Laboratory # 4

Images, Compression, and Coding

4.1 Introduction

A common application of signals and systems is in the production, manipulation, storage and distribution of images. For example, image transmission is an important aspect of communication (especially on the internet), and we would like to be able to distribute high quality images quickly over low-bandwidth connections. To do so, images must be *encoded* into a sequence or file of bits, which can be digitally transmitted or stored. When display of the image is required, the sequence/file of bits must be *decoded* into a reproduction of the image.

Systems or algorithms that do the encoding and decoding are called *source coders*, *coders*, *data compressors*, or *compressors*. They are called *source coders* because they encode the data from a *source*, e.g. a camera. They are also called data compressors, because their encoders usually produce fewer bits than were produced by the original data collector. For example, JPEG is a commonly used, standardized image compressor. You've probably downloaded many JPEG encoded images over the internet — any image with filename extension .jpg. FAX machines use a different image compression algorithm.

In this lab, we will experiment with some basic data compression techniques as applied to images. Typically, there is a tradeoff between the number of bits an encoder produces and the quality of the decoded reproduction. With more bits we can usually obtain better quality at the expense of greater storage or bandwidth requirements. When we assess how well these techniques work, we will count the number of bits their encoders produce(fewer is better), and as a measure of quality, and we will compute the *root-mean squared* (RMS) error as a measure of the quality of the decoded reproduction (low RMS error means high quality, or equivalently, low distortion).

The next section provides an introduction to images and data compression. It is followed in Section 4.3 by a brief listing of things that will be demonstrated in the lab session. Section 4.4 describes the actual assignments for this lab.

4.2 Background

4.2.1 Images

We will restrict attention to black-white images, i.e. images with various gray levels, but no color. Such images are ordinarily represented as a two-dimensional signal $x(t,s)$, $0 \le t \le H$, $0 \le s \le W$, where $x(t,s)$ denotes the *intensity* or *brightness* or *value* of the image at the position with vertical coordinate t and horizontal coordinate s, and where H and W are the height and width of the image, respectively. The values of $x(t,s)$ are nonnegative. A small value of $x(t,s)$ close to zero corresponds to black while larger values correspond to progressively lighter shades of gray.

In *digital* image processing, the image is assumed to be sampled at regularly spaced intervals creating a *discrete-space* image $x[m,n]$:

$$x[m,n] = x(mS, nS) \ ,$$

where S is the sampling interval. For simplicity we consider only square images. Thus for our purposes an image is simply an array or matrix of numbers $x[m,n]$, where m and n are integers ranging from 1 to some integer N. Each $x[m,n]$ is called a *pixel*. We adopt the usual convention that $x[1,1]$ is upper left pixel, $x[1,N]$ is the upper right, $x[N,1]$ is the lower left, and $x[N,N]$ is the lower right.

We shall also adopt the common, but not universal, convention of digital image processing that pixel values, often called *levels* are integers ranging from 0 to 255. The reason the pixel values are integers is that computers cannot store real-valued quantities. Instead the raw pixel values must be *quantized* to values from a finite set. The usual practice is to scale the raw image pixel values by some constant so the maximum value is close to 255 and then to round each pixel value to the nearest integer, thereby obtaining an image whose values are integers between 0 and 255. Why 0 to 255? There are two reasons. One is that these values can be conveniently represented with one byte, i.e. 8 bits.[1] Another reason is that it has the effects of rounding to 256 possible levels is not ordinarily observable, whereas rounding to a significantly smaller number, say 128, is sometimes noticeable.

We next describe some important numerical characteristics of an image. The *mean* or *average value* of an $N \times N$ image is

$$\mathrm{MV} = \frac{1}{N^2} \sum_{m=1}^{N} \sum_{n=1}^{N} x[m,n] \ .$$

The *variance* (VAR) and *standard deviation* (STD) of the image are defined, respectively, as

$$\begin{aligned} \mathrm{VAR} &= \frac{1}{N^2} \sum_{m=1}^{N} \sum_{n=1}^{n} (x[m,n] - \mathrm{MV})^2 \\ \mathrm{STD} &= \sqrt{\mathrm{MSV}} \ . \end{aligned}$$

Notice that STD is like an RMS value, except that the mean value has been subtracted from the data. VAR and STD are measures of how widely varying are the values of the image. If they are small, it means that the signal values (and signal value distribution) is tightly clustered around the mean value, while if they are large, the signal values range widely.

[1] To store an integer in a computer, it must be represented with a binary sequence. If binary sequences of length b are used, then 2^b levels can be represented, because there are 2^b distinct binary sequences of length b. Thus, it takes 8 bits to store the 256 levels from 0 to 255.

If $y[m,n]$ is a reproduction or other approximate version of $x[m,n]$, then it is customary to measure the *distortion* in $y[m,n]$ with the *mean-squared error* (MSE) and *root mean-squared error* (RMSE), which are defined, respectively, as

$$\begin{aligned} \text{MSE} &= \frac{1}{N^2}\sum_{n=1}^{N}\sum_{m=1}^{n}(x[m,n]-y[m,n])^2 \\ \text{RMSE} &= \sqrt{\text{MSE}}\,. \end{aligned}$$

4.2.2 Data compression

There are two primary types of data compressors: *lossless* and *lossy*. A lossless compressor will encode and decode in such a way that the decoded reproduction is exactly the same as the original (8 bits per pixel) image. A lossy compressor will encode and decode in such a way that the decoder produces only an approximation to the original image. On the one hand, lossless is better because it is, well, lossless. This is essential when compressing computer files. UNIX *compress*, *gzip*, and the PKZip compression formats are all lossless. On the other hand, if a small amount of distortion is permitted, lossy compressors can attain much larger amounts of compression, i.e. their encoders can produce many fewer bits. For multimedia, lossy compression is usually acceptable. Examples of lossy compression schemes that you may have used include MP3 (for audio), JPEG (for photos), and MPEG (for movies). In this lab you will experiment only with lossy coders.

We also mention that some data compressors are *fixed-length* meaning that the number of bits per pixel they produce is a constant that is known in advance e.g. 2 bits/pixel, whereas others are *variable-length* meaning that the number of bits per pixel they produce is not known in advance. For simplicity, in this lab you will experiment only with fixed-length data compressors. On the other hand, JPEG and all lossless compressors are variable-length.

Lossy compressors typically consist of three parts: a preprocessor such as a transform, one more quantizers, and a binary coder. These are described in the following subsection.

4.2.3 Quantization

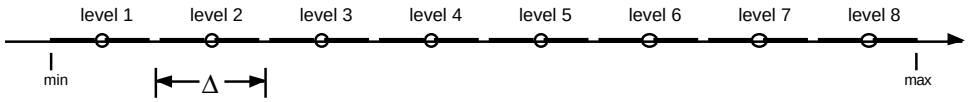
Quantization is the most elementary form of lossy data compression. When a number x is *quantized to M levels*, we mean that its value is replaced by (or quantized to) the nearest member of a set of M *quantization levels*. Here, we consider *uniform scalar quantization*[2]. For the uniform scalar quantization used here:

- We define a *quantizer range* defined by values x_{min} and x_{max}
- We divide this range into M equally sized segments, each with size $\Delta = \frac{x_{max}-x_{min}}{M}$.
- We place the quantization level for a given segment in the middle of that segment.

The quantizer is illustrated with the figure shown below, which shows $M = 8$ segments of width $\Delta = (x_{max} - x_{min})/8$ as thick lines and the corresponding levels within each segment as circles.

If it should happen that x is larger than x_{max}, then x is quantized to the largest level, namely, $x_{max} - \Delta/2$. Similarly, when x is smaller than x_{min}, then x is quantized to the smallest level, namely, $x_{min} + \Delta/2$.

[2]There are sometimes advantages to using quantizers with unequal level spacings, but we will not deal with such quantizers in this lab.



One can see that if x is within the quantizer range, then its quantized value will differ from x by at most $\Delta/2$, so that the quantizer introduces only a small error. On the other hand, when x is outside the range, the quantizer can introduce a large error. Thus when designing a quantizer it is important to choose the quantizer range so that it includes most values of x. Making the range large will do this. However, we don't want to make it too large, because the larger range, then the larger is Δ, which in turn controls the sizes of the errors introduced when x lies within the range of the quantizer.

When a quantizer is applied to some data such as an image $x[m,n]$ and produces a reproduction $y[m,n]$, as indicated earlier we compute the RMS error (RMSE) between $x[m,n]$ and $y[m,n]$ as a measure of the distortion introduced by the quantizer. Alternatively, it will sometimes be more convenient to measure the mean-squared error (MSE).

Elementary theory predicts that when the quantizer range includes most values of the image $x[m,n]$ and when Δ is much smaller than the RMS value of the image, then the MSE induced by quantizing with level spacing Δ can be approximated as follows.

$$\begin{aligned} \text{MSE} &\approx \frac{1}{12}\,\Delta^2 \\ &= \frac{1}{12}\left(\frac{x_{max}-x_{min}}{M}\right)^2 \end{aligned} \tag{4.3}$$

This shows that if we were to shrink Δ by a factor of 2, as would happen if M were doubled and the range were held constant, then the MSE would decrease by a factor of four.

When a quantizer is applied to data whose signal value distribution is fairly constant over a given range, then it is usually good practice to choose the quantizer range to match the data range. This is the case when directly quantizing images. That is, we choose $x_{min} = 0$ and $x_{max} = 255$.

On the other hand, when quantizing data whose signal value distribution is quite uneven, then it may be best to choose the quantizer range to be a subset of the data range. For example, in the transform coding described later, it often happens most of the data to be quantized is near zero but there are a few very, very large values. In such cases, experience has shown that to design a quantizer with small MSE, one should normally choose the width of the range to be proportional to the standard deviation (STD) of the data being quantized, i.e. $(x_{max} - x_{min}) = c \times \text{STD} = c\sqrt{\text{VAR}}$, where the constant of proportionality c is usually between 2 and 6. Smaller values of c work well for smaller values of M, and larger values of c work well for large values of M. Using this relation in (4.3), we find

$$\begin{aligned} \text{MSE} &\approx \frac{1}{12}\,\Delta^2 = \frac{1}{12}\left(\frac{x_{max}-x_{min}}{M}\right)^2 \\ &\approx \frac{1}{12}\,c^2\,\frac{\text{VAR}}{M^2}\ . \end{aligned} \tag{4.4}$$

This shows that quantizer MSE is proportional to the variance of the data and inversely proportional to M^2.

4.2.4 Binary coding

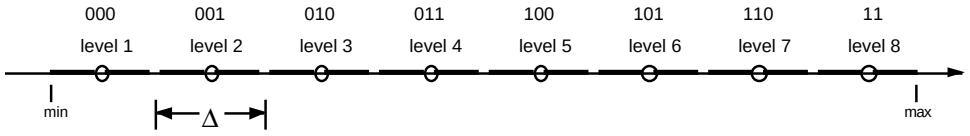
The output of an encoder should be a sequence or file of bits. We have so far described how a quantizer replaces a data value such as $x[m,n]$ by the closest quantization level. The

next step is to represent this quantization level with a *binary codeword*, which is simply a sequence of bits. It is the binary codeword that is the actual output of the encoder when quantizing the data. The decoder will, eventually, receive this binary codeword and output the corresponding quantization level as the reproduction of the original piece of data.

If the number of levels is a power of two, i.e. $M = 2^b$ where b is an integer. Then the simplest approach is to make each codeword have b bits. It does not matter which b-bit sequence is assigned to which level, but the usual scheme, as illustrated below, is to assign the binary sequence representing 0 to be the smallest level, the binary sequence representing 1 to the next largest level, and so on. With this type of binary coding, the encoder is *fixed-rate* in the sense described earlier. Often, a better scheme is to use shorter codewords for the quantization levels that occur more frequently, and longer ones for those that are used less frequently. Such variable-length codes are used in JPEG and other high efficiency schemes, but we will not try them in this lab assignment.

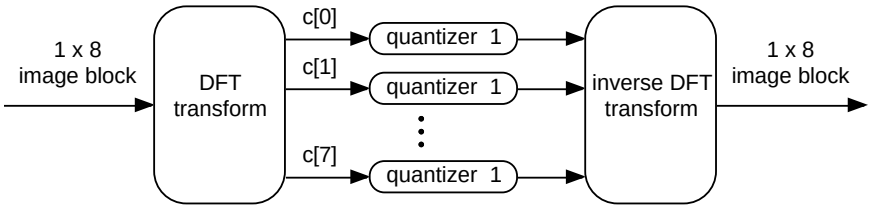
In summary, a lossy data compressor that consists of a quantizer and a binary coder has an encoder and a decoder that operate as follows on an image: the encoder quantizes successive pixels (usually taken in raster order) into quantization levels, and the binary coder outputs the corresponding binary codewords. Successive binary codewords are concatenated to form the encoded/compressed data stream or file. The decoder receives the compressed data stream or file, decodes each codeword into a quantization level, and outputs successive quantization levels as reproductions of the original image pixels. In this lab we will experiment with the quantizer, but not the binary coder. That is, we will create quantized images, but not compressed data files. (MATLABdoes not work easily with bits.) We will, however, count the number of bits that would be in such compressed data files.

In the next subsection, we add an additional element to the encoder and decoder.



4.2.5 Transform coding[3]

Efficient lossy data compressors typically apply quantization only after the data has been preprocessed in some way. One very common preprocessing step is to apply a *transform*, and such compressors are called *transform coders*. For example, JPEG is a transform coder based on the discrete cosine transform (DCT). Here we'll apply quantization after first preprocessing by applying the discrete Fourier transform (DFT) on small groups of image pixels, usually called *blocks*. A block diagram of our transform coder is shown below.



In this lab, the DFT will be applied to 1×8 blocks of pixels, for example to $x[1,0], ..., x[1,7]$.

[3]Transform coding is used only in Part 4 of the lab assignment. This section, which discusses transform coding, is not needed for Parts 1, 2 and 3. It can be skimmed or skipped until you are ready to start Part 4.

Recall that for a signal of length 8, the DFT is defined as

$$X[k] = \frac{1}{8}\sum_{n=0}^{7} x[n]e^{-j2\pi\frac{k}{8}n} \tag{4.5}$$

It produces eight DFT coefficients, $X[0], ..., X[7]$. In general, these coefficients are complex: $X[k] = u[k] + jv[k]$. Recall, however, that $X[0]$ is the DC coefficient, i.e. the average of the eight pixels. As a result, $v[0] = 0$ and $X[0] = u[0]$. For $k \geq 1$, coefficient $X[k]$ indicates the complex amplitude of the component of the block at frequency $k/8$. Recall also that the DFT has the property that $X[8-k] = X^*[k]$, so that $u[8-k] = u[k]$, and $v[8-k] = -v[k]$. Finally, note that for $k = 4$, $X[4] = X^*[4]$, i.e. $X[4] = u[4]$ is real-valued. It follows from these relationships that the DFT actually produces only eight *independent* coefficients, which we take to be

$$c[0] = u[0],\ \sqrt{2}c[1] = u[1],\ \sqrt{2}c[2] = u[2],\ c[3] = \sqrt{2}u[3],\ c[4] = u[4]$$
$$c[5] = \sqrt{2}v[1],\ c[6] = \sqrt{2}v[2],\ c[7] = \sqrt{2}v[3].$$

It is these eight coefficients, $c[0], \ldots, c[7]$, that must actually be quantized and binary encoded for each block. The $\sqrt{2}$ factors have been included to take into account that one coefficient is, in effect, standing in for two. It can be shown that with these factors

$$\sum_{n=0}^{7} x^2(1,n) = 8\sum_{k=0}^{7} c^2(k)\ , \tag{4.6}$$

which is an often useful fact.

To make this transform coder work well, the quantizers must be individually designed for each of the eight types of (independent) coefficients. Indeed, it turns out that if one quantizes all eight types of coefficients with the same number of levels, then the transform coder will not work substantially better than direct scalar quantization (quantization without preprocessing). In short, for each of the eight types of coefficients, one must carefully choose the number of quantization levels M and the quantizer range limits x_{min} and x_{max}. We let $M[k]$, $x_{min}[k]$, and $x_{max}[k]$ denote the parameters of the quantizer for the $c[k]$'s. We will take $M[k]$ to be a power of two, i.e. $M[k] = 2^{b[k]}$, where $b[k]$ is the number of bits allocated to quantizing $c[k]$.

Clearly, choosing large $M[k]$'s will permit the transform coder to do better. However, the total number of bits produced by the encoder is the number of blocks, $N^2/8$ times the number of bits to encode one block, $\sum_{k=0}^{8} b[k]$. We will frequently be interested in the number of encoded bits divided by the number of pixels, which is

$$\frac{1}{8}\sum_{k=0}^{8} b[k]\ . \tag{4.7}$$

Thus, if the total number of encoded bits per pixel is specified in advance, the real issue is how to divide these bits among the eight types of coefficients, i.e. how to choose the $b[k]$'s.

Using (4.6), it can be shown that.

$$\text{MSE} = \sum_{k=0}^{7} \text{MSE[k]}\ , \tag{4.8}$$

where MSE$[k]$ is the MSE of the quantizer for $c[k]$. In other words, the MSE of the transform coder is approximately the average of the MSE's of the quantizers for the different coefficients.

Let us now consider a transform coder where each type of coefficient is quantized with the same number of bits/pixel, i.e. $b[0] = b[1] = \ldots = b[7]$. We assert without proof that such a transform coder has roughly the same MSE as that of direct scalar quantization with the sum number of bits/pixel. We will now argue that changing the $b[k]$'s so that some are larger than others will make the transform coder work better than direct scalar quantization.

From (4.4) we have that

$$\begin{aligned} \text{MSE[k]} &\approx \frac{1}{12}\, c^2\, \frac{\text{VAR}[k]}{M^2} \\ &= \frac{1}{12}\, c^2\, \text{VAR}[k] 2^{-2b[k]} \ , \end{aligned} \tag{4.9}$$

where VAR$[k]$ denotes the variance of the $c[k]$ values. One can see from the above that the coefficients with larger variance will be quantized with larger mean-squared error. In particular the DC coefficients usually have the largest variance; so they will have the largest MSE. On the other hand, the $c[5]$'s and $c[7]$'s usually have the smallest variance and distortion. Now suppose we increase $b[0]$ by one and decrease $b[7]$ by one. From (4.7) we see that this will have no net effect on the number of bits produced by the coder. However, from (4.9) we see that this decreases the (large) MSE of the DC coefficients by a factor of 4, and increases the (small) MSE of the $c[7]$ coefficients by a factor of 4. Is it beneficial to decrease one MSE by 4 when another one increases by 4 beneficial? We can see from (4.8) that indeed it is beneficial, because decreasing a larger MSE by the factor 4 decreases the average in (4.8) more than increasing a small MSE by the factor of 4 increases the average.[4] Thus, what we want to do is shift bits towards the coefficients with larger variances. This will make MSE smaller than if all coefficients were quantized with the same number of bits and, therefore, smaller than the distortion of direct scalar quantization. More generally, in a well designed transform code, all of the MSE$[k]$'s will be approximately the same. (If they were quite different one could take a bit from a coefficient with small MSE to one with large MSE and make a net decrease in overall MSE.) In this light, one can see that the role of the transform is to make the variances of the coefficients as different as possible. Some should be large, and others should be small.

4.3 Demonstrations in the lab session

1. Displaying images and measuring their signal properties.
2. Noise images.
3. Lossless image compression using a Lempel-Ziv (LZ) compressor.
4. Lossy image compression using JPEG.
5. Lossy compression by downsampling.
6. Lossy compression by quantization.
7. Lossy transform coding based on the DFT and quantization.

Note that the specific lab assignments given below relate to Items 1, 2, 6 and 7.

[4] For example, $24 + 8$ is larger than $24/4 + 8 \times 4$.

4.4 Laboratory assignment

1. **Image display and properties.**

   Each problem in this lab deals with the same image, "cameraman." This image is built into MATLAB.

   (a) Load and display the image "cameraman". (If your computer does not have the Image Processing Toolbox, you'll need to download the file "cameraman.tif" from the web page). To load the image, use the command

   ```
   img = double(imread('cameraman.tif')) ;
   ```

   To display an image, for instance one called `your_image`, use the following sequence of commands:

   ```
   imagesc(your_img); colormap(gray); colorbar; axis image
   ```

   Every image that you display in this lab must have a colorbar. This means that every individual subplot must have a colorbar as well. Points will be deducted for each image that lacks a colorbar.

   - [6][5] Include the plot of the image in your report.[6]
   - [2] Calculate the size of the image (the number of rows and columns) and the total number of pixels and include these values in your report.

   (b) Familiarize yourself with values of the image by plotting the signal value distribution of the image using the command:

   ```
   hist(img(:),256)
   ```

   - [5] Include this plot in your report.
   - [3] From this histogram, what signal values occur the most often in this image?

   (c) Download the M-file `display_rect.m`, and use it to display the pixel values in several rectangular segments of the image. Find, approximately, the smallest rectangle of pixels that includes the black tip of the camera lens.

   - [2] From this display, what are the row and column indices of this rectangle?
   - [2] From this display, what are minimum and maximum values within this rectangle?
   - [3] Include in your report a plot from `display_rect.m` showing the pixel values of the rectangle you found.
   - [1] By hand, circle the region that you selected on the image displayed in Problem 1a.

   (d) Assuming 8 bits are used to represent the value of each pixel:

   - [2] How many bits are required to describe the entire image?

   (e) MATLAB ordinarily uses 64-bit double-precision to represent numbers, such as pixel values. Assuming this to be the case:

   - [2] How many bits are needed for the entire image?
   - [2] How many possible pixel values can a 64-bit number represent?

2. **Additive noise.**

   Throughout this lab we will consider the original "cameraman" image to be noise-free, and any modifications or distortions that are introduced will be considered as noise. In general, our model is

   $$v = s + n$$

[5] The numbers in brackets indicate how many points an item is worth.
[6] 'Bullets' show items to be included in your report.

where s is a signal, v is a corrupted or distorted ("noisy") version of s, and n is a *noise signal*. In later parts of the lab, the "noise" will actually be the errors caused by data compression. Here, however, we will simply add random noise.

(a) Create a noisy image with the command

```
img2 = img + 20*randn(size(img));.
```

- [6] Using `subplot`, display and include a plot of the original image, the noisy image, and the noise image (which can be found by subtracting the original image from the noisy image). Be sure to label which image is which.

(b) Calculate and include in your report:

- [3] the RMS value[7] of the original image.
- [3] the RMS value of the noisy image.
- [3] the RMS value of the noise image.

(c) Look at noisy images with smaller RMS noise values by adjusting the multiplicative factor in front of the `randn` command. Find a multiplier that produces noise that is "just noticeable" when compared to the original image (that is, a smaller value yields noise that you cannot see)? (Hint: Plot the original and noisy images next to one another using `subplot`. The answer to this question is somewhat subjective, but try to be as objective as possible.)

- [5] What is the multiplicative factor that you found?

3. **Quantization.**

(a) Download the function `quantize.m`. This function, which implements uniform scalar quantization, takes a signal, a number of quantization levels (M), a minimum value quantizer range value, and a maximum quantizer range value. For instance, to quantize the image (i.e. all pixels of the image) to 64 levels, use the command

```
q_img = quantize(img,64,0,255)
```

Quantize the image using 64 levels, 16 levels, and 4 levels.

- [8] Display and include in your report the three resulting quantized images along with the original using `subplot`.
- [6] Use `hist` to plot the signal value distribution for each of the three images. Use 256 bins (the second parameter of `hist`). Display these jointly using `subplot` and include them in your report.
- [1] Can you see the effects of quantization in these plots?

(b) Use MATLAB to make a plot of the function being implemented by `quantize.m` . For example, for the 64 level quantizer, run `quantize(x,64,0,255)` for x ranging from 0 to 255, and plot the resulting values versus x.

- [6] Display and include in your report the quantization functions for the 4, 16 and 64 level quantizers together using subplot.

(c) For the 4, 16, and 64 level quantizers,

- [3] How many bits are needed to represent each of these quantized images?
- [3] How many bits are needed to represent each pixel in one of these images?

(d) Using our signal-plus-noise model from above, find the error ("noise") image corresponding to each of these quantized images.

- [6] Using `subplot`, display and include in your report the three error images in the same plot.

[7] The RMS value of a set of numbers $z_1, z_2, \ldots, z_N$ is $\sqrt{\frac{1}{N}\sum_{i=1}^{N}(z_i)^2}$.

- [1] Can you see aspects of the original images in these plots?
- [3] Calculate and include the RMS error values for each image.

(e) Elementary theory predicts that the RMS error induced by quantizing with level spacing Δ is $\Delta/\sqrt{12}$. (Recall that the formula for Δ is given in Section 4.2.3.) Calculate RMS error predicted using this formula for each of the three sets of quantization levels.

- [3] What are the predicted RMS errors for 4, 16, and 64 level quantizers?
- [6] Plot both the measured and predicted RMS error values versus the required number of bits per pixel. (Each line on this plot will only have three points).
- [2] For what number of bits per pixel is this prediction most accurate?

4. **Transform Coding** (This part is to be turned in with Lab 5. It will also be graded as a part of Lab 5.)

In this part of the assignment, you will experiment with DFT based transform coding.

(a) Download the M-file `dft_block.m` . Run the function using the command

```
A = dft_block(img)
```

This applies the DFT to successive 1×8 blocks of the image. If the image is $N \times N$, this function produces a series of eight $N \times N/8$ *band images*. For $k = 1, \ldots, 8$, the k^{th} band image contains the $c[k-1]$ coefficients for each block. The eight band images are stored in `A` as a three-dimensional array. To access the third band image, for instance, we would call

```
band3 = A(:,:,3);
```

Each of these eight matrices can itself be viewed as an image, although all but the first one (containing DC values) have negative as well as positive values.

- [4][8] Use `subplot` to simultaneously display all eight band images. Use `axis square` rather than `axis image` when you display these band images.[9]
- [2] Discuss the appearances of the various band images. For example, can you see any features of the original cameraman image in any or all of them?

(b) Download and run the M-file `inverse_dft_block.m` using the command

```
recon = inverse_dft_block(A)
```

This collects the coefficients in A and applies the inverse DFT to reconstruct an image.

- [2] Compute the RMS error between the original and the transformed/inverse transformed image. (It should be negligibly small.)

(c) Download the M-file `dft_coder.m`, which implements the transform coder and decoder shown in the figure of Section 4.2.5 . It uses `dft_block(img)` to create the band images previously described. It then quantizes each band image using `quantize.m`. Finally, it makes a reconstruction of the image by applying `inverse_dft_block.m` to the quantized band images. It also prints to the command window a table showing the performance of the coder and some data about the 8 types of coefficients and their quantizers.the

Our goal in using `dft_coder` is to find appropriate parameters for the eight quantizers. Through intelligent design, we hope to achieve lower RMS error than with direct scalar quantization of the image using the same number of bits. We

[8] These points will count towards your grade for Lab 5

[9] Remember to include in your report the things specified in bullets.

do this by allocating bits to each of our eight quantizers independently. For instance, if we call `dft_coder` like this[10]

```
coded = dft_coder(img,[8 6 6 6 6 4 4 4]);,
```

we quantize our $c[0]$'s (DC) coefficients using 8 bits, the next four coefficients with 6 bits each, and the three imaginary coefficients using 4 bits each. Note that the number of bits per pixel required to represent the coded image is one-eighth of the sum of the number of bits allocated to each coefficient. Thus, this particular coder design uses 5.5 bits per pixel. Note also that when `quantize.m` is called to quantize the k^{th} band image, the quantizer range is chosen to be symmetric about the mean of the coefficients in that band and to have width equal to 5 times the standard deviation of the band.[11]

Find a 4 bits per pixel design with as small an RMS error as you can. You should be able to get an RMS error less than 4.2. (Hint: As a general rule of thumb from Section 4.2.5, bigger coefficients should get more bits.)

- [4] What bit allocation did you use, and what was the resulting RMS error?
- [3] Display the reconstructed image and the error image on the same figure using `subplot`.
- [2] Compare your RMS error to the RMS error of 4 bits per pixel uniform scalar quantization like those that you performed in problem 3.
- [1] Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the scalar quantizer.

(d) Find a 3 bits per pixel design with as small an RMS error as you can. You should be able to get an RMS error less than 7.

- [4] What bit allocation did you use, and what was the resulting RMS error?
- [3] Display the reconstructed image and the error image on the same figure using `subplot`.
- [2] Compare your RMS error to the RMS error of a 3 bits per pixel uniform scalar quantization like those that you performed in problem 3.
- [1] Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the scalar quantizer.

(e) Find a 2 bits per pixel design with as small an RMS error as you can. You should be able to get an RMS error less than 13.

- [4] What bit allocation did you use, and what was the resulting RMS error?
- [3] Display the reconstructed image and the error image on the same figure using `subplot`.
- [2] Compare your RMS error to the RMS error of a 2 bits per pixel uniform scalar quantization like those that you performed in problem 3.
- [1] Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the scalar quantizer.

(f) Given your experimentation with this transform coder,

- [4] Comment on the relative performances of direct scalar quantization and transform coding as the number of bits/pixel changes.

[10] Though more advanced coders may allow the allocation of fractions of bits, note that for this coder you must allocate a whole number of bits to each coefficient. You can, however, assign no bits to a coefficient. In this case, that coefficient is simply set to a constant value.

[11] Note that you can change this factor to something else if you wish. Note also that in an actual coder, the means and standard deviations of the bands would have to encoded and sent to the decoder. However, we can ignore this detail because it requires very few bits relative to those produced by the quantizers.