Chapter 6

Image Compression

JORGE REBAZA

One of the central issues in information technology is the representation of data by arrays of bits in the most efficient way possible, a never-ending quest for improvement in the representation of bits that are smaller, faster and cheaper. This is exactly the role of data compression: to convert strings of bits into shorter ones for more economical transmission, storage and processing. Abundant applications require such compression process: medical imaging, publishing, graphic arts, digital photography, wire photo transmission, etc.

For the past few years, the Joint Photographic Experts Group (**JPEG**) has been working to keep an international compression standard for both, grayscale and color images. No surprise that a strong mathematical research in this direction has been going on since then, and it is important to remark that when JPEG conducted a first selection process in 1988, they reported that a proposal based on the Discrete Cosine Transform had produced the best picture quality. As a matter of fact, JPEG is a format for image compression based on the discrete cosine transform, which is used to reduce the file size of an image as much as possible without affecting the quality of the image as experienced by the human sensory system.

In this chapter we present an elegant application of mathematical tools and concepts (in particular from linear algebra and numerical analysis) to the problem of image compression, and illustrate how certain theoretical mathematical results can be effectively used in applications that include the response of our vision system to changes in the image representation.

6.1 Compressing with Discrete Cosine Transform

We will study two main techniques for compressing images. First we introduce a technique that is currently used for compressing most images available on the Internet and that uses a square orthogonal matrix of order eight to perform the corresponding transformation of coordinates (from space to frequency). Later on, and for completion, we study image compression as an application of the SVD factorization of a matrix A. Thus, orthogonality is present in both approaches.

We are interested in the two-dimensional discrete cosine transform, but we start with its one-dimensional version.

6.1.1 1-d Discrete cosine transform

Through the discrete cosine transform we can combine and apply concepts such as orthogonality, interpolation, least squares, as well as linear combination of basis functions in vector spaces. This transform is a very special mathematical tool that will allow us to separate and order an image into parts of differing importance, with respect to the image visual quality. We start with the one-dimensional case.

Definition 6.1 *Define the following $n \times n$ orthogonal matrix*

$$C = \sqrt{\frac{2}{n}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & \cdots & \frac{1}{\sqrt{2}} \\ \cos\frac{\pi}{2n} & \cos\frac{3\pi}{2n} & \cdots & \cos\frac{(2n-1)\pi}{2n} \\ \cos\frac{2\pi}{2n} & \cos\frac{6\pi}{2n} & \cdots & \cos\frac{2(2n-1)\pi}{2n} \\ \vdots & \vdots & \cdots & \vdots \\ \cos\frac{(n-1)\pi}{2n} & \cos\frac{(n-1)3\pi}{2n} & \cdots & \cos\frac{(n-1)(2n-1)\pi}{2n} \end{bmatrix}. \tag{6.1}$$

Given a vector $x = [x_0 \cdots x_{n-1}]^T$, the discrete cosine transform (DCT) of x is the vector $y = [y_0 \cdots y_{n-1}]^T$ given by

$$y = Cx. \tag{6.2}$$

Example 6.1.1 *For n=8, the matrix C in (6.1) is*

$$\begin{bmatrix} 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 \\ 0.4904 & 0.4157 & 0.2778 & 0.0975 & -0.0975 & -0.2778 & -0.4157 & -0.4904 \\ 0.4619 & 0.1913 & -0.1913 & -0.4619 & -0.4619 & -0.1913 & 0.1913 & 0.4619 \\ 0.4157 & -0.0975 & -0.4904 & -0.2778 & 0.2778 & 0.4904 & 0.0975 & -0.4157 \\ 0.3536 & -0.3536 & -0.3536 & 0.3536 & 0.3536 & -0.3536 & -0.3536 & 0.3536 \\ 0.2778 & -0.4904 & 0.0975 & 0.4157 & -0.4157 & -0.0975 & 0.4904 & -0.2778 \\ 0.1913 & -0.4619 & 0.4619 & -0.1913 & -0.1913 & 0.4619 & -0.4619 & 0.1913 \\ 0.0975 & -0.2778 & 0.4157 & -0.4904 & 0.4904 & -0.4157 & 0.2778 & -0.0975 \end{bmatrix} \tag{6.3}$$

We can readily verify that this matrix (up to rounding) is in fact orthogonal, that is, $C^TC = I$. Now define the vector

$$x = [1 \quad 2 \ -2 \quad 0 \quad 1 \quad 4 \quad 0 \ -1]^T$$

Then, the DCT of x is $y = Cx$, where

$$y = [1.7678 \quad 0.0480 \ -0.4619 \quad 3.8565 \ -1.0607 \ -1.4262 \ -0.1913 \ -2.3645]^T.$$

Note: Observe the sign pattern in the rows or columns of the matrix C in (6.3).

To appreciate how the DCT will allow us to compress data, we introduce a theorem that through interpolation of an input vector x, it explicitly arranges the elements of its DCT $y = Cx$ in order of importance, as coefficients of a linear combination of (basis) cosine functions.

Theorem 6.2 (DCT Interpolation Theorem) *Let C be the matrix in (6.1), and let $x = [x_0 \cdots x_{n-1}]^T$. If $y = [y_0 \cdots y_{n-1}]^T$ is the DCT of x ($y = Cx$), then the function*

$$P_n(t) = \frac{1}{\sqrt{n}}\, y_0 \ + \ \sqrt{\frac{2}{n}} \sum_{k=1}^{n-1} y_k \cos \frac{k(2t+1)\pi}{2n} \tag{6.4}$$

satisfies

$$P_n(i) = x_i, \qquad \text{for} \quad i = 0, \ldots, n-1.$$

That is, $P_n(t)$ interpolates the data $(0, x_0), (1, x_1), \ldots, (n-1,\, x_{n-1})$, i.e. $P_n(t)$ passes through the n points $(i,\, x_i)$.

Proof. From (6.4) we have

$$\begin{aligned} P_n(0) &= \tfrac{1}{\sqrt{n}} y_0 + \sqrt{\tfrac{2}{n}} \sum_{k=1}^{n-1} y_k \cos \tfrac{k\pi}{2n} \\ P_n(1) &= \tfrac{1}{\sqrt{n}} y_0 + \sqrt{\tfrac{2}{n}} \sum_{k=1}^{n-1} y_k \cos \tfrac{3k\pi}{2n} \\ \vdots &\quad \vdots \\ P_n(n-1) &= \tfrac{1}{\sqrt{n}} y_0 + \sqrt{\tfrac{2}{n}} \sum_{k=1}^{n-1} y_k \cos \tfrac{k(2n-1)\pi}{2n}. \end{aligned}$$

Using orthogonality, $y = Cx$ implies $x = C^T y$. Then, the equations above can be written as

$$\begin{bmatrix} P_n(0) \\ \vdots \\ P_n(n-1) \end{bmatrix} = C^T \begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix} = C^T y = x = \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix}.$$

□

Remark 6.3 *In terms of linear algebra, the DCT interpolation statement in (6.4) is nothing else but expressing $P_n(t)$ as a unique linear combination of n cosine basis functions of increasing frequencies (the first term $\frac{1}{\sqrt{n}}\, y_0$ corresponds to cosine of zero frequency), weighted by appropriate coefficients. For $n = 8$, in Figure 6.1 we plot the cosine basis functions in (6.4) and the corresponding eight-point basis (denoted with '$\times$') from the rows of the matrix C in Example 6.1.1.*

Before we present an example of a one-dimensional DCT interpolation, let us stress the fact that in the proof of Theorem 6.2 we have exploited the fact that the matrix C is orthogonal and therefore $C^T C = I$. Thus, from (6.2), we can take $C^T y = C^T C x = x$. That is, we can recover x as

$$x = C^T y, \tag{6.5}$$

which is known as the (one-dimensional) *inverse discrete cosine transform* of y.

Now for a moment let us take $n = 3$. Then, (6.5) is

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \sqrt{\frac{2}{3}} \begin{bmatrix} \frac{1}{\sqrt{2}} & \cos\frac{\pi}{6} & \cos\frac{2\pi}{6} \\ \frac{1}{\sqrt{2}} & \cos\frac{3\pi}{6} & \cos\frac{6\pi}{6} \\ \frac{1}{\sqrt{2}} & \cos\frac{5\pi}{6} & \cos\frac{10\pi}{6} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix},$$

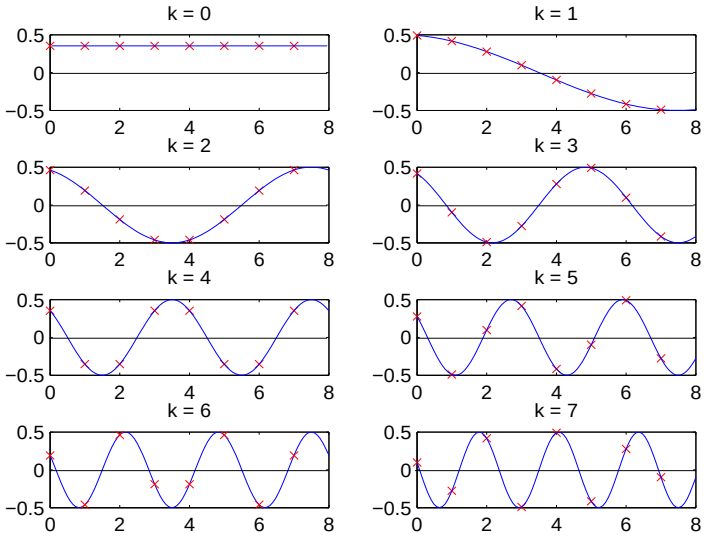

Figure 6.1: Cosine basis functions for $n = 8$

thus, componentwise we have

$$\begin{aligned} x_0 &= \tfrac{1}{\sqrt{3}}\, y_0 + \sqrt{\tfrac{2}{3}}\left[y_1 \cos\tfrac{\pi}{6} + y_2 \cos\tfrac{2\pi}{6}\right], \\ x_1 &= \tfrac{1}{\sqrt{3}}\, y_0 + \sqrt{\tfrac{2}{3}}\left[y_1 \cos\tfrac{3\pi}{6} + y_2 \cos\tfrac{6\pi}{6}\right], \\ x_2 &= \tfrac{1}{\sqrt{3}}\, y_0 + \sqrt{\tfrac{2}{3}}\left[y_1 \cos\tfrac{5\pi}{6} + y_2 \cos\tfrac{10\pi}{6}\right]. \end{aligned} \tag{6.6}$$

These equations (6.6) are nothing else but (6.4) with the interpolation property $P_n(j) = x_j$, for $j = 0, \ldots, n-1$. This illustrates a general fact about the connection between the DCT interpolation (6.4) and the inverse DCT given by (6.5).

Example 6.1.2 *Interpolate the points*

$$(0, 2),\ (1, 0),\ (2, -1),\ (3, 0),\ (4, 0.25),\ (5, -1.5),\ (6, -2)$$

using the DCT.

In this case we use the DCT matrix (6.1) with n=7, and then for the vector $x = [2 \quad 0 \quad -1 \quad 0 \quad 0.25 \quad -1.5 \quad -2]^T$ *we compute (after rounding)*

$$y = Cx = [-0.8504 \quad 2.4214 \quad 0.0715 \quad 1.9751 \quad 0.8116 \quad -0.3764 \quad 0.1387]^T.$$

Then, from Theorem 6.2, the function interpolating the seven data points is

$$\begin{aligned} P_7(t) &= \tfrac{1}{\sqrt{7}}\,(-0.8504) \,+\, \sqrt{\tfrac{2}{7}}\,\Big[\, 2.4214\cos\tfrac{(2t+1)\pi}{14} + 0.0715\cos\tfrac{2(2t+1)\pi}{14} \\ &\quad + 1.9751\cos\tfrac{3(2t+1)\pi}{14} + 0.8116\cos\tfrac{4(2t+1)\pi}{14} - 0.3764\cos\tfrac{5(2t+1)\pi}{14} \\ &\quad + 0.1387\cos\tfrac{6(2t+1)\pi}{14}\,\Big]\,. \end{aligned}$$

The interpolant $P_7(t)$, which is a combination of the seven cosine basis functions is shown in Figure 6.2 as a solid curve, where the data points are represented by stars.

There are a couple of remarks to point out about the interpolation via DCT. Firstly, the frequencies of the cosine functions in (6.4) are in increasing order, and the coefficients y_k act as weights of these cosine functions. As it will turn out, the terms with the highest frequencies (the last terms in the expansion) will be the least important in terms of accuracy of interpolation, so that they can be safely dropped without substantially altering the final interpolation, resulting in a saving of terms (and storage).

Secondly, when using the interpolating polynomial $P_n(t)$ in (6.4), the coefficients y_k of the interpolation are easily computed through a matrix-vector multiplication $y = Cx$. Finding such coefficients is precisely the difficult part when finding other interpolating functions (such as Lagrange polynomials, splines, etc.). In addition, the basis functions are just cosines of increasing frequency. This makes DCT interpolation very simple and inexpensive to compute.

Finally, a more remarkable fact about DCT interpolation is that we can drop some of the last terms in the polynomial $P_n(t)$, and the error involved will be minimum in the sense of least squares. This is exactly our first step into compression.

Theorem 6.4 DCT Least Squares Approximation. *Let C be the matrix in (6.1). For $x = [x_0 \cdots x_{n-1}]^T$, let y be its DCT, that is, $y = [y_0 \cdots y_{n-1}]^T$ with $y = Cx$, and let m be an integer with $1 \le m < n$. Then, choosing the first m coefficients $y_0, \ldots, y_{m-1}$ to form*

$$P_m(t) = \frac{1}{\sqrt{n}}\, y_0 \,+\, \sqrt{\frac{2}{n}} \sum_{k=1}^{m-1} y_k \cos\frac{k(2t+1)\pi}{2n} \tag{6.7}$$

minimizes the error $\sum_{i=0}^{n-1}(P_m(i)-x_i)^2$, *when approximating the* n *data points.*

Proof. We are trying to find coefficients $y_0,\ldots,y_{m-1}$ so that the error in matching the equations

$$P_m(i)=\frac{1}{\sqrt{n}}\,y_0\;+\;\sqrt{\frac{2}{n}}\sum_{k=1}^{m-1}y_k\cos\frac{k(2i+1)\pi}{2n}=x_i$$

is minimum. Following the notation in the proof of Theorem 6.2, the last equality above can be written as

$$C_m^T y=x,$$

where C_m is the matrix formed with the first m rows of C. This means that the columns of C_m^T are orthonormal and therefore $I=(C_m^T)^T C_m^T=C_mC_m^T$. The equation $C_m^T y=x$ is an overdetermined linear system and therefore we can find its least squares solution by using the corresponding normal equations. This gives

$$C_mC_m^T y=C_mx,\quad\text{or}\quad y=C_mx.$$

Thus, the minimum least square error is obtained by choosing the first m coefficients $y_0,\ldots,y_{m-1}$.

□

Example 6.1.3 *Consider the data vector* x *from Example 6.1.2. We perform DCT least squares approximation by dropping the last two terms of* $P_7(t)$ *to obtain*

$$\begin{aligned}P_5(t)\;&=\;\tfrac{1}{\sqrt{7}}\,(-0.8504)\;+\;\sqrt{\tfrac{2}{7}}\,\Big[\,2.4214\cos\tfrac{(2t+1)\pi}{14}+0.0715\cos\tfrac{2(2t+1)\pi}{14}\\&\qquad+1.9751\cos\tfrac{3(2t+1)\pi}{14}+0.8116\cos\tfrac{4(2t+1)\pi}{14}\,\Big].\end{aligned}$$

According to Theorem 6.4, this new function $P_5(t)$ *(although not an interpolant anymore) approximates the data points with a minimum error in the sense of least squares. Figure 6.2 shows* $P_5(t)$ *as a dashed curve.*

Thus, the DCT gives not just a more general application of least squares to problems such as the one studied in linear regression, but more importantly, it provides with an approximation with terms arranged in a very special fashion.

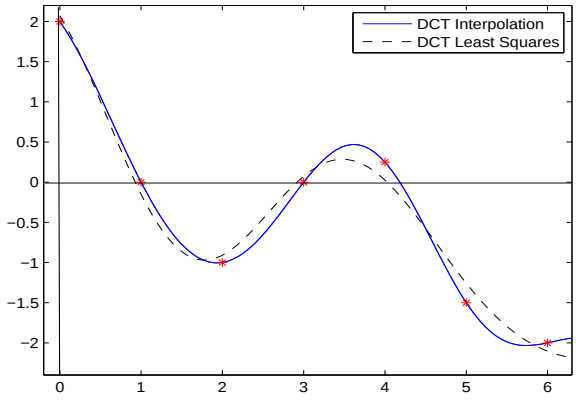

Figure 6.2: Interpolation and least squares using DCT

Remark 6.5 *Dropping terms in such a way for another class of interpolating functions such as Lagrange polynomials or splines would completely alter the interpolant, resulting in a function that is very far from being an approximation of the data points. However, we are able to do this with a DCT interpolating function because the terms are already arranged in order of importance.*

Since we are interested in image compression, we need to move to a 2-dimensional framework. But the above introduction to the one-dimensional DCT has given us a clear overall idea involved in the process of compressing. Now we just need to extend everything to two dimensions.

6.1.2 The 2-D discrete cosine transform

We start with the definition of the 2-dimensional version of the DCT, which is simply speaking the 1-d DCT applied twice. Given an input matrix X, the DCT is applied to the rows of X, and then the DCT is applied again to the rows of the resulting matrix. That is, we perform the matrix multiplications $C(CX^T)^T = C\,X\,C^T$. More formally, we have the following

Definition 6.6 *Let C be the $n \times n$ matrix defined in (6.1), and let X be an arbitrary $n \times n$ real matrix. Then, the 2-d DCT of X is defined as*

$$Y = C\,X\,C^T. \tag{6.8}$$

Remark 6.7 *Recall that the DCT matrix C is orthogonal and square, which implies that $C^{-1} = C^T$. Thus, the expression in (6.8) is a statement of similarity of matrices (see (2.63)), or more properly, a change of coordinates.*

One of the goals in image processing is to be able to recover the original image stored in an input matrix X. Here is where again the concept of orthogonality plays a crucial role. Observe that we can first multiply (6.8) by C^T from the left and then multiply by C from the right to obtain $C^TY\,C = C^TC\,X\,C^TC = I\,X\,I = X$. That is, we have

$$X = C^TY\,C. \tag{6.9}$$

The matrix X is then what is known as the 2-d Inverse Discrete Cosine Transform **(IDCT)** of the $n \times n$ matrix Y.

In a similar way as we did in Section 6.1.1, here we illustrate how the mathematical concept of interpolation is related to the IDCT in (6.9), and both in turn related to the technique of compressing images.

In a general one-dimensional interpolation problem, a function is found so that its graph is a curve that passes through a given set of points (t_i, x_i), $i = 0, \ldots, n-1$ in $\mathbb{R}^2$. For the case of 1-dimensional interpolation with DCT studied above, those points were (i, x_i), $i = 0, \ldots, n-1$. The two-dimensional case is similar, but now given a set of points (t_i, t_j, x_{ij}), $i = 1, \ldots, n$ in $\mathbb{R}^3$, we want to find a function whose graph is a surface that passes through the given points. See Figure 6.3. For the particular case of 2-dimensional interpolation with DCT those points are (i, j, x_{ij}), with $i, j = 0, \ldots, n-1$.

Theorem 6.8 (2-d DCT Interpolation) *Let C be the matrix in (6.1), and let X be any real $n \times n$ real matrix. If Y is the 2-d DCT of X, then the function*

$$P_n(s,t) = \frac{2}{\sqrt{n}} \sum_{k=0}^{n-1} \sum_{l=1}^{n-1} y_{kl} a_k a_l \cos \frac{k(2s+1)\pi}{2n} \cos \frac{l(2t+1)\pi}{2n} \tag{6.10}$$

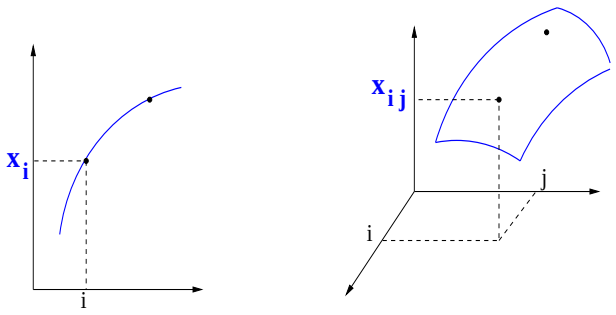

Figure 6.3: 1-D and 2-D interpolation

satisfies $P_n(i,j) = x_{ij}$, *for* $i,j = 0,\ldots,n-1$, *where*

$$a_k = \begin{cases} 1/\sqrt{2}, & k = 0 \\ 1, & k > 0. \end{cases}$$

In other words, the function $P_n(s,t)$ *interpolates the input data* (i,j,x_{ij}), *for* $i,j = 0,1,\ldots,n-1$.

Example 6.1.4 *Consider the input data matrix*

$$X = \begin{bmatrix} 1.0 & 0.8 & 1.0 & 1.0 & 0.8 & 1.0 \\ 1.0 & 0.5 & 0.3 & 0.0 & 0.5 & 1.0 \\ 1.0 & 0.3 & 0.2 & 0.0 & 0.3 & 1.0 \\ 1.0 & 0.2 & 0.0 & 0.0 & 0.2 & 1.0 \\ 1.0 & 0.3 & 0.2 & 0.0 & 0.3 & 1.0 \\ 1.0 & 0.8 & 1.0 & 1.0 & 0.8 & 1.0 \end{bmatrix}.$$

We want to perform 2-dimensional interpolation of this data by using the DCT through Theorem 6.8. We can consider each entry x_{ij} *of the matrix* X *as an assigned value at each grid point* (i,j), *like in Figure 6.3. First, we compute the DCT of* X, $Y = C\,X\,C^T$:

$$Y = \begin{bmatrix} 3.7500 & 0.0427 & 1.4901 & -0.1167 & 0.6010 & 0.1594 \\ 0.1077 & 0.0106 & -0.0354 & -0.0289 & -0.0911 & 0.0394 \\ 1.2247 & -0.0149 & -0.9500 & 0.0408 & -0.0866 & -0.0558 \\ -0.1500 & -0.0183 & 0.0612 & 0.0500 & 0.1061 & -0.0683 \\ 0.4950 & -0.0345 & -0.4619 & 0.0943 & 0.1000 & -0.1288 \\ 0.0077 & 0.0106 & -0.0354 & -0.0289 & 0.0503 & 0.0394 \end{bmatrix}.$$

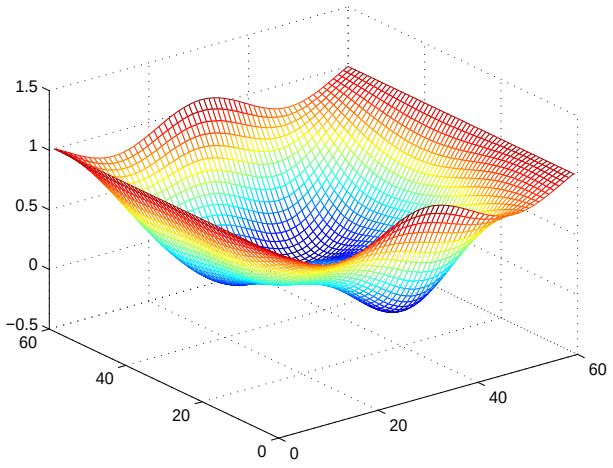

Figure 6.4: 2-D DCT interpolation of Example 6.1.4

Then, we compute the function $P(s,t)$ as in (6.10):

$$P_6 = \frac{2}{6}\left[\frac{1}{2}(3.75) + \frac{1}{\sqrt{2}}(0.0427)\cos\frac{(2t+1)\pi}{12} + \frac{1}{\sqrt{2}}(1.4901)\cos\frac{2(2t+1)\pi}{12} + \right.$$
$$\cdots + \frac{1}{\sqrt{2}}(0.0503)\cos\frac{5(2s+1)\pi}{16}\cos\frac{4(2t+1)\pi}{16}$$
$$\left. + \frac{1}{\sqrt{2}}(0.0394)\cos\frac{5(2s+1)\pi}{16}\cos\frac{5(2t+1)\pi}{16}\right].$$

This function, which passes through all the points (i,j,x_{ij}), is plotted in Figure 6.4

There are several important facts that need to be explained from Theorem 6.8, and in particular from (6.10). We start by realizing that what we observed in Section 6.1.1 for the one dimensional case also applies here. Namely, by recalling the definition of the DCT matrix C in (6.1) and by performing the matrix multiplication $X = C^T Y\, C$ in (6.8) componentwise, we can easily deduce that this gives exactly (6.10) with the property $P_n(i,j) = x_{ij}$, establishing the connection between the IDCT and the Interpolation Theorem 6.8. Once again,

we remark the fact that the coefficients y_{kl} of the interpolation in (6.10) are easily obtained through matrix multiplication $Y = C\,X\,C^T$, that is, by computing the DCT of the input matrix X.

By Remark 6.7, applying the DCT to an input matrix X amounts to a similarity transformation, or in other words, the DCT can be understood as a change of coordinates. In fact, in the applications language (say, image processing), the DCT is understood as a technique to convert a spatial domain waveform into its constituent frequency components (represented by a set of coefficients). Thus, the DCT is a change from spatial to frequency coordinates. It is exactly in the frequency framework where compression can take place, as we will see in the next section.

2-d DCT Least Squares Approximation. With the obvious modifications, Theorem 6.4 still applies here. Namely, we can zero some coefficients corresponding to large frequencies (some of the last few terms in (6.10)) and the error involved will be minimum in the sense of least squares. As in the 1-d case, the function obtained is not an interpolant anymore but it approximates the data points in an optimal way.

Since we are walking our way towards image compression, we are interested in dropping terms with high frequency. Now, given two distinct terms like $\cos(4t)\cos(5t)$ and $\cos(t)\cos(6t)$, which one has higher frequency, and therefore should be dropped? We can use the convention that the frequency of the term is given by the sum of the individual frequencies. Thus, e.g. $\cos(4t)\cos(5t)$ has a "total" frequency that is higher than that of $\cos(t)\cos(6t)$. Then, following the index notation in (6.10), that is, considering the matrix Y as having elements y_{kl}, with $k, l = 0, 1, \ldots n-1$, we want to zero e.g. those elements for which $k + l > m$, for a given $m < n$.

Example 6.1.5 *Consider the input data x_{ij} from Example 6.1.4, where $n = 6$. Out of a total of 36 terms in the interpolation function $P_6(s,t)$, first, we zero a total of 21 by requiring that we keep only those terms for which $k + l \leq 4$, and then we are less demanding and impose $k + l \leq 6$ which still eliminates 10 terms. Both least squares approximations are shown in Figure 6.5 Compare those approximations with the original interpolation of Figure 6.4.*

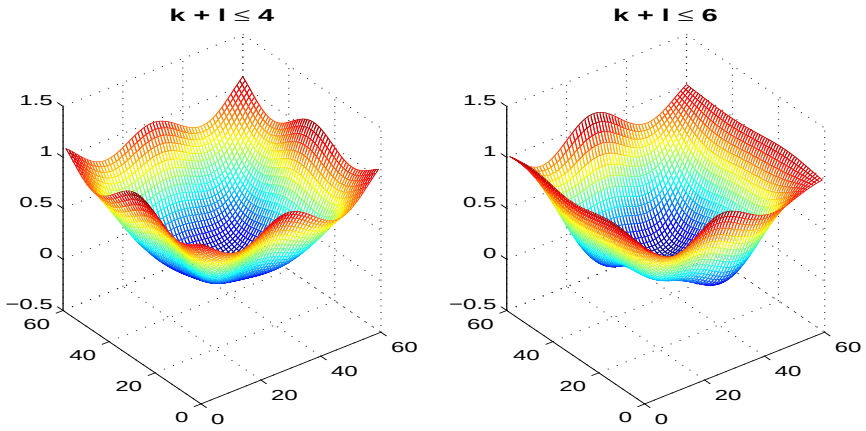

Figure 6.5: 2-d DCT least squares approximation

6.1.3 Image compression and the human visual system

To illustrate the idea of the great need of compressing images, consider a color picture measuring three by five inches that you shot using your digital camera, and at a resolution (which defines the quality of the image) of 400 dots per inch (dpi). The image is 15 in^2, and since each square inch has 400× 400=160,000 dots (or pixels), the image will contain a total of 2,400,000 pixels. Now, each pixel requires 3 bytes of data to store the different colors in the picture. Therefore, the image would require 7,200,000 bytes, which is about 7 MB of memory. Thus, storing such digital images without compression would mean using huge amounts of memory. In addition, these images would require large transfer times when sent electronically, especially for those with slow connection speeds. Several compression techniques have been developed to deal with this problem, all of them with the goal of compressing images to several times less than their original size, allowing for easier storage and transmission.

There are two main types of data compression: lossless and lossy. In **lossless compression** (such in zip and tar files) one is able to regain the original data after compression so that the quality of the image is not sacrificed; in fact, only redundant data is removed from the original file. In **lossy compression** some of the data is lost during the compression process, resulting in a final image that is of a lower quality than the original image, but such a loss of quality is in general not easily perceived by the human eye. Obviously, compression rates

in this case are much higher than those achieved with lossles compression.

Here we mostly discuss lossy compression, in particular the most common form of image compression, known as JPEG (about 80% of all web images today are JPEG encoded). A newer and more sophisticated version, JPEG 2000, has been developed but it is not yet widely used.

A word on the human visual system. The central idea in image processing is to exploit the unique characteristics of the human visual system in order to deliver images of optimal quality in color and detail, at a minimum cost. The human vision is sensitive to the visible portion of the electromagnetic spectrum we know as light. The incident light is focused on the retina, which contains photoreceptors called rods and cones. Rods give us the ability to see at very low light levels, whereas at relatively higher light levels, cones take over. However, we have fewer cones than rods. This may explain why we can discern fewer colors than we can discern a larger number of shades of gray.

We will see later that black and white pixels can be represented by a single number denoting the so called luminance. However colors have three attributes: brightness, hue and saturation and therefore they cannot be specified by a single number. We also know that the human vision has the highest sensitivity to yellow-green light, the lowest sensitivity to blue light, and red somewhere in between. In fact, evidence shows that the cones of the human retina can be classified into three types, with overlapping spectral sensitivities centered at about 700nm (red), 535 nm (green) and 445 nm (blue). See Figure 6.6. According to the tri-stimulus theory, the color of light entering the human visual system may be specified by only three numbers associated to three independent color sources. In optical systems these colors are Red, Green and Blue (RGB).

6.1.4 Basis functions and images

If we consider an input matrix X as an image block of grayscale values (0-255), the statements of interpolation (6.10) and that of the IDCT (6.9) tell us that such image can be written as the unique linear combination of basis functions given in terms of cosines (and it is possible to visualize such basis functions for any value of n). We want to start by illustrating the case $n = 4$.

In the vector space of 4×4 real matrices, we have the canonical basis

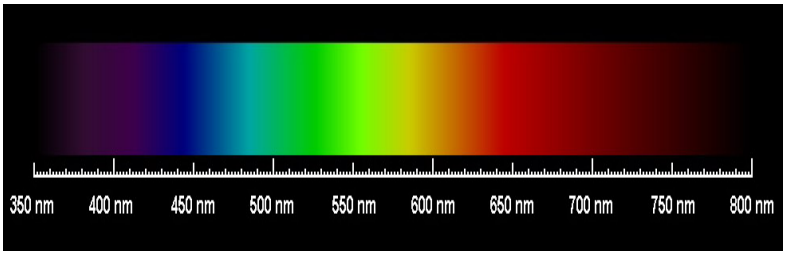

Figure 6.6: Visible spectrum

$$\mathcal{B} = \left\{ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad \cdots \quad , \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right\}. \tag{6.11}$$

Thus, any 4×4 real matrix A can be expressed as unique linear combination of the matrices of the basis $\mathcal{B}$. Following the change of coordinates reasoning, a simple way to obtain and visualize the standard DCT 4×4 basis functions is to compute the 2-d DCT of each matrix X in $\mathcal{B}$ as $Y = C\,X\,C^T$, where C is the matrix in (6.1), with $n = 4$, and then display Y as an image, through the MATLAB commands

$$\begin{array}{l} Y = C * X * C'; \\ \mathsf{colormap}(\mathsf{gray}); \\ \mathsf{imagesc}(Y). \end{array}$$

See Figure 6.7, where we show the DCT 4×4 basis functions resulting from these calculations. For illustration, here are the Y matrices in the new basis, corresponding to the first two X matrices in the basis $\mathcal{B}$ above.

$$\begin{bmatrix} 0.2500 & 0.2500 & 0.2500 & 0.2500 \\ 0.2500 & 0.2500 & 0.2500 & 0.2500 \\ 0.2500 & 0.2500 & 0.2500 & 0.2500 \\ 0.2500 & 0.2500 & 0.2500 & 0.2500 \end{bmatrix}, \quad \begin{bmatrix} 0.3266 & 0.1353 & -0.1353 & -0.3266 \\ 0.3266 & 0.1353 & -0.1353 & -0.3266 \\ 0.3266 & 0.1353 & -0.1353 & -0.3266 \\ 0.3266 & 0.1353 & -0.1353 & -0.3266 \end{bmatrix}.$$

This means that each of the 16 images in Figure 6.7 is the image display of the 2-d DCT of each of the corresponding matrices in the basis $\mathcal{B}$. But more importantly, this means that any 4×4 grayscale image block can be obtained

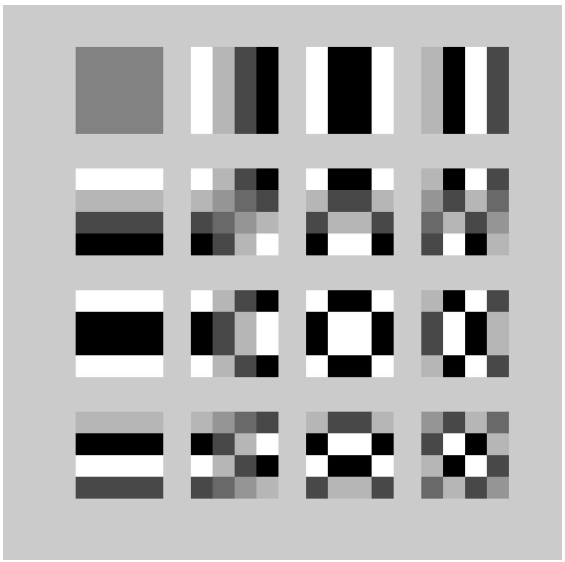

Figure 6.7: DCT 4×4 basis images

or expressed as a unique linear combination of the 16 basis functions shown in Figure 6.7.

Remark 6.9 *The* MATLAB *command* imagesc(Y) *rescales the entries of* Y *to values in the interval* $[0,\ 255]$ *and then prints grayscales corresponding to each entry, where* 0 *corresponds to black and* 255 *corresponds to white.*

Example 6.1.6 *Consider the* 4×4 *image of Figure 6.8, call it* Y*. Then,* Y *can be uniquely written as the linear combination of the DCT* 4×4 *basis elements shown in Figure 6.7. More precisely, if we name such basis as* $\{Y_0, Y_1, \ldots, Y_{15}\}$*, then the image can be decomposed as*

$$Y = Y_0 + 0.5Y_1 - 3Y_2 + 4Y_3 + 2Y_4 - 1.5Y_5 - 2.5Y_6 - Y_7 - Y_8 + 4Y_9$$

$$-3Y_{10} - 0.5Y_{11} + 2.5Y_{12} + 3.5Y_{13} + 3Y_{14} - Y_{15}.$$

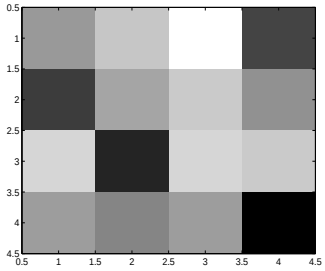

Figure 6.8: 4×4 image Y of Example 6.1.6

Similarly, any other 4×4 grayscale image can also be written as a unique combination of the basis functions $Y_0, \ldots, Y_{15}$.

But beyond this illustration, we are mostly interested in 8×8 images, because for image compression, an arbitrary figure will be decomposed into hundreds or thousands of 8×8 pixel values. To obtain the DCT 8×8 basis shown in Figure 6.9, we can proceed in exactly the same way we did to obtain the basis in the 4×4 case, and any 8×8 grayscale image will be the unique linear combination of such 64 basis elements.

Remark 6.10 *There are other transforms that can be used in a similar fashion for the purpose of image compression, like the Haar transform (Exercise 6.17) or the Hadamard transform (Exercise 6.18), but it has been shown that with the DCT, the mean square error between the original image and the reconstructed image decreases fastest as the number of basis images used increases.*

6.1.5 Low-pass filtering

We want to start by considering grayscale images, and later on we will generalize this discussion to color images. Any digital image is composed of pixels, which can be thought of as small dots on the screen. Consider for instance the image of Figure 6.10(a), which is a 512×512 array of pixels. Mathematically speaking, this grayscale image is a 512×512 matrix X (the input matrix), where each entry has a value between 0 and 255, corresponding to how dark or bright the pixel at the corresponding position should be; the value 0 corresponds to black and 255 to white. Thus, if we zoom-in the picture enough, we will see only

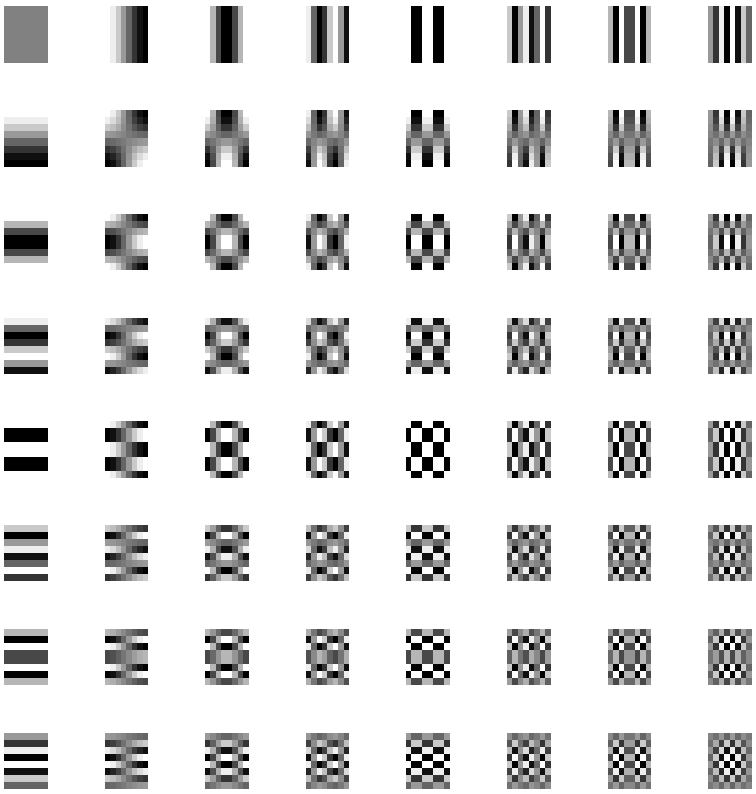

Figure 6.9: DCT 8×8 basis images

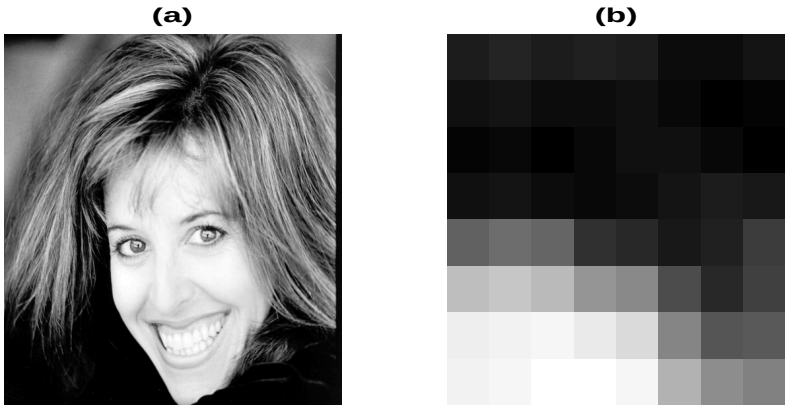

Figure 6.10: Original image and one 8× 8 block

small boxes of different grayscales, such as the one in Figure 6.10(b), which corresponds to an area around the the left eye of the person in the picture.

Assume the picture is stored as a JPEG image face.jpg. Then, we can import it into MATLAB through the command

```
A=imread('face.jpg');
```

Although the DCT can be applied to the whole matrix A at once, we will consider this matrix A as composed of several 8×8 image blocks, like the one showed in Figure 6.10 (b), and we will successively apply the DCT to each block. At the same time, this will allow us to better illustrate how the DCT works on such matrices. The grayscales of Figure 6.10 (b) are the entries in the matrix

$$X = \begin{bmatrix} 30 & 35 & 30 & 32 & 31 & 17 & 17 & 24 \\ 20 & 25 & 19 & 17 & 22 & 14 & 10 & 12 \\ 12 & 15 & 10 & 16 & 20 & 21 & 14 & 7 \\ 22 & 23 & 17 & 15 & 17 & 25 & 29 & 28 \\ 84 & 91 & 86 & 45 & 40 & 27 & 33 & 55 \\ 154 & 160 & 151 & 124 & 115 & 66 & 41 & 58 \\ 190 & 195 & 198 & 187 & 175 & 111 & 75 & 76 \\ 194 & 198 & 203 & 205 & 198 & 145 & 116 & 107 \end{bmatrix}, \tag{6.12}$$

which we can be thought of as an input matrix. Before applying the DCT to X, there is an optional and technical step called level shifting, which changes the

values in the interval [0, 255] to [−128, 127], thus converting them into signed bytes, centered around zero. This can be achieved by subtracting 128=2^7 (in general, we subtract 2^{n-1}, where 2^n is the maximum number of gray levels). The shifted matrix, which we still call X, is

$$X = \begin{bmatrix} \text{-98} & \text{-93} & \text{-98} & \text{-96} & \text{-97} & \text{-111} & \text{-111} & \text{-104} \\ \text{-108} & \text{-103} & \text{-109} & \text{-111} & \text{-106} & \text{-114} & \text{-118} & \text{-116} \\ \text{-116} & \text{-113} & \text{-118} & \text{-112} & \text{-108} & \text{-107} & \text{-114} & \text{-121} \\ \text{-106} & \text{-105} & \text{-111} & \text{-113} & \text{-111} & \text{-103} & \text{-99} & \text{-100} \\ \text{-44} & \text{-37} & \text{-42} & \text{-83} & \text{-88} & \text{-101} & \text{-95} & \text{-73} \\ 26 & 32 & 23 & \text{-4} & \text{-13} & \text{-62} & \text{-87} & \text{-70} \\ 62 & 67 & 70 & 59 & 47 & \text{-17} & \text{-53} & \text{-52} \\ 66 & 70 & 75 & 77 & 70 & 17 & \text{-12} & \text{-21} \end{bmatrix} \tag{6.13}$$

Our first step consists on applying the DCT to the matrix X. This gives (after rounding):

$$Y = CXC^T = \begin{bmatrix} -455 & 148 & -35 & -16 & 14 & -24 & -2 & 10 \\ -440 & -129 & 45 & 12 & -15 & 10 & -3 & -9 \\ 179 & 32 & -49 & 6 & 16 & 0 & -6 & 1 \\ 27 & 56 & 17 & -22 & 5 & -12 & 4 & 6 \\ -14 & -38 & 21 & -4 & -6 & 6 & 0 & 0 \\ 4 & -1 & -16 & 7 & 4 & 4 & -2 & -3 \\ 5 & 2 & -4 & 4 & 2 & -1 & -1 & -2 \\ 4 & 6 & 3 & -6 & -2 & 0 & 2 & 2 \end{bmatrix}. \tag{6.14}$$

It is now evident that using X, the DCT has produced a matrix Y such that its entries around the upper left corner have the largest magnitude, whereas the ones around the lower right corner have the lowest one. (and this will be true for any given input matrix X). The entries of the matrix Y are known as the **DCT coefficients**. By recalling the discussion about equations (6.9) and (6.10), that is, the image X can be represented as a combination of cosine basis functions with the DCT coefficients acting as weights, we observe that the largest weights are associated with the basis elements with lower frequency (the upper left corner), and the smallest weights are associated with the basis elements with higher frequency (lower right corner).

We are at the heart of the DCT action: it has produced a change of coordinates from the image input signal to the frequency coordinates, and it has arranged it in increasing order of frequency. In terms of image compression, it is ideal,

since the human visual system is more sensitive to lower frequencies than to higher ones. Thus, in terms of human vision, the first terms in (6.10) are far more important to the last terms. Accordingly, the DCT has therefore given (through the entries in the upper left corner of Y) more weight to those functions with lower frequency.

Now we can try our first compression strategy, in a similar way as we did for the one-dimensional case: drop some terms in the lower right corner of Y (that is, a few of the last terms in (6.10)) to obtain a new matrix $\overline{Y}$ and then apply the IDCT to this new matrix. Of course, we will not obtain the original image but a compressed one, technically of lower quality, but still for the most part the difference is not very much perceived by the human eye. This simple technique is called low-pass filtering.

Thus, suppose we decide to zero the diagonal and the lower triangular part of Y. Then, this filtering gives

$$\overline{Y} = \begin{bmatrix} -455 & 148 & -35 & -16 & 14 & -24 & -2 & 0 \\ -440 & -129 & 45 & 12 & -15 & 10 & 0 & 0 \\ 179 & 32 & -49 & 6 & 16 & 0 & 0 & 0 \\ 27 & 56 & 17 & -22 & 0 & 0 & 0 & 0 \\ -14 & -38 & 21 & 0 & 0 & 0 & 0 & 0 \\ 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{6.15}$$

To reconstruct the (compressed) image, we apply the IDCT to this matrix, that is we calculate $C^T \overline{Y} C$ and then we add back 128 to each entry to obtain (after rounding)

$$\overline{X} = \begin{bmatrix} 32 & 34 & 31 & 30 & 27 & 18 & 18 & 26 \\ 19 & 23 & 21 & 23 & 25 & 17 & 8 & 6 \\ 10 & 14 & 10 & 11 & 21 & 21 & 13 & 10 \\ 24 & 30 & 19 & 10 & 17 & 22 & 26 & 34 \\ 82 & 89 & 75 & 54 & 43 & 32 & 33 & 48 \\ 153 & 163 & 153 & 131 & 105 & 66 & 46 & 56 \\ 190 & 198 & 195 & 188 & 166 & 113 & 75 & 78 \\ 194 & 199 & 199 & 208 & 201 & 150 & 108 & 109 \end{bmatrix}.$$

Obviously, the compressed image is not exactly the same as the original image ($\overline{X} \neq X$), but the difference between them is not easily perceived by the human

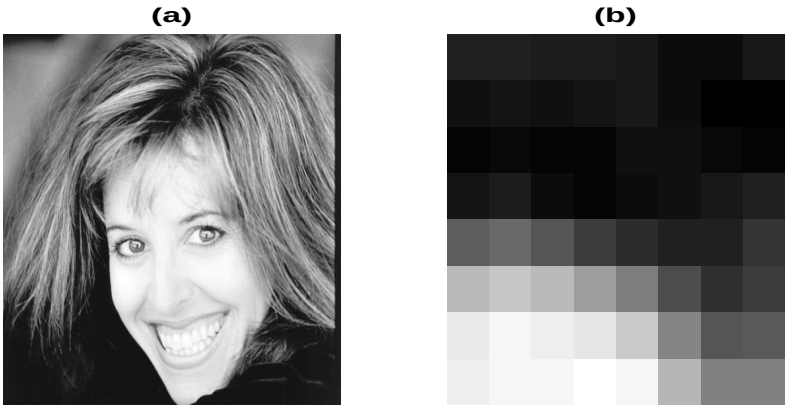

Figure 6.11: DCT low-pass filtering

eye. Compare Figure 6.10 (b) with Figure 6.11 (b). Even more, these 8×8 image blocks are just very small pieces of a given actual image. Thus, we can expect not to notice the small changes in the compressed image even though, as in this case, we have reduced storage requirements by about 50%. To actually apply this method to an entire image, we apply the above technique to each 8×8 block and then build up the compressed image from the compressed blocks. Figure 6.11 (a) shows the compressed image, which should be compared with the original Figure 6.10(a).

Remark 6.11 *This low-pass filtering technique is related to the 2-d DCT interpolation and least squares approximation discussed in Section 6.1.2; that is, the error involved when dropping some of the last DCT coefficients is minimum in the sense of least squares.*

6.1.6 Quantization

The low-pass filtering compression technique presented above is effective but it sure can be improved. While still trying to zero the DC coefficients associated with the largest frequencies, now at the same time we want to rescale the remaining nonzero coefficients in such a way that fewer bits are necessary for their storage. Since the lower frequency terms are the most important ones, we would

like to apply a moderate rescaling to them, while applying a more aggressive rescaling (if possible down to zero) to the higher frequency terms.

There are several possible methods to perform this nonuniform rescaling, which in the language of compression is known as quantization. Most of these methods define a so called quantization matrix Q so that Y is entrywise divided by Q and then rounded to obtain a new quantized matrix

$$Y_Q = \text{round}\left(\frac{y_{kl}}{q_{kl}}\right). \tag{6.16}$$

Clearly, here an error is introduced due to rounding; this is why this technique falls into the category of lossy compression. One such quantization matrix Q can be defined as

$$q_{kl} = 8s\,(k+l+1), \quad 0 \le k, l \le 7.$$

That is,

$$Q = s\begin{bmatrix} 8 & 16 & 24 & 32 & 40 & 48 & 56 & 64 \\ 16 & 24 & 32 & 40 & 48 & 56 & 64 & 72 \\ 24 & 32 & 40 & 48 & 56 & 64 & 72 & 80 \\ 32 & 40 & 48 & 56 & 64 & 72 & 80 & 88 \\ 40 & 48 & 56 & 64 & 72 & 80 & 88 & 96 \\ 48 & 56 & 64 & 72 & 80 & 88 & 96 & 104 \\ 56 & 64 & 72 & 80 & 88 & 96 & 104 & 112 \\ 64 & 72 & 80 & 88 & 96 & 104 & 112 & 120 \end{bmatrix}. \tag{6.17}$$

Thus, for larger values of s more compression will be applied. Observe that the entries of Q at the upper left corner are small, because we expect to have large values in the matrix Y at those positions, and entrywise division by Q and rounding will merely rescale to numbers of smaller magnitude, requiring therefore smaller bits for storage. At the same time, the elements at the lower right corner of Q are large, and since we expect to have small values in the matrix Y at those positions, entrywise division by Q and rounding will set most of them to zero and the rest will be rescaled to smaller magnitude and require fewer bits for storage. This is clearly more efficient than low-pass filtering.

The JPEG standard in its Appendix K ("Examples and Guidelines") recommends quantization matrices that are based on psychovisual thresholding and derived empirically in experiments with the human visual system, and therefore, from the practical point of view, are more reliable. For the case of grayscales, the so called **luminance quantization matrix** they recommend is

$$Q = s \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \tag{6.18}$$

Now let us apply this luminance quantization to (6.14), first with the parameter $s = 1$, by using (6.16). This gives

$$Y_Q = \begin{bmatrix} -28 & 13 & -4 & -1 & 1 & -1 & 0 & 0 \\ -37 & -11 & 3 & 1 & -1 & 0 & 0 & 0 \\ 13 & 2 & -3 & 0 & 0 & 0 & 0 & 0 \\ 2 & 3 & 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6.19}$$

Compare this matrix Y_Q with the one in (6.15) form low-pass filtering.

To recover back the (compressed) image, we apply the reverse process; that is, we first multiply entrywise Y_Q by Q (this is where an error is introduced) to obtain a modified $Y = QY_Q$. Then we apply the IDCT to Y : $X = C^T Y\, C$, and finally we add back 128 to X. In Figure 6.12 we show the images obtained when using $s = 1$ and $s = 4$. For $s = 1$ the compressed image is quite similar to the original one in Figure 6.10 (b), while for $s = 4$, some differences are already noticeable.

We can now compress the image of Figure 6.10(a) by applying the above process to each 8×8 image block and then reconstruct the image by putting together the compressed blocks. Figure 6.13 shows the results for $s = 1$ and $s = 4$.

As illustration to estimate how much memory in terms of bits we save by applying luminance quantization, consider an arbitrary 8×8 image block, and as a worst case scenario, assume each entry in $Y = CXC^T$ is the number 255, the largest possible. If we apply quantization through (6.16) and (6.18) to Y, we

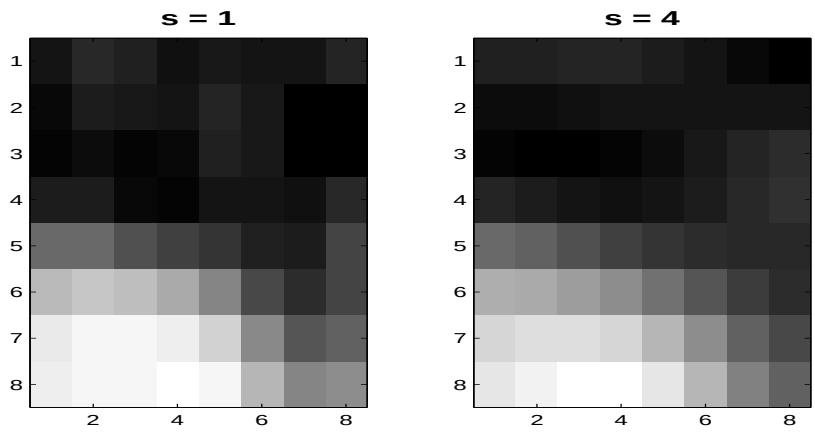

Figure 6.12: DCT luminance quantization using (6.18)

Figure 6.13: DCT luminance quantization using (6.18)

obtain the matrix Y_Q below.

$$Y_Q = \begin{bmatrix} 16 & 23 & 26 & 16 & 11 & 6 & 5 & 4 \\ 21 & 21 & 18 & 13 & 10 & 4 & 4 & 5 \\ 18 & 20 & 16 & 11 & 6 & 4 & 4 & 5 \\ 18 & 15 & 12 & 9 & 5 & 3 & 3 & 4 \\ 14 & 12 & 7 & 5 & 4 & 2 & 2 & 3 \\ 11 & 7 & 5 & 4 & 3 & 2 & 2 & 3 \\ 5 & 4 & 3 & 3 & 2 & 2 & 2 & 3 \\ 4 & 3 & 3 & 3 & 2 & 3 & 2 & 3 \end{bmatrix}, \quad \text{Bits} : \begin{bmatrix} 6 & 6 & 6 & 6 & 5 & 4 & 4 & 4 \\ 6 & 6 & 6 & 5 & 5 & 4 & 4 & 4 \\ 6 & 6 & 6 & 5 & 4 & 4 & 4 & 4 \\ 6 & 5 & 5 & 5 & 4 & 3 & 3 & 4 \\ 5 & 5 & 4 & 4 & 4 & 3 & 3 & 3 \\ 5 & 4 & 4 & 4 & 3 & 3 & 3 & 3 \\ 4 & 4 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

Since the number r of bits necessary to store a given number n can be estimated as

$$r = \lfloor \log_2(n) \rfloor + 2,$$

where the function $\lfloor x \rfloor$ is the largest integer less than or equal to x, we have calculated the bits necessary to represent the entries in Y_Q, and shown them on the matrix next to it. Thus, adding up the 64 numbers in the matrix of bits, we get 266, which is about half the bits necessary to store the original 8×8 image without compression.

Thus, for general images, by applying quantization with $s = 1$, we can save about 50% of memory storage and still obtain a compressed image that to the human eye has been perfectly reconstructed. For larger values of the parameter s, more compression can be applied to the image, resulting in smaller file size and therefore in more memory saving, but at the same time it also means losing more quality. Thus, it all depends on the application at hand, or on how much quality we want to trade off for memory. In any case, the parameter s in (6.18) allows flexibility and an easy way for testing different compression rates (for $s = 4$ the number of total number of bits is 166, which is about 32% of the original size).

6.1.7 Compression of color images

When we think of matrices, we automatically think of 2-dimensional arrays of rows and columns, just like the ones we have been working with so far. However, if we import a color picture into MATLAB, say through the command `A=imread('face.jpg')`, and then check the size of A we observe that such matrix is 3-dimensional, e.g. $512 \times 512 \times 3$. This matrix can be understood as three layers

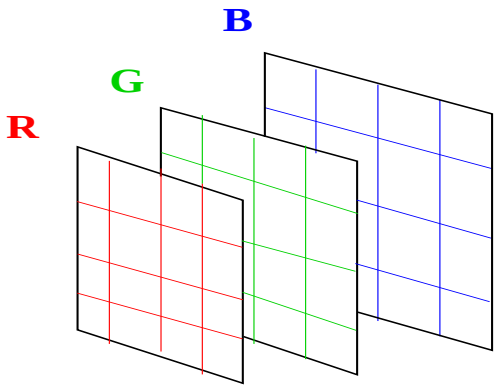

Figure 6.14: Three dimensional array of a color image

of two-dimensional matrices. In particular, for color images, the three layers correspond to Red, Green and Blue **(RGB)** intensities (see Figure 6.14). For black and white images, each pixel corresponds to a number between 0 and 255 acoording to its grayscale. For color images, each pixel is given three numbers representing the three color intensities.

Several approaches can be taken to compress color images. The simplest one would be to treat each color (or layer) independently, that is, compression can be applied to each color as if we were dealing with grayscale intensities and then reconstruct the (compressed) image from the superposition of the colors. This works, but it is not efficient. A second, and very popular approach is the one outlined by the so called Baseline JPEG. The central idea again comes from the practical point of view: the human eye is more sensitive to luminance (changes in brightness) than to chroma (changes in color). This real fact gives a hint: we should be able to perform higher rates of compression in the chrominance coordinates and still make it unnoticeable to the human eye. Recall that for grayscale images we only had luminance.

Therefore, instead of working just with plain colors (RGB), we perform a change of coordinates to color differences, or chroma **(YUV)**:

$$Y = 0.299R + 0.587G + 0.114B, \qquad U = B - Y, \quad V = R - Y. \tag{6.20}$$

Remark 6.12 *The coefficients of R, G and B in the Y coordinate agree with the fact that out of the three colors, the human eye is most sensitive to green and least sensitive to blue, with red somewhere in the middle.*

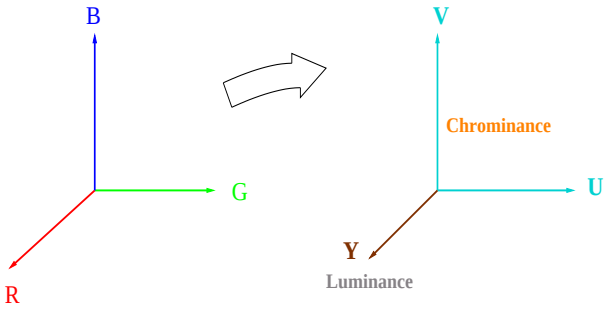

Figure 6.15: Change of coordinates from RGB to Y-UV

Through this change of coordinates, the color image can be represented in (y, u, v) form. We perform compression in the Y coordinate as if we were working with grayscale images, that is, we can quantize the data using the luminance matrix in (6.18). Then, independently we can perform a more aggressive compression in the UV coordinates, by using a less conservative quantization matrix.

JPEG recommends the following **Chrominance matrix**

$$Q_c = s \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}. \tag{6.21}$$

The difference between the luminance and chrominance matrices (6.18) and (6.21) respectively is obvious.

For compression of a color image, we proceed in the following way

- group pixel values for each of the three components into 8×8 blocks.
- transform each block X by applying the DCT to it to obtain the DCT coefficients.
- apply luminance quantization (6.18) to the Y coordinates, and chrominance quantization (6.21) to the U and V coordinates (6.20).

Figure 6.16: Original image and 8× 8 block

The DCT coefficients either will be zeroed or reduced in size for storage. The decompression process is just the reverse algorithm, as explained before, except that now we also need to go back to the RGB coordinates through the equations

$$\begin{aligned} B &= U + Y \\ R &= V + Y \\ G &= (Y - 0.299R - 0.114B)/0.587 \end{aligned}$$

We are going to apply these ideas to the color image of Figure 6.16 (a); the 8×8 block was taken from a part of the lady's hat. As usual, we load the image into MATLAB with the command imread, which represents the image in this case as a $512 \times 512 \times 3$ matrix: 512 rows, 512 columns and (combinations of) three colors: red, green and blue. Figures 6.17 (a) and (b) show the results of applying the recommended JPEG luminance and chrominance quantization matrices for $s = 3$.

Note. To obtain a similar quality of compressed image for the grayscale case, we had to take just $s = 1$ and no higher.

Figure 6.17: Compressed images with $s = 3$

6.2 Huffman Coding

A central step in the lossy compression technique described above consists on applying a quantization matrix of the type (6.18) or (6.21) to the DCT of the original image matrix X to obtain a matrix Y of quantized coefficients of the form (6.19). The entries of this matrix Y with a large number of zeros are the numbers we need to store. But this has to be done in an efficient way. This is the point where an additional modest amount of compression can be achieved, but this time it is lossless compression.

Those entries of Y will be stored as binary digits, or bits, that is, as sequences of 0's and 1's. We want to explain how this is done.

The first question to answer is the following: given a general string of symbols, what is the minimum number of bits needed to code such string? It turns out that the answer is related to the probability with which each symbol occurs in the string. Suppose there are n different symbols available, and let p_i be the probability of the occurrence of symbol i in any given string. Then we define the **entropy** of the string as

$$H = -\sum_{i=1}^{n} p_i \log_2(p_i). \tag{6.22}$$

This entropy H tries to quantify the average minimum number of bits per symbol needed to code the string.

Example 6.2.1 *Let us find the entropy of the string BDABBCDB. The probabilities of the symbols A,B,C,D are respectively:* $p_1 = 1/8$, $p_2 = 4/8$, $p_3 = 1/8$, $p_4 = 2/8$, *or expressed as powers of two:* $p_1 = 2^{-3}$, $p_2 = 2^{-1}$, $p_3 = 2^{-3}$, $p_4 = 2^{-2}$. *Then, the entropy of the string is*

$$H = -\sum_{i=1}^{4} p_i \log_2(p_i) = \frac{1}{8}(3) + \frac{4}{8}(1) + \frac{1}{8}(3) + \frac{2}{8}(2) = \frac{14}{8} = 1.75$$

Thus, the entropy formula indicates that the minimum number of bits per symbol needed to code the string BDABBCDB is 1.75.

Taking this as a starting point, several coding techniques have been developed to code strings of symbols, but it is the Huffman coding the one that comes closer to achieve this minimum. This process is better explained through a detailed example.

Suppose we have the following symbols and their corresponding probabilities of occurrence in a string

Symbol	Probability
A	0.35
B	0.25
C	0.14
D	0.11
E	0.11
F	0.04

Then, to code these symbols we proceed as follows:

Building the tree. (See Figure 6.18)

1. We combine two symbols with the lowest probabilities, say E and F, to obtain the symbol EF with probability 0.15.

2. Now we have five symbols left, A, B, C, D and EF. We combine the two with lowest probabilities, C and D to obtain the symbol CD with probability 0.25.

3. From the four symbols left A, B, CD and EF we combine two with the lowest probabilities, say CD and EF to obtain the symbol CDEF with probability 0.40.

4. Now we have three symbols left, A, B and CDEF. We combine the two with the lowest probabilities, A and B, to obtain the symbol AB with probability 0.60.

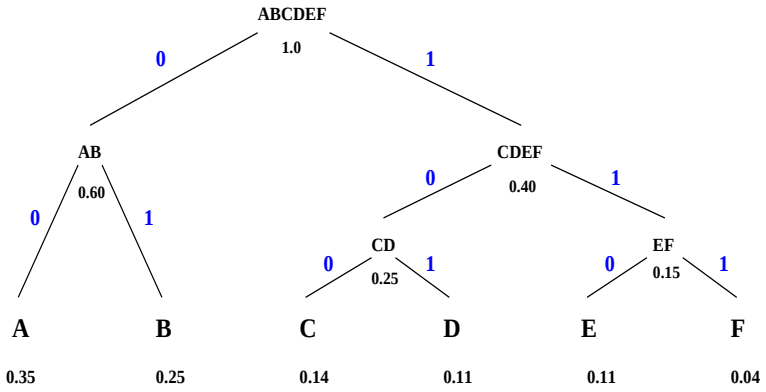

Figure 6.18: Huffman coding tree

5. Finally, we combine the remaining two symbols AB and CDEF to obtain the symbol ABCDEF with probability 1.0.

Assigning the codes.

At this step we translate the string of symbols into a bit stream, by first obtaining the Huffman code for each symbol. This is done by arbitrarily assigning a bit of 0 to a left branch and a bit of 1 to a right branch. Once this is done, we start at the top of the tree and we read the symbols as:

A=00	C=100	E=110
B=01	D=101	F=111

Now we can translate a string of those symbols into bits. By instance, the string

ACDAACBBEB

is translated as

(00)(100)(101)(00)(00)(100)(01)(01)(110)(01)

This bit stream has length 24 and therefore it uses 24/10=2.4 bits per symbol.

Uniqueness. In step 1 of building the tree we could have also combined the symbols D and F first. Similarly, in step 3 we could have chosen to combine the symbols B and EF instead of combining CD and EF. The idea is to combine arbitrary symbols with the lowest probabilities. By picking different choices we

obtain in general different codes for the symbols, which implies that a Huffman code is not unique. However, the average size will remain the same. For the example above it will always be 2.4 bits per symbol.

6.2.1 Huffman coding and JPEG

With the basic background introduced above we can now explain how to encode the DCT coefficients, (the entries of the quantized matrix Y). Recall that we partition the matrices in 8×8 blocks so that in fact we are dealing with 64 DCT coefficients at a time. We also know that the first of these coefficients, which is known as the DC coefficient, is the most important as it has the largest weight or magnitude, and that all other 63 coefficients are smaller and decrease in magnitude as we read the matrix Y toward the lower right corner (in fact the majority are zeros). These 63 coefficients are known as AC coefficients. Because of this main difference they are coded separately.

6.2.1.1 Coding the DC coefficients

Since we expect some correlation between neighboring 8×8 blocks, instead of coding individual DC coefficients for each block the strategy is to code their differences (see Figure 6.19). The larger the correlation, the smaller the difference. That is, we will code the difference D between the DC coefficients of two neighboring blocks k and $k+1$:

$$D = (DC)_{k+1} \; - \; (DC)_k, \quad k = 1, 2, \ldots, \tag{6.23}$$

where $(DC)_k$ is initially set to zero.

The DC coefficient difference D will be represented as two symbols, the first one for its bit size and the second one for its value D in (6.23). That is,

$$\begin{pmatrix} \text{Symbol 1 for} \\ \text{Bit Size} \end{pmatrix} \begin{pmatrix} \text{Symbol 2 for} \\ \text{Diff Value } D \end{pmatrix}. \tag{6.24}$$

Here we define the bit size of an integer z as

$$S = \begin{cases} \lfloor \log_2 |z| \rfloor + 1, & z \neq 0 \\ 0 & z = 0. \end{cases} \tag{6.25}$$

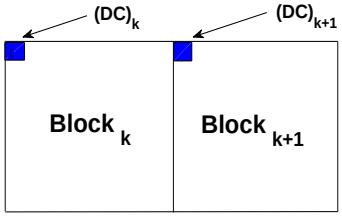

Figure 6.19: DC coefficients of neighboring blocks

Bit Size	Code	Bit Size	Code
0	00	6	1110
1	010	7	11110
2	011	8	111110
3	100	9	1111110
4	101	10	11111110
5	110	11	111111110

Table 6.1: Codes for DC symbol 1

Given a particular DC coefficient difference we first find the bit size S of that difference through (6.25). Next, to get symbol 1 for S we use Table 6.1, where the codes shown were obtained by building a tree similar to that in Figure 6.18 (see Exercise 6.27).

Example 6.2.2 *Suppose the DC coefficient difference between two neighboring blocks is $D = 9$. From (6.25), its bit size is $S = 4$, and according to Table 6.1, symbol 1 should be* **101**.

To obtain symbol 2 in (6.24) we use n bits if the bit size S of the difference is n. But since there are several integer coefficients (positive and negative) that have the same size S, they are grouped together by bit size. Then each one in the group is assigned a unique combination of 0's and 1's according to Table 6.2.

S	Difference Value D	Code
0	0	
1	-1, 1	0, 1
2	-3, -2, 2, 3	00, 01, 10, 11
3	-7, -6, -5, -4, 4, 5, 6, 7	000, 001, 010, 011, 100, 101, 110, 111
4	-15, -14, . . . , -8, 8, . . . ,14, 15	0000, 0001,. . . ,0111, 1000,. . . ,1110, 1111
5	-31, -30,. . . ,-16, 16,. . . ,30, 31	00000, 00001,. . . ,01111, 10000,. . . ,11110, 11111
6	-63, -62,. . . ,-32, 32,. . . ,62, 63	000000, 0000001,. . . ,011111, 100000,. . . ,111110, 111111
7	-127, -126,. . . ,-64, 64,. . . ,126, 127	0000000, 0000001,. . . ,0111111, 1000000,. . . ,1111110, 1111111
.	.	.
.	.	.
.	.	.

Table 6.2: Codes for DC/AC symbol 2

Example 6.2.3 *In Example 6.2.2 we had $D = 9$, with $S = 4$. Then, by looking at Table 6.2 we conclude that symbol 2 is* **1001**. *Thus, from (6.24) the complete representation of the DC coefficient difference $D = 9$ is*

$$(101)(1001),$$

where the parenthesis is only for notational convenience and clarity.

6.2.1.2 Coding the AC coefficients

We know that a great majority of the 63 AC coefficients will likely be zero, as a result of the quantization process, and it is very likely that a high frequency coefficient will be zero given that its predecessors are zero. This implies that there will be runs of zeros in the AC coefficients. We exploit the presence of these runs of zeros by using a zigzag scanning as illustrated in Figure 6.20 when reading the coefficients, because this scanning tends to group longer runs of zeros.

The AC coefficient will be coded as two symbols. We use the first symbol to represent the pair

$$(r, S),$$

where r is the length of a run of zeros, that is, the number of consecutive zero AC coefficients, and S is the bit size of the next nonzero entry. The corresponding code for each pair is obtained from Table 6.3. The second symbol represents the value of the AC coefficient; the corresponding code for this value comes as before from Table 6.2.

Thus, the representation has the form

$$\begin{pmatrix}\text{Symbol 1 for}\\(r,S)\end{pmatrix}\begin{pmatrix}\text{Symbol 2 for}\\\text{AC Value}\end{pmatrix}, \tag{6.26}$$

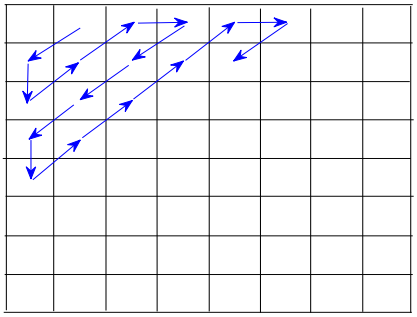

Figure 6.20: Zigzag pattern for AC coefficients

where as usual, S comes from (6.25).

Example 6.2.4 *Suppose we have the following AC coefficients*

$$9, 6, 0, 0, 0, 0, -3.$$

For the first coefficient 9 we have $(r, S) = (0, 4)$ *because it contains no zeros and because from (6.25) the size of 9 is* $S = 4$*. Thus, from Table 6.3 its symbol 1 is* **1011**. *For symbol 2 we conclude from Table 6.2 that the code for 9 is* **1001**.

Similarly, for the coefficient 6 we have $(r, S) = (0, 3)$*, because* $r = 0$ *and from (6.25) the size of 6 is* $S = 3$*. Thus, from Table 6.3 symbol 1 is* **100**. *To obtain symbol 2 we observe from Table 6.2 that the code for 6 is* **110**.

Next we have four consecutive zeros followed by -3*. Then, we have* $(r, S) = (4, 2)$*, because the run of zeros has length 4, and the size of* -3 *is* $S = 2$*. Thus, from Table 6.3 symbol 1 is* **1111111000**. *Finally, we observe from Table 6.2 that the code for* -3 *is* **00**.

Thus, the given seven AC coefficients are coded as

$$(1011)(1001)\ (100)(110)\ (1111111000)(00),$$

where again the parentheses are just for notational convenience and clarity.

Note: In the example above, if -3 was the very last coefficient from the quantized matrix, then the code above must be finished with EOB (end of block), that is, with (1010). See Table 6.3.

(r, S)	Code	(r, S)	Code
(0,1)	00	(5,1)	1111010
(0,2)	01	(5,2)	11111110111
(0,3)	100	(5,3)	1111111110011110
(0,4)	1011	(5,4)	1111111110011111
(0,5)	11010	(5,5)	1111111110100000
⋮	⋮	⋮	⋮
(1,1)	1100	(6,1)	1111011
(1,2)	11011	(6,2)	111111110110
(1,3)	1111001	(6,3)	1111111110100110
(1,4)	111110110	(6,4)	1111111110100111
(1,5)	11111110110	(6,5)	1111111110101000
⋮	⋮	⋮	⋮
(2,1)	11100	(7,1)	11111010
(2,2)	11111001	(7,2)	111111110111
(2,3)	1111110111	(7,3)	1111111110101110
(2,4)	111111110100	(7,4)	1111111110101111
(2,5)	1111111110001001	(7,5)	1111111110110000
⋮	⋮	⋮	⋮
(3,1)	111010	(8,1)	111111000
(3,2)	111110111	(8,2)	111111111000000
(3,3)	111111110101	(8,3)	1111111110110110
(3,4)	1111111110001111	(8,4)	1111111110110111
(3,5)	1111111110010000	(8,5)	1111111110111000
⋮	⋮	⋮	⋮
(4,1)	111011	(9,1)	111111001
(4,2)	1111111000	(9,2)	1111111110111110
(4,3)	1111111110010110	(9,3)	1111111110111111
(4,4)	1111111110010111	(9,4)	1111111111000000
(4,5)	1111111110011000	(9,5)	1111111111000001
⋮	⋮	⋮	⋮
		EOB	1010

Table 6.3: AC table, symbol 1

6.3 Compression with SVD

In Section 4.3 we introduced a matrix factorization of a general matrix $A_{m\times n}$ as the product of two orthogonal matrices and a diagonal one. This factorization is expressed as

$$A = U\Sigma V^T, \tag{6.27}$$

where $U_{m\times m}$ and $V_{n\times n}$ are orthogonal matrices and $\Sigma_{m\times n}$ is a diagonal matrix whose diagonal entries σ_i are known as the *singular values* of the matrix A.

We learned that this factorization provides with plenty of information about the matrix A: It gives orthonormal bases for $\mathrm{col}(A)$ and $\mathrm{row}(A)$, it reveals the rank of A, it provides with the spectral norm of A, etc. One very important result was that the factorization (6.27) can be written as

$$A = \sigma_1 u_1 v_1^T + \cdots + \sigma_r u_r v_r^T, \tag{6.28}$$

where r is the rank of A and u_i, v_i represent the i-th columns of the matrices U and V respectively.

Writing the SVD of a matrix A as the expansion (6.28) allowed us to introduce low-rank approximations of the matrix A. A rank-k matrix A_k that approximates A with minimum error in the sense of least squares is given by a truncation of the expansion (6.28) to k terms. That is,

$$A_k = \sigma_1 u_1 v_1^T + \cdots + \sigma_k u_k v_k^T, \qquad k \leq r. \tag{6.29}$$

We have already studied two direct applications of these SVD low-rank approximations, namely in information retrieval (Section 4.5) and simple substitution cryptograms (Section 4.6). Now we discuss one more application of SVD low-rank approximations, this time to image compression, of both, gray scale and color images.

6.3.1 Compressing grayscale images

As remarked before, given a grayscale image, this can be understood as an $m \times n$ matrix X whose entries are values between 0 and 255 indicating different gray intensities between black (0) and white (255). Let us assume that such matrix X has rank r. Then, using the notation in (6.28), its SVD factorization $X = U\Sigma V^T$ can be written as

$$X = \sigma_1 u_1 v_1^T + \cdots + \sigma_r u_r v_r^T. \tag{6.30}$$

One very important fact to remember about this factorization is that the singular values satisfy the inequalities

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0.$$

This means that the importance of the terms in the expansion (6.30) decreases as more terms are considered, or equivalently, the first terms of the expansion must contain the most important information about the matrix A. This remarkable fact about the SVD of X is exactly what we can exploit to achieve compression: instead of storing the whole expansion (6.30), we can try to store just a truncation of such expansion to k terms, with $k < r$, dropping all terms with coefficients $\sigma_{k+1}, \ldots, \sigma_r$.

For a chosen value of $k < r$, we know from Theorem 4.21 that

$$X_k = \sigma_1 u_1 v_1^T + \cdots + \sigma_k u_k v_k^T$$

is an optimal rank-k approximation to X, and therefore we expect that the compressed image X_k will look very similar to the original one X.

As expected, we have a trade-off between quality and storage savings. The lower the value of k, the more we save and compress, but at the same time we may be losing some quality of the compressed image. We want to illustrate this with an example

Example 6.3.1 *Consider again the image in Figure 6.10. We show in Figure 6.21 this original image along with three different compression rates, corresponding to the rank-k approximations. The original matrix has rank* $r = 462$. *Observe that with* $k = 95$ *we already obtain a very good approximation to the original image. This means that from the 462 terms in (6.30) we can drop* $462 - 95 = 367$ *terms and still obtain a good quality image.*

6.3.2 Compressing color images

A very simple approach to compress color images via low-rank SVD approximations is to treat each color coordinate in (R, G, B) independently. Recall that a color image is understood as a three-dimensional array (see Figure 6.14). Since

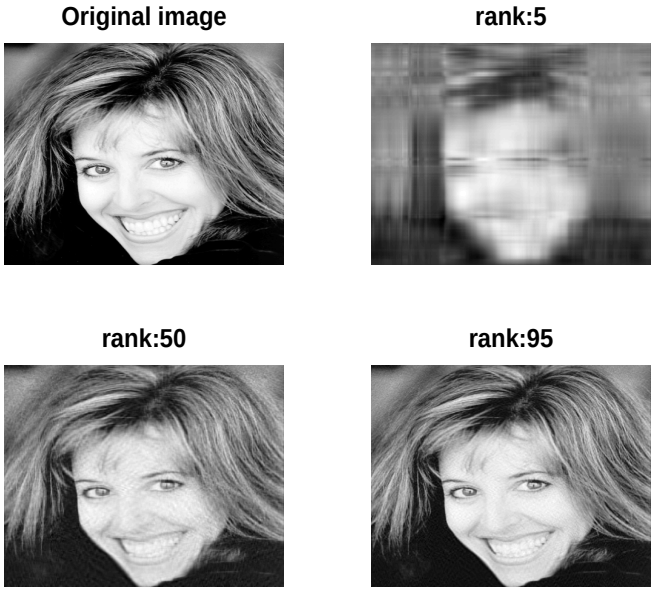

Figure 6.21: Compression with low-rank SVD approximations

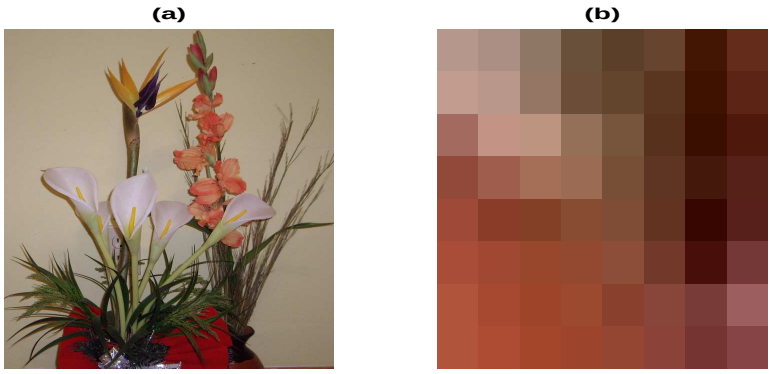

Figure 6.22: Original image and 8×8 block

each layer is a usual two-dimensional matrix, we compute the SVD factorization of each one of them, and then apply low-rank approximation just the way we did to grayscale images. The final step is to reconstruct the whole (compressed) image by putting the three layers back together again.

Example 6.3.2 *Consider the color image in Figure 6.22 and one of its* 8×8 *blocks. We want to apply SVD compression to both images by truncating the the SVD expansion (6.30) on each coordinate of* (R, G, B)*. Each of these layers has rank* $r = 1773$*. With only* $k = 55$ *we are able to get a good quality compressed image. See Figure 6.23.*

6.4 Final Remarks and Further Reading

In this section we have studied two different approaches to grayscale and color image compression. The first and most important one is done via the discrete cosine transform, which is currently used by JPEG. The second approach is presented for completion, as an application of the singular value decomposition.

The topic of image compression is discussed on a large list of books and articles. A great reference on data compression in general is the book by D. Salomon [49]. The book by K. Thyagarajan [55] offers a detailed discussion on image

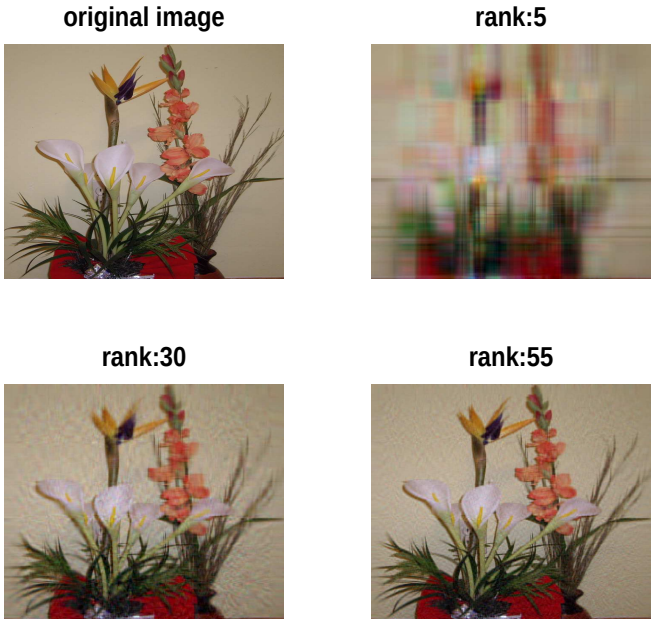

Figure 6.23: Original image and low-rank approximations

processing, including applications to digital cinema. A brief and clear exposition of image and sound compression can also be found in the book by T. Sauer [50]. The reader can always have full access to documents online with detailed and complete tables for Huffman coding and other information. See for example http://www.w3.org/Graphics/JPEG/itu-t81.pdf

Although not widely used yet, JPEG2000, based on wavelets, is the latest effort in achieving even more efficiency when compressing images. However, its time has not come yet as the standard choice for images in web browsers.

Ultimately, both, JPEG and SVD approaches are excellent and current real-world applications of linear algebra and numerical analysis, and represent a very interesting topic to convey to students.

6.5 Exercises

Exercise 6.1 *Let C be the orthogonal matrix in (6.1) and define $A_{n\times n}$ as*

$$A = \begin{bmatrix} 1 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 1 \end{bmatrix}.$$

Show that the columns of C^T are unit eigenvectors of A.

Exercise 6.2 *Observe the DCT matrix in (6.3) row by row. What pattern do you see in the signs of the entries, and how is this related to low-high frequencies?*

Exercise 6.3 *Following Exercise 6.2, make up a matrix $C_{4\times 4}$ of positive and negative 1's that would follow a pattern similar to that in (6.3). Then normalize it to make it orthogonal. Apply this transform to the matrix corresponding to a 4×4 grayscale image (CXC^T).*

Exercise 6.4 *Prove Theorem 6.8.*

Exercise 6.5 *Interpolate the following data*

$$(0,3),\ (1,1),\ (2,-1),\ (3,3),\ (4,1.5),\ (5,-0.5),\ (6,-2)$$

using the DCT. Plot the data and the interpolating function together.

Exercise 6.6 *Consider the data of Exercise 6.5. Apply DCT least squares approximation by dropping the last two terms of its interpolating polynomial. Plot both the least squares approximation and the interpolating polynomials as well as the data points.*

Exercise 6.7 *Consider the input data matrix*

$$X = \begin{bmatrix} -3.50 & -1.50 & -0.75 & -0.70 & -0.75 & -1.50 & -3.50 \\ -3.50 & -1.25 & -0.65 & -0.60 & -0.65 & -1.25 & -3.50 \\ -3.50 & -1.50 & -1.00 & -0.50 & -1.00 & -1.50 & -3.50 \\ -3.50 & -1.00 & -0.40 & 0.60 & -0.40 & -1.00 & -3.50 \\ -3.50 & -1.25 & -0.25 & 0.10 & -0.25 & -1.25 & -3.50 \\ -3.50 & -2.00 & -0.25 & 0.00 & -0.25 & -2.00 & -3.50 \\ -3.50 & -3.00 & -2.50 & -2.00 & -2.50 & -3.00 & -3.50 \end{bmatrix}.$$

Find the DCT of X and plot the graph of the interpolating function.

Exercise 6.8 *Consider again the data X of Exercise 6.7. By following Example 6.1.5, compute two least squares approximations by requiring that $k+l \leq 4$ and $k+l \leq 6$.*

Exercise 6.9 *Try the following compression technique: given a grayscale image, crop it so that the number of rows and columns is a multiple of 8. Then, replace each entry of each 8×8 block with its corresponding average pixel value in that block. Plot both, the original and the compressed image.*

Exercise 6.10 *True or False? Any file can be compressed.*

Exercise 6.11 *Find a scaling function that transforms an arbitrary interval $[a,\ b]$ into the interval $[0,\ 255]$.*

Exercise 6.12 *Obtain and plot 4-bit (16 variations) and 8-bit (256 variations) black to white gradients.*

Exercise 6.13 *Consider the 4×4 block image Y of Example 6.1.6. Plot the original image together with three approximations to it, according to the number of basis images Y_i used: a) $i = 0$, b) $i = 0, 1, 2, 3$, c) $i = 0, 1, \ldots, 8$.*

Exercise 6.14 *Import a grayscale image into* MATLAB .

(a) Extract an 8×8 block from the image and compress it by using the quantization matrices (6.17) and (6.18), with $s = 3$. Compare your results.

(b) Apply the same process to the whole image.

Exercise 6.15 *Repeat Exercise 6.14 but now using the following quantization matrix, for* $s = 5$.

$$K = s \begin{bmatrix} 5 & 5 & 5 & 5 & 5 & 6 & 6 & 8 \\ 5 & 5 & 5 & 5 & 5 & 6 & 7 & 8 \\ 5 & 5 & 5 & 5 & 6 & 7 & 8 & 9 \\ 5 & 5 & 5 & 6 & 7 & 8 & 9 & 10 \\ 5 & 5 & 6 & 7 & 8 & 9 & 11 & 12 \\ 6 & 6 & 7 & 8 & 9 & 11 & 13 & 14 \\ 6 & 7 & 8 & 9 & 11 & 13 & 15 & 16 \\ 8 & 8 & 9 & 10 & 12 & 14 & 16 & 19 \end{bmatrix}.$$

Exercise 6.16 *Denote with* X *the original image and with* Z *the compressed one. If there was no loss in the compression, then* Z *is identical to* X *and the image* $A = X - Z$ *is a matrix of zeros and therefore black. For the* 8×8 *block of Exercise 6.14, obtain the corresponding matrices* A *corresponding to both quantization matrices (6.17) and (6.18), and display their images. Which one is farther from a black image?*

Exercise 6.17 *Consider the following orthogonal (Haar) matrix*

$$H = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\ 2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 \end{bmatrix}.$$

Starting with the canonical basis of the vector space of 8×8 *matrices, obtain the basis images associated to* H*, to obtain an image similar to Figure 6.9.*

Exercise 6.18 *Repeat Exercise 6.17 for the (Hadamard) matrix*

$$H = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}.$$

Exercise 6.19 *Consider the following* 8×8 *block of pixels values*

$$X = \begin{bmatrix} 154 & 161 & 188 & 197 & 200 & 181 & 134 & 111 \\ 153 & 155 & 185 & 199 & 199 & 191 & 145 & 108 \\ 154 & 149 & 176 & 196 & 198 & 194 & 161 & 112 \\ 160 & 150 & 168 & 190 & 200 & 196 & 173 & 122 \\ 164 & 157 & 167 & 188 & 201 & 201 & 181 & 130 \\ 165 & 162 & 168 & 186 & 197 & 195 & 188 & 142 \\ 173 & 166 & 162 & 181 & 194 & 191 & 191 & 160 \\ 184 & 171 & 155 & 176 & 197 & 198 & 191 & 176 \end{bmatrix}.$$

Compute its DCT transform $Y = CXC^T$ *and verify that the sums of squares of the entries in both matrices* X *and* Y *are equal. Why is this true?*

Exercise 6.20 *We know that we can compress an image by filtering out high frequency terms, retaining only the low frequency ones, which are the most important to the human eye. Experiment compressing a grayscale image but this time filtering out the low frequency terms and retaining the high frequency ones.*

Exercise 6.21 *Suppose you have a color RGB image. Change it to grayscale by expressing the grayscale intensities as a combination of the three coordinates* R, G *and* B*. Then, compare your result to the one obtained with the* MATLAB *command* rgb2gray*.*

Exercise 6.22 *By setting* $x = [R \quad G \quad B]^T$ *and* $z = [Y \quad U \quad V]^T$*, write the change of coordinates in (6.20) as the transformation*

$$z = Tx + b,$$

for some matrix T *and some vector* b*.*

Exercise 6.23 *Import a grayscale image into* MATLAB *.*

(a) Extract an 8×8 *block from the image and compress it by individually applying the luminance quantization matrix (6.18) to each color R, G, B.*

(b) Apply the same process to the whole image.

Exercise 6.24 *Import a grayscale image into* MATLAB *.*

(a) Extract an 8×8 *block from the image and compress it by first changing from RGB to YUV coordinates as in (6.20) and then using the luminance quantization matrix (6.18) for* Y *and the chrominance matrix (6.21) for* UV*.*

(b) Apply the same process to the whole image.

Exercise 6.25 *Assume we have the following set of symbols:* $\{A, B, C, D, E\}$*, with probabilities:* $A = 0.25$*,* $B = 0.10$*,* $C = 0.15$*,* $D = 0.15$*,* $E = 0.35$*. Find the entropy (6.22). According to this entropy, find out the optimal number of bits needed to code the string DECEEEAA.*

Exercise 6.26 *Refer to the Huffman tree of Figure 6.18. Build a different tree for the same symbols, but this time taking different choices, e.g. at step 1 combine D and F instead of E and F. What codes do you get for the symbols A, B, C, D, E? Next, translate the string ACDAACBBEB into a bit string. How many bits per symbol are needed?*

Exercise 6.27 *Construct a Huffman tree that generates the Table 6.1.*

Exercise 6.28 *Suppose two neighboring* 8×8 *blocks have the DC coefficients* $DC_4 = 35$*,* $DC_5 = 42$*. Find the coding of the difference coefficient* $D = DC_5 - DC_4$ *as given in (6.24).*

Exercise 6.29 *From Table 6.2 find the codes for a)*12*, b)*-60*.*

Exercise 6.30 *Suppose we have the following AC coefficients*

$$8, -5, 0, 0, 0, 0, 0, 4.$$

Translate this into a bit stream following to (6.26).

Exercise 6.31 *Consider the following quantized matrix*

$$Y_Q = \begin{bmatrix} -35 & 14 & -2 & -1 & 0 & 0 & 0 & 0 \\ -27 & -11 & 3 & 0 & 0 & 0 & 0 & 0 \\ 12 & 6 & -1 & 0 & 0 & 0 & 0 & 0 \\ 3 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Following the zigzag pattern of Figure 6.20, find the Huffman code for all the quantized coefficients.

Exercise 6.32 *Import a grayscale image into* MATLAB .

(a) Extract an 8×8 *block from the image and apply rank-k SVD approximations to the corresponding* 8×8 *matrix for three different values of* k*. Print the rank-k images together with the original image in a* 4×4 *figure.*

(b) Apply the same process to the whole image.

Exercise 6.33 *Use SVD compression on a color image by first individually compressing each color (RGB) as if you were dealing with grayscale intensities, and then reconstruct the compressed image from the superposition of the three colors.*