and represented when the next non-zero value occurs. In terms of the coefficient matrix, encoding is performed on each column representing a 8x8 block of transform data. However, the result of the algorithm is a single vector of symbols where each representation of a block is ended by a double zero End-of-Block (EOB).

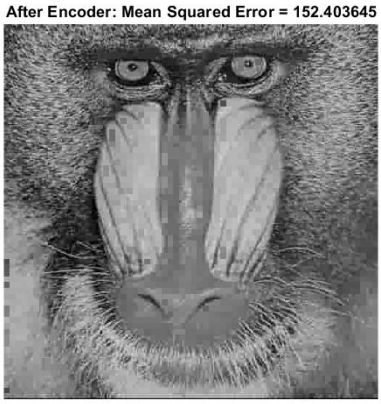
After the arithmetic encoding has been performed, this described process then needs to be reversed. This ultimately means converting the single vector back into a coefficient matrix to be put through the inverse DCT and quantization blocks in the system. This is made easy by the fact the triplets allow parsing the data to be easy and fast for reconverting the data's format. The code for this portion of the lab can be found in the Assignment 6 and Lab 5 Driver Script Code Appendix.

Assignment 7

The final portion of the algorithm as a whole is to perform the arithmetic encoding and decoding on the vector of symbols which is compression. In general encoding is a method of taking a particular symbol and representing it in a different form, where the original intention can still be deciphered with the correct operations. One example that is familiar is Huffman codes which encodes single characters based on their frequency of use to represent common symbols with less memory. Arithmetic coding is a similar approach, but instead of looking at discrete elements of a symbol, we can look at larger structures to encode as a whole. An example would be to encode an entire string instead of just a single character. The operation is performed in the forward direction on the symbol vector using the MATLAB *arithenco*()

function and in the inverse direction using *arithdeco*(). To verify that the forward and inverse directions worked properly the image was plotted before and after to show that it is relatively the same with some error due to the encoding. Below is this process for the mandril photo:

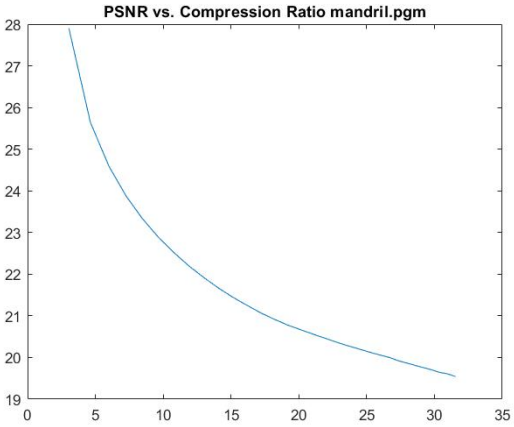


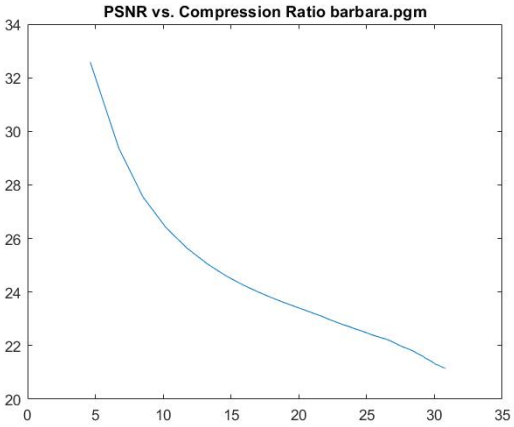


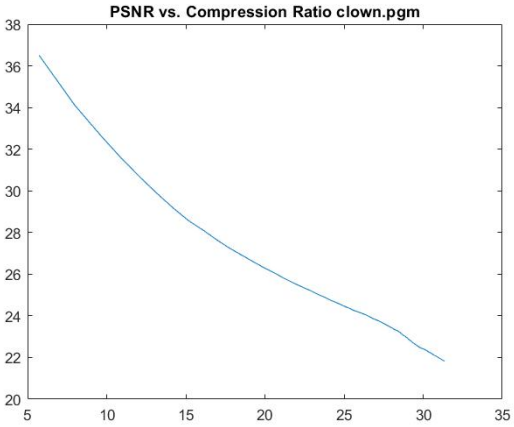
It can be seen that the approach works, but with some error associated with the reconstruction which can be seen by the mean squared error in the image after the encoder and visually by the added blurriness and blocky pixels. This is likely due to some loss caused by trying to compress various parts of the symbol as a whole and undo that operation within the program. This would vary, however, based on the characteristics of the image. The code for this encoding can be found in the Assignment 7 and under the Assignment 7 section in the Lab 5 Driver Script Code Appendix

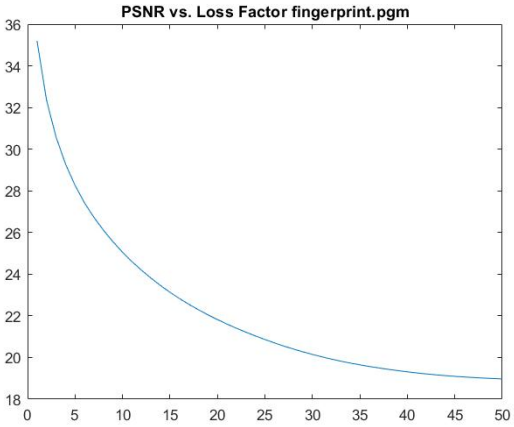
Assignment 8

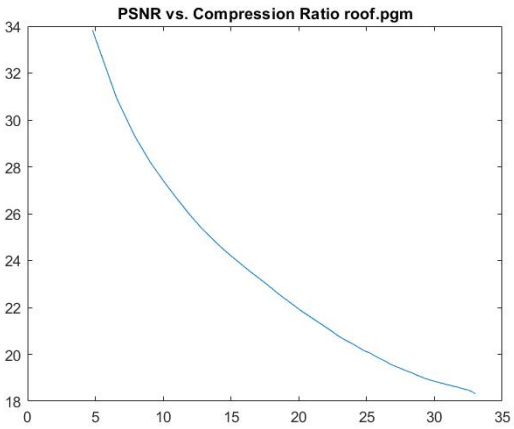
The final portion of this lab consists of analyzing the quality of the arithmetic encoder compression based on processing the 6 test images. This is again quantified and shown graphically using the peak signal to noise ratio, but with the x axis this time as compression ratio of the image as a whole after compression. The expectation is that the PSNR will decrease as the compression ratio increases and look similar to the previous PSNR plots due to the fact that an increased compression ratio results in more damage to the image. Below shows this plot for the 6 images:

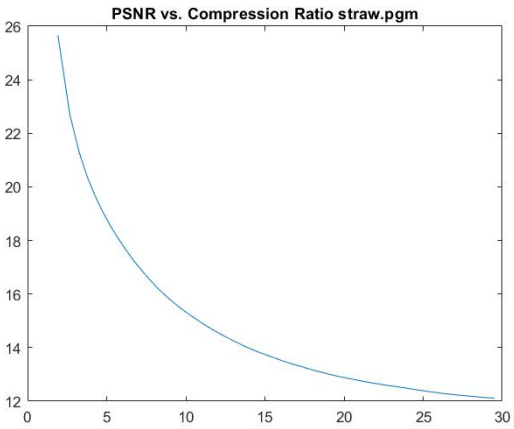

PSNR vs. Compression Ratio mandril.pgm


PSNR vs. Compression Ratio barbara.pgm


PSNR vs. Compression Ratio clown.pgm


PSNR vs. Loss Factor fingerprint.pgm


PSNR vs. Compression Ratio roof.pgm


PSNR vs. Compression Ratio straw.pgm

As expected the PSNR decreases as compression increases which makes sense due to the damage to the signal itself. These plots follow a similar trend to the previous PSNR plots where the straw and mandril photos start and end with a much lower PSNR due to the finer detail in the image. Conversely, all of the others start with higher PSNR values. One interesting observation is the fact that the fingerprint photo has one of the larger signal integrities. By looking at the image, it looks like there are many edges from the ridges on the print that would represent high frequencies that are greatly reduced in the system. However, the solid blacks in the photo must help the signal maintain its PSNR. Another possibility is that the image was already slightly blurry, which it looks like it may be, which would result in a smaller impact by the processing. Another interesting note is how the clown PSNR is the closest to being a linearly decreasing trend. This means that effectively the compression scales with the damage that it does, which would be a very nice characteristic if all images followed this, but is only true for images of certain characteristics. Finally, the roof photo shows a good mixture of edges and transitions in on the roof itself, but also lower frequencies in the solid color of the sky. This explains why, despite the greater detail, this photo also has a larger magnitude. The final conclusion of this analysis is really that the compression and processing needs to be catered to the specific application, which is what the various variants of this algorithm and other image compression attempt to provide. The code for this section can be found in the Assignment 8 section of the Lab 5 Driver Script Code Appendix.

Code Appendix

Lab 5 Driver Script

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: Driver Script
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Setup and Variable Assignment
filename = 'straw.pgm';
im = double(imread(filename));
loss_factor = 5;
N = size(im,1);

%% Assignment 1 and 3: DCTMGR

%Use the DCT to create a matrix of DCT coefficients
coeff = dctmgr(im,loss_factor);

%% Assignment 2 and 3: IDCTMGR
%Reconstruct the transformed coefficients
reconst = idctmgr(coeff,loss_factor);

%Find the mean squared error between the original and reconstruction
error = immse(im,reconst);

%Plot the original and reconstructed images
figure,imagesc(im, [0 255]),colormap gray,axis square, axis off, title('Original');
figure,imagesc(reconst, [0 255]),colormap gray,axis square, axis off,title(sprintf("Reconstruction: Loss Factor=%d\n Mean Squared Error = %f",loss_factor,error));

%% Assignment 5 PSNR
PSNR = zeros(1,50);

%Calculate the PSNR for of an image over 50 loss factors
for i=1:50
    transform = dctmgr(im,i);
    reconst = idctmgr(transform,i);
    PSNR(i) = 10*log10(255^2/((1/N^2)*sum(sum(abs(im(1:N,1:N)-reconst(1:N,1:N)).^2))));
end

%Plot the result of PSNR as a function of loss factor
figure,plot(1:50,PSNR);
title(sprintf("PSNR vs. Loss Factor %s",filename));

%% Assignment 6 Pre-Post Processor

%Compute the triplets and store into int16 vector
symb = int16(run_bits_value(dctmgr(im,loss_factor))');

%Invert the symb array back to a matrix of coeffs
coeff2 = irun_bits_value(symb);

%% Assignment 7 Arithmetic Encoder

[decoded encoded totalbits] = arithEncode(symb);
coeff2 = irun_bits_value(decoded);
reconstructed = idctmgr(coeff2,loss_factor);

%Find the mean squared error between the original and reconstruction
encError = immse(reconstructed,reconst);

figure,imagesc(reconst, [0 255]),colormap gray,axis square, axis off,title(sprintf("Before Encoder: Loss Factor=%d",loss_factor));
figure,imagesc(reconstructed, [0 255]),colormap gray,axis square, axis off,title(sprintf("After Encoder: Mean Squared Error = %f",encError));

%% Assignment 8 PSNR Compression
PSNR2 = zeros(1,50);

%Perform the arithmetic encoding over 100 loss factors and collect
%the compression ratio
for i=1:100
    transform = dctmgr(im,i);
    symb = int16(run_bits_value(transform)');
    [decoded encoded totalbits] = arithEncode(symb);
    reconstructed = idctmgr(transform,i);
    PSNR2(i) = 10*log10(255^2/((1/N^2)*sum(sum(abs(im(1:N,1:N)-reconstructed(1:N,1:N)).^2))));
    compressionRatio(i) = (8*512*512)/totalbits;
end

%Plot the result
plot(compressionRatio,PSNR2);
title(sprintf("PSNR vs. Compression Ratio %s",filename));
```

Assignment 1

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: DCT Manager
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function coeff = dctmgr(pic,loss_factor)

    %Create the array of transform coefficients to return
    coeff = zeros(64,(size(pic,1)*size(pic,2))/64);
    s=1;

    %Loop over 64 blocks in rows and columns
    for i=1:size(pic,1)/8
        for j=1:size(pic,2)/8
            %Run the forward DCT and zigzag scan and store the size 64
            %result
            coeff(:,s) = zigzagscan(lab5_dct(pic(((i-1)*8)+1:i*8,(((j-1)*8)+1:j*8)),'forward',loss_factor)
                )';
            s=s+1;
        end
    end

    %Do the DC differential encoding
    coeff = dcDiffEncoding(coeff,'encode');

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: DCT Function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function transform = lab5_dct(X,direction,loss_factor)
    % DCT Matrix Persists so it only needs to be calculated once
    persistent A N quantizationTable;

    %If this is true, compute the basis vectors
    if(isempty(A))
        % Number of Rows and Columns
        N = 8;
        %Quantization Table Provided in Lab
        quantizationTable = [16 11 10 16 24 40 51 61; 12 12 14 19 26 58 60 55; ...
            14 13 16 24 40 57 69 56; 14 17 22 29 51 87 80 62; 18 22 37 56 68 109 103 77; ...
            24 35 55 64 81 104 113 92; 49 64 78 87 103 121 120 101; 72 92 95 98 112 100 103 99];

        for i=0:7
            for j=0:7
                if(i == 0)
                    A(i+1,j+1) = sqrt(1/N)*cos(0);
                else
                    A(i+1,j+1) = sqrt(2/N)*cos(2*pi*((i)/(2*N))*j+(pi/(2*N))*(i));
                end
            end
        end
    end

    %Compute the DCT with the 8x8 input block
    if(direction == 'forward') %Forward transform
        transform = double(A)*double(X)*double(A');

        %Perform Quantization
        temp = transform(1,1);
        transform = floor(transform./(loss_factor*double(quantizationTable))+.5);
        transform(1,1) = temp;

    elseif(direction == 'inverse') %Inverse Transform

        %Perform Inverse Quantization
        temp = X(1,1);
        Y = X.*quantizationTable*loss_factor;
        Y(1,1) = temp;
        transform = double(A')*double(Y)*double(A);
    end

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: Zig-Zag Scan
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function zz = zigzagscan(block)
    i=1;
    j=1;
    zz = zeros(1,64);
```

```
    zz(1) = block(1,1);
    size = 1;

    %Upper left section
    while(j<8)
        %Move Right
        j=j+1;
        size=size+1;
        zz(size) = block(i,j);

        %Move downward diagonal
        while(j ~= 1)
           i = i+1;
           j = j-1;
           size=size+1;
           zz(size) = block(i,j);
        end

        %Move downwards
        if(i < 8)
            i=i+1;
            size=size+1;
            zz(size) = block(i,j);
        else
            break; %Stop when we hit the bottom left corner of matrix
        end

        %Move upward diagonal
        while(i ~= 1)
           i=i-1;
           j=j+1;
           size=size+1;
           zz(size) = block(i,j);
        end
    end

    %Bottom right section
    while(j<8 || i<8)
        %Move Right
        j=j+1;
        size=size+1;
        zz(size) = block(i,j);
        if(i==8 && j==8)
            break;
        end

        %Move upward diagonal
        while(j ~= 8)
           i=i-1;
           j=j+1;
           size=size+1;
           zz(size) = block(i,j);
        end

        %Move downwards
        if(i < 8)
            i=i+1;
            size=size+1;
            zz(size) = block(i,j);
        else
            break; %Stop when we hit the bottom left corner of matrix
        end

        %Move downward diagonal
        while(i ~= 8)
           i = i+1;
           j = j-1;
           size=size+1;
           zz(size) = block(i,j);
        end
    end

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: DC Differential Encoding
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function coeffs = dcDiffEncoding(coeffs, direction)
    persistent dc;

    %Store the original DC values
    if(isempty(dc))
       dc = coeffs(1,:);
    end

    %Differentially encode the DC values
    if(direction == 'encode')
        for i=2:size(coeffs,2)
```

```
            coeffs(1,i) = dc(i)-dc(i-1);
        end
    elseif(direction == 'decode')
        for i=2:size(coeffs,2)
            coeffs(1,i) = coeffs(1,i)+dc(i-1);
        end
    end
end
```

Assignment 2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: Inverse DCT Manager
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function reconst = idctmgr(coeffs,loss_factor)
    %Create the array of transform coefficients to return
    reconst = zeros(size(coeffs,2)/8,size(coeffs,2)/8);

    %Undo the DC differential encoding
    coeffs = dcDiffEncoding(coeffs,'decode');

    s=1;

    %Loop over 64 blocks in rows and columns
    for i=1:size(coeffs,1)
        for j=1:size(coeffs,1)
            %Run the inverse DCT and zigzag scan and store in 8x8 block
            reconst(((i-1)*8)+1:i*8,(((j-1)*8)+1:j*8)) = lab5_dct(izigzagscan(coeffs(:,s)'),'inverse',
                loss_factor);
            s=s+1;
        end
    end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: DCT Function
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function transform = lab5_dct(X,direction,loss_factor)
    % DCT Matrix Persists so it only needs to be calculated once
    persistent A N quantizationTable;

    %If this is true, compute the basis vectors
    if(isempty(A))
        % Number of Rows and Columns
        N = 8;
        %Quantization Table Provided in Lab
        quantizationTable = [16 11 10 16 24 40 51 61; 12 12 14 19 26 58 60 55; ...
            14 13 16 24 40 57 69 56; 14 17 22 29 51 87 80 62; 18 22 37 56 68 109 103 77; ...
            24 35 55 64 81 104 113 92; 49 64 78 87 103 121 120 101; 72 92 95 98 112 100 103 99];

        for i=0:7
            for j=0:7
                if(i == 0)
                    A(i+1,j+1) = sqrt(1/N)*cos(0);
                else
                    A(i+1,j+1) = sqrt(2/N)*cos(2*pi*((i)/(2*N))*j+(pi/(2*N))*(i));
                end
            end
        end
    end

    %Compute the DCT with the 8x8 input block
    if(direction == 'forward') %Forward transform
        transform = double(A)*double(X)*double(A');

        %Perform Quantization
        temp = transform(1,1);
        transform = floor(transform./(loss_factor*double(quantizationTable))+.5);
        transform(1,1) = temp;

    elseif(direction == 'inverse') %Inverse Transform

        %Perform Inverse Quantization
        temp = X(1,1);
        Y = X.*quantizationTable*loss_factor;
        Y(1,1) = temp;
        transform = double(A')*double(Y)*double(A);
    end

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: Inverse Zig-Zag Scan
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function block = izigzagscan(zz)
    i=1;
    j=1;
    block = zeros(8,8);
    block(1,1) = zz(1);
    size = 1;

    %Upper left section
    while(j<8)
        %Move Right
        j=j+1;
        size=size+1;
        block(i,j) = zz(size);

        %Move downward diagonal
        while(j ~= 1)
           i = i+1;
           j = j-1;
           size=size+1;
           block(i,j)=zz(size);
        end

        %Move downwards
        if(i < 8)
            i=i+1;
            size=size+1;
            block(i,j) = zz(size);
        else
            break; %Stop when we hit the bottom left corner of matrix
        end

        %Move upward diagonal
        while(i ~= 1)
           i=i-1;
           j=j+1;
           size=size+1;
           block(i,j) = zz(size);
        end
    end

    %Bottom right section
    while(j<8 || i<8)
        %Move Right
        j=j+1;
        size=size+1;
        block(i,j) = zz(size);
        if(i==8 && j==8)
            break;
        end

        %Move upward diagonal
        while(j ~= 8)
           i=i-1;
           j=j+1;
           size=size+1;
           block(i,j) = zz(size);
        end

        %Move downwards
        if(i < 8)
            i=i+1;
            size=size+1;
           block(i,j) = zz(size);
        else
            break; %Stop when we hit the bottom left corner of matrix
        end

        %Move downward diagonal
        while(i ~= 8)
           i = i+1;
           j = j-1;
           size=size+1;
           block(i,j) = zz(size);
        end
    end

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: DC Differential Encoding
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function coeffs = dcDiffEncoding(coeffs, direction)
    persistent dc;
```

```
9
10      %Store the original DC values
11      if(isempty(dc))
12          dc = coeffs(1,:);
13      end
14
15      %Differentially encode the DC values
16      if(direction == 'encode')
17          for i=2:size(coeffs,2)
18              coeffs(1,i) = dc(i)-dc(i-1);
19          end
20      elseif(direction == 'decode')
21          for i=2:size(coeffs,2)
22              coeffs(1,i) = coeffs(1,i)+dc(i-1);
23          end
24      end
25  end
```

Assignment 3

```
1   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2   % Clint Olsen
3   % ECEN 4532: DSP Lab
4   % Lab 5: DCT Function
5   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
7   function transform = lab5_dct(X,direction,loss_factor)
8       % DCT Matrix Persists so it only needs to be calculated once
9       persistent A N quantizationTable;
10
11      %If this is true, compute the basis vectors
12      if(isempty(A))
13          % Number of Rows and Columns
14          N = 8;
15          %Quantization Table Provided in Lab
16          quantizationTable = [16 11 10 16 24 40 51 61; 12 12 14 19 26 58 60 55; ...
17              14 13 16 24 40 57 69 56; 14 17 22 29 51 87 80 62; 18 22 37 56 68 109 103 77; ...
18              24 35 55 64 81 104 113 92; 49 64 78 87 103 121 120 101; 72 92 95 98 112 100 103 99];
19
20          for i=0:7
21              for j=0:7
22                  if(i == 0)
23                      A(i+1,j+1) = sqrt(1/N)*cos(0);
24                  else
25                      A(i+1,j+1) = sqrt(2/N)*cos(2*pi*((i)/(2*N))*j+(pi/(2*N))*(i));
26                  end
27              end
28          end
29      end
30
31      %Compute the DCT with the 8x8 input block
32      if(direction == 'forward') %Forward transform
33          transform = double(A)*double(X)*double(A');
34
35          %Perform Quantization
36          temp = transform(1,1);
37          transform = floor(transform./(loss_factor*double(quantizationTable))+.5);
38          transform(1,1) = temp;
39
40      elseif(direction == 'inverse') %Inverse Transform
41
42          %Perform Inverse Quantization
43          temp = X(1,1);
44          Y = X.*quantizationTable*loss_factor;
45          Y(1,1) = temp;
46          transform = double(A')*double(Y)*double(A);
47      end
48
49  end
```

Assignment 6

```
1   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2   % Clint Olsen
3   % ECEN 4532: DSP Lab
4   % Lab 5: Pre-Processor
5   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
7   function symb = run_bits_value(coeff)
8       %Variable to track zeros before current value
9       zero_tracker = 0;
10
11      %Keep track of symb size
12      symb_size = 1;
13
14      %Loop through each column of the coeff matrix
15      for cols=1:size(coeff,2)
```

```
        for rows=1:size(coeff,1)
            if(rows==1)%Store the DC part
                symb(symb_size) = 0;
                symb(symb_size+1) = 12;
                symb(symb_size+2) = coeff(rows,cols);
                symb_size = symb_size + 3; % Increment size of symb by 3
            elseif(coeff(rows,cols) == 0) %Count sequential zeros
                zero_tracker = zero_tracker + 1;
            else
                symb(symb_size) = zero_tracker;
                zero_tracker = 0; %Reset zero tracker
                symb(symb_size+1) = 11; %11 bits for all AC components
                symb(symb_size+2) = coeff(rows,cols);
                symb_size = symb_size + 3; % Increment size of symb by 3
            end
        end

        %Add EOB after processing each column of coeffs.
        symb(symb_size) = 0;
        symb(symb_size+1) = 0;
        symb_size = symb_size + 2;
        zero_tracker = 0;
    end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: Post-Processor
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function newcoeff = irun_bits_value(symb)
    %Declare coeff matrix
    newcoeff = zeros(64,(512*512)/64);

    %Counter variables for coeff matrix
    r = 1;
    c = 1;
    i = 1;

    %Loop through all items in symb vector in groups of 3
    while i < length(symb)-2

        % Increment compensation for EOB
        if(symb(i) == 0 && symb(i+1) == 0)
            i = i+2;
        end

        %Enter appropriate number of zeros into the coeff matrix
        if(symb(i) ~= 0)
           for z=1:symb(i)
              newcoeff(r+(z-1),c) =  0;
           end
        end

        %Enter the value into the coeff matrix at that location in symb
        newcoeff(r+symb(i),c) = symb(i+2);

        %Handle EOB by jumping to next column in coeff
        if(symb(i+3) == 0 && symb(i+4) == 0)
            if(i+4 == length(symb))
                return;
            end
            newcoeff(r:64,c) = 0; %Fill remaining with zeros

            %Set new col and row
            c = c + 1;
            r = 1;
        else
            r = r+symb(i)+1;
        end
        i = i+3;
    end
end
```

Assignment 7

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clint Olsen
% ECEN 4532: DSP Lab
% Lab 5: Arithmetic Encoder/Decoder
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [recon_symb bitstream totalbits] = arithEncode(symb)

    minval = min( symb );              % find the minimum value
    symb = symb - (minval - 1);    % guarantee no non-negative integers in symb
    maxval = max( symb );           % the number of possible values symb can assume
    histvals = histcounts(symb, maxval ); % this is needed by arithmetic encoder
```

```
13      histvals = histvals + 1;          % can?t have 0 values in this array either
14                                        % i.e., no symbols have probability 0
15      %histogram(symb, [0:maxval] ); % if you want to see a plot of the histogram
16      bitstream = arithenco(symb, histvals); % returns the encoded bitstream
17      totalbits = length(bitstream);
18
19      symb_length = length(symb);
20      recon_symb = arithdeco(bitstream, histvals, symb_length);
21      recon_symb = cast(recon_symb, 'double') + double(minval - 1);   % undo the previous operation
22
23  end
```