Laboratory 5

Images, Compression, and Coding

5.1 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. A block diagram of a general data compression system, with an encoder and decoder, is shown in Figure 5.1.

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 or scanner. 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

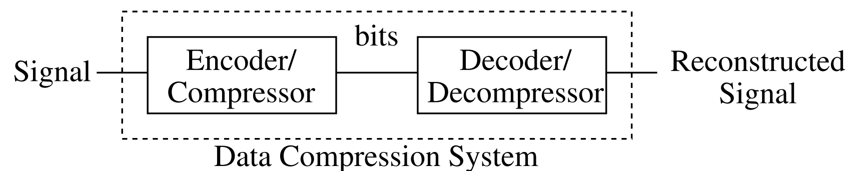


Figure 5.1: A block diagram of a general data compression system.

 The University of Michigan, All rights reserved

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 mean-squared or RMS error as a measure of the quality of the decoded reproduction (low error means high quality, or equivalently, low distortion).

5.1.1 "The Question"

- How can we compress images so that they take up less storage space and/or less bandwidth?

5.2 Background

5.2.1 Images

So far, we have dealt entirely with *one-dimensional* signals. That is, these signals are indexed by only one independent variable (usually time). In this lab, we will start to consider *two-dimensional* signals. An image is an example of a two-dimensional signal. In an image, we usually index the signal based on horizontal and vertical position — two dimensions that are needed to find the "signal value" at any given point.

In this lab, we will generally restrict our attention to gray-scale images[1]. We mathematically represent such an image (in *continuous-space*) as a signal $x(t, s)$, where $0 \leq t \leq H$, $0 \leq s \leq W$. $x(t, s)$ denotes the *intensity*, *brightness*, or *value* of the image at the position with vertical coordinate t and horizontal coordinate s, and H and W are the height and width of the image, respectively. The values of $x(t, s)$ are generally nonnegative. Thus, 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 $x[m, n]$:

$$x[m, n] = x(mT_s, nT_s), \tag{5.1}$$

where T_s is the sampling interval, given in units of *distance*. Thus, in discrete-space, an image is simply an $M \times N$ array or matrix of numbers $x[m, n]$, where m and n are integers in the range $[1, M]$ and $[1, N]$, respectively. Each $x[m, n]$ is called a *pixel*. We adopt the usual convention that $x[1, 1]$ is the upper left pixel, $x[1, N]$ is the upper right, $x[M, 1]$ is the lower left, and $x[M, N]$ 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

[1] Color images are often represented as three *separate* signals (or *channels*), one each for red, green, and blue.

The University of Michigan, All rights reserved

values can be conveniently represented with one byte, i.e. 8 bits.[2] Another reason is that the effects of rounding to 256 possible levels are not ordinarily observable, whereas rounding to a significantly smaller number, say 128, is sometimes noticeable.

5.2.2 Signal statistics for images

Two-dimensional signals in general, and images in particular, have the same sorts of statistics that one-dimensional images have. Generalizing from the one-dimensional case is often quite straightforward. We will also introduce two new statistics for both one- and two-dimensional signals.

1. **Average Value.** The *mean* or *average value*, M, of a discrete-space image $x[m,n]$ is

$$M(x) = \frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} x[m,n] \tag{5.2}$$

2. **Mean-squared value.** The *mean-squared value* (or MSV), MS, of a discrete-space image $x[m,n]$ is

$$MS(x) = \frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} x^2[m,n] \tag{5.3}$$

3. **Root mean-squared value.** The *root mean-squared value* (or RMS value) of a discrete-space image $x[m,n]$ is

$$RMS(x) = \sqrt{\frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} x^2[m,n]} \tag{5.4}$$

$$= \sqrt{MS(x)} \tag{5.5}$$

4. **Variance.** The *variance* of a discrete-space image $x[m,n]$ is

$$Var(x) = \frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} (x[m,n] - M(x))^2 \tag{5.6}$$

$$= MS(x - M(x)) \tag{5.7}$$

$$= MS(x) - (M(x))^2 \tag{5.8}$$

5. **Standard deviation.** The *standard deviation* of a discrete-space image $x[m,n]$ is

$$Std(x) = \sqrt{\frac{1}{NM}\sum_{m=1}^{M}\sum_{n=1}^{N} (x[m,n] - M(x))^2} \tag{5.9}$$

$$= \sqrt{Var(x)} \tag{5.10}$$

$$= RMS(x - M(x)) \tag{5.11}$$

$$= \sqrt{MS(x) - (M(x))^2} \tag{5.12}$$

[2] 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.

 The University of Michigan, All rights reserved

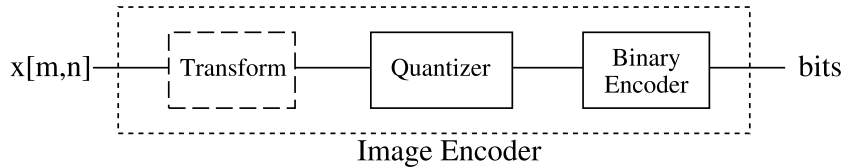


Figure 5.2: A block diagram of a general image encoder/compressor.

Notice the relationship between the variance and standard deviation, equation (5.10), and the relationship between these statistics and the MS and RMS values[3], equations (5.8) and (5.12). The variance and standard deviation measure how widely varying are the values of a signal. If they are small, it means that the signal values (and thus the signal value distribution) is tightly clustered around the mean value, while if they are large, the signal values range widely.

Recall from Laboratory 1 that we often use the MS and RMS values to measure *distortion* of a signal. We will be doing this for images in this laboratory. If $y[m, n]$ is a distorted version of $x[m, n]$, then we can measure the *mean-squared error* (MSE) and *root mean-squared error* (RMSE), using

$$MSE = \frac{1}{NM} \sum_{n=1}^{N} \sum_{m=1}^{M} (x[m,n] - y[m,n])^2 \tag{5.13}$$

$$RMSE = \sqrt{\text{MSE}}. \tag{5.14}$$

5.2.3 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 often acceptable. Examples of lossy compression schemes that you may have used include MP3 (for audio), JPEG (for photos), and MPEG (for movies). These three schemes are all examples of *transform coding* methods; we will examine a simple transform coding scheme in this laboratory. MP3 encoding is also an example of so-called *perceptual coding*. Perceptual coding methods often introduce significant amounts of distortion, but do so in a way that is nearly imperceptible.

Consider the generalized image encoder/compressor shown in Figure 5.2. It consists of three main components: the *transform*, the *quantizer* and the *binary encoder*. These are described briefly now, and in more detail in the next three subsections. The input to the encoder is a sampled image, $x[m, n]$. We generally consider that the pixels of $x[m, n]$ take on a continuum of real values.

[3]Equation (5.8) is not something that is immediately obvious, but it is something that can be straightforwardly derived. (Doing so is an interesting exercise.)

The University of Michigan, All rights reserved

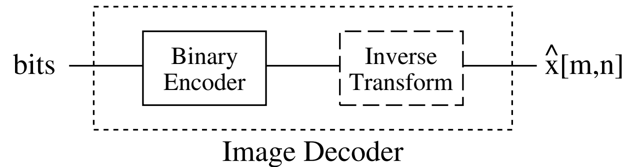


Figure 5.3: A block diagram of a general image decoder/decompressor.

The first component, the *transform*, is optional. When it is included, it is usually a spatial-domain to frequency-domain transformation like the Discrete Fourier Transform (DFT). Applying the transform to short segments or *blocks* of the signal tends to concentrate the energy of the signal into just a few coefficients, which permits the quantizer and binary encoder to be more effective.

The next component is *quantization*. Quantization takes the input sample/pixel and "rounds" it to one of a finite set of *levels*. It is a lossy or noninvertible operation in the sense that one cannot recover the original sample/pixel from the quantized sample/pixel. As an example, we noted previously that digital images often have pixel values ranging from 0 to 255. This is because each *raw* pixel value, as produced by some camera, has already been rounded to the nearest of a set of 256 quantization levels. Lossless image compression schemes work by operating on image that are already quantized; additional quantization is not permitted. However, in lossy compression schemes, additional quantization is performed, in order to obtain greater compression. For example, each image pixel may be quantized to the nearest of a set of only 64 levels.

The final component is *binary coding*, which assigns a sequence of bits called a *codeword*, to each level produced by the quantizer. In some systems, called *fixed-length coders*, the number of bits used to represent each pixel is known in advance (e.g. 2 bits per pixel). This is the simplest type of coder. Other systems, called *variable-length coders*, assign binary codewords of different lengths to each pixel value, usually based on the frequency of occurrence. More frequently occuring levels are assigned shorter codewords. This allows the compression system to achieve additional compression. Many advanced compression systems, including JPEG, use variable length coders. MP3 coding has provisions for both fixed-length and variable-length coding.

As illustrated in Figure 5.3, the decoder/decompressor corresponding to the encoder/compressor just described has two components. The input to the decoder is the bits produced by the encoder. The first component, the *binary decoder*, inverts the operation of the binary encoder, and produces the levels originally produced by the quantizer. The quantization operation produced by the encoder is not invertible, so there is not corresponding decoding step. Instead the last step is the inverse transform, which as the name suggests performs the inverse of the encoding transform. The output of the decoder, called an *encoded or decoded image or reproduction* can be displayed on a monitor or printed on paper, as desired.

5.2.4 Transformation

Efficient lossy data compressors typically perform some sort of preprocessing on the data to be compressed. One very common preprocessing step is a *transform*, and such compressors are called *transform coders*. For example, JPEG is a transform coder based on the dis-

 The University of Michigan, All rights reserved

crete cosine transform (DCT), which is a spectral transformation similar to the DFT. The transform is typically applied to small groups of pixels called *blocks*. In this lab, we will experiment with a simple DFT-based transform coder that uses short 1×8 pixel blocks. That is, we use an N-point DFT with $N = 8$. Recall that the synthesis and analysis formulas for an 8-point DFT are given by

$$x[n] = \sum_{k=0}^{7} X[k] e^{j\frac{2\pi k}{8}n} \tag{5.15}$$

$$X[k] = \frac{1}{8} \sum_{n=0}^{7} x[n] e^{-j\frac{2\pi k}{8}n} \tag{5.16}$$

Here, $X[k]e^{j\frac{2\pi k}{8}n}$ is the "spatial" frequency component at frequency $\hat{\omega} = \frac{2\pi k}{8}$. The DFT synthesis formula shows that an image block $x[n]$ can be viewed as the sum of such components. More specifically,

$$\begin{aligned}(x[0], x[1], x[2], \ldots, x[7]) = \; & X[0]\,(1,\ 1,\ 1,\ \ldots,\ 1) \\ & + X[1]\,(\, e^{j\frac{2\pi}{N}0},\ e^{j\frac{2\pi}{N}1},\ e^{j\frac{2\pi}{N}2},\ \ldots,\ e^{j\frac{2\pi}{N}7}\,) \\ & + X[2]\,(\, e^{j\frac{2\pi 2}{N}0},\ e^{j\frac{2\pi 2}{N}1},\ e^{j\frac{2\pi 2}{N}2},\ \ldots,\ e^{j\frac{2\pi 2}{N}7}\,) \\ & + \ldots \\ & + X[7]\,(\, e^{j\frac{2\pi 2}{N}0},\ e^{j\frac{2\pi 2}{N}1},\ e^{j\frac{2\pi 2}{N}2},\ \ldots,\ e^{j\frac{2\pi 2}{N}7}\,)\end{aligned} \tag{5.17}$$

Note that we are NOT presuming that these blocks are in any way periodic.

Why do we perform a spectral transformation before compressing a signal? As suggested in Section 5.2.3, the transformation serves to shift around the energy in a given block so that it is easier to compress. Consider, for instance, a single block of an image given by

$$x[n] = (\ 165,\ 168,\ 167,\ 166,\ 167,\ 165,\ 168,\ 166\ )$$

This block is roughly constant, so we expect its 8-point DFT to have a large $X[0]$ (i.e., DC) component. Since there is little other variation, though, the rest of the DFT coefficients will be relatively small. By only storing the $X[0]$ coefficient, for instance, and throwing away the rest, we only need $1/8^{\text{th}}$ as much storage as if we had stored all of the coefficients for this block; further, we have introduced only a small amount of distortion (as measured by the mean-squared error). While we won't go so far as to throw away the rest, this example suggests the basic idea for how to use the transform to compress a signal. If some transformed coefficients tend to be smaller than others, we can store them more efficiently.

An 8-point DFT produces 8 complex numbers, $X[0], X[1], \ldots, X[7]$. This actually translates into 16 real numbers (the 8 real and 8 imaginary parts) that we need to consider storing. Thus taking the transfrom would at first seem to be a bad idea, because we now need twice as much storage to represent a block! However, there are symmetry properties that we can exploit so that we only need to store 8 of these numbers. Since the input signal is real, recall that the 8-point DFT, $X[k]$ has the conjugate symmetry property:

$$X[8-k] = X^*[k] \tag{5.18}$$

This means that knowing the real and imaginary parts of $X[1]$, for instance, completely determines the real and imaginary parts of $X[N-1]$. Thus, though we need to store the real

and imaginary parts of $X[0], X[1], X[2], X[3], X[4]$, we do *not* need to store $X[5], X[6], X[7]$, because these values can be recovered from the previous four. Further, the coefficients $X[0]$ and $X[4]$ are purely real (that is, $X[0] = X^*[0]$ and $X[4] = X[8-4] = X^*[4]$). Thus, $X[0]$ and $X[4]$ each require the storage of a single real number. From this argument, we can see that the 8-point DFT produces a total of only eight real numbers that must be stored.

In our implementation of the transform encoder, the eight numbers $c[0], \ldots, c[7]$ that we choose to represent the 8-point DFT $X[0], \ldots, X[7]$, of an image block are

$$\begin{aligned} c[0] &= X[0] \\ c[1] &= \sqrt{2}\ \mathrm{Re}\{X[1]\} \\ c[2] &= \sqrt{2}\ \mathrm{Re}\{X[2]\} \\ c[3] &= \sqrt{2}\ \mathrm{Re}\{X[3]\} \\ c[4] &= X[4] \\ c[5] &= \sqrt{2}\ \mathrm{Im}\{X[1]\} \\ c[6] &= \sqrt{2}\ \mathrm{Im}\{X[2]\} \\ c[7] &= \sqrt{2}\ \mathrm{Im}\{X[3]\} \end{aligned}$$

The $\sqrt{2}$ factors have been included where one coefficient is, in effect, standing in for two. It can be shown that with these factors

$$\sum_{n=0}^{7} x^2[1,n] = 8 \sum_{k=0}^{7} c^2[k], \tag{5.19}$$

which is an often useful fact. This is derived using Parseval's relation, as given in Lab 4[4].

5.2.5 Quantization

Quantization is the most elementary form of lossy data compression, while also forming a fundamental part of more advanced lossy compression schemes such as transform coding. We may quantize an image directly, or we may quantize the results of a transformation as described in Section 5.2.4. When a number x is *quantized to L levels*, we mean that its value is replaced by (or quantized to) the nearest member of a set of L *quantization levels*. Here, we consider *uniform quantization*[5]. For the uniform quantization used here:

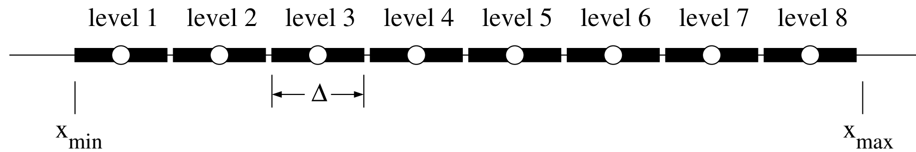
- We define a *quantizer range* defined by values x_{min} and x_{max}
- We divide this range into L equally sized segments, each with size $\Delta = \frac{x_{max}-x_{min}}{L}$.
- We place the quantization level for a given segment in the middle of that segment.

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

[4] For this derivation, see the document titled "Notes: The Distortion of Transform Coding" by D.L. Neuhoff.

[5] There are sometimes advantages to using quantizers with unequal level spacings, but we will not deal with such quantizers in this lab. Uniform quantizers are sometimes called *uniform scalar quantizers* to distinguish them from more sophisticated quantizers that do not operate independently on successive data samples.

 The University of Michigan, All rights reserved



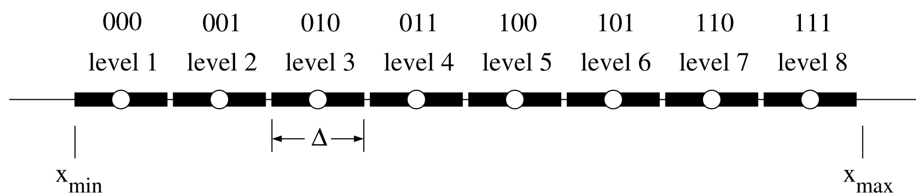
Given a pixel $x[m,n]$, the quantizer operates by outputing the nearest level. Equivalently, if $x[m,n]$ lies in the ith segment, then quantizer outputs the ith level. If x is larger than x_{max}, then x is quantized to the largest level, namely, $x_{max} - \Delta/2$. Similarly, if x is smaller than x_{min}, then x is quantized to the smallest level, namely, $x_{min} + \Delta/2$.

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 the range too large. Larger ranges mean that $\Delta = (x_{max} - x_{min})/L$ is larger, which in turn increases the maximum possible error introduced when x lies within the range of the quantizer.

5.2.6 Binary coding

The output of a data compression encoder must always be bits, not quantized samples or pixels. Thus, the quantizer is always followed by a *binary encoder*, as illustrated in Figure 5.4. A compressor that consists simply of a quantizer followed by a binary encoder will be called a *direct quantizer*, in contrast to a transform coder or some other coder that involves a preprocessing step.

A binary encoder operates by assigning a distinct sequence of bits, called a *codeword* to each level of the quantizer. For example, an assignment of codewords to levels is shown below



Such binary codewords are the output of the encoder when quantizing the data. The decoder will, eventually, receive a binary codeword and output the corresponding quantization level as the reproduction of the original piece of data. For instance, if the image pixel $x[m,n]$ lies in the third segment of the quantizer, the binary encoder will produce 010, which when received, the decoder will produce level 3 as the reproduction of $x[m,n]$.

If, as often happens, 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 above, is to assign the binary sequence representing 0 to the smallest level, the binary sequence

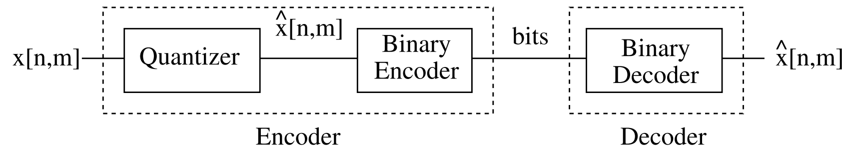


Figure 5.4: Block diagram of a direct quantizer

representing 1 to the next largest level, and so on. With this type of binary coding, the encoder is fixed-length (or *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.

5.2.7 Performance

There are two ways that we measure the performance of a compression system. First, we want to know how many bits are required to store an image. The total number of bits produced by the encoder is equal to the number of blocks multiplied by the number of bits required to encode one block. More commonly, we report the number of bits required to store a single pixel. This is called the *coding rate*, R. The coding rate is equal to the number of bits required to code a single block divided by the number of pixels in a block. Naturally, we prefer a lower coding rate.

The second performance measure is the amount of distortion introduced by the coder. Generally, we measure this distortion by computing the mean-squared (MSE) or RMS error (RMSE). We also prefer to have low distortion, and equivalently low error.

Unfortunately, we generally have to trade off between these two performance measures. That is, we can produce a highly compressed (with a low coding rate) image, but this generally introduces a large RMS error. Alternatively, we can have a very high-quality representation of an image (with low distortion), but such a representation requires many bits to encode. Figure 5.5 shows an illustration of the tradeoff between the two performance measures. In the laboratory assignment, you will produce a plot similar to this for compression using uniform quantization.

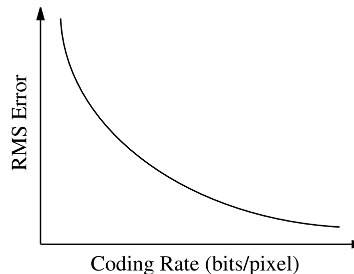


Figure 5.5: There is an inherent tradeoff between coding rate and distortion.

 The University of Michigan, All rights reserved

Performance of direct quantizers

Let us now analyze the performance of a direct quantizer, where the quantizer is uniform with $L = 2^b$ levels and range $[x_{min}, x_{max}]$.

Since the binary encoder for such a system assigns b bits to each level, the coding rate is

$$R = b \;\text{ bits/pixel (bpp)} \tag{5.20}$$

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 standard deviation of the image, then the MSE induced by quantizing with level spacing Δ can be approximated as follows[6]/

$$\begin{aligned} MSE &\approx \frac{1}{12}\Delta^2 && (5.21)\\ &= \frac{1}{12}\left(\frac{x_{max}-x_{min}}{L}\right)^2 && (5.22)\\ &= \frac{1}{12}(x_{max}-x_{min})^2 2^{-2R}\;, && (5.23)\end{aligned}$$

This shows that if we were to shrink Δ by a factor of 2, as would happen if L were doubled and the range were held constant, then the MSE would decrease by a factor of four. Equivalently, the last equation shows that this factor of four reduction comes by increasing the coding rate by one bit per pixel.

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 generally the case when directly quantizing images, so we will generally 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 transform coding, 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 of the data being quantized, i.e. $(x_{max} - x_{min}) = c \times Std(x) = c\sqrt{Var(x)}$. The constant of proportionality c is usually between 2 and 6. Smaller values of c work well for smaller values of L, and larger values of c work well for large values of L. Using this relation in (5.21), we find

$$\begin{aligned} \text{MSE} &\approx \frac{1}{12}\Delta^2 = \frac{1}{12}\left(\frac{x_{max}-x_{min}}{L}\right)^2 && (5.24)\\ &\approx \frac{c^2}{12}\,\frac{Var(x)}{L^2}. && (5.25)\\ &\approx \frac{c^2}{12}\,Var(x)2^{-2R}. && (5.26)\end{aligned}$$

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

[6] See the document "Note: The $\Delta^2/12$ Formula" by D.L. Neuhoff.

The University of Michigan, All rights reserved

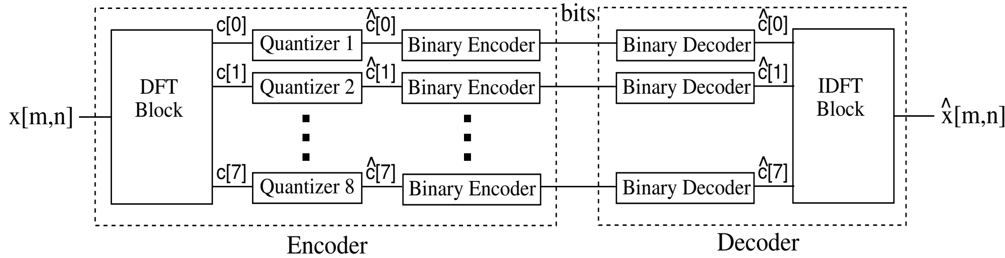


Figure 5.6: A block diagram of a transform code. The encoder divides the incoming image, $x[m,n]$ into 1×8 blocks and transforms each block into a sequence of 8 coefficients $c[0],\ldots,c[7]$. These coefficients are then quantized to yield $\widehat{c}[0],\ldots,\widehat{c}[7]$, and encoded into a binary representation. The decoder creates a reconstruction of the image, $\widehat{x}[m,n]$ by decoding the binary codewords and inverting the transformation.

5.2.8 Designing a transform coder

In the previous sections, we have described the three main components of transform type data compression system. In particular, in Section 5.2.4, we described a transform that uses an 8-point DFT of 1×8 blocks of image pixels. A block diagram of the system based on this transform can be found in Figure 5.6.

To make this transform coder work well, though, the quantizers must be individually designed for each of the eight types of (independent) coefficients. Indeed, if we quantize all eight types of coefficients with the same number of levels, then the transform coder will not work substantially better than direct quantization (quantization without preprocessing). Thus, for each of the eight types of coefficients, we must carefully choose the number of quantization levels, L, and the quantizer range limits, x_{min} and x_{max}.

Let L_k be the number of levels for each coefficient $c[k]$. Further, let each L_k be a power of two such that $L_k = 2^{b_k}$, where b_k is the number of bits that we allocate to the transformed coefficient $c[k]$.

It should be clear that choosing large L_k's will permit the transform coder to encode with less distortion. However, the total number of bits produced by the encoder is the number of blocks, $\frac{N^2}{8}$, times the number of bits to encode one block, $\sum_{k=0}^{7} b_k$. Thus, higher L_k's require more bits to store the signal, and thus a higher coding rate. For this transform coder, we can calculate the coding rate, R, as

$$R = \frac{1}{8}\sum_{k=0}^{7} b_k \quad \text{bpp} . \tag{5.27}$$

In many situations, we are given a desired coding rate R, e.g. $R = 2$ bpp. In this case, the question becomes how we should divide these bits among the eight types of coefficients, i.e. how to choose the b_k's, so they average to the desired coding rate R, yet cause the distortion in the reproduction produced by the transform code to be as small as possible.

 The University of Michigan, All rights reserved

Using (5.19), it can be shown that[7].

$$\text{MSE} = \sum_{k=0}^{7} \text{MSE[k]} \ , \qquad (5.28)$$

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

Let us first 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 quantization with the same number of bits/pixel. Now, 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 quantization.

From (5.26) we have that

$$\text{MSE[k]} \approx \frac{1}{12} c^2 \, Var(c[k]) 2^{-2b_k} \ , \qquad (5.29)$$

where $Var(c[k])$ 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 $C[0]$ usually have the largest variance; so they will have the largest MSE. On the other hand, the $c[3]$'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 (5.27) we see that this will have no net effect on the number of bits produced by the coder. However, from (5.29) we see that this decreases the (large) MSE of the DC coefficients $c[0]$ 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? We can see from (5.28) that indeed it is beneficial. Decreasing a larger MSE by the factor 4 decreases the average in (5.28) more than increasing a small MSE by the factor of 4 increases the average.[8] 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 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, we could move a bit from a coefficient with small MSE to one with large MSE and achieve a net decrease in overall MSE. In this light, we 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.

5.3 Some MATLAB commands for this lab

- **Making a matrix into a vector.** Especially when working with images, it is often useful to be able to convert a matrix into a vector containing the same elements. In MATLAB, we can do this for a matrix `x` in the following manner:

```
>> y = x(:);
```

[7]See the document titled "Notes: The Distortion of Transform Coding" by D.L. Neuhoff.
[8]For example, $24 + 4$ is larger than $24/4 + 4 \times 4$.

After this operation, `y` contains a "vectorized" version of `x`. Specifically, if `x` is an $M \times N$ matrix, `y` is a vector whose first M elements are the first column of `x`, whose second M elements are the second column of `x`, and so on. This is especially useful for calculating many of the signal statistics presented in this laboratory.

- **Calculating signal statistics on images.** To compute these statistics on images, we first need "vectorize" the image.

  1. Average value, $M(x)$:

     ```
     >> M = mean(x(:))
     ```

  2. Mean-squared value, $MS(x)$:

     ```
     >> MS = mean(x(:).^ 2)
     ```

  3. Root-mean squared value, $RMS(x)$:

     ```
     >> RMS = sqrt(mean(x(:).^ 2))
     ```

  4. Variance, $Var(x)$.

     ```
     >> variance = var(x(:));
     ```

  5. Standard deviation, $Std(x)$.

     ```
     >> std_dev = std(x(:));
     ```

  6. Signal value distribution. To compute a histogram with 256 bins centered at integers from 0 to 255, use the command

     ```
     >> hist(x(:),0:255);
     ```

- **Loading images.** Images are generally stored in some sort of standard file format, like TIFF or JPEG. To load such an image file into MATLAB, we use the command `imread`. Unfortunately, `imread` generally returns images as arrays of integers. This is unfortunate because MATLAB puts some heavy restrictions on the use of integers. In particular, to prevent integer overflow you cannot perform arithmetic on integers. Thus, we need to convert our loaded images into double precision arrays using the command `double`. To load and convert an image in the file `my_img.tif`, for instance, use the command

  ```
  >> x = double(imread('my_img.tif'));
  ```

  Note that the `imread` command will load many standard image file formats, including JPEG, PNG, BMP, TIFF, PCX, and a host of others.

- **Displaying images.** To display an image in MATLAB, there are actually a number of commands that must be used simultaneously. To display the image itself, we use `imagesc` command. To tell MATLAB to display the image as a gray-scale image, we use command `colormap(gray)`. To set the axes so that the aspect ratio is correct, use the command `axis image`. Finally, to add a "color bar" that relates image values to colors, use the `colorbar` command. *Every image that you produce for this course must have a color bar; you will lose points for* every *image you display without a color bar.* To do all of these things at once to display an image `x`, use the following code:

 The University of Michigan, All rights reserved

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

You will be using this sequence of commands often, so you might wish to write a short function that executes all of these commands simultaneously.

- **Quantizing an image:** The function `quantize_fcn.m`, which we provide to you for this lab, implements a uniform quantizer for images and transform coefficients. It takes a signal, the desired number of quantization levels (L), and the two numbers that define the quantization range, x_{min} and x_{max}. For instance, to quantize an image, `img`, to 64 levels, use the command

  ```
  >> [q_img, delta] = quantize_fcn(img,64,0,255);
  ```

  `q_img` contains the quantized image, while `delta` contains the Δ value used for quantization. Here, note that $x_{min} = 0$ and $x_{max} = 255$. This separately quantizes each pixel of `img` to one of 64 levels, in accordance with the procedure described in the background section.

- **Using the DFT Coder:** The DFT-based transform coder that we have described in this laboratory is provided as three separate functions.

  - `dft_block.m` breaks the image into 1×8 blocks and computes the DFT of each block. If the image is $N \times N$, this function produces a series of eight *band images*. For $k = 1, \ldots, 8$, the kth band image contains the $c[k-1]$ coefficients for each block. For example, the $k = 1$ band image, contains the $c[0]$, or DC, coefficients from each block. Each band image has size $M \times N/8$.

    The eight band images are returned as a three-dimensional array. To produce the band images for an image, `img` and then access the third band image, for instance, we would use the commands

    ```
    >> A = dft_block(img);
    >> A(:,:,3);
    ```

    Note that except for the first one, each band image contains both positive and negative values. However, we can still display them using `imagesc`.

  - `inverse_dft_block.m` reconstructs the image from the matrix of band images returned by `dft_block.m`.

  - `dft_coder.m` puts both of these blocks together by calling `dft_block`, quantizing the coefficient matrix, and reconstructing the image with `inverse_dft_block`.

    `dft_coder` takes several input parameters, all of which are optional except the first one. The first parameter is the image to encode. The second is a vector of bit allocations, b_k. For instance, if we call `dft_coder` like this

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

    we quantize our $c[0]$ (DC) coefficients using 8 bits, the next four (real) coefficients with 6 bits each, and the last three (imaginary) coefficients using 4 bits each[9].

[9] Though more advanced coders may allow the allocation of fractions of bits, 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.

The University of Michigan, All rights reserved

Note that the number of bits required to encode a single pixel is equal to the average value of all of the b_k's. Thus, the example above uses 5.5 bits per pixel. When run, `dft_coder` returns the decoded image and also displays a table of useful statistics corresponding to each coefficient $c[k]$. To see this table, make sure that you put a semicolon at the end of your call to `dft_coder`.

5.4 Demonstrations in the Lab Section

1. Images are signals too.
2. Signal compression
3. The "Almost JPEG" DFT Coder
4. Designing the coder

5.5 Laboratory assignment

1. (Images in MATLAB) In this problem, you'll familiarize yourself with the image capabilities of MATLAB along with one particular image, the "cameraman."

   (a) (Display an image) Load the image "cameraman.tif". (If your computer does not have the Image Processing Toolbox, you'll need to download the file from the web page).

   - Display the image and include the resulting figure in your report.
   - Calculate the size of the image (the number of rows and columns) and the total number of pixels in the image.
   - Find the minimum and maximum pixel values, x_{min} and x_{max} in the image.

   (b) (Produce and interpret a histogram) Estimate the signal value distribution of this image by generating a histogram with 256 bins centered at integers from 0 to 255.

   - Include the resulting plot in your report.
   - From this histogram, what signal values occur the most often in this image?
   - In words, describe which part(s) of the image corresponds to these signal values.

   (c) (Examining signal values) It is useful to be able to think of images in terms of the signal values that make them up. Download the M-file `display_square.m`, which will help this process. Use this function to display the pixel values in several rectangular segments of the "cameraman" image. Find, approximately, the smallest rectangle of pixels that includes the black tip of the camera lens.

   - Include in your report a plot from `display_square.m` showing the pixel values of the rectangle you found.
   - From this display, what are the row and column indices of this rectangle?
   - From this display, what are minimum and maximum values within this rectangle?

 The University of Michigan, All rights reserved

(d) (Signal representations) We know that this image takes on only integer values over a finite range, but there are still a few different ways we can represent the image. In the original file, for instance, each pixel is represented using 8 bits. In MATLAB, though, we convert the image into 64-bit double precision values.

- How many bits are required to describe the entire image at 8 bits per pixel?
- How many bits are required to describe the entire image at 64 bits per pixel?
- How many possible pixel values can a 64-bit number represent?

2. (Direct quantization) In this problem, we will experiment with direct quantization as an image compression mechanism. Download the function `quantize_fcn.m`.

(a) (Quantize an image) Use `quantize_fcn` to quantize the "cameraman" image using 64 levels, 16 levels, and 4 levels. Assume $x_{min} = 0$ and $x_{max} = 255$.

- Display and include in your report the three resulting quantized images along with the original using `subplot`. Again, make sure that you indicate which image is which.
- Describe the effects of the quantization in these plots.

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

- Plot the quantization function for the 16 level quantizer.
- Also, plot the histogram of image quantized with 16 levels, using 256 bins centered at integers from 0 and 255.

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

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

(d) (Measuring quantization error) Find the "error image" corresponding to each of these quantized images.

- Using `subplot`, display and include in your report the three error images in the same plot.
- Can you see aspects of the original images in these plots?
- Calculate the RMS error for each quantization of the image.

(e) (Evaluating RMS error predictions) Now, we want to compare the actual RMS error for "cameraman" versus the predicted RMS error (based on the derivation in Section 5.2.7) for quantizers with 2, 4, 8, 16, 32, 64, and 128 levels.

- Calculate the actual RMS error for each of these quantizers.
- Calculate the predicted RMS errors for these quantizers.
- Plot both the actual and predicted RMS error values versus the required number of bits per pixel.
- For what number of bits per pixel is this prediction most accurate?

3. (Compression using a transform coder) In this problem, you will experiment with the DFT-based transform coder that is described in the background section.

(a) (Create and examine band images) Download the M-file `dft_block.m`. Use it to generate the matrix of band images for the "cameraman" image.

- Use `subplot` to simultaneously display all eight band images. Use `axis square` rather than `axis image` when you display these band images.
- 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) (Reconstruct a coded image) Download the M-file `inverse_dft_block.m`. Use this function to reconstruct the original image from the set of band images produced by `dft_block`.

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

(c) (Designing coders for image compression) Download the M-file `dft_coder.m`. Our goal in using `dft_coder` is to find appropriate parameters for the eight quantizers when compressing the "camerman" image. Through intelligent design, we hope to achieve lower RMS error than with direct quantization of the image using the same number of bits. We do this by allocating bits to each of our eight quantizers independently.

i. (Design a 4 bpp coder) 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. (Hint: As a general rule of thumb from Section 5.2.8, bigger coefficients should get more bits.)

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

ii. (Design a 3 bpp coder) 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 6.4.

- What bit allocation did you use, and what was the resulting RMS error?
- Display the reconstructed image and the error image on the same figure using `subplot`.
- Compare your RMS error to the RMS error of the 3 bits per pixel uniform quantization that you performed in problem 2e.
- Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the direct quantizer. (Note: You have not yet displayed the 3 bpp image, so you will need to generate it for comparison.)

iii. (Design a 2 bpp coder) 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 10.8.

- What bit allocation did you use, and what was the resulting RMS error?
- Display the reconstructed image and the error image on the same figure using `subplot`.

 The University of Michigan, All rights reserved

- Compare your RMS error to the RMS error of a 2 bits per pixel uniform quantization that you performed in problem 2e.
- Compare the qualitative appearance of the reconstruction produced by the transform coder to that produced by the direct quantizer.

iv. (Comment on coder design) Given your experimentation with this transform coder,

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

Food for Thought: In this lab, we've used a 1-dimensional transform for our coder. We can achieve significantly better compression if we use a 2-dimensional transform. MATLAB *implements a two-dimensional DFT with the command* `fft2`*. As a challenging project, consider modifying the transform coder provided here to work on* 4×4 *or* 8×8 *blocks of an image. How much compression can you achieve with this modified coder?*

4. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

Postscript: JPEG Compression

In this lab, we've presented a transform coder here which uses some of the same basic ideas as JPEG compression. However, JPEG achieves much higher compression rates that what we have seen, and with much less distortion. How is this achieved? There are several modifications used in JPEG coding.

1. JPEG uses a *two-dimensional transform.* This allows much greater compaction of the data into a few transform coefficients.
2. JPEG uses a transform called the *discrete cosine transform*, which purely real, rather than the DFT. This removes some of the redundancies in our coding method.
3. JPEG uses a technique called *run-length encoding.* This allows a coder to store a "run" of similar values by indicating the value and the number of repetitions.
4. JPEG uses a variable-length coding scheme (often Huffman coding, which you may study in an intermediate programming course on data structures and algorithms) to produce a bit stream for the final coded representation.

All of these improvements allow images to be significantly compressed with relatively small distortion. For more information about JPEG coding, you might wish to look at the JPEG Tutorial:

`http://www.ece.purdue.edu/~ace/jpeg-tut/jpegtut1.html`

The University of Michigan, All rights reserved