

Vladimir Britanak "Discrete Cosine and Sine Transforms"
The Transform and Data Compression Handbook
Ed. K. R. Rao et al.
Boca Raton, CRC Press LLC, 2001

Chapter 4

Discrete Cosine and Sine Transforms

Vladimir Britanak

Slovak Academy of Sciences

4.1 Introduction

The discrete cosine transform (DCT) and discrete sine transform (DST) are members of a family of sinusoidal unitary transforms. They have found applications in digital signal and image processing and particularly in transform coding systems for data compression/decompression. Among the various versions of DCT, types II and III have received much attention in digital signal processing. Besides being real, orthogonal, and separable, its properties are relevant to data compression and fast algorithms for its computation have proved to be of practical value. Recently, DCT has been employed as the main processing tool for data compression/decompression in international image and video coding standards [31]. An alternative transform used in transform coding systems is DST. In fact, the alternate use of modified forms of DST and DCT has been adopted in the international audio coding standards MPEG-1 and MPEG-2 (Moving Picture Experts Group) [31].

In this chapter, the definitions and basic mathematical properties of four even types of DCT and the DST are discussed. Then, the properties of DCT and DST relevant to data compression are briefly outlined. For each DCT and DST, a fast computational algorithm is described, and a corresponding regular generalized signal flow graph is shown, followed by its implementation in C. Finally, to illustrate the compression capability of DCT, a real DCT-based data compression application is considered. The simple and efficient JPEG (Joint Photographic Experts Group) DCT-based image compression and decompression system [31] and its implementation is described in detail. Generally, this chapter contains many implemented algorithms that can be useful not only in data compression applications but also in any other DCT- and DST-related applications.

4.2 The Family of DCTs and DSTs

DCTs and DSTs are members of the class of sinusoidal unitary transforms developed by Jain [1]. A sinusoidal unitary transform is an invertible linear transform whose kernel describes a set of complete, orthogonal discrete cosine and/or sine basis functions. The well-known Karhunen–Loève transform (KLT) [30], generalized discrete Fourier transform [2], generalized discrete Hartley transform [3] or equivalently generalized discrete W transform [4], and various types of the DCT and DST are members of this class of unitary transforms.

The set of DCTs and DSTs introduced by Jain [1] is not complete. The complete set of DCTs and DSTs, so-called discrete trigonometric transforms, has been described by Wang and Hunt [4]. The family of discrete trigonometric transforms consists of 8 versions of DCT and corresponding 8 versions of DST [13, 14]. Each transform is identified as even or odd and of type I, II, III, and IV. All present digital signal and image processing applications (mainly transform coding and digital filtering of signals) involve only even types of the DCT and DST. Therefore, this chapter considers four even types of DCT and DST.

4.2.1 Definitions of DCTs and DSTs

In subsequent sections, N is assumed to be an integer power of 2, i.e., $N = 2^m$. A subscript of a matrix denotes its order, while a superscript denotes the version number.

Four normalized even types of DCT in the matrix form are defined as [4]

$$DCT - I : \quad [C_{N+1}^I]_{nk} = \sqrt{\frac{2}{N}} \left[\epsilon_n \epsilon_k \cos \frac{\pi nk}{N} \right], \quad (4.1a)$$

$$n, k = 0, 1, \dots, N,$$

$$DCT - II : \quad [C_N^{II}]_{nk} = \sqrt{\frac{2}{N}} \left[\epsilon_k \cos \frac{\pi(2n+1)k}{2N} \right], \quad (4.1b)$$

$$n, k = 0, 1, \dots, N-1,$$

$$DCT - III : \quad [C_N^{III}]_{nk} = \sqrt{\frac{2}{N}} \left[\epsilon_n \cos \frac{\pi(2k+1)n}{2N} \right], \quad (4.1c)$$

$$n, k = 0, 1, \dots, N-1,$$

$$DCT - IV : \quad [C_N^{IV}]_{nk} = \sqrt{\frac{2}{N}} \left[\cos \frac{\pi(2n+1)(2k+1)}{4N} \right], \quad (4.1d)$$

$$n, k = 0, 1, \dots, N-1,$$

where

$$\epsilon_p = \begin{cases} \frac{1}{\sqrt{2}} & p = 0 \text{ or } p = N \\ 1 & \text{otherwise} \end{cases}$$

and the corresponding four normalized even types of the DST are defined as [4]

$$DST - I : \quad \left[S_{N-1}^I \right]_{nk} = \sqrt{\frac{2}{N}} \left[\sin \frac{\pi(n+1)(k+1)}{N} \right], \quad (4.2a)$$

$$n, k = 0, 1, \dots, N-2,$$

$$DST - II : \quad \left[S_N^{II} \right]_{nk} = \sqrt{\frac{2}{N}} \left[\epsilon_k \sin \frac{\pi(2n+1)(k+1)}{2N} \right], \quad (4.2b)$$

$$n, k = 0, 1, \dots, N-1,$$

$$DST - III : \quad \left[S_N^{III} \right]_{nk} = \sqrt{\frac{2}{N}} \left[\epsilon_n \sin \frac{\pi(2k+1)(n+1)}{2N} \right], \quad (4.2c)$$

$$n, k = 0, 1, \dots, N-1,$$

$$DST - IV : \quad \left[S_N^{IV} \right]_{nk} = \sqrt{\frac{2}{N}} \left[\sin \frac{\pi(2n+1)(2k+1)}{4N} \right], \quad (4.2d)$$

$$n, k = 0, 1, \dots, N-1,$$

where

$$\epsilon_q = \begin{cases} \frac{1}{\sqrt{2}} & q = N-1 \\ 1 & \text{otherwise.} \end{cases}$$

The DCT-I introduced by Wang and Hunt [5] is defined for the order $N+1$. It can be considered a special case of symmetric cosine transform introduced by Kitajima [6]. The DST-I introduced by Jain [7] is defined for the order $N-1$ and constitutes the basis of a technique called recursive block coding [35]. The DCT-II and its inverse, DCT-III, first reported by Ahmed, Natarajan, and Rao [8], has an excellent energy compaction property, and among the currently known unitary transforms it is the best approximation for the optimal KLT. The DST-II and its inverse, DST-III, have been introduced by Kekre and Solanki [9]. DST-II is a complementary or alternative transform to DCT-II used in transform coding. DCT-IV and DST-IV introduced by Jain [1] have found applications in the fast implementation of lapped orthogonal transform for the efficient transform/subband coding [12].

4.2.2 Mathematical Properties

The basic mathematical properties of discrete transforms are fundamental for their use in practical applications. Thus, properties such as scaling, shifting, and convolution are readily applied in the discrete transform domain. In the following, we briefly summarize the most relevant mathematical properties of the family of DCTs and DSTs.

DCT and DST matrices are real and orthogonal. All DCTs and DSTs are separable transforms; the multidimensional transform can be decomposed into successive application of one-dimensional (1-D) transforms in the appropriate directions.

The Unitarity Property

The following relations hold for inverse DCT matrices

$$\left[C'_{N+1}\right]^{-1} = \left[C'_{N+1}\right]^T = \left[C'_{N+1}\right] \quad (4.3a)$$

$$\left[C''_N\right]^{-1} = \left[C''_N\right]^T = \left[C'''_N\right] \quad (4.3b)$$

$$\left[C'''_N\right]^{-1} = \left[C'''_N\right]^T = \left[C''_N\right] \quad (4.3c)$$

$$\left[C^{IV}_N\right]^{-1} = \left[C^{IV}_N\right]^T = \left[C^{IV}_N\right] \quad (4.3d)$$

and for inverse DST matrices

$$\left[S'_{N-1}\right]^{-1} = \left[S'_{N-1}\right]^T = \left[S'_{N-1}\right] \quad (4.4a)$$

$$\left[S''_N\right]^{-1} = \left[S''_N\right]^T = \left[S'''_N\right] \quad (4.4b)$$

$$\left[S'''_N\right]^{-1} = \left[S'''_N\right]^T = \left[S''_N\right] \quad (4.4c)$$

$$\left[S^{IV}_N\right]^{-1} = \left[S^{IV}_N\right]^T = \left[S^{IV}_N\right] \quad (4.4d)$$

If the nonsingular matrix is real and orthogonal, its inverse is obtained as its transpose. In the definitions of DCT and DST, matrices given by Eqs. (4.1a)–(4.1d) and Eqs. (4.2a)–(4.2d), respectively, the normalization factors $\sqrt{(2/N)}$ can be merged as $2/N$, and it can be moved to either the forward or inverse transform. By merging these normalization factors, the family of DCT and DST matrices are no longer orthonormal. They are, however, still orthogonal. The DCT-I, DCT-IV, DST-I, and DST-IV matrices are involutory, i.e., they are orthogonal and symmetric. The symmetry of an orthogonal matrix indicates that algorithms for the forward and inverse transform computation will be the same except for the normalization. On the other hand, DCT-II and DCT-III are inverses of each other. The same property holds for DST-II and DST-III.

The Linearity Property

Since matrix multiplication is a linear operation, i.e.,

$$M (\alpha \mathbf{g} + \beta \mathbf{f}) = \alpha M \mathbf{g} + \beta M \mathbf{f} \quad (4.5)$$

for a matrix M , constants α and β , and vectors \mathbf{g} and \mathbf{f} , all DCTs and DSTs are linear transforms.

The Convolution-Multiplication Property

All DCTs and DSTs possess convolution — multiplication property which is a powerful tool for performing digital filtering in the transform domain. The convolution operation in the transform domain realized by taking an inverse transform of the

product of forward transforms of two data sequences is equivalent to symmetric convolution of those symmetrically extended sequences in the spatial domain [13, 14]. Let $\{x_n\}$ and $\{y_n\}$ be two input data sequences to be convolved. Generally, the relation between the symmetric convolution and transform domain convolution-multiplication property can be expressed by the following equation

$$\{x_n\} < \mathbf{sc} > \{y_n\} = \mathcal{T}_c^{-1} [\mathcal{T}_a \{x_n\} \times \mathcal{T}_b \{y_n\}] , \quad (4.6)$$

where $< \mathbf{sc} >$ is the operator of symmetric convolution, \times denotes element-by-element multiplication of its operands, and $\mathcal{T}_a \{x_n\}$ denotes a specified transform \mathcal{T}_a of the sequence $\{x_n\}$. As an example, the convolution-multiplication property for the DCT-II is obtained by substituting $\mathcal{T}_a = \mathcal{T}_b = [C_N'']$ and $\mathcal{T}_c = [C_{N+1}']^{-1}$ into Eq. (4.6). Definition of the symmetric convolution and convolution-multiplication properties for the entire family of discrete trigonometric transforms are given in references [13, 14], and [15].

The Shift Property, Scaling, and Difference Property

For the family of DCTs and DSTs, the reader can find the complete derivations of the shift property in references [10, 11], and [30] and scaling in time and the difference property in [30].

4.2.3 Relations to the KLT

The performance of DCTs and DSTs, particularly important in transform coding, is associated with the KLT. KLT is an optimal transform for data compression in a statistical sense because it decorrelates a signal in the transform domain, packs the most information in a few coefficients, and minimizes mean-square error between the reconstructed and original signal compared to any other transform. However, KLT is constructed from the eigenvalues and the corresponding eigenvectors of a covariance matrix of the data to be transformed; it is signal-dependent, and there is no general algorithm for its fast computation. There is asymptotic equivalence of the family of DCTs and DSTs with respect to KLT for a first-order stationary Markov process in terms of the transform size and the adjacent (interelement) correlation coefficient ρ . For finite length data, DCTs and DSTs provide different approximations to KLT, and the best approximating transform varies with the value of correlation coefficient ρ . For example, when $\rho = 1$ the KLT is reduced to DCT-II (DCT-III) [16, 17, 30], for $\rho = 0$ the KLT is reduced to DST-I [7, 17, 18], and for $\rho = -1$ it is reduced to DST-II (DST-III) [19]. On the other hand, if the transform size N increases (i.e., N tends to infinity), it can be shown that KLT is reduced to DCT-I or DCT-IV [30]. This asymptotic behavior implies that DCTs and DSTs can be used as substitutes for KLT of certain random processes.

In general, there are several characteristics that are desirable in a transform when it is used for the purpose of data compression [36]:

- **Data decorrelation:** The ideal transform completely decorrelates the data in a sequence/block; i.e., it packs the most amount of energy in the fewest number of coefficients. In this way, many coefficients can be discarded after quantization and prior to encoding. It is important to note that the transform operation itself does not achieve any compression. It aims at decorrelating the original data and compacting a large fraction of the signal energy into relatively few transform coefficients.
- **Data-independent basis functions:** Owing to the large statistical variations among data, the optimum transform usually depends on the data, and finding the basis functions of such transform is a computationally intensive task. This is particularly a problem if the data blocks are highly nonstationary, which necessitates the use of more than one set of basis functions to achieve high decorrelation. Therefore, it is desirable to trade optimum performance for a transform whose basis functions are data-independent.
- **Fast implementation:** The number of operations required for an n -point transform is generally of the order $\mathcal{O}(n^2)$. Some transforms have fast implementations, which reduce the number of operations to $\mathcal{O}(n \log n)$. For a separable $n \times n$ 2-D transform, performing the row and column 1-D transforms successively reduces the number of operations from $\mathcal{O}(n^4)$ to $\mathcal{O}(2n^2 \log n)$.

Among the family of DCTs and DSTs, the performance of DCT-II is closest to the statistically optimal KLT based on a number of performance criteria (variance distribution, energy packing efficiency, residual correlation, rate distortion, and maximum reducible bits and generalized Wiener filtering) [30]. The importance of DCT-II is further accentuated by its superiority in bandwidth compression (redundancy reduction) of a wide range of signals and by existence of fast algorithms for its implementation. Owing to powerful performance in the bit-rate reduction, DCT-II and its inversion, DCT-III, have been employed in the international image/video coding standards: JPEG for compression of still images, MPEG for compression of motion video including HDTV (High Definition Television), H.261 for compression of video telephony and teleconferencing, and H.263 for visual communication over ordinary telephone lines [31].

4.3 A Unified Fast Computation of DCTs and DSTs

The DCT and DST matrices defined in Section 4.2 are orthonormal. The normalization factor $\sqrt{(2/N)}$ in the forward and the inverse transforms can be merged as $2/N$ and moved to the forward transform. By merging these normalization factors the family of DCT and DST matrices are orthogonal. Without loss of generality, in this section orthogonal DCT and DST matrices will be considered.

A unified fast computation of even types of DCT (DCT-I, -II, -III, -IV) and DST (DST-I, -II, -III, -IV) is based on a universal computational structure both for DCT-

II/DST-II and DCT-III/DST-III computation [26]. This DCT-II/DST-II (DCT-III/DST-III) universal computational structure is used as the basic computational unit (a potential DCT/DST processor) in fast algorithms defined by sparse matrix factorizations. The fast algorithms are simple, numerically stable and efficient. For each type of the DCT and DST computation, the corresponding regular generalized signal flow graph is shown. Generalized signal flow graphs are enabled to realize computation of given DCT and DST for any $N = 2^m$, $m > 0$ (N being the length of the data sequence). The unified fast computation of DCTs and DSTs provides simple and compact transform building blocks. Finally, computer programs for each even type of the DCT and DST computation are presented.

4.3.1 Definitions of Even-Odd Matrices

Even-Odd Transform Matrix

$$A_J = \begin{bmatrix} I_{\frac{J-1}{2}} & \bar{I}_{\frac{J-1}{2}} \\ \bar{I}_{\frac{J-1}{2}} & -I_{\frac{J-1}{2}} \end{bmatrix} \text{ for } J \text{ odd}, \quad (4.7)$$

where I_N is the identity matrix. Blanks in the even-odd transform matrix Eq. (4.7) represent null submatrices and

$$\bar{I}_N = \begin{bmatrix} 0 & \cdots & 0 & 0 & 0 & 1 \\ 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & \cdots & 1 & 0 & 0 & 0 \\ \vdots & & \vdots & \vdots & \vdots & \vdots \\ 1 & \cdots & 0 & 0 & 0 & 0 \end{bmatrix} \quad (4.8)$$

is the reflection matrix. The orthogonal even-odd transform matrix Eq. (4.7) converts data sequences into their symmetric (even) and anti-symmetric (odd) parts.

Even-Odd Permutation Matrices

$$P_J = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ & & \vdots & & & & & \vdots & \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ & & \vdots & & & & & \vdots & \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \text{ for } J \text{ even}, \quad (4.9a)$$

$$P_J = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ & & \vdots & & & & & \vdots & \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ & & \vdots & & & & & \vdots & \\ 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \quad \text{for } J \text{ odd.} \quad (4.9b)$$

The permutation matrix P_J reorders the data sequence such that the first half of even-numbered data is arranged in the natural order, while the last half of odd-numbered data is arranged in the reversed order.

4.3.2 DCT-II/DST-II and DCT-III/DST-III Computation

The DCT-II for a given data sequence $\{x_n\}$, $n = 0, 1, \dots, N-1$ is defined as [8]

$$z_k^{II} = \frac{2\epsilon_k}{N} \sum_{n=0}^{N-1} x_n \cos \left[\frac{\pi(2n+1)k}{2N} \right], \quad k = 0, 1, \dots, N-1 \quad (4.10)$$

and the inverse DCT-II (DCT-III) is defined by

$$x_n = \sum_{k=0}^{N-1} \epsilon_k z_k^{II} \cos \left[\frac{\pi(2n+1)k}{2N} \right], \quad n = 0, 1, \dots, N-1, \quad (4.11)$$

where

$$\epsilon_k = \begin{cases} \frac{1}{\sqrt{2}} & k = 0 \\ 1 & \text{otherwise.} \end{cases}$$

DCT-II and its inverse, DCT-III, given by Eqs. (4.10) and (4.11), respectively, can be rewritten as [23]

$$z_k^{II} = \frac{2\epsilon_k}{N} \sum_{n=0}^{N-1} \tilde{x}_n \cos \left[\frac{\pi(4n+1)k}{2N} \right], \quad k = 0, 1, \dots, N-1, \quad (4.12)$$

$$\tilde{x}_n = \sum_{k=0}^{N-1} \epsilon_k z_k^{II} \cos \left[\frac{\pi(4n+1)k}{2N} \right], \quad n = 0, 1, \dots, N-1, \quad (4.13)$$

where

$$\begin{aligned} \tilde{x}_n &= x_{2n} \\ \tilde{x}_{N-n-1} &= x_{2n+1}, \quad n = 0, 1, \dots, \frac{N}{2} - 1. \end{aligned} \quad (4.14)$$

The reordering in Eq. (4.14) corresponds to the permutation matrix P_N given by Eq. (4.9a) applied to the input data vector.

Let C_N^{II} be the $N \times N$ orthogonal DCT-II matrix. Then a reordered DCT-II matrix \hat{C}_N^{II} with permuted rows and columns is given by

$$\hat{C}_N^{II} = R_N C_N^{II} [P_N]^T, \quad (4.15)$$

where R_N is the bit reversal permutation matrix and $[P_N]^T$ is the transpose of the permutation matrix P_N . A fast, recursive algorithm for DCT-II (DCT-III) computation with a regular structure is based on a block matrix factorization of the reordered DCT-II matrix \hat{C}_N^{II} . The reordered DCT-II matrix \hat{C}_N^{II} has a recursive structure; higher order matrices can be generated from lower order ones, and its block matrix factorization has the following form [28, 30]

$$\hat{C}_N^{II} = \begin{bmatrix} I_{\frac{N}{2}} & 0 \\ 0 & K_{\frac{N}{2}} \end{bmatrix} \begin{bmatrix} \hat{C}_{\frac{N}{2}}^{II} & 0 \\ 0 & \hat{C}_{\frac{N}{2}}^{II} \end{bmatrix} \begin{bmatrix} I_{\frac{N}{2}} & 0 \\ 0 & Q_{\frac{N}{2}} \end{bmatrix} \begin{bmatrix} I_{\frac{N}{2}} & I_{\frac{N}{2}} \\ I_{\frac{N}{2}} & -I_{\frac{N}{2}} \end{bmatrix}, \quad (4.16)$$

where $K_{\frac{N}{2}}$ is an $\frac{N}{2} \times \frac{N}{2}$ matrix given by

$$K_{\frac{N}{2}} = R_{\frac{N}{2}} L_{\frac{N}{2}} R_{\frac{N}{2}}, \quad (4.17)$$

where $R_{\frac{N}{2}}$ is the bit reversal permutation matrix, $L_{\frac{N}{2}}$ is the lower triangular matrix

$$L_{\frac{N}{2}} = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 2 & 0 & \cdots & 0 \\ -1 & 2 & -2 & 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ -1 & 2 & -2 & 2 & \cdots & 2 \end{bmatrix},$$

$Q_{\frac{N}{2}}$ is the $\frac{N}{2} \times \frac{N}{2}$ diagonal matrix

$$Q_{\frac{N}{2}} = \text{diag} [\cos \phi_m],$$

$$\phi_m = \left(m + \frac{1}{4}\right) \left(\frac{2\pi}{N}\right), \quad m = 0, 1, \dots, \frac{N}{2} - 1. \quad (4.18)$$

The block matrix factorization Eq. (4.16) defines Hou's fast, recursive, and numerically stable algorithm for DCT-II (DCT-III) computation which can be represented in the matrix form as [23]

$$\begin{bmatrix} \hat{\mathbf{z}}_e^{II} \\ \hat{\mathbf{z}}_o^{II} \end{bmatrix} = \frac{2}{N} \hat{C}_N^{II} \begin{bmatrix} \tilde{\mathbf{x}}_p \\ \tilde{\mathbf{x}}_r \end{bmatrix}, \quad (4.19)$$

where

$$\begin{aligned}
\tilde{\mathbf{x}}_p &= [x_0, x_2, x_4, \dots, x_{N-4}, x_{N-2}]^T, \\
\tilde{\mathbf{x}}_r &= [x_{N-1}, x_{N-3}, x_{N-5}, \dots, x_3, x_1]^T, \\
\hat{\mathbf{z}}_e^{II} &= R_{\frac{N}{2}} \mathbf{z}_e^{II}, \\
\hat{\mathbf{z}}_o^{II} &= R_{\frac{N}{2}} \mathbf{z}_o^{II}, \\
\mathbf{z}_e^{II} &= [z_0, z_2, z_4, \dots, z_{N-4}, z_{N-2}]^T, \\
\mathbf{z}_o^{II} &= [z_1, z_3, z_5, \dots, z_{N-3}, z_{N-1}]^T,
\end{aligned}$$

where \mathbf{z}_e^{II} is the even half and \mathbf{z}_o^{II} is the odd half of the DCT-II transformed sequence both arranged in the natural order. T denotes transposition.

A regular generalized signal flow graph based on this algorithm for DCT-II and its inverse, DCT-III, for any $N = 2^m$, $m > 0$ has been described by Britanak [24]. It is shown for $N = 16$ in Fig. 4.1. Full lines represent transfer factors $+1$, while broken lines represent transfer factors -1 . \bigcirc represents addition, \downarrow represents multiplication by cosine coefficients $C_n^k = \cos(k\phi_n)$, $\phi_n = \frac{\pi(4n+1)}{2N}$, and \rightarrow represents multiplication by 2. The normalization factor is not included in the signal flow graph. The generalized signal flow graph consists of two regular parts. The first part is related to the butterfly structure, and the second one, after bit-reversal permutation, is mapped into a pipeline structure. This pipeline structure is related to a simple recurrent relation for any $N = 2^m$, $m > 0$ [24].

The DST-II for a given data sequence $\{x_n\}$, $n = 0, 1, \dots, N-1$ is defined as [9]

$$s_k^{II} = \frac{2\epsilon_k}{N} \sum_{n=0}^{N-1} x_n \sin \left[\frac{\pi(2n+1)(k+1)}{2N} \right], \quad k = 0, 1, \dots, N-1 \quad (4.20)$$

and the inverse DST-II (DST-III) is defined as

$$x_n = \sum_{k=0}^{N-1} \epsilon_k s_k^{II} \sin \left[\frac{\pi(2n+1)(k+1)}{2N} \right], \quad n = 0, 1, \dots, N-1, \quad (4.21)$$

where

$$\epsilon_k = \begin{cases} \frac{1}{\sqrt{2}} & k = N-1 \\ 1 & \text{otherwise.} \end{cases}$$

Let C_N^{II} and S_N^{II} be orthogonal $N \times N$ DCT-II and DST-II matrices, respectively. According to Wang [20] S_N^{II} is related to C_N^{II} by

$$S_N^{II} = \bar{I}_N C_N^{II} D_N, \quad (4.22)$$

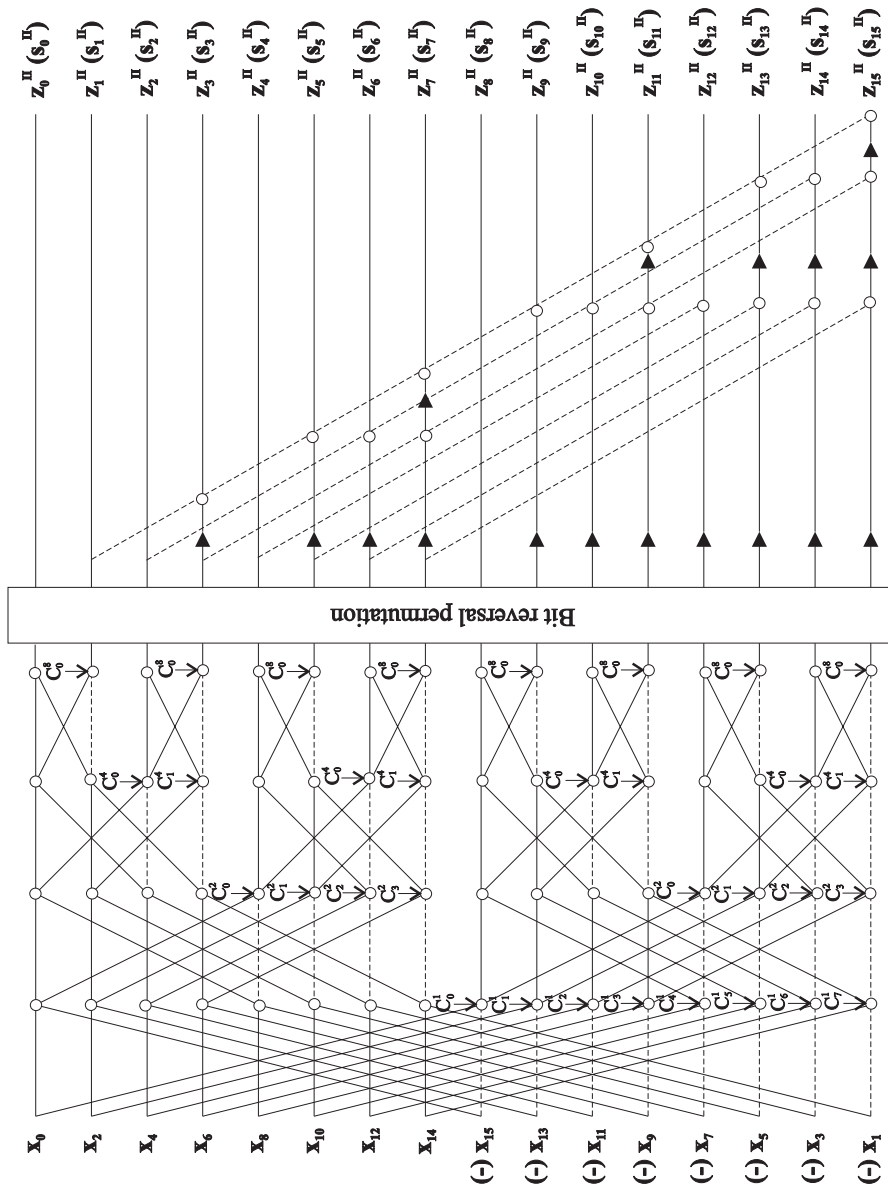


FIGURE 4.1
DCT-II/DST-II (DCT-III/DST-III) universal computational structure for $N = 16$. ©Slovak Academic Press Ltd.

D_N is the diagonal odd sign-changing matrix

$$D_N = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & -1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -1 \end{bmatrix}. \quad (4.23)$$

The D_N matrix applied to the input data sequence given by Eq. (4.14) corresponds to the reordering and sign changes:

$$\begin{aligned} \tilde{x}_n &= x_{2n} \\ \tilde{x}_{N-n-1} &= -x_{2n+1}, \quad n = 0, 1, \dots, \frac{N}{2} - 1. \end{aligned} \quad (4.24)$$

From Eq. (4.22) it follows that the generalized signal flow graph for the DCT-II computation can also be used for the DST-II computation for any $N = 2^m$, $m > 0$. The output DST-II transformed sequence, after the DCT-II computation for the input data sequence given by Eq. (4.24), is in reversed order; i.e., the final DST-II transformed sequence is obtained as

$$s_k^{II} = z_{N-1-k}^{II}, \quad k = 0, 1, \dots, N-1. \quad (4.25)$$

Hence, by the same computational structure, both the DCT-II and DST-II computation can be effectively realized for any $N = 2^m$, $m > 0$ simply by changing the input and output data sequences. Because both DCT-II and DST-II are orthogonal transforms, the algorithm for DST-III computation is obtained by transposing of Eq. (4.22). The generalized signal flow graph for DCT-II/DST-II and their inverse computations, so called DCT-II/DST-II (DCT-III/DST-III) universal computational structure, is shown for $N = 16$ in Fig. 4.1. The symbols in brackets correspond to DST-II (DST-III) computation. DCT-II/DST-II (DCT-III/DST-III) universal computational structure [25] represents the unified DCT-II/DST-II and their inverse computations, DCT-III/DST-III for any $N = 2^m$, $m > 0$. The universality of DCT-II/DST-II computational structure is related to the fact that it can be used as the basic computational unit for the fast implementation of the entire class of discrete sinusoidal transforms, i.e., generalized discrete Fourier transform, generalized discrete Hartley transforms, and the other types of the DCT and DST, respectively [26, 27]. We note that for fast computation of other discrete sinusoidal transforms, the bidirectional DCT-II/DST-II (DCT-III/DST-III) universal computational structure is used without the proper normalization. If DCT-II (DCT-III) or DST-II (DST-III) computation is required, the proper normalization should be applied to the input and output data sequences.

4.3.3 DCT-I and DST-I Computation

DCT-I for a given data sequence $\{x_n\}$, $n = 0, 1, \dots, N$ is defined as [5]

$$z_k^I = \frac{2\epsilon_k}{N} \sum_{n=0}^N \epsilon_n x_n \cos \left[\frac{\pi nk}{N} \right], \quad k = 0, 1, \dots, N \quad (4.26)$$

and the inverse DCT-I (IDCT-I) is defined by

$$x_n = \epsilon_n \sum_{k=0}^N \epsilon_k z_k^I \cos \left[\frac{\pi nk}{N} \right], \quad n = 0, 1, \dots, N, \quad (4.27)$$

where

$$\epsilon_p = \begin{cases} \frac{1}{\sqrt{2}} & p = 0 \text{ or } p = N \\ 1 & \text{otherwise.} \end{cases}$$

DCT-I and IDCT-I are defined for data sequences of length $N + 1$. Let C_{N+1}^I be the orthogonal DCT-I matrix of order $N + 1$. Then for $N = 2^m$, $m \geq 1$, C_{N+1}^I can be decomposed into the following recursive matrix form [21]

$$C_{N+1}^I = P_{N+1} \begin{bmatrix} C_{\frac{N}{2}+1}^I & 0 \\ 0 & \bar{I}_{\frac{N}{2}} C_{\frac{N}{2}}^{III} \bar{I}_{\frac{N}{2}} \end{bmatrix} A_{N+1}, \quad (4.28)$$

where A_{N+1} and P_{N+1} are matrices given by Eq. (4.7) and Eq. (4.9b), respectively. $C_{\frac{N}{2}+1}^I$ is the DCT-I matrix of order $\frac{N}{2} + 1$. The matrix product $\bar{I}_{\frac{N}{2}} C_{\frac{N}{2}}^{III} \bar{I}_{\frac{N}{2}}$ denotes $\frac{N}{2} \times \frac{N}{2}$ DCT-III matrix with reversed order for both its rows and columns. The permutation matrix P_{N+1} applied to the data vector corresponds to the reordering:

$$\begin{aligned} \tilde{x}_0 &= x_0 \\ \tilde{x}_{n+1} &= x_{2n+2} \\ \tilde{x}_{N-n} &= x_{2n+1}, \quad n = 0, 1, \dots, \frac{N}{2} - 1. \end{aligned} \quad (4.29)$$

Because C_{N+1}^I is a symmetric matrix, the algorithms for the DCT-I and IDCT-I computation are the same except for the normalization. The generalized signal flow graph for the DCT-I and IDCT-I computation for $N + 1 = 17$ is shown in Fig. 4.2. Here $\alpha = \frac{1}{\sqrt{2}}$, and the normalization factor is again not included in the signal flow graph.

The DST-I for a given data sequence $\{x_n\}$, $n = 0, 1, \dots, N - 2$ is defined as [7]

$$s_k^I = \frac{2}{N} \sum_{n=0}^{N-2} x_n \sin \left[\frac{\pi(n+1)(k+1)}{N} \right], \quad k = 0, 1, \dots, N - 2, \quad (4.30)$$

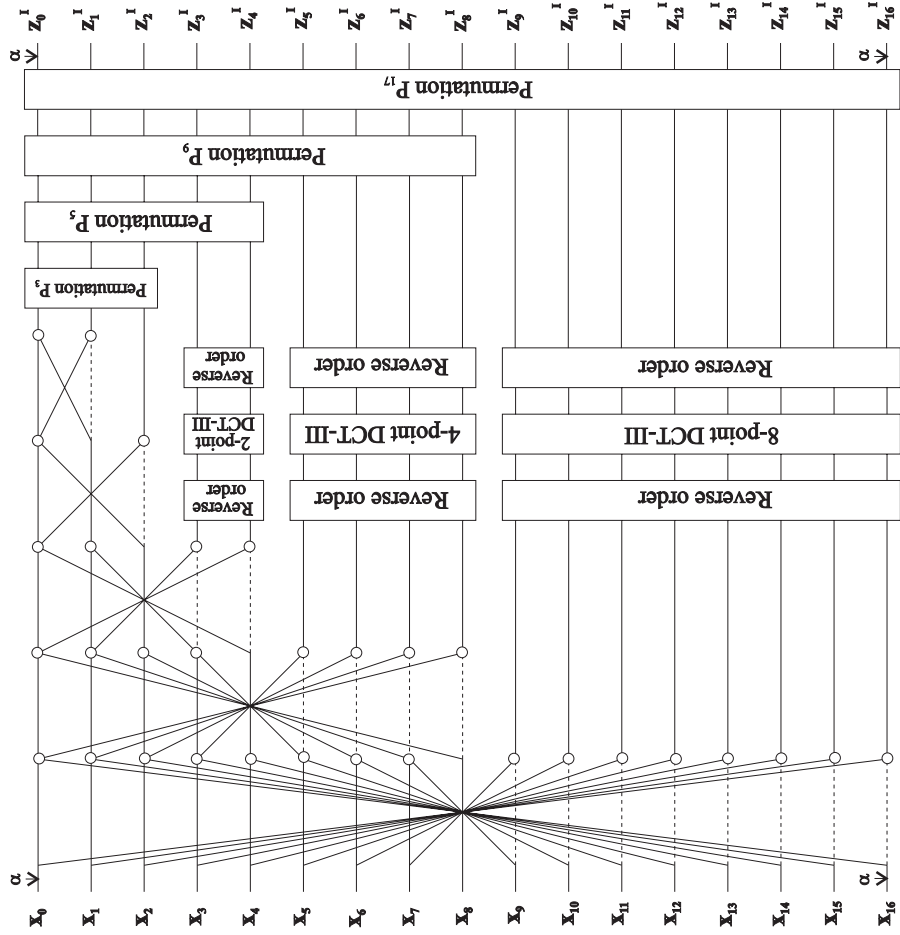


FIGURE 4.2
DCT-I and IDCT-I generalized signal flow graph for $N + 1 = 17$. ©Slovak Academic Press Ltd.

and the inverse DST-I (IDST-I) is defined by

$$x_n = \sum_{k=0}^{N-2} s_k^I \sin \left[\frac{\pi(n+1)(k+1)}{N} \right], \quad n = 0, 1, \dots, N-2. \quad (4.31)$$

DST-I and IDST-I are defined for data sequences of length $N - 1$. Let S_{N-1}^I be the orthogonal DST-I matrix of order $N - 1$. Then for $N = 2^m$, $m > 1$, S_{N-1}^I can be

decomposed into the following recursive matrix form [21]

$$S'_{N-1} = P_{N-1} \begin{bmatrix} S_{\frac{N}{2}}^{III} & 0 \\ 0 & \bar{I}_{\frac{N}{2}-1} S'_{\frac{N}{2}-1} \bar{I}_{\frac{N}{2}-1} \end{bmatrix} A_{N-1}, \quad (4.32)$$

where $S_{\frac{N}{2}}^{III}$ is the $\frac{N}{2} \times \frac{N}{2}$ DST-III matrix. The matrix product $\bar{I}_{\frac{N}{2}-1} S'_{\frac{N}{2}-1} \bar{I}_{\frac{N}{2}-1}$ denotes the DST-I matrix of order $\frac{N}{2} - 1$ with reversed order for both its rows and columns. The permutation matrix P_{N-1} applied to the data vector corresponds to the reordering

$$\begin{aligned} \tilde{x}_0 &= x_0 \\ \tilde{x}_{n+1} &= x_{2n+2} \\ \tilde{x}_{N-2-n} &= x_{2n+1}, \quad n = 0, 1, \dots, \frac{N}{2} - 2. \end{aligned} \quad (4.33)$$

Because S'_{N-1} is a symmetric matrix, the algorithms for the DST-I and IDST-I are the same except for the normalization. The generalized signal flow graph for the DST-I and IDST-I computation for $N - 1 = 15$ is shown in Fig. 4.3. The normalization factor is not included in the signal flow graph.

4.3.4 DCT-IV/DST-IV Computation

The DCT-IV for a given data sequence $\{x_n\}$, $n = 0, 1, \dots, N - 1$ is defined as [1]

$$z_k^{IV} = \frac{2}{N} \sum_{n=0}^{N-1} x_n \cos \left[\frac{\pi(2n+1)(2k+1)}{4N} \right], \quad k = 0, 1, \dots, N - 1 \quad (4.34)$$

and the inverse DCT-IV (IDCT-IV) is defined by

$$x_n = \sum_{k=0}^{N-1} z_k^{IV} \cos \left[\frac{\pi(2n+1)(2k+1)}{4N} \right], \quad n = 0, 1, \dots, N - 1. \quad (4.35)$$

Let C_N^{IV} be the orthogonal $N \times N$ DCT-IV matrix. Then for $N = 2^m$, $m \geq 1$, C_N^{IV} can be decomposed into the following sparse matrix product [22]

$$C_N^{IV} = T_N \begin{bmatrix} C_{\frac{N}{2}}^{III} & 0 \\ 0 & \bar{I}_{\frac{N}{2}} S_{\frac{N}{2}}^{III} \bar{I}_{\frac{N}{2}} \end{bmatrix} P_N B_N, \quad (4.36)$$

where $C_{\frac{N}{2}}^{III}$ is the $\frac{N}{2} \times \frac{N}{2}$ DCT-III matrix. The matrix product $\bar{I}_{\frac{N}{2}} S_{\frac{N}{2}}^{III} \bar{I}_{\frac{N}{2}}$ denotes the $\frac{N}{2} \times \frac{N}{2}$ DST-III matrix with reversed order for both its rows and columns. T_N is the

$$T_N = \begin{bmatrix} \cos \frac{\pi}{4N} & & & & \sin \frac{\pi}{4N} \\ & \ddots & & & \\ & & \cos \frac{(N-1)\pi}{4N} & \sin \frac{(N-1)\pi}{4N} & \\ & & \sin \frac{(N-1)\pi}{4N} & -\cos \frac{(N-1)\pi}{4N} & \\ & & & & \ddots \\ \sin \frac{\pi}{4N} & & & & -\cos \frac{\pi}{4N} \end{bmatrix} \quad (4.37)$$

and B_N is the tridiagonal matrix given by

$$B_N = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -1 & 0 & \cdots & 0 \\ 0 & 1 & 1 & 0 & \cdots & 0 \\ \vdots & & & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & -1 & 0 \\ 0 & \cdots & 0 & 1 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (4.38)$$

As can be seen, the decomposition of the matrix C_N^{IV} depends on the DCT-III and DST-III matrices of half size. Because C_N^{IV} is a symmetric matrix, the algorithms for the DCT-IV and IDCT-IV computation are the same except for the normalization. The generalized signal flow graph for the DCT-IV and IDCT-IV computation for $N = 16$ is shown in Fig. 4.4. The normalization factor is not included in the signal flow graph. The matrix product $P_N B_N$ can be realized by one butterfly stage in the generalized signal flow graph.

The DST-IV for a given data sequence $\{x_n\}$, $n = 0, 1, \dots, N - 1$ is defined as [1]

$$s_k^{IV} = \frac{2}{N} \sum_{n=0}^{N-1} x_n \sin \left[\frac{\pi(2n+1)(2k+1)}{4N} \right], \quad k = 0, 1, \dots, N - 1, \quad (4.39)$$

and the inverse DST-IV (IDST-IV) is defined by

$$x_n = \sum_{k=0}^{N-1} s_k^{IV} \sin \left[\frac{\pi(2n+1)(2k+1)}{4N} \right], \quad n = 0, 1, \dots, N - 1. \quad (4.40)$$

Let C_N^{IV} and S_N^{IV} be the $N \times N$ DCT-IV and DST-IV matrices, respectively. The matrix S_N^{IV} is related to C_N^{IV} by [21]

$$S_N^{IV} = \tilde{I}_N C_N^{IV} D_N. \quad (4.41)$$

Because S_N^{IV} is also a symmetric matrix, the algorithms for the DST-IV and IDST-IV computation are the same except for the normalization. From relation Eq. (4.41) it follows that the generalized signal flow graph for the DCT-IV computation can be also used for the DST-IV computation by changing only the input and output data sequences. The output DST-IV transformed sequence, after the DCT-IV computation for the input data sequence given by Eq. (4.24), is order reversed; the final DST-IV transformed data sequence is obtained as

$$s_k^{IV} = z_{N-1-k}^{IV}, \quad k = 0, 1, \dots, N - 1. \quad (4.42)$$

The generalized signal flow graph for the DCT-IV/DST-IV and IDCT-IV/IDST-IV computation for $N = 16$ is shown in Fig. 4.4, where the symbols in brackets correspond to DST-IV/IDST-IV computation. This generalized signal flow graph represents the unified DCT-IV/DST-IV and their inverse computations for any $N = 2^m$, $m > 0$.

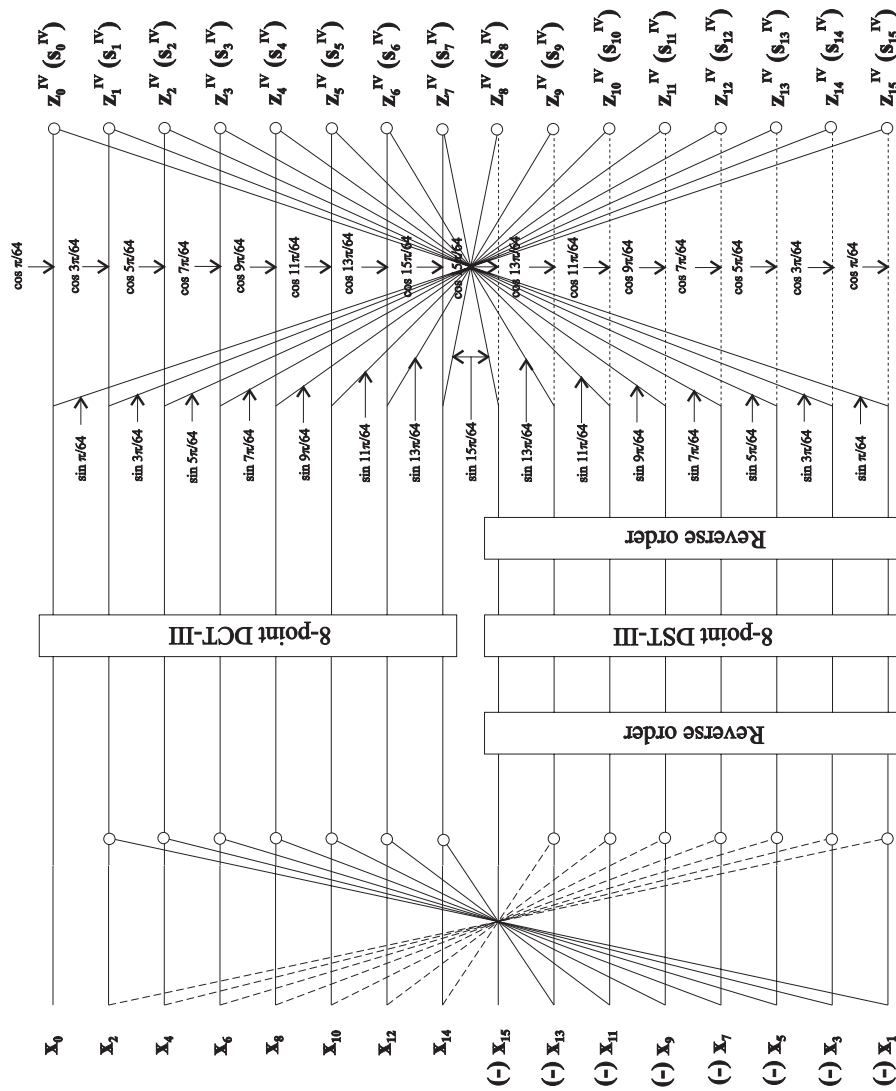


FIGURE 4.4
DCT-IV/DST-IV and IDCT-IV/IDST-IV generalized signal flow graph for $N = 16$. ©Slovak Academic Press Ltd.

4.3.5 Implementation of the Unified Fast Computation of DCTs and DSTs

All developed algorithms have been implemented in the C language, and they can be used in practical applications. Implemented algorithms are able to compute the DCT/DST orthogonal transform of a given type for real data sequence up to

size 1024. By minor modification (macro SIZE and LOG2SIZE) in program modules, any DCT/DST can be computed for the required size. In the implementation of DCT-II/DST-II (DCT-III/DST-III) universal computational structure, the normalization is optional. All computations are performed in double precision.

The orthonormal versions of the DCT and DST have the normalization factor $\sqrt{2/N}$ in both the forward and inverse transforms. Therefore, for the computation of orthonormal DCTs and DSTs, the implemented algorithms can be easily modified.

Computer Program for the Fast DCT-II/DST-II and DCT-III/DST-III Computation

```

/*-----*
*Module:   The 1-D Fast Discrete Cosine II and III *
*          Transform (DCT) and Discrete Sine II   *
*          and III Transform (DST)                 *
*          *                                       *
*Algorithm: DCT/DST universal computational        *
*           structure for the 1-D DCT-II/DST-II and *
*           DCT-III/DST-III Transform Computation *
*          *                                       *
*Note that the DCT/DST universal computational    *
*structure in algorithms for discrete sinusoidal   *
*transforms computation is used without the        *
*normalization. This module simulates a potential *
*DCT/DST processor.                               *
*-----*/
/*--- Prototypes to be included in calling program---*/
int dct_processor (
    double *pdct, /* input/output vector of length 2**m */
    int m,        /* m = log_2 (N)
                  E.g. for N = 256 --> m = 8 */
    int norm,     /* norm = 0 normalization is disabled
                  norm != 0 normalization is enabled */
    int flag);   /* Transform computation:
                  flag = 1  1-D DCT-II
                  flag = -1 1-D DCT-III
                  flag = 2  1-D DST-II
                  flag = -2 1-D DST-III */
/* NOTE: Function returns into calling program
   following value:
       0 - successful processing
      -1 - invalid length of input vector or
           invalid type transform computation */
/*----- Includes -----*/
#include <math.h>
/*----- Defines -----*/
#define SIZE 1024 /* max length 1024 */
#define LOG2SIZE 10 /* log_2 (SIZE) */
#define PI 3.141592653589793 /* pi */
#define DCT_II 1
#define DCT_III -1
/*----- Local Variables -----*/
static double ac [SIZE]; /* working vector */
static double cs [SIZE-1]; /*table of cos coefficients*/
static int length;
/* --- Beginning of the DCT/DST processor module ---*/

```

```

int dct_processor(double *pdct, int m, int norm,
                 int flag)
{
    int i,j,k,n,n1,n2,r,s,f0,f1,f2,f3,ip,ic,half,base;
    double arg,fi,scale,tmp,*pcl,*pc2,*pac = &ac [0];
    /* Verification of the input vector length (SIZE) */
    if ( m < 1 || m > LOG2SIZE )
        return (-1);
    /* Verification of the transform type computation */
    if ( flag < -2 || flag == 0 || flag > 2 )
        return (-1);
    /* Initialize input vector length and variables */
    n = 1 << m;
    n1 = n - 1;
    n2 = n >> 1;
    /* Generate the table of cosine coefficients Table
       is updated for new value of N */
    if ( length != n )
    {
        scale = 1.0 / (double) (n << 1);
        for ( s = base = 0; s < m; s++, base += ip )
        {
            half = n >> s;
            ip = half >> 1;
            ic = n / half;
            arg = (double) ic * PI * scale;
            for ( i = 0; i < ip; i++ )
            {
                fi = (double) (4 * i + 1) * arg;
                cs [base+i] = cos (fi);
            }
        }
        length = n;
    }
    /* Test type of computation - Forward or Inverse
       transform */
    if ( flag < 0 )
        goto inv;
/*
=====
THE 1-D FAST DCT-II OR DST-II TRANSFORM
=====
*/
/* Reordering of the original input data sequence */
for ( i = 0; i < n2; i++ )
{
    *(pac + i) = *(pdct + 2 * i);
    if ( flag == DCT_II )
        *(pac + n - 1 - i) = *(pdct + 2 * i + 1);
    else
        *(pac + n - 1 - i) = - *(pdct + 2 * i + 1);
}
/* Implementation of the butterfly structure */
for ( s = base = 0; s < m; s++, base += ip )
{
    half = n >> s;
    ip = half >> 1;
    for ( j = 0; j < ip; j++ )
        for ( i = j; i < n; i += half )

```

```

        {
            pc1 = pac + i;
            pc2 = pc1 + ip;
            tmp = *pc1 + *pc2;
            *pc2 = (*pc1 - *pc2) * cs [base+j];
            *pc1 = tmp;
        }
    }
/* Bit reversal permutation */
for ( i = 1; i < n1; i++ )
{
    for ( k = j = 0, r = i; k < m; k++ )
    {
        s = r >> 1;
        j = j + j + r - s - s;
        r = s;
    }
    if ( i < j )
    {
        tmp = *(pac + i);
        *(pac + i) = *(pac + j);
        *(pac + j) = tmp;
    }
}
/* Implementation of the pipeline structure */
if ( m > 1 )
{
    for ( i = 0; i < m - 1; i++ )
    {
        f0 = n / (1 << i);
        f1 = f0 >> 1;
        f2 = f1 >> 1;
        f3 = ((1 << i) - 1) << 1;
        for ( j = 1; j <= f2; j++ )
        {
            ip = f0 - j;
            ic = f1 - j;
            pc1 = pac + ip;
            pc2 = pac + ic;
            *pc1 += *pc1 - *pc2;
            k = 1;
            while ( k <= f3 )
            {
                ip += f1;
                ic += f1;
                pc1 = pac + ip;
                pc2 = pac + ic;
                *pc1 += *pc1 - *pc2;
                k++;
            }
        }
    }
}

/*-----
The normalization of the transformed data sequence.
If DCT-II/DST-II transform is required, then parameter
norm != 0. The block is not used for other discrete
sinusoidal transforms computation. Then norm = 0.
-----*/

```

```

if ( norm )
{
    scale = 2.0 / (double) n;
    *pac *= 1.0 / sqrt (2.0);
    for ( i = 0; i < n; i++ )
        *(pac + i) *= scale;
}
/* Reverse order of the data sequence for DST-II */
if ( flag == DCT_II )
    for ( i = 0; i < n; i++ )
        *(pdct + i) = *(pac + i);
else
    for ( i = 0; i < n; i++ )
        *(pdct + i) = *(pac + n - 1 - i);
return (0);
/*
=====
THE 1-D FAST DCT-III OR DST-III TRANSFORM
=====
*/
inv:
/* Reverse order of the data sequence for DST-III */
if ( flag == DCT_III )
    for ( i = 0; i < n; i++ )
        *(pac + i) = *(pdct + i);
else
    for ( i = 0; i < n; i++ )
        *(pac + n - 1 - i) = *(pdct + i);
/*
-----
The normalization of the DC term. If DCT-III/DST-III
transform is required, then parameter norm != 0. The
block is not used for other discrete sinusoidal
transforms computation. Then norm = 0.
-----*/
if ( norm )
    *pac *= 1.0 / sqrt (2.0);
/* Implementation of the pipeline structure */
if ( m > 1 )
{
    for ( i = m - 2; i >= 0; i-- )
    {
        f0 = n / (1 << i);
        f1 = f0 >> 1;
        f2 = f1 >> 1;
        f3 = ((1 << i) - 1) << 1;
        for ( j = f2; j > 0; j-- )
        {
            k = f3;
            ip = f0 - j + k * f1;
            ic = f1 - j + k * f1;
            pc1 = pac + ic;
            pc2 = pac + ip;
            *pc1 -= *pc2;
            *pc2 += *pc1;
            while ( k > 0 )
            {
                k--;
                ip -= f1;
                ic -= f1;
                pc1 = pac + ic;
                pc2 = pac + ip;
            }
        }
    }
}

```

```

        *pc1 -= *pc2;
        *pc2 += *pc2;
    }
}
}
/* Bit reversal permutation */
for ( i = 1; i < n1; i++ )
{
    for ( k = j = 0, r = i; k < m; k++ )
    {
        s = r >> 1;
        j = j + j + r - s - s;
        r = s;
    }
    if ( i < j )
    {
        tmp = *(pac + i);
        *(pac + i) = *(pac + j);
        *(pac + j) = tmp;
    }
}
/* Implementation of the butterfly structure */
for ( s = 0, base = n - 2; s < m; s++, base -= half )
{
    half = 1 << (s + 1);
    ip = half >> 1;
    for ( j = 0; j < ip; j++ )
        for ( i = j; i < n; i += half )
        {
            pc1 = pac + i;
            pc2 = pc1 + ip;
            tmp = *pc2 * cs [base+j];
            *pc2 = *pc1 - tmp;
            *pc1 = *pc1 + tmp;
        }
}
/* Reordering of output samples for DCT-III/DST-III */
for ( i = 0; i < n2; i++ )
{
    *(pdct + 2 * i) = *(pac + i);
    if ( flag == DCT_III )
        *(pdct + 2 * i + 1) = *(pac + n - 1 - i);
    else
        *(pdct + 2 * i + 1) = - *(pac + n - 1 - i);
}
return (0);
}
/*----- End of the DCT/DST processor module -----*/

```

Computer Program for the Fast DCT-I Computation

```

/*-----*
*Module:   The 1-D Fast Discrete Cosine I Transform *
*
*Algorithm: The Forward and Inverse 1-D DCT-I      *
*           Transform Computation                  *
*
*Note that the DCT-I matrix of order N + 1 and it is *
*symmetric. Thus, the forward and inverse transforms *

```



```

    *are the same except for the normalization.
    *-----*/
/*--- Prototypes to be included in calling program ---*/
int fdctild (
    double *x, /* input/output vector of length 2**m+1 */
    int m,      /* m = log2 (N)
                  E.g. for N = 256 --> m = 8 */
    int flag); /* Forward or Inverse DCT-I computation:
                  flag = 0 Forward 1-D DCT-I
                  flag = 1 Inverse 1-D DCT-I */
/* NOTE: Function returns into calling program
   following
   value:
       0 - successful processing
      -1 - invalid length of input vector
   */
/*----- Includes -----*/
#include <math.h>
/*----- Defines -----*/
#define SIZE 1024 /* max length SIZE+1 */
#define LOG2SIZE 10 /* log2 (SIZE) */
/* NOTE: Actual transform size is SIZE + 1 */
int dct_processor (double *, int, int, int);
/*----- Local Variables -----*/
static double y [SIZE+1];
/*----- Beginning of the 1-D Fast DCT-I module-----*/
int fdctild (double *x, int m, int flag)
{
    int i,j,n,n1,n2,n3,nc;
    double scale,tmp;
    /* Verification of the input vector length (SIZE+1) */
    if ( m < 1 || m > LOG2SIZE )
        return (-1);
    /* Initialize the input vector length */
    n = 1 << m;
    /* Multiply x[0] and x [n] by 1 / sqrt(2) */
    scale = 1.0 / sqrt (2.0);
    x [0] *= scale;
    x [n] *= scale;
    /* Implementation of generalized signal flow graph */
    n1 = n >> 1;
    n2 = n;
    n3 = n << 1;
    nc = m - 1;
    do
    {
        /* Butterflies for even-odd transform matrix A(N) */
        for { i = 0; i < n1; i++ )
            {
                tmp = x [i];
                x [i] = tmp + x [n2 - i];
                x [n2 - i] = tmp - x [n2 - i];
            }
        /* Reverse order of the input data sequence */
        for ( i = n1 + 1, j = i + n1 - 1; i < j; i++, j-- )
            {
                tmp = x [i];
                x [i] = x [j];
            }
    }
}

```

```

        x [j] = tmp;
    }
    /* Compute the DCT-III transform */
    dct_processor (&x [n1+1],nc,0,-1);
    /* Reverse order of the transformed data sequence */
    for ( i = n1 + 1, j = i + n1 - 1; i < j; i++, j-- )
    {
        tmp    = x [i];
        x [i]  = x [j];
        x [j]  = tmp;
    }
    n1 >>= 1;
    n2 >>= 1;
    nc--;
    /* The last butterfly - 2x2 transform matrix */
    if ( n2 == 1 )
    {
        tmp    = x [0];
        x [0]  = tmp + x [1];
        x [1]  = tmp - x [1];
    }
    while ( n2 > 1 );
    /* Reorder data sequence by permutation matrix P(N) */
    n2 = 2;
    n1 = n2 >> 1;
    do
    {
        for ( i = 0; i < n2 + 1; i++ )
            y [i] = x [i];
        for ( i = 0; i < n1; i++ )
        {
            x [2 * i + 2] = y [i + 1];
            x [2 * i + 1] = y [n2 - i];
        }
        n2 <<= 1;
        n1 <<= 1;
    }
    while ( n2 < n3 );
    /* Multiply x[0] and x [n] by 1 / sqrt(2) */
    x [0] *= scale;
    x [n] *= scale;
    /* Normalization of the transformed data sequence */
    if ( !flag )
    {
        scale = 2.0 / (double) n;
        for ( i = 0; i < n + 1; i++ )
            x [i] *= scale;
    }
    return (0);
}
/*----- End of the 1-D Fast DCT-I module-----*/

```

Computer Program for the Fast DST-I Computation

```

/*-----*
*Module:   The 1-D Fast Discrete Sine I Transform  *
*          *                                         *
*Algorithm: The Forward and Inverse 1-D DST-I      *
*          Transform Computation                    *

```

```

*
*Note that the DST-I matrix of order N - 1 and it is
*symmetric. Thus, the forward and inverse transforms
*are the same except for the normalization.
*-----*/
/*--- Prototypes to be included in calling program ---*/
int fdstild (
    double *x, /* input/output vector of length 2**m-1 */
    int m,      /* m = log_2 (N)
                  E.g. for N = 256 --> m = 8 */
    int flag); /* Forward or Inverse DST-I computation:
                  flag = 0 Forward 1-D DST-I
                  flag = 1 Inverse 1-D DST-I */
/*NOTE: Function returns into calling program following
value:
0 - successful processing
-1 - invalid length of input vector
/*----- Defines -----*/
#define SIZE 1024 /* max length SIZE-1 */
#define LOG2SIZE 10 /* log_2 (SIZE) */
/* NOTE: Actual transform size is SIZE - 1 */
int dct_processor (double *, int, int, int);
/*----- Local Variables -----*/
static double y [SIZE-1];
/* working vector of length N-1 */
/*----- Beginning of the 1-D Fast DST-I module-----*/
int fdstild (double *x, int m, int flag)
{
    int i,j,n,n1,n2,nb,nc;
    double scale,tmp;
    /* Verification of the input vector length (SIZE-1) */
    if ( m < 2 || m > LOG2SIZE )
        return (-1);
    /* Trivial case m = 1 */
    if ( m == 1 )
        return (0);
    /* Initialize the input vector length */
    n = 1 << m;
    /* Implementation of generalized signal flow graph */
    n1 = n >> 1;
    n2 = n;
    nc = m - 1;
    nb = 0;
    while ( n2 > 2 )
    {
        /* Butterflies for even-odd transform matrix A(N) */
        if ( n == n2 )
            for ( i = 0; i < n1 - 1; i++ )
            {
                tmp = x [i];
                x [i] = tmp + x [n - 2 - i];
                x [n - 2 - i] = tmp - x [n - 2 - i];
            }
        /* Butterflies for even-odd transform matrix A(N)
        with reversed order of its columns */
        else
            for ( i = 0; i < n1 - 1; i++ )

```

```

        tmp      = x [nb + i];
        x [nb + i] = x [n - 2 - i] + tmp;
        x [n - 2 - i] = x [n - 2 - i] - tmp;
    }
    /* Compute the DST-III transform */
    dct_processor (&x [nb],nc,0,-2);
    n2 >>= 1;
    n1 >>= 1;
    nb += (1 << nc);
    nc--;
}
/* Reorder of data sequence by permutation matrix P(N) */
n2 = 2;
nc = 1;
nb = n - 2 - (1 << nc);
while ( n2 < n )
{
    for ( i = nb; i < n - 1; i++ )
        y [i] = x [i];
    for ( i = 0; i < n2 - 1; i++ )
    {
        x [nb + 2 * i + 2] = y [nb + i + 1];
        x [nb + 2 * i + 1] = y [n - 2 - i];
    }
}
/* Reverse order of the permuted data sequence */
if ( nb != 0 )
    for ( i = nb, j = n - 2; i < j; i++, j-- )
    {
        tmp      = x [i];
        x [i] = x [j];
        x [j] = tmp;
    }
    n2 <=< 1;
    nc++;
    nb -= (1 << nc);
}
/* Normalization of the transformed data sequence */
if ( !flag )
{
    scale = 2.0 / (double) n;
    for ( i = 0; i < n - 1; i++ )
        x [i] *= scale;
}
return (0);
}
/*----- End of the 1-D Fast DST-I module-----*/

```

Computer Program for the Fast DCT-IV/DST-IV Computation

```

/*-----*
*Module:      The 1-D Fast Discrete Cosine IV and      *
*            Discrete Sine IV Transform                *
*            *                                          *
*Algorithm:   The Forward and Inverse 1-D DCT-IV/DST-IV*
*            Transform Computation                    *
*            *                                          *
*Note that the DCT-IV and DST-IV matrices are          *
*symmetric. Thus, the forward and inverse transforms  *
*are the same except for the normalization.            *
*-----*/

```

```

/* --- Prototypes to be included in calling program ---*/
int fdcstivld (
    double *x, /* input/output vector of length 2**m */
    int m,      /* log_2 vector length
                  E.g. N = 256 --> m = 8 */
    int flag); /* Forward or Inverse DCT-IV/DST-IV
                  computation:
                  flag = 1  Forward 1-D DCT-IV
                  flag = -1 Inverse 1-D DCT-IV
                  flag = 2  Forward 1-D DST-IV
                  flag = -2 Inverse 1-D DST-IV */
/* NOTE: Function returns into calling program
   following
   value:
   0 - successful processing
   -1 - invalid length of input vector
       invalid type transform computation */
/* ----- Includes -----*/
#include <math.h>
/* ----- Defines -----*/
#define SIZE 1024 /* max length 1024 */
#define LOG2SIZE 10 /* log_2 (SIZE) */
#define PI 3.141592653589793 /* pi */
int dct_processor (double *, int, int, int);
/* ----- Local Variables ----- */
static double y [SIZE]; /* working vector of length N */
static double as [SIZE/2]; /* table of sine values */
static double cc [SIZE/2]; /* table of cosine+sine values*/
static double ss [SIZE/2]; /* table of sine-cosine values*/
static int length;
/*-- Beginning of the 1-D Fast DCT-IV/DST-IV module --*/
int fdcstivld (double *x, int m, int flag)
{
    int i,j,n,n2,n4;
    double arg,dev,argc,args, scale,tmp;
    /* Verification of the input vector length (SIZE) */
    if ( m < 1 || m > LOG2SIZE )
        return (-1);
    /* Verification of the type transform computation */
    if ( flag < -2 || flag == 0 || flag > 2 )
        return (-1);
    /* Initialize the input vector length and variables */
    n = 1 << m;
    n2 = n >> 1;
    n4 = n >> 2;
    /* Generate tables of sines and cosines for rotation
       matrix R(N). Table is updated for new value of N */
    if ( length != n )
    {
        arg = PI / (double) (n << 2);
        dev = PI / (double) (n << 1);
        for ( i = 0; i < n2; i++, arg += dev )
        {
            argc = cos (arg);
            args = sin (arg);
            as [i] = args;

```

```

        cc [i] = argc + args;
        ss [i] = args - argc;
    }
    length = n;
}
/* Reordering of data sequence by permutation matrix
   P(N). For DST-IV computation odd-numbered samples
   are sign-changed */
for ( i = 0; i < n2; i++ )
{
    y [i] = x [2 * i];
    if ( flag == 1 || flag == -1 )
        y [n - 1 - i] = x [2 * i + 1];
    else
        y [n - 1 - i] = - x [2 * i + 1];
}
/* Butterflies corresponding to the matrix product
   P(N) B(N) */
for ( i = 1; i < n2; i++ )
{
    tmp      = y [n - i] - y [i];
    y [i]     = y [n - i] + y [i];
    y [n - i] = tmp;
}
/* Get DCT-III transform of the first n/2 samples */
dct_processor (&y [0],m-1,0,-1);
/* Reverse order of the last n/2 samples */
for ( i = n2, j = n - 1; i < j; i++, j-- )
{
    tmp      = y [i];
    y [i]     = y [j];
    y [j]     = tmp;
}
/* Get the DST-III of the last n/2 samples */
dct_processor (&y [n2],m-1,0,-2);
/* Reverse order of the last n/2 samples */
for ( i = n2, j = n - 1; i < j; i++, j-- )
{
    tmp      = y [i];
    y [i]     = y [j];
    y [j]     = tmp;
}
/* Butterflies for the rotation matrix T(N) */
for ( i = 0; i < n2; i++ )
{
    tmp      = (y [i] - y [n - 1 - i]) * as [i];
    x [i]     = y [i] * cc [i] - tmp;
    x [n - 1 - i] = y [n - 1 - i] * ss [i] + tmp;
}
/* DST-IV computation -
   reverse order of data sequence */
if ( flag == 2 || flag == -2 )
    for ( i = 0, j = n - 1; i < j; i++, j-- )
    {
        tmp      = x [i];
        x [i]     = x [j];
        x [j]     = tmp;
    }

```

```

/* Normalization of the transformed data sequence */
if ( flag > 0 )
{
    scale = 2.0 / (double) n;
    for ( i = 0; i < n; i++ )
        x [i] *= scale;
}
return (0);
}
/*----- End of the 1-D Fast DCT-IV/DST-IV module -----*/

```

4.4 The 2-D DCT/DST Universal Computational Structure

Section 4.3 presented fast algorithms for 1-D computation of a given type of DCT/DST (I, II, III, IV) together with their implementations. For digital image processing applications, the fast 2-D algorithms are more significant than 1-D ones. For simplicity, in this section DCT and DST refer to types II and III only. The 2-D DCT and its inverse are used as the basic processing elements in international image/video coding standards [31].

Generally, there are two approaches to computation of the 2-D DCT: indirect and direct. In the indirect approach, the 2-D DCT computation can be realized via other 2-D discrete orthogonal transforms, such as the discrete Fourier transform or the Walsh–Hadamard transform [30]. There are two methods of direct approach which is based on direct 2-D DCT computation. The first, a so called row-column method, is based on the separability property of the 2-D DCT kernel, which sequentially uses any fast 1-D DCT algorithm on rows and columns of the input data matrix. The second is a vector radix method which uses a 2-D decomposition process. An algorithm obtained by this method outperforms the conventional row-column method in computational efficiency and works directly on 2-D data sets.

In this section, a generalized signal flow graph, the 2-D DCT/DST universal computational structure, is described. It represents a unified approach to the direct 2-D DCT and 2-D DST computation and their inverses for any square block of size $2^m \times 2^m$. The computer program implementing the direct 2-D DCT/DST is also presented.

4.4.1 The Fast Direct 2-D DCT/DST Computation

The 2-D DCT for an $N \times N$ input data matrix $\{x_{m,n}\}$, $m, n = 0, 1, \dots, N-1$ is defined by the following relation [30]

$$z_{k,l} = \frac{4\epsilon_k \epsilon_l}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x_{m,n} \cos \left[\frac{\pi(2m+1)k}{2N} \right] \cos \left[\frac{\pi(2n+1)l}{2N} \right], \quad (4.43)$$

$$k, l = 0, 1, \dots, N-1,$$

and the inverse 2-D DCT (2-D IDCT)

$$x_{m,n} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \epsilon_k \epsilon_l z_{k,l} \cos \left[\frac{\pi(2m+1)k}{2N} \right] \cos \left[\frac{\pi(2n+1)l}{2N} \right], \quad (4.44)$$

$$m, n = 0, 1, \dots, N-1,$$

where

$$\epsilon_p = \begin{cases} \frac{1}{\sqrt{2}} & p = 0 \\ 1 & \text{otherwise} \end{cases}$$

and N is assumed to be an integer power of 2. The corresponding 2-D DST is defined by

$$s_{k,l} = \frac{4\epsilon_k \epsilon_l}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x_{m,n} \sin \left[\frac{\pi(2m+1)(k+1)}{2N} \right] \sin \left[\frac{\pi(2n+1)(l+1)}{2N} \right], \quad (4.45)$$

$$k, l = 0, 1, \dots, N-1,$$

and the inverse 2-D DST (2-D IDST)

$$x_{m,n} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \epsilon_k \epsilon_l s_{k,l} \sin \left[\frac{\pi(2m+1)(k+1)}{2N} \right] \sin \left[\frac{\pi(2n+1)(l+1)}{2N} \right], \quad (4.46)$$

$$m, n = 0, 1, \dots, N-1,$$

where

$$\epsilon_p = \begin{cases} \frac{1}{\sqrt{2}} & p = N-1, \\ 1 & \text{otherwise.} \end{cases}$$

The recursive 1-D DCT/DST algorithm and its corresponding generalized signal flow graph with regular structure for any value of $N = 2^m$ (1-D DCT/DST universal computational structure) enable the formulation by the vector radix method of direct 2-D DCT/DST fast, recursive algorithm that possesses a regular structure for any $N \times N$ block size. By extension of reordering Eq. (4.14) to a 2-D case, the 2-D DCT and 2-D IDCT defined by Eqs. (4.43) and (4.44), respectively, can be rewritten in the following form [30]

$$z_{k,l} = \frac{4\epsilon_k \epsilon_l}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \tilde{x}_{m,n} \cos \left[\frac{\pi(4m+1)k}{2N} \right] \cos \left[\frac{\pi(4n+1)l}{2N} \right], \quad (4.47)$$

$$k, l = 0, 1, \dots, N-1,$$

$$\tilde{x}_{m,n} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \epsilon_k \epsilon_l z_{k,l} \cos \left[\frac{\pi(4m+1)k}{2N} \right] \cos \left[\frac{\pi(4n+1)l}{2N} \right], \quad (4.48)$$

$$m, n = 0, 1, \dots, N-1,$$

where

$$\begin{aligned} \tilde{x}_{m,n} &= x_{2m,2n} \\ \tilde{x}_{m,N-n-1} &= x_{2m,2n+1} \\ \tilde{x}_{N-m-1,n} &= x_{2m+1,2n} \\ \tilde{x}_{N-m-1,N-n-1} &= x_{2m+1,2n+1}, \quad m, n = 0, 1, \dots, \frac{N}{2} - 1. \end{aligned} \quad (4.49)$$

By reordering Eq. (4.49) an $N \times N$ input data matrix \mathbf{X} is decomposed into four $\frac{N}{2} \times \frac{N}{2}$ submatrices, as even-even, even-odd, odd-even, and odd-odd indexed elements. After reordering the input data and output transform matrix, a fast recursive algorithm for direct $N \times N$ 2-D DCT/DST computation is given in matrix form as [28]

$$\begin{bmatrix} \hat{\mathbf{z}}_{ee} \\ \hat{\mathbf{z}}_{eo} \\ \hat{\mathbf{z}}_{oe} \\ \hat{\mathbf{z}}_{oo} \end{bmatrix} = (\hat{C}_N \otimes \hat{C}_N) \begin{bmatrix} \tilde{\mathbf{x}}_{pp} \\ \tilde{\mathbf{x}}_{pr} \\ \tilde{\mathbf{x}}_{rp} \\ \tilde{\mathbf{x}}_{rr} \end{bmatrix}, \quad (4.50)$$

where

$$\begin{aligned} \hat{\mathbf{z}}_e &= (R \otimes R) \mathbf{z}_e, \quad \hat{\mathbf{z}}_e = \begin{bmatrix} \hat{\mathbf{z}}_{ee} \\ \hat{\mathbf{z}}_{eo} \end{bmatrix}, \quad \mathbf{z}_e = \begin{bmatrix} \mathbf{z}_{ee} \\ \mathbf{z}_{eo} \end{bmatrix}, \\ \hat{\mathbf{z}}_o &= (R \otimes R) \mathbf{z}_o, \quad \hat{\mathbf{z}}_o = \begin{bmatrix} \hat{\mathbf{z}}_{oe} \\ \hat{\mathbf{z}}_{oo} \end{bmatrix}, \quad \mathbf{z}_o = \begin{bmatrix} \mathbf{z}_{oe} \\ \mathbf{z}_{oo} \end{bmatrix}, \\ \tilde{\mathbf{x}} &= \begin{bmatrix} \tilde{\mathbf{x}}_{pp} \\ \tilde{\mathbf{x}}_{pr} \\ \tilde{\mathbf{x}}_{rp} \\ \tilde{\mathbf{x}}_{rr} \end{bmatrix} = (P_N \otimes P_N) \mathbf{x}. \end{aligned}$$

\otimes denotes the Kronecker matrix product. \mathbf{z}_e and \mathbf{z}_o are vectors consisting of transposed even and odd row vectors of the output transform matrix, both of which are arranged in the natural order, respectively. \mathbf{x} denotes the vector consisting of transposed row vectors of the input data matrix. The direct product $R \otimes R$ performs 2-D bit reversal permutation, and $P_N \otimes P_N$ performs 2-D rearrangement defined by

Eq. (4.49). For clarity of Eq. (4.50), an example for $N = 4$ is shown

$$\begin{bmatrix} z_{00} \\ z_{02} \\ z_{01} \\ z_{03} \\ \text{---} \\ z_{20} \\ z_{22} \\ z_{21} \\ z_{23} \\ \text{---} \\ z_{10} \\ z_{12} \\ z_{11} \\ z_{13} \\ \text{---} \\ z_{30} \\ z_{32} \\ z_{31} \\ z_{33} \end{bmatrix} = (\hat{C}_4 \otimes \hat{C}_4) \begin{bmatrix} x_{00} \\ x_{02} \\ x_{03} \\ x_{01} \\ \text{---} \\ x_{20} \\ x_{22} \\ x_{23} \\ x_{21} \\ \text{---} \\ x_{30} \\ x_{32} \\ x_{33} \\ x_{31} \\ \text{---} \\ x_{10} \\ x_{12} \\ x_{13} \\ x_{11} \end{bmatrix}.$$

Substituting the block matrix factorization of the DCT matrix \hat{C}_N Eq. (4.16) into Eq. (4.50) and using properties of the Kronecker matrix product the direct, fast and recursive 2-D DCT/DST algorithm is developed [28]

$$\begin{aligned} \hat{C}_N \otimes \hat{C}_N = & \left\{ \begin{bmatrix} I_{\frac{N}{2}} & 0 \\ 0 & K_{\frac{N}{2}} \end{bmatrix} \otimes \begin{bmatrix} I_{\frac{N}{2}} & 0 \\ 0 & K_{\frac{N}{2}} \end{bmatrix} \right\} \left\{ \begin{bmatrix} \hat{C}_{\frac{N}{2}} & 0 \\ 0 & \hat{C}_{\frac{N}{2}} \end{bmatrix} \otimes \begin{bmatrix} \hat{C}_{\frac{N}{2}} & 0 \\ 0 & \hat{C}_{\frac{N}{2}} \end{bmatrix} \right\} \\ & \left\{ \begin{bmatrix} I_{\frac{N}{2}} & 0 \\ 0 & Q_{\frac{N}{2}} \end{bmatrix} \otimes \begin{bmatrix} I_{\frac{N}{2}} & 0 \\ 0 & Q_{\frac{N}{2}} \end{bmatrix} \right\} \left\{ \begin{bmatrix} I_{\frac{N}{2}} & I_{\frac{N}{2}} \\ I_{\frac{N}{2}} & -I_{\frac{N}{2}} \end{bmatrix} \otimes \begin{bmatrix} I_{\frac{N}{2}} & I_{\frac{N}{2}} \\ I_{\frac{N}{2}} & -I_{\frac{N}{2}} \end{bmatrix} \right\}, \end{aligned} \quad (4.51)$$

where $K_{\frac{N}{2}}$ and $Q_{\frac{N}{2}}$ are $\frac{N}{2} \times \frac{N}{2}$ matrices given by Eqs. (4.17) and (4.18), respectively. From Eq. (4.22) it follows that by this algorithm the direct 2-D DST computation can be realized merely by sign changes on the input data matrix (direct product $D_N \otimes D_N$) and after the 2-D DCT computation, reversing order along both rows and columns of the output transformed DCT data matrix (direct product $\bar{I}_N \otimes \bar{I}_N$).

The detailed analysis of the intrinsic structure of the algorithm given by Eqs. (4.50) and (4.51) results in a highly regular 2-D DCT/DST generalized signal flow graph, the 2-D DCT/DST universal computational structure, representing the unified direct 2-D DCT and 2-D DST computation and their inverses for any $N \times N$ block size [29]. It is shown for a 16×16 block in Fig. 4.6. The 2-D DCT/DST universal computational structure consists of two regular parts. The first part is related to

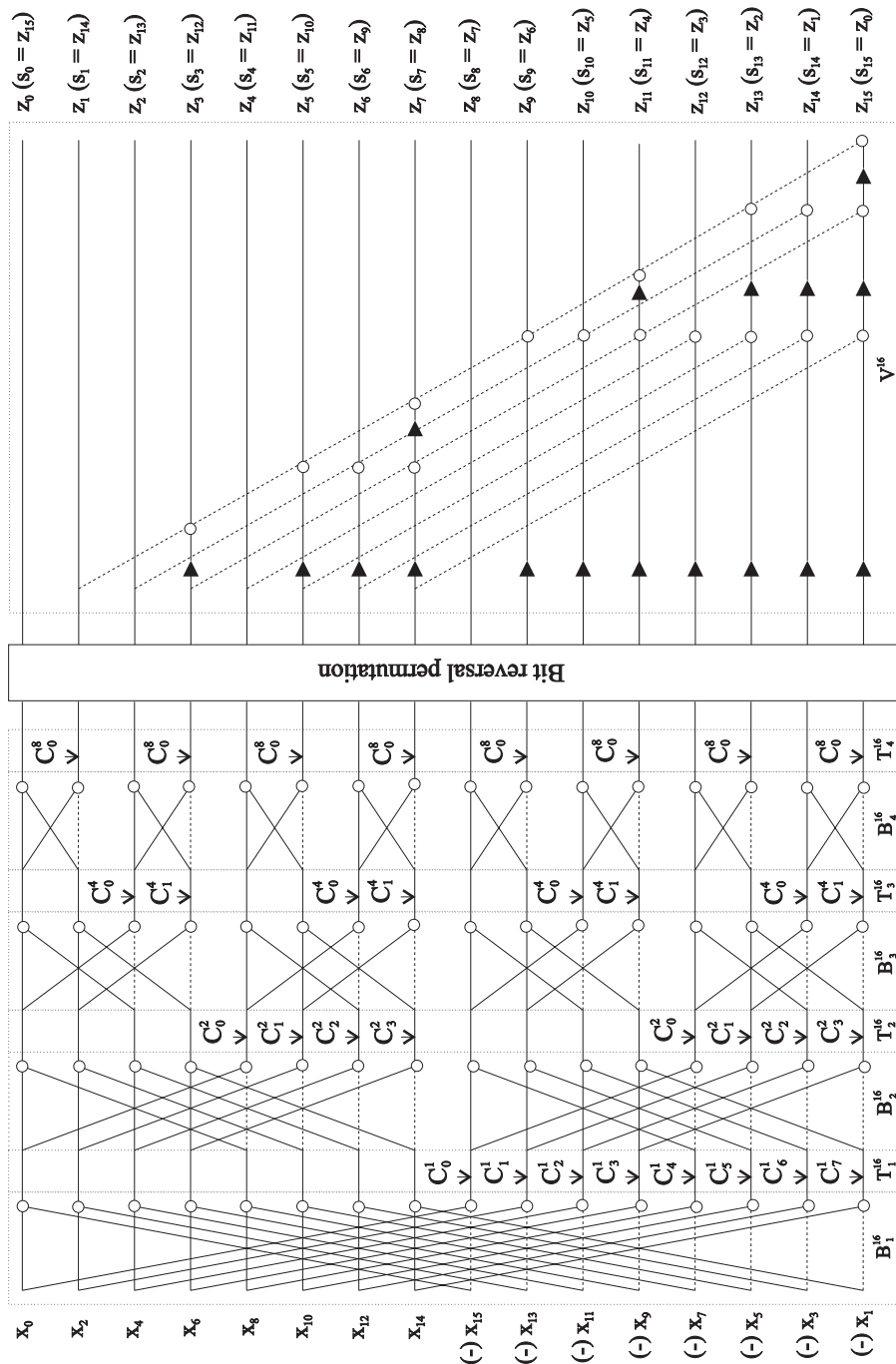


FIGURE 4.5
1-D DCT/DST universal computational structure for $N = 16$. ©Springer-Verlag London Ltd.

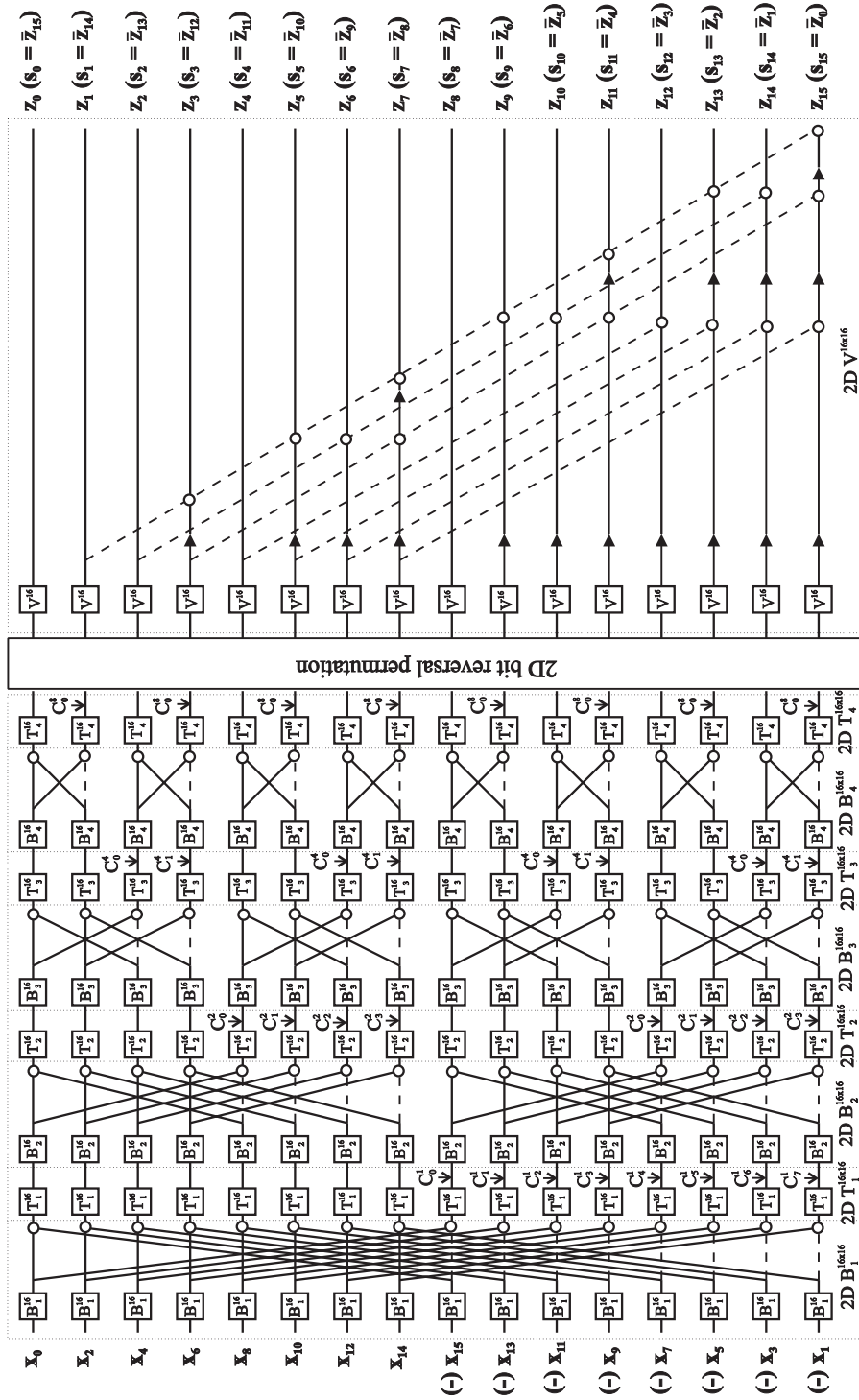


FIGURE 4.6
2-D DCT/DST universal computational structure for 16×16 block size.
 ©Springer–Verlag London Ltd.

the 2-D butterfly structure, and the second one, after the 2-D bit reversal permutation, is mapped into a 2-D pipeline structure. This 2-D pipeline structure can be represented by a regular computational scheme of the same type for any block size $2^m \times 2^m$ [29]. In order to show a one-to-one relationship between the 2-D DCT/DST universal computational structure and its 1-D counterpart, for a given $N \times N$ block size it is partitioned into blocks 2-D $B_i^{N \times N}$, 2-D $T_i^{N \times N}$, $i = 1, 2, \dots, \log_2 N$ related to the 2-D butterfly structure and the block 2-D $V^{N \times N}$ related to the 2-D pipeline structure. All blocks indicated by B_i^N , T_i^N , $i = 1, 2, \dots, \log_2 N$ and the block V^N are defined in the 1-D DCT/DST universal computational structure (Fig. 4.5). Heavy lines in Fig. 4.6 denote vector operations on rows of the input data matrix, $\mathbf{x}_i = [x_{i,0}, x_{i,2}, \dots, x_{i,N-2}, x_{i,N-1}, \dots, x_{i,3}, x_{i,1}]^T$ and $\mathbf{z}_i = [z_{i,0}, z_{i,1}, \dots, z_{i,N-2}, z_{i,N-1}]^T$ for $i = 0, 1, \dots, N-1$. The symbols in brackets correspond to the 2-D DST computation and $\bar{\mathbf{z}} = \bar{I} \mathbf{z}$.

Recall that in the international image/video coding standards [31] the 2-D DCT and its inverse are defined for fixed 8×8 blocks as [43]

$$z_{k,l} = \frac{\epsilon_k \epsilon_l}{4} \sum_{m=0}^7 \sum_{n=0}^7 x_{m,n} \cos \left[\frac{\pi(2m+1)k}{16} \right] \cos \left[\frac{\pi(2n+1)l}{16} \right], \quad (4.52)$$

$$k, l = 0, 1, \dots, 7$$

$$x_{m,n} = \frac{1}{4} \sum_{k=0}^7 \sum_{l=0}^7 \epsilon_k \epsilon_l z_{k,l} \cos \left[\frac{\pi(2m+1)k}{16} \right] \cos \left[\frac{\pi(2n+1)l}{16} \right], \quad (4.53)$$

$$m, n = 0, 1, \dots, 7$$

The 2-D DCT given by Eq. (4.52) is identical to Eq. (4.43) for $N = 8$ except for a scaling factor of 4.

4.4.2 Implementation of the Direct 2-D DCT/DST Computation

The 2-D DCT/DST universal computational structure has been implemented in C. It can compute 2-D DCT or 2-D DST and their inverses for any square $2^m \times 2^m$, $m > 0$ block size. The cosine coefficients for a given $N = 2^m$ are precomputed and stored in tables. The tables are updated if the program calls for a new value of N . If a larger block size is required for 2-D DCT/DST computation, then macros SIZE and LOG2SIZE should be redefined in the program. In the implementation of the 2-D DCT/DST universal computational structure, the normalization is optional. All computations are performed in double precision.

The transposition of the input data matrix required in Eq. (4.50) and its reordering given by Eq. (4.49) can be realized simultaneously as follows:

$$\begin{aligned} \tilde{x}_{n,m} &= x_{2m,2n} \\ \tilde{x}_{n,N-m-1} &= x_{2m,2n+1} \\ \tilde{x}_{N-n-1,m} &= x_{2m+1,2n} \\ \tilde{x}_{N-n-1,N-m-1} &= x_{2m+1,2n+1}, \quad m, n = 0, 1, \dots, \frac{N}{2} - 1. \end{aligned} \quad (4.54)$$

```

/*-----*
* Module:   The 2-D Fast Discrete Cosine/Sine      *
*           Transform (2-D DCT/DST Universal       *
*           Computational Structure)                *
*-----*
* Algorithm: The Forward and Inverse 2-D DCT/DST   *
*            computation by vector-radix structured *
*            approach for block sizes N x N, i.e.,  *
*            square blocks. N is assumed to be an  *
*            integer powers of 2.                   *
*-----*/
/* --- Prototypes to be included in calling program ---*/
int fdcst2d (
    double **x, /* input/output matrix of dimension NxN */
    int m,      /* m = log_2 (N) for N x N block size
                  e.g., length = 8 -> m = 3 */
    int norm,   /* norm = 0 normalization is disabled
                  norm != 0 normalization is enabled */
    int flag); /* Forward or Inverse DCT/DST computation:
                  flag = 1  2-D DCT-II
                  flag = -1 2-D DCT-III
                  flag = 2  2-D DST-II
                  flag = -2 2-D DST-III
    -----
    DECLARATION OF THE INPUT MATRIX: Let N = 8 --> then
    m = 3. Input matrix 8x8 must be declared in calling
    program as follows:
        double block [8*8]; /declarations
        double *x    [8];
        for ( i = 0; i < 8; i++ )
            x [i] = block + i * 8; /pointers to rows
                                   of the block
        fdcst2d (&x,3,1, 1); / DCT-II computation
        fdcst2d (&x,3,1,-1); /IDCT-II computation
        fdcst2d (&x,3,1, 2); / DST-II computation
        fdcst2d (&x,3,1,-2); /IDST-II computation
    -----
    NOTE: Function returns into calling program following
    value:
        0 - successful processing
        -1 - invalid dimension of input matrix
        -2 - invalid transform type
    -----*/
/* ----- Includes -----*/
#include <math.h>
/* ----- Defines -----*/
#define SIZE 32 /* max dimension 32x32 */
#define LOG2SIZE 5 /* log_2 max dimension */
#define PI 3.141592653589793 /* pi */
#define SQRT2 0.707106781186547 /* sqr (1/2) */
#define DCT 1
#define IDCT -1
/* ----- Local Variables -----*/
static double ac [SIZE*SIZE]; /* working array */
static double *z [SIZE]; /* array of pointers */
static int ntab_cs = 0;
static double tc1 [SIZE-1]; /* tables of cos coefficients*/
static double tc2 [SIZE*SIZE/3];
static int tab1_len = 0;
static int tab2_len = 0;

```

```

/*----- Beginning of the Fast 2-D DCT/DST module ----- */
int fdcst2d (double **x, int m, int norm, int flag)
{
    int    i,j,k,n,n1,n2,r,s,t,u,f0,f1,f2,f3,ip,ic,half;
    int    b1,b2;
    double arg,fi1,fi2,scale,scl,tmp,*ptr,*z1,*z2;
    /* Verification of the input matrix dimension
       (SIZE x SIZE) */
    if ( m < 0 || m > LOG2SIZE )
        return (-1);
    /* Verification of the transform type computation */
    if ( flag < -2 || flag == 0 || flag > 2 )
        return (-2);
    /* Trivial transform if m = 0 */
    if ( m == 0 )
        return (0);
    /* Initialize input matrix dimension and variables */
    n = 1 << m;
    n1 = n - 1;
    n2 = n >> 1;
    /* Initialize pointers on rows of the input matrix */
    for ( i = 0; i < n; i++ )
        z[i] = ac + i * n;
    /* Compute tables of cosine coefficients for new
       value of N */
    if ( ntab_cs != n )
    {
        b1 = b2 = tab1_len = tab2_len = 0;
        scale = 1.0 / (double) (n << 1);
        for ( s = ip = 1; s <= m; s++, ip <= 1 )
        {
            ic = n >> s;
            arg = (double) ip * PI * scale;
            for ( i = 0; i < ic; i++ )
            {
                fi1 = (double) (4 * i + 1) * arg;
                tc1 [b1+i] = cos (fi1);
                tab1_len++;
            }
            for ( i = u = 0; i < ic; i++, u = i * ic )
            {
                for ( j = 0; j < ic; j++ )
                {
                    fi2 = (double) (4 * j + 1) * arg;
                    tc2 [b2+u+j] = tc1 [b1+i] * cos (fi2);
                    tab2_len++;
                }
                b1 += ic;
                b2 += ic * ic;
            }
            ntab_cs = n;
        }
    }
    /* Test type of 2-D DCT/DST computation */
    if ( flag < 0 )
        goto inv;
/*
=====
THE 2-D FAST FORWARD DISCRETE COSINE/SINE TRANSFORM
=====
*/

```

```

/* Reordering and transposition of input data matrix
----- */
for ( i = 0; i < n2; i++ )
    for ( j = 0; j < n2; j++ )
    {
        z [j] [i] = x [2*i] [2*j];
        z [n-j-1] [n-i-1] = x [2*i+1] [2*j+1];
        if ( flag == DCT )
        {
            z [n-j-1] [i] = x [2*i] [2*j+1];
            z [j] [n-i-1] = x [2*i+1] [2*j];
        }
        else
        {
            z [n-j-1] [i] = -x [2*i] [2*j+1];
            z [j] [n-i-1] = -x [2*i+1] [2*j];
        }
    }

/* Implementation of the 2-D butterfly structure
----- */
for ( s = b1 = b2 = 0; s < m; s++ )
{
    half = n >> s;
    ip = half >> 1;
    /* Butterflies along rows of the data matrix */
    for ( i = 0, z1 = z [0]; i < n; i++, z1 = z [i] )
        for ( j = 0; j < ip; j++ )
            for ( k = j; k < n; k += half )
            {
                tmp = z1 [k] + z1 [k+ip];
                z1 [k+ip] = z1 [k] - z1 [k+ip];
                z1 [k] = tmp;
            }
    /* Butterflies between rows of the data matrix */
    for ( j = u = 0; j < ip; j++, u = j*ip )
        for ( k = j; k < n; k += half )
        {
            z1 = z [k];
            z2 = z [k+ip];
            for ( i = 0; i < n; i++ )
            {
                tmp = *z1 + *z2;
                *z2++ = *z1 - *z2;
                *z1++ = tmp;
            }
        }
    /* Multiplications by cosine coefficients */
    z1 = z [k];
    z2 = z [k+ip];
    for ( r = 0; r < ip; r++ )
        for ( t = r; t < n; t += half )
        {
            z1 [t+ip] *= tc1 [b1+r];
            z2 [t] *= tc1 [b1+j];
            z2 [t+ip] *= tc2 [b2+u+r];
        }
    b1 += ip;
    b2 += ip * ip;
}

```



```

/* The 2-D bit reversal permutation
----- */
for ( t = 0, z1 = z [0]; t < n; t++, z1 = z [t] )
    for ( i = 1; i < n1; i++ )
    {
        for ( k = j = 0, r = i; k < m; k++ )
        {
            s = r >> 1;
            j = j + j + r - s - s;
            r = s;
        }
        if ( i < j )
        {
            tmp      = z1 [i];
            z1 [i] = z1 [j];
            z1 [j] = tmp;
        }
    }
for ( i = 1; i < n1; i++ )
{
    for ( k = j = 0, r = i; k < m; k++ )
    {
        s = r >> 1;
        j = j + j + r - s - s;
        r = s;
    }
    if ( i < j )
    {
        ptr      = z [i];
        z [i] = z [j];
        z [j] = ptr;
    }
}
/* Implementation of the 2-D pipeline structure
----- */
if ( m > 1 )
{
    /* Pipelines along rows of the data matrix */
    for ( i = 0; i < m - 1; i++ )
    {
        f0 = n / (1 << i);
        f1 = f0 >> 1;
        f2 = f1 >> 1;
        f3 = ((1 << i) - 1) << 1;
        z1 = z [0];
        for ( t = 0; t < n; t++, z1 = z [t] )
            for ( j = 1; j <= f2; j++ )
            {
                ip = f0 - j;
                ic = f1 - j;
                z1 [ip] += z1 [ip] - z1 [ic];
                k = 1;
                while ( k <= f3 )
                {
                    ip += f1;
                    ic += f1;
                    z1 [ip] += z1 [ip] - z1 [ic];
                    k++;
                }
            }
    }
}

```

```

/* Pipelines between rows of the data matrix */
for ( j = 1; j <= f2; j++ )
{
    ip = f0 - j;
    ic = f1 - j;
    z1 = z [ip];
    z2 = z [ic];
    for ( t = 0; t < n; t++, z1++ )
        *z1 += *z1 - *z2++;
    k = 1;
    while ( k <= f3 )
    {
        ip += f1;
        ic += f1;
        z1 = z [ip];
        z2 = z [ic];
        for ( t = 0; t < n; t++, z1++ )
            *z1 += *z1 - *z2++;
        k++;
    }
}
}

/*-----
The normalization of the transformed data sequence.
If DCT-II/DST-II is required, then parameter
norm != 0. The block is not used for other discrete
sinusoidal transforms computation. Then norm = 0.
-----*/
if ( norm )
{
    scale = 4.0 / ((double) n * (double) n);
    for ( i = 0, z [0] [0] *= SQRT2; i < n; i++ )
        for ( j = 0; j < n; j++ )
        {
            z [i] [j] *= scale;
            if ( i == 0 || j == 0 )
                z [i] [j] *= SQRT2;
        }
}

/* Reverse rows and columns of the transformed data
matrix for the DST
----- */
for ( i = 0; i < n2; i++ )
    for ( j = 0; j < n; j++ )
        if ( flag == DCT )
        {
            x [j] [i] = z [i] [j];
            x [n-1-j] [n-1-i] = z [n-1-i] [n-1-j];
        }
        else
        {
            x [j] [i] = z [n-1-i] [n-1-j];
            x [n-1-j] [n-1-i] = z [i] [j];
        }
    return (0);
}

/*
=====
THE 2-D FAST INVERSE DISCRETE COSINE/SINE TRANSFORM
=====

```

```

*/
inv: /* Reverse rows and columns of the transformed data
      matrix for the IDST
      ----- */
for ( i = 0; i < n2; i++ )
    for ( j = 0; j < n; j++ )
        if ( flag == IDCT )
        {
            z [j      ] [i      ] = x [i      ] [j      ];
            z [n-1-j] [n-1-i] = x [n-1-i] [n-1-j];
        }
        else
        {
            z [j      ] [i      ] = x [n-1-i] [n-1-j];
            z [n-1-j] [n-1-i] = x [i      ] [j      ];
        }
/* -----
The normalization of the DC term. If DCT-III/DST-III
is required, then parameter norm != 0. The block is
not used for other discrete sinusoidal transforms
computation. Then norm = 0.
----- */
if ( norm )
    for ( i = 0, z [0] [0] *= SQRT2; i < n; i++ )
        for ( j = 0; j < n; j++ )
            if ( i == 0 || j == 0 )
                z [i] [j] *= SQRT2;
/* Implementation of the 2-D pipeline structure
----- */
if ( m > 1 )
{
/* Pipelines between rows of the data matrix */
for ( i = m - 2; i >= 0; i-- )
{
    f0 = n / (1 << i);
    f1 = f0 >> 1;
    f2 = f1 >> 1;
    f3 = ((1 << i) - 1) << 1;
    for ( j = f2; j > 0; j-- )
    {
        k = f3;
        u = k * f1;
        ip = f0 - j + u;
        ic = f1 - j + u;
        z1 = z [ip];
        z2 = z [ic];
        for ( t = 0; t < n; t++, z2++ )
        {
            *z2 -= *z1;
            *z1 += *z1++;
        }
        while ( k > 0 )
        {
            k--;
            ip -= f1;
            ic -= f1;
            z1 = z [ip];
            z2 = z [ic];
            for ( t = 0; t < n; t++, z2++ )

```

```

        {
            *z2 -= *z1;
            *z1 += *z1++;
        }
    }
}

/* Pipelines along rows of the data matrix */
z1 = z [0];
for ( t = 0; t < n; t++, z1 = z [t] )
    for ( j = f2; j > 0; j-- )
    {
        k = f3;
        u = k * f1;
        ip = f0 - j + u;
        ic = f1 - j + u;
        z1 [ic] -= z1 [ip];
        z1 [ip] += z1 [ip];
        while ( k > 0 )
        {
            k--;
            ip -= f1;
            ic -= f1;
            z1 [ic] -= z1 [ip];
            z1 [ip] += z1 [ip];
        }
    }
}

/* The 2-D bit reversal permutation
----- */
for ( t = 0, z1 = z [0]; t < n; t++, z1 = z [t] )
    for ( i = 1; i < n1; i++ )
    {
        for ( k = j = 0, r = i; k < m; k++ )
        {
            s = r >> 1;
            j = j + j + r - s - s;
            r = s;
        }
        if ( i < j )
        {
            tmp = z1 [i];
            z1 [i] = z1 [j];
            z1 [j] = tmp;
        }
    }
for ( i = 1; i < n1; i++ )
{
    for ( k = j = 0, r = i; k < m; k++ )
    {
        s = r >> 1;
        j = j + j + r - s - s;
        r = s;
    }
    if ( i < j )
    {
        ptr = z [i];
        z [i] = z [j];
        z [j] = ptr;
    }
}

```

```

    }
/* Implementation of the 2-D Butterfly structure
----- */
b1 = tab1_len;
b2 = tab2_len;
for ( s = 0; s < m; s++ )
{
    half = 1 << (s + 1);
    ip   = half >> 1;
    b1 -= ip;
    b2 -= ip * ip;
/* Multiplications by cosine coefficients */
    for ( j = u = 0; j < ip; j++, u = j*ip )
        for ( k = j; k < n; k += half )
        {
            z1 = z [k];
            z2 = z [k+ip];
            for ( r = 0; r < ip; r++ )
                for ( t = r; t < n; t += half )
                {
                    z1 [t+ip] *= tc1 [b1+r];
                    z2 [t]    *= tc1 [b1+j];
                    z2 [t+ip] *= tc2 [b2+u+r];
                }
        }
/* Butterflies between rows of the data matrix */
        z1 = z [k];
        z2 = z [k+ip];
        for ( i = 0; i < n; i++ )
        {
            tmp    = *z2;
            *z2++ = *z1 - tmp;
            *z1++ = *z1 + tmp;
        }
}
/* Butterflies along rows of the data matrix */
z1 = z [0];
for ( i = 0; i < n; i++, z1 = z [i] )
    for ( j = 0; j < ip; j++ )
        for ( k = j; k < n; k += half )
        {
            tmp    = z1 [k+ip];
            z1 [k+ip] = z1 [k] - tmp;
            z1 [k]    = z1 [k] + tmp;
        }
}
/* Reordering and transposition of DCT/DST output
data matrix
----- */
for ( i = 0; i < n2; i++ )
    for ( j = 0; j < n2; j++ )
    {
        x [2*i] [2*j] = z [j] [i];
        x [2*i+1] [2*j+1] = z [n-j-1] [n-i-1];
        if ( flag == IDCT )
        {
            x [2*i] [2*j+1] = z [n-j-1] [i];
            x [2*i+1] [2*j] = z [j] [n-i-1];
        }
        else

```

```

        {
            x [2*i ] [2*j+1] = -z [n-j-1] [i ] ;
            x [2*i+1] [2*j ] = -z [j ] [n-i-1] ;
        }
    }
    return (0);
}
/*----- End of Fast 2-D DCT/DST module ----- */

```

4.5 DCT and Data Compression

The amount of information in its many forms (images, text, speech, video, audio, etc.) that is handled is increasing at a phenomenal rate. As a result, the ability to access, store, and transmit information in an efficient manner has become crucial, particularly in the case of digital images. Although representing images in digital form allows visual information to be easily manipulated in useful and novel ways, there is one potential problem with digital images — the large number of bits required to represent even a single digital image directly. In order to utilize digital images effectively, specific techniques are needed to reduce the number of bits required for their representation. Fortunately, digital images in their canonical representation generally contain a significant amount of redundancy (spatial, spectral, or temporal redundancy). Image data compression (the art/science of efficient coding of the picture data) aims at taking advantage of this redundancy to reduce the number of bits required to represent an image. This can result in significantly reducing the memory needed for image storage and channel capacity for image transmission [36].

The need for image compression becomes apparent when we compute the number of bits per image resulting from typical sampling and quantization schemes. We consider the amount of storage for the “Lena” digital image shown in Fig. 4.7. The monochrome (grayscale) version of this image with a resolution $512 \times 512 \times 8$ bits/pixel requires a total of 2,097,152 bits, or equivalently 262,144 bytes. The color version of the same image in RGB format (red, green, and blue color bands) with a resolution of 8 bits/color requires a total of 6,291,456 bits, or 786,432 bytes. Such an image should be compressed for efficient storage or transmission.

Image compression methods can be classified into two fundamental groups: lossless and lossy [34, 36, 37]. In lossless compression, the reconstructed image after compression is identical to the original image. However, only a modest amount of compression is possible; typically 1:2 or 1:3 compression ratios are achieved. In lossy compression, the reconstructed image contains degradations relative to the original. Generally, more compression is obtained at the expense of more distortion. As a result, much higher compression can be achieved by lossy techniques than by lossless techniques. The most used lossy compression technique is transform coding [32]. A general transform coding scheme involves subdividing an $N \times N$ image into smaller nonoverlapping $n \times n$ sub-image blocks and performing a unitary transform on each



FIGURE 4.7

Monochrome $512 \times 512 \times 8$ bits/pixel “Lena” digital image. Reproduced by Special Permission of *Playboy* magazine. Copyright ©1972, 2000 by Playboy.

block. The transform operation itself does not achieve any compression. It aims at decorrelating the original data and compacting a large fraction of the signal energy into a relatively small set of transform coefficients (energy packing property). In this way, many coefficients can be discarded after quantization and prior to encoding.

Most practical transform coding systems are based on DCT of types II and III, which provides good compromise between energy packing ability and computational complexity. The energy packing property of DCT is superior to that of any other unitary transform. Transforms that redistribute or pack the most information into the fewest coefficients provide the best sub-image approximations and, consequently, the smallest reconstruction errors. DCT basis images are fixed (image independent) as opposed to the optimal KLT which is data dependent. Moreover, when compared to the other image independent transforms, DCT has the advantages of having been implemented in a single integrated circuit [30] and minimizing the blocklike appearance (blocking artifact) that results when the boundaries between sub-image blocks become visible. This last property is particularly important in comparison with the other sinusoidal transforms [34]. Important properties of DCT have proved to be of practical value, and, therefore, it has become the basic processing unit for data compression in the international image/video coding standards [30, 31, 39, 40, 41, 42].

4.5.1 DCT-Based Image Compression/Decompression

For the purposes of using DCT in real data compression applications, we have selected the JPEG DCT-based image compression and decompression technique. There are several reasons for this selection. JPEG is the first established/emerging international digital compression standard for continuous-tone (multilevel) still images, both monochrome and color [31, 43, 44]. It has been recently recognized as the

most popular, simple, and efficient transform coding technique that yields a satisfactory solution to most of the practical image coding problems. Furthermore, the JPEG standard played a considerable role in the development of other international video coding standards. From the methodological viewpoint, the JPEG standard enables one to simply illustrate the compression capability of DCT. Finally, the JPEG DCT-based coding approach is the basis of hybrid intraframe/interframe MC (motion compensated)/DPCM (differential pulse code modulation)/DCT coding scheme used in the international video coding standards: H.261 video coder, MPEG-1 audiovisual coder for digital storage media, MPEG-2/H.262 digital video coder, MPEG-4 and H.263 coders for very low-bit rate video coding, digital HDTV standards, and the CMTT.723 digital broadcasting standard for transmission of television signals [31].

The JPEG standard specifies the basic encoding and decoding operations by means of specific functions and defines the syntax and semantics of encoded bit stream [31, 43, 44]. Detailed requirements such as file format, spatial resolution, and color space are not defined by the standard. It is only necessary that the encoding processes comply with the functions defined by the standard and they produce the valid bit stream. Thus, there is freedom and flexibility in the actual design and development of the JPEG compression and decompression system.

The JPEG standard has four main processing modes: sequential, progressive, lossless, and hierarchical. The sequential mode provides the variability of coding operations from a baseline system to an extended one. For simplicity, we consider the JPEG sequential baseline system. The extended system allows the baseline system to satisfy a broader range of applications. Input and output data precision in the baseline system is limited to 8 bits. RGB color images prior to compression are converted into a monochrome compatible luminance component and two chrominance components. The luminance component contains the shades of gray and is a monochrome image. Two chrominance components together contain the color information. Encoding/decoding operations in the JPEG baseline system are performed for luminance and chrominance components.

All compression systems consist of two distinct structural blocks: an encoder and a decoder. An input image is fed into the encoder, which creates encoded compressed representation of the input data. After transmission over the channel, the encoded representation is fed into the decoder, where the reconstructed output image is generated.

The block diagram of the encoder and decoder for JPEG DCT-based image compression and decompression is shown in Fig. 4.8. For processing the luminance component of an image the algorithm generally consists of the following steps [31, 34, 36, 43, 47]:

- The source image is partitioned into nonoverlapping $n \times n$ pixel blocks which are processed sequentially in a raster scan fashion, left to right and top to bottom. The JPEG standard uses the fixed block size 8×8 . Each block is first level shifted and transformed using DCT. In principle, DCT introduces no loss to the source samples, it merely transforms them to a domain in which they can be more efficiently encoded.

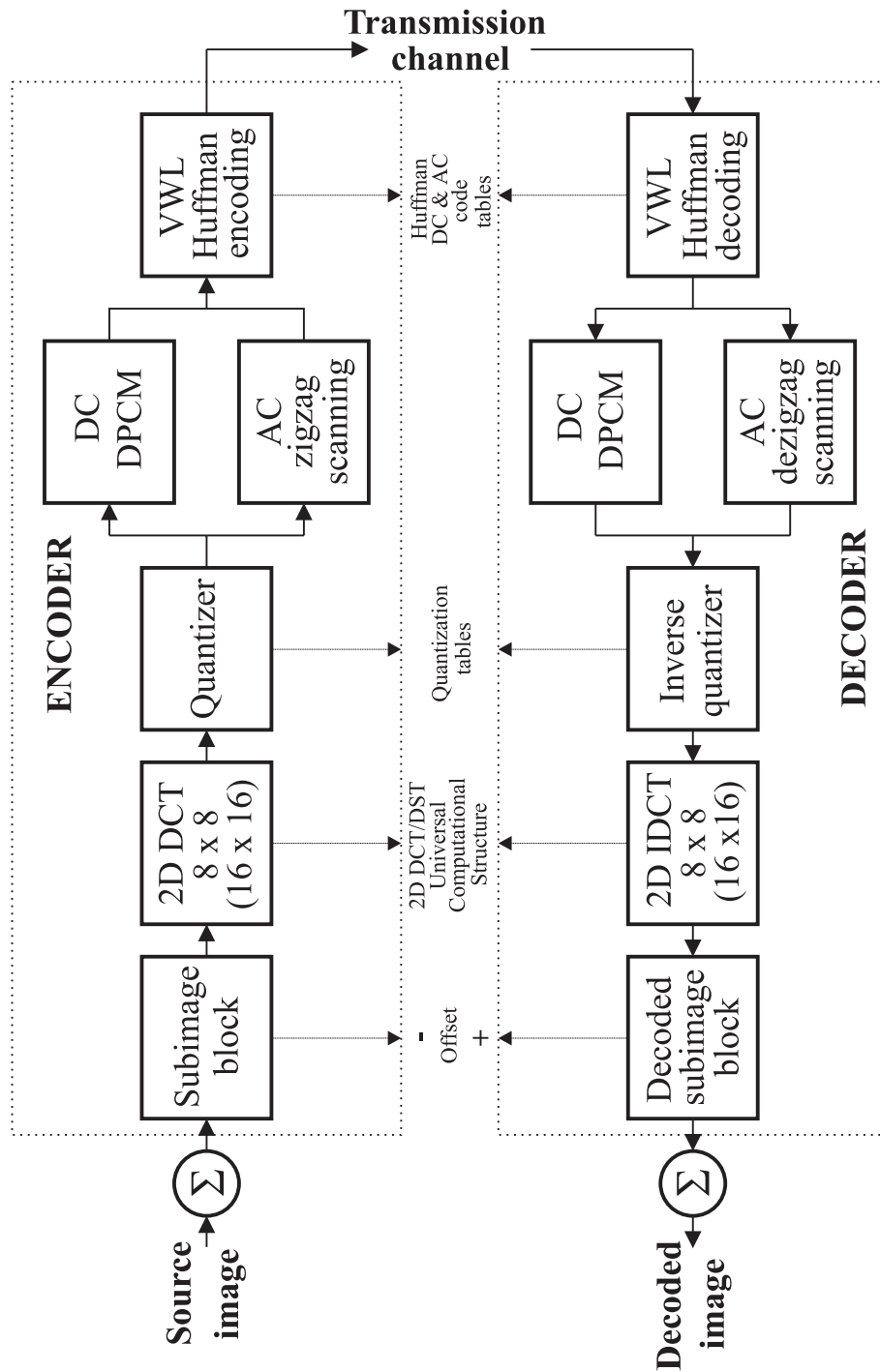


FIGURE 4.8
Block diagram of encoder and decoder for JPEG DCT-based image compression and decompression.

- The 2-D DCT array of coefficients is uniformly quantized. The top left coefficient in the 2-D DCT array with zero frequency in both dimensions is referred to as the *DC coefficient*, and it is proportional to the average brightness of the spatial block. The remaining coefficients are called the *AC coefficients*. Prior to quantization, transform coefficients can be weighted according to their visual importance using HVS (Human Visual System) sensitivity models [47, 48].
- The quantization of the AC coefficients produces many zeros, especially at the higher frequencies. To take advantage of these zeros, the 2-D DCT array of quantized coefficients is reordered using a zigzag pattern [see Fig. 4.9(a)] to form a 1-D sequence. This rearranges the coefficients in approximately decreasing order of their average energy (as well as in order of increasing spatial frequency) with the aim of creating large runs of zero values. The quantization is a key operation because the combination of the quantization and runlength coding contributes to most of the compression.
- The final processing step at the encoder is entropy coding. This step achieves additional compression losslessly by encoding the quantized coefficients more compactly based on their statistical characteristics. The quantized DCT coefficients are variable-length coded using two global different predetermined Huffman coding tables, one for DC and one for AC coefficients.

At the decoder, after the encoded bit stream is Huffman decoded and the 2-D array of quantized DCT coefficients is recovered and dezigzag reordered, each coefficient is inverse quantized. The resulting array is transformed by inverse 2-D DCT and inverse level shifted to yield an approximation of the original sub-image block. The same quantization table and Huffman coding tables are used in both the encoder and decoder.

Each chrominance component of a color image is processed and encoded independently in the same way as the luminance component, except that it is downsampled by a factor of two or four in both horizontal and vertical directions prior to DCT operation. At the decoder, the reconstructed chrominance component is bilinearly interpolated to the original size.

The following sections describe the JPEG DCT-based image compression and decompression system. The description is restricted to one sub-image block only because the same encoding and decoding operations are performed on each block. Although required algorithms in the JPEG standard are based on fixed block size (8×8), the system described in this chapter can use larger blocks. In fact, the 2-D DCT/DST universal computational structure offers the flexibility of computing the 2-D DCT and its inverse for any $2^m \times 2^m$ block size. The encoding and decoding operations are described in detail followed by an implementation in C. Where necessary, the input and output data samples are provided; they can be useful for verification of the correctness of a given program module. Low-level routines — setting quantization table, computation of Huffman coding/decoding tables, Huffman encoding and Huffman decoding — are based on shareware generated by Independent JPEG group (Thomas G. Lane) [49]. Program modules together provide the simple, efficient, and

low-cost image compression and decompression system which the reader can use in his or her own data compression applications.

4.5.2 Data Structures for Compression/Decompression

One of the most important aspects of image/video coding standards is to define data structures so that a decoder can decode the received bit stream efficiently and without any ambiguity. This section shows header files that contain definitions and declarations of data structures for an image compression and decompression system.

The header file JPEGDEF.H contains macro definitions and the definition of data structure for the Huffman coding/decoding table.

```

/*
 * JPEGDEF.H
 */
#define SIZE 16 /* max dimension of the block */
#define I_LEVEL 256 /* the number of gray levels */
#define DCT 1 /* 2-D DCT computation */
#define DISABLE_NORM 0 /* disable DCT normalization */
#define SQRT2 0.707106781186547 /* sqrt (2) */
#define LOOKAHEAD 8 /* # of bits of lookahead */
#define MIN_GET_BITS 15 /* minimum allowable value */
/* -----
/* Huffman coding and decoding table
/* -----
struct huff_table {
/* bits [k] = # of symbols with codes of length k bits,
/* bits [0] is unused
unsigned char bits [17];
/* Symbols in order of incremental code length
unsigned char hufval [256];
/* ENCODING TABLES
unsigned int hufcode [256]; /* code for each symbol
char hufsize [256]; /* and its length
/* DECODING TABLES
/* Basic tables: element [0] of each array is unused
long int mincode [17]; /* smallest code of length k
long int maxcode [18];
/* and largest code (-1 if none)
/* Index of 1st symbol of length k
int valptr [17];
/* Lookahead tables: indexed by the next
LOOKAHEAD bits of the input data stream. If the next
Huffman code is no more than LOOKAHEAD bits long, it
can be obtained its length and the corresponding
symbol directly from these tables
int look_nbits [1<<LOOKAHEAD];
/* # bits, or 0 if too long
unsigned int look_sym [1<<LOOKAHEAD];
/* symbol, or unused
};

```

The header file JPEGDATA.H contains declarations of variables and arrays for the image compression and decompression system. Declarations for JPEG luminance sample quantization table, zigzag, and dezigzag scanning patterns are shown for 8×8

block size only. For larger block sizes, the user must specify the corresponding arrays for a given block size. The JPEG DCT-based image compression and decompression system has two optional parameters: the block size and a quality factor for scaling the quantization table.

```

/*
 * JPEGDATA.H
 */
unsigned char out_buffer [256];
/* output bit stream buffer */
int bytes_in_buf;
/* and # of bytes in it */
int encode_bits;
/* # of bits for compressed block */
int exp_val; /* log2 value of block size */
int blk_size; /* block size */
int center_samp; /* center sample value */
int tdc_last; /* the last DC value for encoder */
int rdc_last; /* the last DC value for decoder */
int q_factor; /* quality factor */
long int total_bits;
/* total # of bits for original data */
long int total_bytes;
/* total # of bytes for original data */
long int cmprs_bits;
/* total # of bits for compressed data */
long int cmprs_bytes;
/* total # of bytes for compressed data */
double dct_block [SIZE*SIZE];
/* 2-D DCT block of coefficients */
double *dctptr [SIZE]; /* pointers to its rows */
double scaling; /* scale factor for DCT normalization */
double bit_rate; /* the # of bits per pixel (bpp) */
double cmprs_ratio; /* compression ratio */
/* # of symbols with codes of length k bits
(lumbits [k]) and symbols in order of incremental
code length (lumval [k]) for DC luminance
values - valid for 8-bit data precision */
unsigned char dc_lumbits [17] =
{0,0,1,5,1,1,1,1,1,0,0,0,0,0,0,0,0};
unsigned int dc_lumval [12]
= {0,1,2,3,4,5,6,7,8,9,10,11};
/* # of symbols with codes of length k bits
(lumbits [k]) and symbols in order of incremental
code length (lumval [k]) for AC luminance
values - valid for 8-bit data precision */
unsigned char ac_lumbits [17] =
{0,0,2,1,3,3,2,4,3,5,5,4,4,0,0,1,0x7d};
unsigned char ac_lumval [162] =
{
0x01, 0x02, 0x03, 0x00, 0x04, 0x11, 0x05, 0x12,
0x21, 0x31, 0x41, 0x06, 0x13, 0x51, 0x61, 0x07,
0x22, 0x71, 0x14, 0x32, 0x81, 0x91, 0xa1, 0x08,
0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52, 0xd1, 0xf0,
0x24, 0x33, 0x62, 0x72, 0x82, 0x09, 0x0a, 0x16,
0x17, 0x18, 0x19, 0x1a, 0x25, 0x26, 0x27, 0x28,
0x29, 0x2a, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39,
0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49,
0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59,

```

```

        0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69,
        0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79,
        0x7a, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
        0x8a, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98,
        0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
        0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6,
        0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3, 0xc4, 0xc5,
        0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2, 0xd3, 0xd4,
        0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xe1, 0xe2,
        0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9, 0xea,
        0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
        0xf9, 0xfa };
struct huff_table dc_table; /* Huffman DC code table */
struct huff_table ac_table; /* Huffman AC code table */
/* luminance sample quantization table for 8 x 8 DCT */
int qbase8_tbl [8*8] =
{
    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, 59, 64, 81, 104, 113, 92,
    49, 64, 78, 87, 103, 121, 120, 101,
    72, 92, 95, 98, 112, 100, 103, 99 };
/* zigzag scanning pattern for an 8 x 8 DCT transform */
int zag8 [8*8] =
{
    0, 1, 5, 6, 14, 15, 27, 28,
    2, 4, 7, 13, 16, 26, 29, 42,
    3, 8, 12, 17, 25, 30, 41, 43,
    9, 11, 18, 24, 31, 40, 44, 53,
    10, 19, 23, 32, 39, 45, 52, 54,
    20, 22, 33, 38, 46, 51, 55, 60,
    21, 34, 37, 47, 50, 56, 59, 61,
    35, 36, 48, 49, 57, 58, 62, 63 };
/* dezagzag scanning pattern for an
8 x 8 DCT transform */
int dezag8 [8*8] =
{
    0, 1, 8, 16, 9, 2, 3, 10,
    17, 24, 32, 25, 18, 11, 4, 5,
    12, 19, 26, 33, 40, 48, 41, 34,
    27, 20, 13, 6, 7, 14, 21, 28,
    35, 42, 49, 56, 57, 50, 43, 36,
    29, 22, 15, 23, 30, 37, 44, 51,
    58, 59, 52, 45, 38, 31, 39, 46,
    53, 60, 61, 54, 47, 55, 62, 63 };

```

4.5.3 Setting the Quantization Table

JPEG gives simple and easy quantization methods and suggests informative tables for DC and AC coefficients [31]. One such informative quantization table for the luminance component is shown in the header file JPEGDATA.H. Although default quantization tables are provided by the JPEG standard for both luminance and chrominance processing, the user is free to design custom tables which can be adapted to the characteristics of the image to be compressed.

The quantization of the DCT coefficients is based on properties of the HVS which tolerates more quantization errors at higher frequencies than at lower frequencies. It means that the transform coefficients have different visual sensitivities; visual per-

ception is less sensitive to the high frequency coefficients and more sensitive to low frequency coefficients. Thus, the weighting factors are selected to produce coarser quantization of high frequency coefficients and finer quantization of the low frequency coefficients.

The quantization table can be scaled to provide a variety of compression levels. JPEG specifies the following possible bit rates and quality rates [31]:

- 0.25 ~ 0.50 bpp: moderate to good quality
- 0.50 ~ 0.75 bpp: good to very good quality
- 0.75 ~ 1.50 bpp: excellent images
- 1.50 ~ 2.00 bpp: indistinguishable images (visually lossless)

The quantization table in the JPEG DCT-based image compression and decompression system is scaled according to a specified quality factor. The quality factor takes values in the range 0–100 (given as percentage) with the scaling value of 50 corresponding to the basic quantization table. The value of 100 will cause elements of the quantization table to be equal to 1 for an 8×8 block size and to equal to 2 for a 16×16 block size. The elements of the quantization table are in the range from 1 to 255.

The following program sets the user quantization table according to the specified quality factor.

```
/*-----
SET USER QUANTIZATION TABLE ACCORDING TO DEFINED
'QUALITY'
Set a quantization table equal to the basic table times
a scale factor (given as a percentage). The basic table
is used as-is (scaling 100) for a quality of 50. Values
of the basic table produce "good" quality, and when
divided by 2, "very good" quality. These two settings
are selected by quality = 50 and quality = 75,
respectively. Qualities 50 ... 100 are converted to
scaling percentage 200 - 2*Q. Note that at Q = 100 the
scaling is 0, which will cause qnt_tbl to make all the
table entries 1 (no quantization loss).
*/
#include "jpegdef.h"
void set_qtable (
    int *qnt_tbl, /* user quantization table */
    int blksize, /* block size */
    int *qbase_tbl, /* basic quantization table */
    int quality ) /* quality factor */
{
    int i;
    long int temp;
    /* Safety limit on quality factor (convert 0 to 1 to
    avoid zero divide) */
    if ( quality <= 0 )
        quality = 1;
    else
        if ( quality > 100 )
```

```

        quality = 100;
/* Convert a user-specified quality rating 0-100 to a
percentage scaling factor. Qualities 1 ... 50 are
converted to scaling percentage 5000/Q */
if ( quality < 50 )
    quality = 5000 / quality;
else
    quality = 200 - quality * 2;
/* Set quantization table equal to the qbasic tbl
times a scale factor. Limit the values to the
valid range */
for ( i = 0; i < blksize * blksize; i++ )
{
    temp = ((long int) qbase_tbl [i]
        * quality + 50L) / 100L;
    if ( temp <= 0L )
    {
        temp = 1L;
        if ( blksize == SIZE )
            temp = 2L;
    }
    if ( temp > 255L )
        temp = 255L;
    qnt_tbl [i] = (int) temp;
}
}

```

4.5.4 Standard Huffman Coding/Decoding Tables

The JPEG baseline system uses only the Huffman coding method for encoding the quantized DCT coefficients, and it suggests standard Huffman coding tables for the luminance and chrominance DCT coefficients, two DC and two AC Huffman coding tables [31].

Based on data structures defined in the header file JPEGDATA.H for DC and AC luminance values (structures specifying the number of symbols with codes of length k bits and code symbols), the following program generates standard Huffman coding/decoding tables. The program must be called separately for the DC and AC coding tables (see Section 4.5.7). These DC and AC Huffman coding/decoding tables are valid for 8-bit data precision and can be found in Rao and Hwang [31].

```

/*-----
----- COMPUTE HUFFMAN CODING AND DECODING TABLES -----
*/
#include <string.h>
#include "jpegdef.h"
void fix_huftbl (
    struct huff_table *htbl ) /* Huffman code table */
{
    int p,i,j,k,lastp,size,lookbits;
    char huffsize [257];
    unsigned int huffcode [257],code;
/* Make table of Huffman code length for each symbol
in code-length order */
    for ( k = 1, p = 0; k <= 16; k++ )

```

```

        for ( i = 1; i <= (int) htbl->bits [k]; i++ )
            huffsize [p++] = (char) k;
        huffsize [p] = 0;
        lastp = p;
/* Generate the codes themselves in code-length order */
        code = p = 0;
        size = huffsize [0];
        while ( huffsize [p] )
        {
            while ( ((int) huffsize [p]) == size )
            {
                huffcode [p++] = code;
                code++;
            }
            code <<= 1;
            size++;
        }
/* Generate encoding tables. These are code and size
indexed by symbol value. Set any codeless symbols
to have code length 0. This allows emit_bits () to
detect any attempt to emit such symbols */
        memset (htbl->hufsize,0,sizeof (htbl->hufsize));
        for ( p = 0; p < lastp; p++ )
        {
            htbl->hufcode [htbl->hufval [p]] = huffcode [p];
            htbl->hufsize [htbl->hufval [p]] = huffsize [p];
        }
/* Generate decoding tables for bit-sequential
decoding */
        for ( k = 1, p = 0; k <= 16; k++ )
        {
            if ( htbl->bits [k] )
            {
                htbl->valptr [k] = p;
                htbl->mincode [k] = huffcode [p]; /* min code */
                p += htbl->bits [k];
                htbl->maxcode [k] = huffcode [p-1]; /* max code */
            }
            else
                htbl->maxcode [k] = -1; /* -1 if no codes */
        }
/* Ensures that huff decode () terminates */
        htbl->maxcode [17] = 0xFFFFFL;
/* Compute lookahead tables to speed up decoding.
First set all the table entries to 0, indicating
"too long"; then iterate through the Huffman codes
that are short enough and fill in all the entries
that correspond to bit sequences starting with that
code; k = current code's length, p = its index in
hufcode [] & hufval []. Generate left-justified code
followed by all possible bit sequences */
        memset (htbl->look_nbits,0,sizeof (htbl->look_nbits));
        for ( k = 1, p = 0; k <= LOOKAHEAD; k++ )
            for ( i = 1; i <= (int) htbl->bits [k]; i++, p++ )
            {
                lookbits = huffcode [p] << (LOOKAHEAD - k);
                for ( j = 1 << (LOOKAHEAD - k); j > 0; j-- )
                {
                    htbl->look_nbits [lookbits] = k;
                    htbl->look_sym [lookbits] = htbl->hufval [p];
                }
            }

```



```

    }
    }
    lookbits++;
}

```

4.5.5 Compression of One Sub-Image Block

Having defined and prepared all required data structures, we can concentrate on the image compression process. For simplicity, we consider the compression of one sub-image block because the same operations are performed for each extracted block from the source image. For processing the luminance component of the image, the following steps are performed at the encoder for each block.

1. The data in the block is first level shifted by subtracting the quantity 2^{p-1} , where 2^p is the maximum number of gray levels and p is the precision parameter of the image intensity in bits. In the JPEG baseline system, $p = 8$ and the level shift is 128.
2. The level-shifted block is transformed by the forward 2-D DCT.
3. The 2-D DCT array of coefficients is uniformly quantized by rounding to the nearest integer. Specifically, the quantized DCT coefficients, \bar{C}_{uv} , are defined by the following equation:

$$\bar{C}_{uv} = \text{nearest integer} \left(\frac{C_{uv}}{Q_{uv}} \right), \quad (4.55)$$

where C_{uv} is the DCT coefficient and Q_{uv} is the corresponding element in the quantization table.

4. The 2-D array of quantized DCT coefficients is scanned and formatted into a 1-D sequence using the zigzag pattern shown in Fig. 4.9(a). The DC coefficient is sensitive to spatial frequency response of the HVS and is treated separately from the remaining AC coefficients. Prior to encoding, the DC coefficient is differenced by the following first-order prediction:

$$DIFF = DC_i - DC_{i-1}, \quad (4.56)$$

where DC_i and DC_{i-1} are DC coefficients in the current and previous blocks, respectively. The initial starting DC value at the beginning of the image is set to zero.

We note that in international image/video coding standards two scan methods of quantized DCT coefficients are used: the zigzag scan [Fig. 4.9(a)] which is typical for progressive (noninterlaced) mode processing (in JPEG, MPEG-1, and H.261 standards) and alternate scan [Fig. 4.9(b)] which is more efficient for interlaced video format (adopted in MPEG-2 and HDTV standards). The structure of an alternate scan seems like a vertical scan since the correlation along the horizontal direction is higher than along the vertical direction [31].

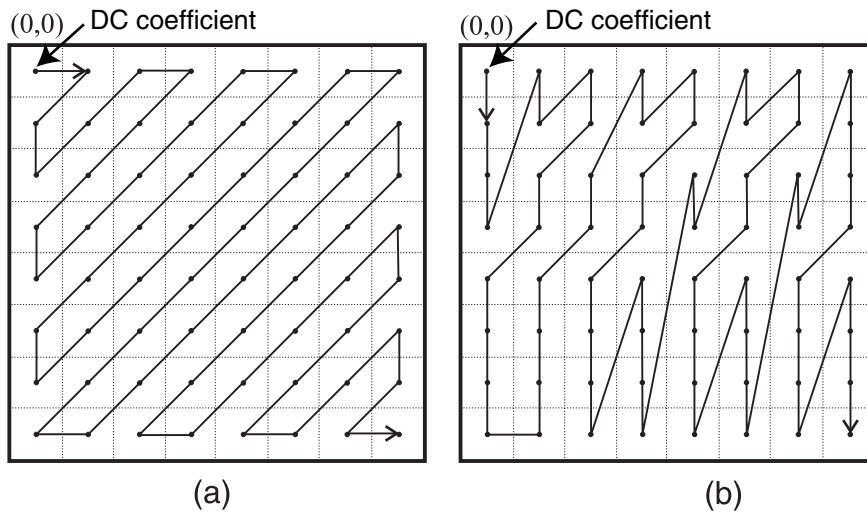


FIGURE 4.9
Scanning patterns of quantized DCT coefficients: (a) zigzag scan; (b) alternate scan.

The following program compresses one sub-image block according to steps described previously.

```

/*-----
      COMPRESSION OF ONE SUB-IMAGE BLOCK
Level shifting, forward 2-D DCT, quantization, zigzag
reordering and Huffman encoding the quantized
coefficients.
-----*/
#include      "jpegdef.h"
extern int    exp_val;      /* log2 value of block size*/
extern int    center_samp; /* center sample value */
extern double scaling; /* scaling for DCT normalization*/
extern int    tdc_last; /*the last DC value for encoder*/
extern int    encode_bits;
              /* # of bits for compressed block */

void cmprs_blk (
    int          *qnt_blk,
    /* input/quantized data block */
    int          blksize,
    /* block size */
    int          *qnt_tbl,
    /* user quantization table */
    int          *zigzag, /* zigzag pattern */
    double       **dctb, /* 2-D DCT block */
    struct huff_table *dctbl, /* DC Huffman code table */
    struct huff_table *actbl /* AC Huffman code table */
)
{
    int i,j,k,temp,*q_ptr;

```

```

        double coef,*dctptr;
/* Level shift of samples in the sub-image block */
for ( i = 0, q_ptr = qnt_blk; i < blksize; i++ )
    for ( j = 0, dctptr = dctb [i]; j < blksize; j++ )
        *dctptr++ = (double) (*q_ptr++ - center_samp);
/* Perform forward 2-D DCT computation and
normalization of transform coefficients */
fdct2d (dctb,exp_val,DISABLE_NORM,DCT);
for ( i = 0, dctb [0] [0] *= SQRT2; i < blksize; i++ )
    for ( j = 0; j < blksize; j++ )
    {
        dctb [i] [j] *= scaling;
        if ( (i == 0) || (j == 0) )
            dctb [i] [j] *= SQRT2;
    }
/* Quantization of the transform DCT coefficients
and zigzag reordering */
for ( i = k = 0; i < blksize; i++ )
    for ( j = 0, dctptr = dctb [i]; j < blksize; j++ )
        if ( (coef = *dctptr++ / *qnt_tbl++) > 0.0 )
            qnt_blk [zigzag [k++]] = (int) (coef + 0.5);
        else
            qnt_blk [zigzag [k++]] = (int) (coef - 0.5);
/* Huffman encoding the quantized coefficients. The DC
coefficient is converted to a difference value */
temp = qnt_blk [0];
qnt_blk [0] -= tdc_last;
tdc_last = temp;
encode_bits = encode_blk(qnt_blk, blksize,
                        dctb1,actb1);
}

```

As an example, the following 8×8 data block is selected from the “Lena” digital image [31]:

```

79 75 79 82 82 86 94 94
76 78 76 82 83 86 85 94
72 75 67 78 80 78 74 82
74 76 75 75 86 80 81 79
73 70 75 67 78 78 79 85
69 63 68 69 75 78 82 80
76 76 71 71 67 79 80 83
72 77 78 69 75 75 78 78

```

After level shifting, this block transformed by the forward 2-D 8×8 DCT is given by

```

-404.375 -29.971  8.623  1.909  1.625 -3.936  0.893  1.516
 23.226  -7.184 -4.327 -0.438  7.346  0.010 -2.266 -3.186
 11.798  -0.278  5.197 -4.772 -3.572  4.160 -0.261 -3.507
  2.299 -10.742  5.495  0.791 -1.029  7.603  3.791  2.820
  6.375   2.511 -1.549 -1.074 -3.625 -0.797  0.506  8.723
  0.739   2.612  0.717  2.530 -0.926  3.206 -2.945 -2.792

```

-9.081	-1.660	-4.511	1.743	2.156	1.549	-1.697	2.055
-3.626	2.241	5.355	-1.960	0.899	-1.370	1.828	-3.314

By applying the basic luminance quantization table (quality factor is equal to 50), the 2-D array of quantized coefficients is

-25	-3	1	0	0	0	0	0
2	-1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	-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	0
0	0	0	0	0	0	0	0

Assuming that the quantized DC coefficient of the previous block is 34, the DC differencing and the reordering 2-D array of quantized coefficients into a 1-D sequence based on zigzag scan result in

-59 -3 2 1 -1 1 0 0 0 0 0 -1 EOB

The 1-D sequence of quantized DCT coefficients is prepared for Huffman encoding. The encoder employs one DC and one AC Huffman table lookups for luminance DCT coefficients. All codes consist of a set of Huffman codes with a maximum length of 16 bits followed by appended additional bits for representing the exact value of the coefficient.

Coding the DC and AC Coefficients

The DIFF values as defined by Eq. (4.56) are classified into 12 categories, each category written by two's complement expression. A Huffman DC coding/decoding table is generated for each category. The difference values in category k are in the range $< -2^k + 1, 2^k - 1 >$, where $0 \leq k \leq 11$. Thus, k denotes the number of bits needed for the magnitude of the coefficient. In the case of $k = 0$ (DIFF = 0), the current DC coefficient is the same as the previous DC coefficient, and additional bits are not required. For the other categories, extra bits are needed to express the exact value in the category, consisting of the sign and magnitude of the DIFF value. When DIFF is positive, the sign bit is 1 and k low-order bits of DIFF are appended to the Huffman code. When DIFF is negative, the sign bit is 0 and k low-order bits of (DIFF-1) are appended to the Huffman code. A (DIFF-1) operation implies one's complement representation to avoid all 1 bits of two's complement operation. This procedure for appending the additional bits is also applied to encoding the AC coefficients.

To encode the AC coefficients, each nonzero coefficient is first described by a composite 8-bit value of the form "RRRRSSSS" in binary notation. The Huffman AC coding/decoding table is generated for each composite value. The four least significant bits, "SSSS," define a category for the coefficient magnitude. The values

in category k are in the range $< -2^k + 1, 2^k - 1 >$, where $1 \leq k \leq 10$ resulting in 10 categories. The four most significant bits in the composite value, "RRRR," give the position of the current coefficient relative to the previous nonzero coefficient, i.e., the runlength of zero coefficients between successive nonzero coefficients. The runlengths specified by "RRRR" can range from 0 to 15, and a separate symbol "11110000" (11-bits ZRL code = 11111111001) is defined to represent a runlength of 16 zero coefficients. If the runlength is greater than 16, it is coded by using multiple symbols. In addition, if all remaining coefficients in the block are zero, a special symbol "00000000" is used to code the end of block (4-bits EOB code = 1010).

By the following program, the 1-D sequence of quantized coefficients is Huffman encoded. The result is stored in the output bit stream buffer.

```

/*-----
HUFFMAN ENTROPY ENCODING ROUTINES
-----*/
#include "jpegdef.h"
extern unsigned char out_buffer [];
extern int bytes_in_buf;
/* bit stream buffer */
/* # of bytes in it */
static long int hufput_buf = 0L;
/* bit accumulator buffer */
static int hufput_bits = 0;
/* # of bits in buffer */
static void emit_bits (unsigned int, int);
/*-----
ENCODE A SINGLE BLOCK OF COEFFICIENTS
It is assumed that DC coefficient in a block was
converted to a difference value. Function returns the
total number of bits for encoded block of
coefficients.-----*/
int encode_blk (
    int *block, /* quantized data block */
    int blksize, /* block size */
    struct huff_table *dctbl, /* DC Huffman code table */
    struct huff_table *actbl) /* AC Huffman code table */
{
    int i,k,nbits,run,temp,temp2,num_bits = 0;
/*-----
ENCODE THE DC COEFFICIENT
-----*/
    if ( (temp = temp2 = block [0]) < 0 )
    {
        temp = -temp; /* abs value of input */
        temp2--; /* negative value is bitwise complement */
    }
/* Find the number of bits for magnitude of the
coefficient */
    nbits = 0;
    while ( temp )

```

```

    {
        nbits++;
        temp >>= 1;
    }
/* Emit the Huffman coded symbol for the number
of bits */
emit_bits (dctbl->hufcode [nbits],
           dctbl->hufsize [nbits]);
num_bits += dctbl->hufsize [nbits];
/* Emit the number of bits of the coefficient value
(positive value) or complement of its magnitude
(negative value). Reject if nbits = 0 */
if ( nbits )
{
    emit_bits ((unsigned int) temp2,nbits);
    num_bits += nbits;
}
/*
=====
ENCORE THE AC COEFFICIENTS
=====
*/
for ( k = 1, run = 0; k < blksize * blksize; k++ )
{
    if ( (temp = block [k]) == 0 )
        run++;
    else
    {
/* If run length > 15 then emit special run-length
codes (0xF0) */
        while ( run > 15 )
        {
            emit_bits (actbl->hufcode [0xF0],
                       actbl->hufsize [0xF0]);
            num_bits += actbl->hufsize [0xF0];
            run -= 16;
        }
        if ( (temp2 = temp) < 0 )
        {
            temp = -temp;
            temp2--;
        }
/* Find the number of bits needed for the magnitude of
the coefficient. The number of bits must be at least
1 bit */
        nbits = 1;
        while ( temp >>= 1 )
            nbits++;
/* Emit the Huffman symbol for
(run length / number of bits) */
        i = (run << 4) + nbits;
        emit_bits (actbl->hufcode [i],
                   actbl->hufsize [i]);
        num_bits += actbl->hufsize [i];
/* Emit the number of bits of the coefficient value
(positive value) or complement of its magnitude
(negative value) */
        emit_bits ((unsigned int) temp2,nbits);
        num_bits += nbits;
        run = 0;
    }
}

```

```

    }
/* If the last coefficients were zero, emit EOB code */
if ( run > 0 )
{
    emit_bits (actbl->hufcode [0],
               actbl->hufsize [0]);
    num_bits += actbl->hufsize [0];
}
/* Fill any partial byte with ones and reset
bit-buffer */
emit_bits (0x7F,7);
hufput_buf = 0L;
hufput_bits = 0;
return (num_bits);
}
/*
-----
OUTPUT HUFFMAN COMPRESSED COEFFICIENTS
Only the right 24 bits of hufput_buf are used.
The valid bits are left justified. At most 16 bits
can be passed to emit_bits () in one call and is
never retained more than 7 bits in accumulator buffer
between calls.-----
*/
static void emit_bits (
    unsigned int code,
    int size )
{
    long int put_buffer = code;
    int put_bits = hufput_bits, byte;
/* Mask off excess bits in put_buffer */
put_buffer &= (((long int) 1) << size) - 1;
put_bits += size; /* new # of bits in buffer */
put_buffer <= 24 - put_bits; /* align incoming bits */
put_buffer |= hufput_buf; /* merge with old buffer */
/* Load byte into output bit stream buffer and count
the number of bytes. Update bit accumulator buffer */
while ( put_bits >= 8 )
{
    byte = (unsigned int) ((put_buffer >> 16) & 0xFF);
    out_buffer [bytes_in_buf++] = (unsigned char) byte;
    put_buffer <= 8;
    put_bits -= 8;
}
hufput_buf = put_buffer;
hufput_bits = put_bits;
}

```

For our example of 1-D sequence of the quantized DCT coefficients, the program generates the following output-encoded bit stream (last unused bits are set to 1):

```

The number of bits      39 (5 bytes)
Bit stream buffer (hex) E1 11 88 3E 95
11100001 00010001 10001000 00111110 10010101/1

```

4.5.6 Decompression of One Sub-Image Block

At the decoder (see Fig. 4.8) for each sub-image block, the inverse operations of the encoder are followed but in reverse order. The quantization table and Huffman coding/decoding tables are the same at both the encoder and decoder.

Each of the Huffman codes is uniquely defined and the quantized DCT coefficients are decoded by the Huffman decoding procedure. The DC coefficient is reconstructed from the differential value. The initial starting DC value at the beginning is set to zero. The reconstructed 1-D sequence of quantized coefficients is dezigzag reordered to form a 2-D array. Each DCT coefficient, \bar{C}_{uv} , in the 2-D array is inverse quantized by multiplying it by the corresponding element of the quantization table as follows:

$$\hat{C}_{uv} = \bar{C}_{uv} \cdot Q_{uv} . \quad (4.57)$$

The resulting array is transformed by the inverse 2-D DCT. Inverse level shift restores the samples in the original block to the unsigned 8-bit representation.

With the following program, the sub-image block is reconstructed from the encoded bit stream.

```
/*-----
  DECOMPRESSION OF ONE SUB-IMAGE BLOCK
  Huffman decoding, inverse quantization, inverse 2-D DCT,
  and reconstruction of the original sub-image block.
  */
#include      "jpegdef.h"
extern int    exp_val; /* log2 value of block size */
extern int    center_samp; /* center sample value */
extern double scaling; /* scaling for DCT normalization */
extern int    rdc_last; /* last DC value for decoder */
void decmprs_blk (
    int          *qnt_blk,
                /* quantized/output data block */
    int          blksize, /* block size */
    int          *qnt_tbl,
                /* user quantization table */
    int          *dezigzag, /* dezigzag pattern */
    double       **dctb, /* 2-D IDCT block */
    struct huff_table *dctbl, /* DC Huffman code table */
    struct huff_table *actbl) /* AC Huffman code table */
{
    int    i,j,k,*q_ptr;
    double pixel,*dctptr;
    /* Huffman decoding the quantized coefficients and
       dezigzag ordering. Convert DC difference to actual
       value and update the last DC value */
    decode_blk (qnt_blk,blksize,dezigzag,dctbl,actbl);
    qnt_blk[0] += rdc_last;
    rdc_last = qnt_blk[0];
    /* Inverse quantization of the coefficients */
    for ( i = k = 0; i < blksize; i++ )
        for ( j = 0, dctptr = dctb[i]; j < blksize; j++ )
            *dctptr++ = (double) (qnt_blk[k++] * *qnt_tbl++);
    /* Perform denormalization and inverse 2-D DCT
```



```

        computation */
        for ( i = 0, dctb [0] [0] *= SQRT2; i < blksize; i++ )
            for ( j = 0; j < blksize; j++ )
            {
                dctb [i] [j] *= scaling;
                if ( (i == 0) || (j == 0) )
                    dctb [i] [j] *= SQRT2;
            }
        fdct2d (dctb,exp_val,DISABLE_NORM,-DCT);
/* Reconstruction of the original sub-image block */
        for ( i = 0, q_ptr = qnt_blk; i < blksize; i++ )
        {
            dctptr = dctb [i];
            for ( j = 0; j < blksize; j++, q_ptr++ )
                if ( (pixel = *dctptr++ + center_samp) > 0.0 )
                {
                    if ( (*q_ptr = (int) (pixel + 0.5))
                        > I_LEVEL - 1 )
                        *q_ptr = I_LEVEL - 1;
                }
                else
                    *q_ptr = 0;
        }
    }
}

```

For our example the inverse quantized block is

400	-33	10	0	0	0	0	0
24	-12	0	0	0	0	0	0
14	0	0	0	0	0	0	0
0	-17	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
0	0	0	0	0	0	0	0

This 2-D array transformed by the inverse 2-D 8×8 DCT is given by

-53.992	-53.111	-51.068	-47.587	-42.784	-37.390	-32.641	-29.846
-51.247	-51.084	-50.368	-48.621	-45.696	-42.036	-38.614	-36.537
-50.225	-50.684	-51.118	-50.873	-49.573	-47.415	-45.143	-43.689
-53.805	-54.156	-54.390	-53.884	-52.300	-49.881	-47.408	-45.846
-58.944	-58.765	-58.018	-56.232	-53.263	-49.563	-46.111	-44.018
-59.846	-59.558	-58.611	-56.564	-53.311	-49.350	-45.697	-43.496
-55.036	-55.370	-55.573	-55.027	-53.401	-50.941	-48.438	-46.859
-49.611	-50.664	-52.194	-53.382	-53.633	-52.908	-51.732	-50.872

and after inverse level shift the reconstructed sub-image block is (for easy comparison the original sub-image block is also given)

74	75	77	80	85	91	95	98	79	75	79	82	82	86	94	94
77	77	78	79	82	86	89	91	76	78	76	82	83	86	85	94
78	77	77	77	78	81	83	84	72	75	67	78	80	78	74	82

74	74	74	74	76	78	81	82	74	76	75	75	86	80	81	79
69	69	70	72	75	78	82	84	73	70	75	67	78	78	79	85
68	68	69	71	75	79	82	85	69	63	68	69	75	78	82	80
73	73	72	73	75	77	80	81	76	76	71	71	67	79	80	83
78	77	76	75	74	75	76	77	72	77	78	69	75	75	78	78

The following program module contains routines for Huffman decoding the quantized DCT coefficients from the encoded bit stream.

```

/* -----
   HUFFMAN ENTROPY DECODING ROUTINES
----- */
#include "jpegdef.h"
extern unsigned char out_buffer [];
static unsigned char *out_buf; /* bit stream buffer */
static long int get_buffer = 0L; /* and pointer to it */
static int bits_left = 0; /* bit-extraction buffer */
static void fill_buf (int); /* # of unused bits */
static int huff_decode (struct huff_table *);
static int slow_decode (struct huff_table *, int);
/*
   *****
   DECODE A SINGLE BLOCK OF COEFFICIENTS
   Data block for the coefficients should be zeroed
   before. Output coefficients are in dezigzagged
   (natural) order.
   *****
*/
void decode_blk (
    int *block, /* decoded block */
    int blksize, /* block size */
    int *dezigzag, /* dezigzag pattern */
    struct huff_table *dctbl, /* DC Huffman code table */
    struct huff_table *actbl) /* AC Huffman code table */
{
    int k,s,r;
    out_buf = out_buffer;
/*
   =====
   DECODE THE DC COEFFICIENT
   Extract Huffman symbol from input bit stream and
   get the number of bits of DC coefficient difference.
   Extract bits of the DC coefficient difference and
   extend sign.
   =====
*/
    if ( s = huff_decode (dctbl) )
    {
        if ( bits_left < s )
            fill_buf (s);
        bits_left -= s;
        r = (int) ((get_buffer >> bits_left))
            & ((1 << s) - 1);
        s = ( r < (1 << (s - 1)) ) ? r
            + ((-1 << s) + 1) : r;
    }
    block [0] = s;

```

```

/* =====
                        DECODE THE AC COEFFICIENTS
Extract Huffman symbol from input bit stream and
get value of (run length / number of bits).
===== */
for ( k = 1; k < blksize * blksize; k++ )
{
    s = huff_decode (actbl);
    r = s >> 4;
    if ( s &= 15 )
    {
        k += r;
/* Extract bits of AC coefficient magnitude and
extend sign */
        if ( bits_left < s )
            fill_buf (s);
        bits_left -= s;
        r = (int) ((get_buffer >> bits_left))
            & ((1 << s) - 1);
        s = ( r < (1 << (s - 1)) ) ? r
            + ((-1 << s) + 1) : r;
        block [dezigzag [k]] = s;
    }
/* The code EOB was detected - the last coefficients
are zeros */
    else
    {
        if ( r != 15 )
            break;
        k += 15;
    }
}
/* Reset bit-extraction buffer to empty */
get_buffer = 0L;
bits_left = 0;
}
/* =====
LOAD UP THE BIT BUFFER TO A DEPTH OF AT LEAST nbits
Source bytes are read into get buffer and bits are
doled out as needed. If get_buffer already contains
enough bits, they are fetched in-line. When there
are not enough bits, fill_buf () is called.
===== */
static void fill_buf (
    int nbits )
{
    int c;
/* Attempt to load at least MIN_GET_BITS into
get_buffer */
    while ( bits_left < MIN_GET_BITS )
    {
/* There are enough bits still left in get_buffer */
        if ( nbits > 0 && bits_left >= nbits )
            break;
/* Load byte from input bit stream buffer into
get_buffer */
        c = *out_buf++;
    }
}

```

```

        get_buffer = (get_buffer << 8) | c;
        bits_left += 8;
    }
}
/*
-----
EXTRACT NEXT HUFFMAN-CODED SYMBOL FROM INPUT BIT STREAM
Lookahead table is used to process codes of up to
LOOKAHEAD bits without looping. Usually, more than 95%
of the Huffman codes will be 8 or fewer bits long. The
few overlength codes are handled with a loop.
-----
*/
static int huff_decode (
    struct huff_table *htbl )
{
    int nb,look,result,b = LOOKAHEAD;
/* 1.The first if-test is coded to call fill_buf ()
   only when necessary.
   2.If the lookahead succeeds, is needed only
   decrement bits_left to remove the proper number
   of bits from get_buffer.
   3.If the lookahead table contains no entry, the
   next code must be more than LOOKAHEAD bits long */
    if ( bits_left >= LOOKAHEAD ||
        (fill_buf (0),bits_left >= LOOKAHEAD) )
    {
        nb = bits_left - b;
        look = (int) ((get_buffer >> nb)) & ((1 << b) - 1);
        if ( (nb = htbl->look_nbits [look]) != 0 )
        {
            bits_left -= nb;
            result = htbl->look_sym [look];
        }
        else
            result = slow_decode (htbl,LOOKAHEAD+1);
    }
    else
        result = slow_decode (htbl,1);
    return (result);
}

static int slow_decode (
    struct huff_table *htbl,
    int min_bits )
{
    int k = min_bits,rs;
    long int code;
/* huff_decode () has determined that the code is at
   least min_bits long, so fetch that many bits in one
   swoop */
    if ( bits_left < k )
        fill_buf (k);
    bits_left -= k;
    code = (int) ((get_buffer >> bits_left))
           & ((1 << k) - 1);
/* Collect the rest of the Huffman code one bit at
   a time */
    while ( code > htbl->maxcode [k] )
    {
        code <<= 1;
    }
}

```

```

        if ( bits_left < 1 )
            fill_buf (1);
        code |= (int) ((get_buffer >> (--bits_left))) & 1;
        k++;
    }
    rs = htbl->valptr [k] + (int)
        (code - htbl->mincode [k]);
    return (htbl->hufval [rs]);
}

```

4.5.7 Image Compression/Decompression

This section shows a sample program for compression and decompression of an image. It performs all described steps of the JPEG DCT-based coding technique for image compression and decompression. One extracted sub-image block is first compressed, immediately decompressed, and displayed on the screen. The displaying routine is not shown. If any dimension of the processed image is not a multiple of the block size, the remaining elements in the block are set to zeros. These additional elements are removed during decompression. No file for the compressed image is created.

```

/* -----
   ----- IMAGE COMPRESSION AND DECOMPRESSION -----
*/
#include <string.h>
#include "jpegdef.h"
extern unsigned char _huge *img_ptr [];
/* ptrs to image rows */
extern unsigned char dc_lumbits [17];
extern unsigned int dc_lumval [12];
extern unsigned char ac_lumbits [17];
extern unsigned int ac_lumval [162];
extern double dct_block [SIZE*SIZE];
extern double *dctptr [SIZE];
extern unsigned char out_buffer [];
extern int bytes_in_buf;
extern int encode_bits;
extern long int cmprs_bits;
extern int tdc_last;
extern int rdc_last;
extern struct huff_table dc_table;
extern struct huff_table ac_table;
static int q_blk [SIZE*SIZE];
/* input/quantized/output block */
static int q_tbl [SIZE*SIZE];
/* user quantization table */
void process_img (
    int xsize, /* image xsize */
    int ysize, /* image ysize */
    int *qbase_tbl, /* basic quantization table */
    int *zag, /* zigzag pattern */
    int *dezag, /* dezigzag pattern */
    int blksize, /* block size */

```

```

    int quality )          /* quality factor */
{
    int            i,j,k,l,m,n,*q_ptr;
    int            xp,yp,xpos,ypos,hblk,hrest,vblk;
    struct huff_table *dctbl = &dc_table;
    struct huff_table *actbl = &ac_table;
/* Set up quantization table according to user
   specified 'quality' factor */
    set_qtable (q_tbl,blksize,qbase_tbl,quality);
/* Compute standard Huffman DC and AC code tables */
    memcpy (&dctbl->bits ,dc_lumbits,sizeof (dc_lumbits));
    memcpy (&dctbl->hufval,dc_lumval ,sizeof (dc_lumval));
    memcpy (&actbl->bits ,ac_lumbits,sizeof (ac_lumbits));
    memcpy (&actbl->hufval,ac_lumval ,sizeof (ac_lumval));
    fix_huftbl (dctbl);
    fix_huftbl (actbl);
/* Initialize variables for compression/decompression */
    tdc_last      = 0; /* the last DC value for encoder */
    rdc_last      = 0; /* the last DC value for decoder */
    cmprs_bits    = 0L; /* # of bits for compressed data */
    bytes_in_buf  = 0; /* # of bytes in output buffer */
/* Set the number of subblocks horizontally
   and vertically */
    hblk = xsize / blksize;
    if ( (hrest = xsize % blksize) )
        hblk++;
    vblk = ysize / blksize;
    if ( ysize % blksize )
        vblk++;
    for ( i = 0; i < blksize; i++ )
        dctptr [i] = dct_block + i * blksize;
/* Extract the 2-D blocks from source image, one block
   at the time and do compression/decompression */
    for ( i = 0; i < vblk; i++ )
    {
        ypos = i * blksize;
        for ( j = 0; j < hblk; j++ )
        {
            xpos = j * blksize;
            memset (q_blk,0,blksize * blksize
                * sizeof (int));
            q_ptr = &q_blk [0];
            for ( m = 0, yp = ypos; m < blksize;
                m++, yp++ )
                if ( yp < ysize )
                    for ( n = 0, xp = xpos; n
                        < blksize; n++, xp++ )
                        if ( xp < xsize )
                            *q_ptr++ = (int)
                                *(img_ptr [yp] + xp);
                        else
                        {
                            q_ptr += (blksize - hrest);
                            break;
                        }
        }
/* Compression of a single sub-image block */
        cmprs_blk (q_blk,blksize,q_tbl,zag,
            &dctptr [0],dctbl,actbl);
    }
}

```

```

        cmprs_bits += encode_bits;
/* Decompression of the single sub-image block */
        memset (q_blk,0,blksize * blksize
                * sizeof (int));
        decmprs_blk (q_blk,blksize,q_tbl,dezag,
                    &dctptr
[0],dcttbl,acttbl);
/* Display reconstructed sub-image block */
        display_block (xsize,ysize,xpos,ypos,hrest,q_blk);
/* Clear output bit stream buffer and byte counter */
        while ( bytes_in_buf > 0 )
            out_buffer [bytes_in_buf--] = 0;
    }
}
}

```

4.5.8 Compression of Color Images

In many imaging applications, it is necessary to deal with color images. Although the RGB representation of images is typical of color displays, it is not the best representation from the viewpoint of compression. RGB images are converted into more suitable $YC_B C_R$ color format using the following equations [31]:

$$\begin{aligned}
 Y &= 0.299 R + 0.587 G + 0.114 B \\
 C_B &= -0.169 R - 0.331 G + 0.500 B = 0.564 (B - Y) \\
 C_R &= 0.500 R - 0.419 G - 0.081 B = 0.713 (R - Y),
 \end{aligned} \tag{4.58}$$

where Y represents a monochrome compatible luminance component, and C_B , C_R represent chrominance components containing color information. Most of image/video coding standards adopt $YC_B C_R$ color format as an input image signal [31]. This color conversion has the desirable property of packing most of the signal energy into Y and significantly less energy into the chrominance components. Furthermore, the HVS is much more sensitive to variations in the luminance component. These properties suggest a compression scheme for color images. The luminance component is encoded with high fidelity while larger errors are allowed in the chrominance components.

We have described the JPEG DCT-based coding technique for the luminance component. The chrominance components are similarly processed except for some minor modifications. Each chrominance component is subsampled by a factor of 2 or 4 in both the horizontal and vertical directions prior to compression. At the decoder, the reconstructed chrominance components are bilinearly interpolated back to their original size. Then, the image in $YC_B C_R$ color format is transformed into RGB format using the following equations:

$$\begin{aligned}
 R &= Y + 1.402 C_R \\
 G &= Y - 0.344 C_B - 0.714 C_R \\
 B &= Y + 1.772 C_B
 \end{aligned} \tag{4.59}$$

For compression and decompression of color images, the user needs the header file `JPEGCOLOR.H` containing the definitions of data structures for computation of stan-

standard Huffman chrominance DC and AC coding and decoding tables and the definition of the chrominance sample quantization table.

```

/*
 * JPEGCOLOR.H
 */
/* # of symbols with codes of length k bits (lumbits
   [k]) and symbols in order of incremental code length
   (lumval [k]) for DC chrominance values - valid
   for 8-bit data precision */
unsigned char dc_chrombits [17] =
    {0,0,3,1,1,1,1,1,1,1,1,0,0,0,0,0,0};
unsigned char dc_chromval [12]
    = {0,1,2,3,4,5,6,7,8,9,10,11};
/* # of symbols with codes of length k bits (lumbits
   [k]) and symbols in order of incremental code length
   (lumval [k]) for AC chrominance values - valid for
   8-bit data precision */
unsigned char ac_chrombits [17] =
    {0,0,2,1,2,4,4,3,4,7,5,4,4,0,1,2,0x77};
unsigned char ac_chromval [162] =
    { 0x00, 0x01, 0x02, 0x03, 0x11, 0x04, 0x05, 0x21,
      0x31, 0x06, 0x12, 0x41, 0x51, 0x07, 0x61, 0x71,
      0x13, 0x22, 0x32, 0x81, 0x08, 0x14, 0x42, 0x91,
      0xa1, 0xb1, 0xc1, 0x09, 0x23, 0x33, 0x52, 0xf0,
      0x15, 0x62, 0x72, 0xd1, 0x0a, 0x16, 0x24, 0x34,
      0xe1, 0x25, 0xf1, 0x17, 0x18, 0x19, 0x1a, 0x26,
      0x27, 0x28, 0x29, 0x2a, 0x35, 0x36, 0x37, 0x38,
      0x39, 0x3a, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,
      0x49, 0x4a, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
      0x59, 0x5a, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68,
      0x69, 0x6a, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
      0x79, 0x7a, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
      0x88, 0x89, 0x8a, 0x92, 0x93, 0x94, 0x95, 0x96,
      0x97, 0x98, 0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5,
      0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4,
      0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3,
      0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2,
      0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda,
      0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8, 0xe9,
      0xea, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, 0xf8,
      0xf9, 0xfa };
struct huff_table dc_ctable; /* Huffman DC code table */
struct huff_table ac_ctable; /* Huffman AC code table */
/* chrominance sample quantization table for
   an 8 x 8 DCT */
int qcbase8 tbl [8*8] =
    { 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, 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, 99 };

```

Assuming that each chrominance component was subsampled, the program from Section 4.5.7 can be used for compression and decompression. It is necessary only to substitute proper identifiers for the quantization table and Huffman coding/decoding tables.

4.5.9 Results of Image Compression

The performance of a compression algorithm can be evaluated in a number of different ways:

- Implementation complexity (algorithm complexity, computational speed, and memory requirements).
- The amount of compression expressed by the compression ratio.
- The average number of bits required to represent a single sample; this is generally referred to as *bit rate*.
- How closely the reconstruction resembles the original; this is related to the reconstructed image quality.

In evaluating the reconstructed image quality, a frequently used measure is the root-mean-square-error (RMSE) as an error metric [36]. Denoting the original $N \times N$ image by f_{ij} and the compressed/decompressed image by \hat{f}_{ij} , RMSE is given by

$$RMSE = \sqrt{\frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (f_{ij} - \hat{f}_{ij})^2},$$

and represents the standard deviation of the error image. Error images represent the difference between the original and reconstructed images. The error image, g_{ij} , can be generated using

$$g_{ij} = k \left| f_{ij} - \hat{f}_{ij} \right|,$$

where the scaling factor k is included to make any error more visible.

The results of applying the JPEG DCT-based coding technique are summarized in [Tables 4.1](#) and [4.2](#) for the monochrome “Lena” image. Recall that the original 512×512 monochrome “Lena” image requires a total of 2,097,152 bits, or 262,144 bytes. In the JPEG DCT-based image compression and decompression system, two block sizes have been used — 8×8 and 16×16 . [Table 4.1](#) summarizes results of compression using the 8×8 block size, and [Table 4.2](#) summarizes results of compression using the 16×16 block size for several values of the quality factor. From the tables it is evident that for 16×16 block size the compression ratios are about two-fold better than for 8×8 block size. On the other hand, at very low bit rates the blocking artifact is more visible for the larger block size. Actual reconstructed and corresponding error images using 8×8 and 16×16 blocks for some values of the quality factor (its definition is given in Section 4.5.3) are shown in Figs. 4.10 and 4.11, respectively.

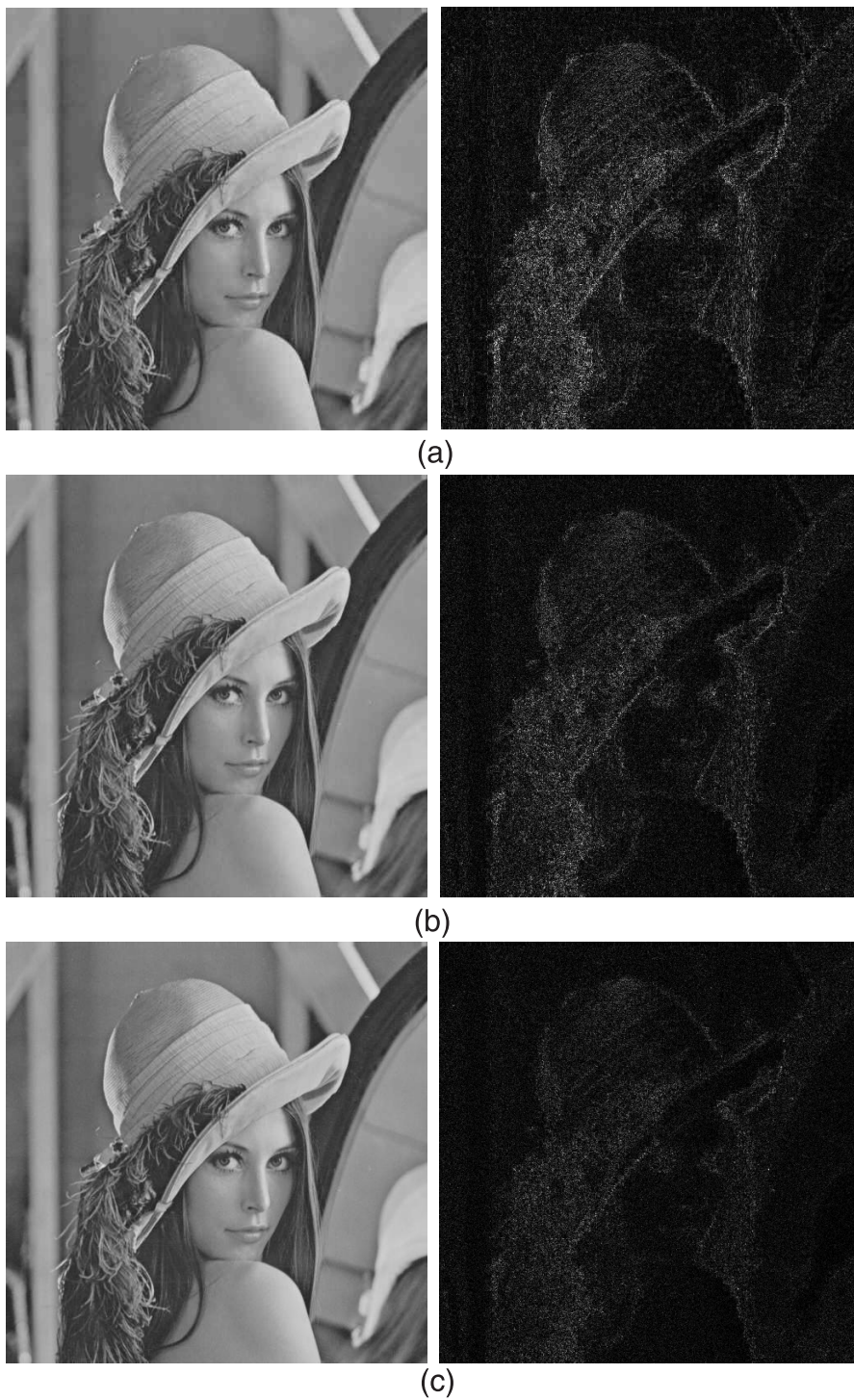


FIGURE 4.10

Reconstructed and corresponding error images using 8×8 DCT block size for the quality factor: (a) 25%, (b) 50%, (c) 75%. Error images are magnified by a factor of 8. Reproduced by Special Permission of *Playboy* magazine. Copyright ©1972, 2000 by Playboy.

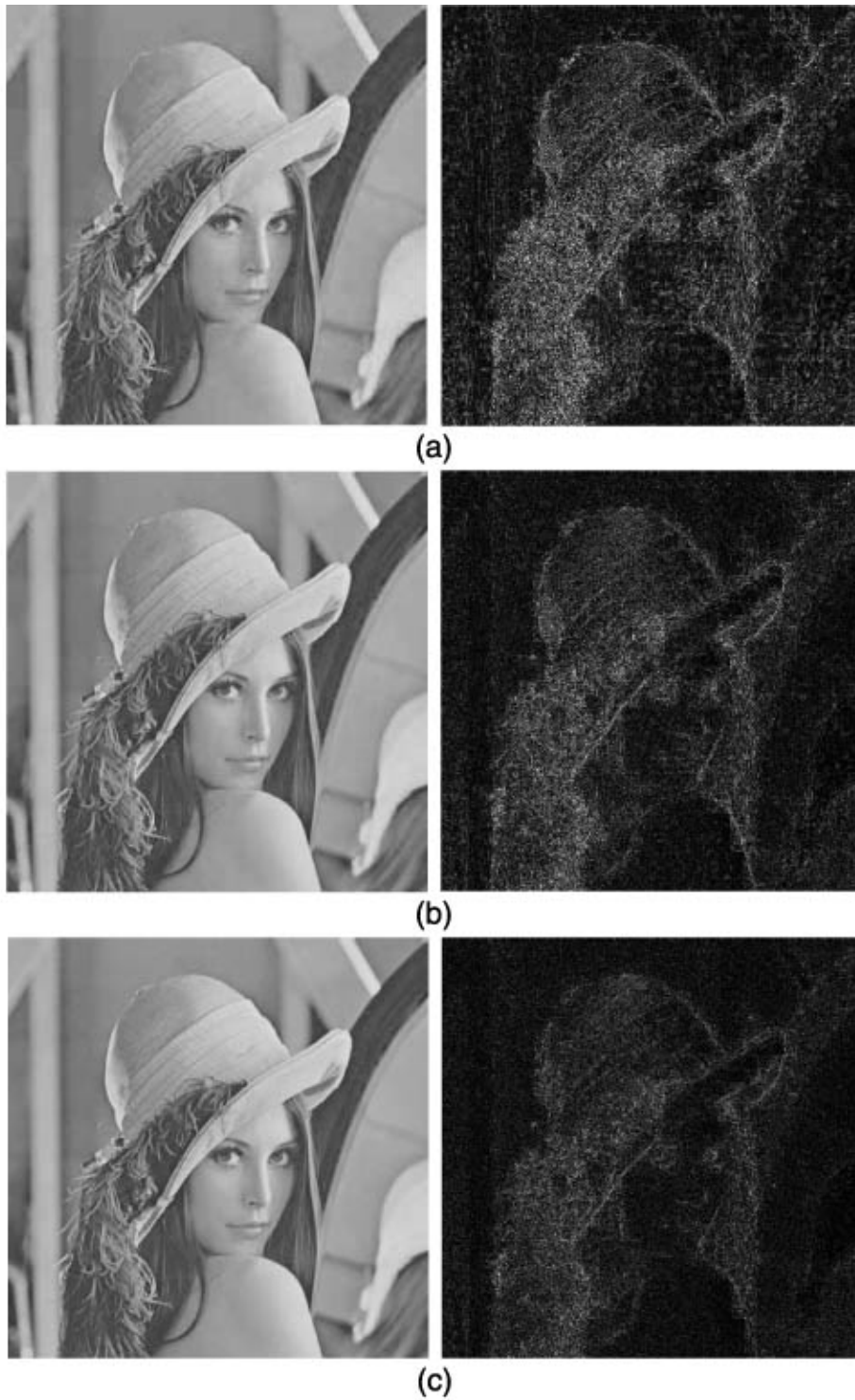


FIGURE 4.11

Reconstructed and corresponding error images using 16×16 DCT block size for the quality factor: (a) 25%, (b) 50%, (c) 75%. Error images are magnified by a factor of 8. Reproduced by Special Permission of *Playboy* magazine. Copyright ©1972, 2000 by Playboy.

Table 4.1 Results of the “Lena” Image Compression for 8×8 Block Size

Quality factor	The number of compressed bits (bytes)	Bit rate	Compression ratio	RMSE error
25%	96 351 (12 044)	0.368	21.766	4.774
50%	148 132 (18 517)	0.565	14.157	3.734
60%	170 878 (21 360)	0.652	12.273	3.470
75%	230 924 (28 866)	0.881	9.082	2.965
90%	422 392 (52 799)	1.611	4.965	2.124
100%	1 212 625 (151 579)	4.626	1.729	0.289

Table 4.2 Results of “Lena” Image Compression for 16×16 Block Size

Quality factor	The number of compressed bits (bytes)	Bit rate	Compression ratio	RMSE error
25%	47 101 (5 888)	0.180	44.525	6.234
50%	77 868 (9 734)	0.297	26.932	4.778
60%	91 272 (11 409)	0.348	22.977	4.428
75%	126 850 (15 857)	0.484	16.533	3.817
90%	244 865 (30 609)	0.934	8.565	2.808
100%	926 585 (115 824)	3.535	2.263	0.644

4.6 Summary

The definitions and properties of four types of the even DCT and corresponding even DST have been discussed and the unified fast computation of DCTs and DSTs has been presented. For each type of DCT and DST, the fast computational algorithm was described and the corresponding regular generalized signal flow graph was shown followed by its implementation in C. Among the DCTs, DCT of types II and III have been employed as the main compression tool in the international image/video coding standards. To illustrate the compression capability of DCT, a real data compression application is considered. The JPEG DCT-based image compression and decompression system with its implementation is described in detail. This simple, efficient, and low-cost image compression and decompression system can be used in real data compression applications. Finally, the results of image compression were presented. We believe that all implemented algorithms will be useful in any other DCT- and DST-related applications.

References

- [1] Jain, A.K., A sinusoidal family of unitary transforms, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 1, 356, 1979.
- [2] Bongiovanni, G., Corsini, P. and Frosini, G., One-dimensional and two-dimensional generalized discrete Fourier transform, *IEEE Trans. on Acoustics, Speech and Signal Processing*, 24, 97, 1976.
- [3] Wang, Z., Comments on generalized discrete Hartley transforms, *IEEE Trans. on Signal Processing*, 43, 1711, 1995.
- [4] Wang, Z. and Hunt, B.R., The discrete W transform, *Applied Mathematics and Computation*, 16, 19, 1985.
- [5] Wang, Z. and Hunt, B.R., The discrete cosine transform — a new version, *Proc. IEEE ICASSP*, Boston, MA, 1256, 1983.
- [6] Kitajima, H., A symmetric cosine transform, *IEEE Trans. on Computers*, 29, 317, 1980.
- [7] Jain, A.K., A fast Karhunen–Loève transform for a class of random processes, *IEEE Trans. on Communications*, 24, 1023, 1976.
- [8] Ahmed, N., Natarajan, T., and Rao, K.R., Discrete cosine transform, *IEEE Trans. on Computers*, 23, 90, 1974.
- [9] Kekre, H.B. and Solanki, J.K., Comparative performance of various trigonometric unitary transforms for transform image coding, *Int. J. Electronics*, 44, 305, 1978.
- [10] Yip, P. and Rao, K.R., On shift property of DCTs and DSTs, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 35, 404, 1987.
- [11] Wu, L.N., Comments on shift property of DCTs and DSTs, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 38, 186, 1990.
- [12] Malvar, H.S., Lapped transforms for efficient transform/subband coding, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 38, 969, 1990.
- [13] Martucci, S.A., Convolution-multiplication properties for the entire family of discrete sine and cosine transforms, *Proc. Twenty-Sixth Annual Conf. on Information Sciences and Systems*, Princeton, NJ, 399, 1992.
- [14] Martucci, S.A., Symmetric convolution and the discrete sine and cosine transforms, *IEEE Trans. on Signal Processing*, 42, 1038, 1994.
- [15] Martucci, S.A., Digital filtering of images using the discrete sine or cosine transform, *Optical Engineering*, 35, 119, 1996.

- [16] Clarke, R.J., Relation between the Karhunen–Loève and cosine transforms, *IEEE Proc. Part F: Communications, Radar and Signal Processing*, 128, 359, 1981.
- [17] Ahmed, N. and Flickner, M., Some considerations of the discrete cosine transform, *16th Asilomar Conf. on Circuits, Systems and Computers*, Pacific Grove, CA, 295, 1982.
- [18] Clarke, R.J., Relation between the Karhunen–Loève and sine transforms, *Electronics Letters*, 20, 12, 1984.
- [19] Zou, F. and Gallagher, R.R., A new transform with symmetrical coding performance for Markov(1) signals, *IEEE Trans. on Signal Processing*, 43, 2195, 1995.
- [20] Wang, Z., A fast algorithm for the discrete sine transform implemented by the fast cosine transform, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 30, 814, 1982.
- [21] Wang, Z., Fast algorithms for the discrete W transform and discrete Fourier transform, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 32, 803, 1984.
- [22] Wang, Z., On computing the discrete Fourier and cosine transform, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 33, 1341, 1985.
- [23] Hou, H.S., A fast recursive algorithm for computing the discrete cosine transform, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 35, 1455, 1987.
- [24] Britanak, V., On the discrete cosine transform computation, *Signal Processing*, 40, 183, 1994.
- [25] Britanak, V., A unified discrete cosine and discrete sine transform computation, *Signal Processing*, 43, 333, 1995.
- [26] Britanak, V., A unified approach to fast computation of discrete sinusoidal transforms I: DCT and DST transforms, *Computers and Artificial Intelligence*, 17, 583, 1998.
- [27] Britanak, V., A unified approach to fast computation of discrete sinusoidal transforms II: DFT and DWT transforms, *Computers and Artificial Intelligence*, 18, 19, 1999.
- [28] Wu, H.R. and Paoloni, F.J., A two-dimensional fast cosine transform algorithm based on Hou's approach, *IEEE Trans. on Signal Processing*, 39, 544, 1991.
- [29] Britanak, V., A generalized signal flow graph for the 2-D DCT computation, *Applied Signal Processing*, 1, 76, 1994.
- [30] Rao, K.R. and Yip, P., *Discrete Cosine Transform: Algorithms, Advantages, Applications*, Academic Press, Boston, 1990.

- [31] Rao, K.R. and Hwang, J.J., *Techniques and Standards for Image, Video and Audio Coding*, Prentice-Hall, Upper Saddle River, NJ, 1996.
- [32] Clarke, R.J., *Transform Coding of Images*, Academic Press, London, 1990.
- [33] Poularikas, A.D., *The Transforms and Application Handbook*, CRC Press and IEEE Press, Boca Raton, FL, 1996.
- [34] Gonzalez, R.C. and Woods, R.E., *Digital Image Processing*, Addison–Wesley, Reading, MA, chap. 6, 1992.
- [35] Jain, A.K., *Fundamentals of Digital Image Processing*, Prentice-Hall, Englewood Cliffs, NJ, chap. 5 and 11, 1989.
- [36] Rabbani, M. and Jones, P.W., *Digital Image Compression Techniques*, Volume TT7 of Tutorial Texts Series, SPIE Optical Engineering Press, Bellingham, WA, 1991.
- [37] Sayood, K., *Introduction to Data Compression*, Morgan Kaufmann, San Francisco, CA, 1996.
- [38] Madisetty, V.K. and Williams, D.B., *The Digital Signal Processing Handbook*, CRC Press and IEEE Press, Boca Raton, FL, 1998.
- [39] Bhaskaran, V. and Konstantinides, K., *Image and Video Compression Standards: Algorithms and Architecture*, Kluwer Academic, Norwell, MA, 1995.
- [40] Kou, W., *Digital Image Compression — Algorithms and Standards*, Kluwer Academic, Hingham, MA, 1995.
- [41] Gibson, J.D., Berger, T., Lookabaugh, T., Lindbergh, D., and Baker, R.L., *Digital Compression for Multimedia: Principles and Standards*, Morgan Kaufmann, San Francisco, CA, 1998.
- [42] Symes, P.D., *Video Compression: Fundamental Compression Techniques and Overview of the JPEG and MPEG Compression Systems*, McGraw-Hill, New York, 1998.
- [43] Wallace, G.K., *JPEG Technical Specification, Revision 5*, JPEG Joint Photographic Experts Group ISO/IEC JTC1/SC2/WG8 CCITT SGVIII, JPEG-8-R5, January 2, 1990.
- [44] Wallace, G.K., The JPEG still picture compression standard, *Communications of the ACM*, 34, 31, 1991.
- [45] Le Gall, D., MPEG: A video compression standard for multimedia applications, *Communications of the ACM*, 34, 47, 1991.
- [46] Liou, M., Overview of the $p \times 64\text{kbit/s}$ video coding standard, *Communications of the ACM*, 34, 60, 1991.
- [47] Rabbani, M. and Dally, S., An optimized image data compression technique utilized in the Kodak SV9600 still video transceiver, *SPIE Proc. Vol. 1071 Optical Sensors and Electronic Photography*, Bellingham, WA, 246, 1989.

- [48] Chitprasert, B. and Rao, K.R., Human visual weighted progressive image transmission, *IEEE Trans. on Communications*, 38, 1040, 1990.
- [49] Murray, J.D. and VanRyper, W., *Encyclopedia of Graphics File Formats*, O'Reilly and Associates, Sebastopol, CA, 1994.