An Introduction to Video Compression in C/C++

Fore June

Chapter 1

Discrete Cosine Transform (DCT)

6.1 Time, Space and Frequency Domains

Data compression techniques can be classified as *lossless* or *lossy*. In lossless compression, we can reverse the process and recover the exact original data. This technique works by exploiting and removing redundancy of data and no information is lost in the process. Typically, lossless compression is used to compress text and binary programs and compression ratios achieved are usually not very high.

In lossy compression, we throw away some information carried by the data and thus the process is not reversible. The technique can give us much higher compression ratios. *Given a set of data, what kind of information should we throw away?* It turns out that choosing the portion of information to throw away is the state-of-the-art of lossy compression. Note that in technical terms, we can have redundant data but **not** redundant information. However, we can have irrelevant information and usually this is the part of the information want to throw away. For example, when we write a story about a marathon runner, we may usually omit the part that tells the time she sleeps, the time she gets up and the time she eats without affecting the story. Given an image, we want to determine which components are not as relevant as other components and discard the less relevant components. In previous chapters, we discussed that by transforming the representation of an image from RGB to 4:2:0 YCbCr format, which separates the intensity from color components, we can easily compress an image by a factor of two without much down grading of the image quality. The underlying principles in this stage of compression is that human eyes are more sensitive to brightness than to color and in practice, we do not need to retain as much information of the color components as we need in presenting an image. However, even after this change in format, we still represent the image in the spatial domain, where a sample value depends on the positon of the two-dimensional space. That is, it is a function of the coordinates (x, y) of a two-dimensional plane. Our eyes do not have any crucial discrimination of a point at any special position in space and thus it is difficult for us to further pick the irrelevant information and throw it away if we need to. On the other hand, if we could represent the image in the frequency domain, we know that our eyes are not very sensitive to high frequency components and if we have to get rid of any information, we would like to get rid of those components first.

It turns out that in nature, any wave can be expressed as a superposition of sine and cosine waves. In other words, any periodic signals can be decomposed into a series of sine and cosine waves with frequencies that are integer multiples of a certain frequency. This is the well-known **Fourier Theorem**, which is one of the most important discoveries in the history of science and technology. It is the foundation of many science and engineer applications. Lossy image compression is one of the many applications that utilize a transform built upon the theorem.

6.2 Discrete Cosine Transform (DCT)

Numerous research has been conducted on transforms for image and video compression. There are a few methods that are practical and popularly used. For static image compression, the Discrete Wavelet Transform (DWT) is the most popular method and can yield good results; it has been incorporated in the JPEG standard. Other popular methods that require less memory to operate include Karhunen-Loeve Transform (KLT), Singular Value Decomposition (SVD), and Discrete Cosine Transform (DCT). For video compression, DCT tends to give very good performance and has been incorporated in the MPEG standard. In this book, DCT will be the only significant transform that we shall discuss. DCT closely relates to Discrete Fourier Transform (DFT). Figure 6-1 shows the operation of two dimensional DCT on a block of $N \times M$ pixels.

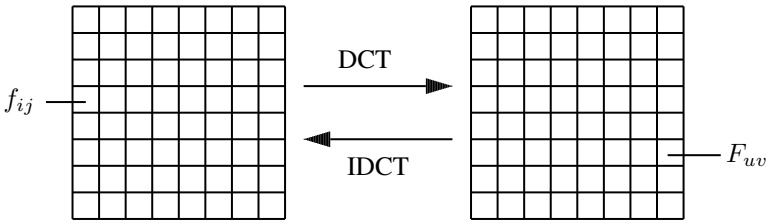

Figure 6-1. DCT and Inverse DCT (IDCT)

In the figure, f_{ij} are the pixel values and F_{uv} are the transformed values. The transform is reversible meaning that if we discount the rounding errors occurred in arithmetic calculations, we can recover the original pixel values by reversing the transformation. The reversed transformation is known as Inverse DCT or IDCT, which is also shown in **Figure 6-1**.

The general equation for a 2D **DCT** of a block of $N \times M$ pixels with values f_{ij}s is defined by the following equation:

$$F_{uv} = N_u M_v \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} f_{ij} cos\frac{(2j+1)v\pi}{2M} cos\frac{(2i+1)u\pi}{2N} \tag{6.1}$$

where

$$K_r = \begin{cases} \sqrt{\frac{1}{K}} & \text{if } r = 0 \\ \sqrt{\frac{2}{K}} & \text{if } r > 0 \end{cases} \tag{6.2}$$

and K is M or N in (6.1).

The corresponding IDCT is given by the following equation:

$$f_{ij} = \sum_{u=0}^{N-1} \sum_{v=0}^{M-1} N_u M_v F_{uv} cos\frac{(2j+1)v\pi}{2M} cos\frac{(2i+1)u\pi}{2N} \tag{6.3}$$

In many applications, $N = M$ and equations (6.1) and (6.2) can be expressed in matrix forms. If **f** and **F** denote the matrices (f_{ij}) and (F_{uv}) respectively, equation (6.1) can be rewritten in the following matrix form:

$$\mathbf{F} = \mathbf{R}\,\mathbf{f}\,\mathbf{R^T} \tag{6.4}$$

where $\mathbf{R^T}$is the transpose of the transform matrix **R**. The matrix elements of **R** are

$$R_{ij} = N_i cos\frac{(2j+1)i\pi}{2N} \tag{6.5}$$

where N_i is defined in (6.2). It turns out that the inverse of $\mathbf{R}$ is the same as its transpose, i.e. $\mathbf{R^{-1}} = \mathbf{R^T}$. Therefore, the inverse transformation, IDCT can be found by:

$$\mathbf{f} = \mathbf{R^T}\,\mathbf{F}\,\mathbf{R} \tag{6.6}$$

Example
Consider N = M = 4. The DCT transform matrix $\mathbf{R}$ is a 4×4 matrix. Using the fact that $cos(\pi - \theta) = -cos\theta$, $cos(\pi + \theta) = -cos\theta$ and $cos(2\pi + \theta) = cos\theta$, we obtain the following matrix:

$$\mathbf{R} = \begin{pmatrix} a & a & a & a \\ b & c & -b & -c \\ a & -a & -a & a \\ c & -b & -b & c \end{pmatrix} \quad where \quad \begin{aligned} a &= \tfrac{1}{2} \\ b &= \sqrt{\tfrac{1}{2}}cos\tfrac{\pi}{8} \\ c &= \sqrt{\tfrac{1}{2}}cos\tfrac{3\pi}{8} \end{aligned} \tag{6.7}$$

Evaluating the cosines, we have

$$\mathbf{R} = \begin{pmatrix} 0.5 & 0.5 & 0.5 & 0.5 \\ 0.653 & 0.271 & -0.271 & -0.653 \\ 0.5 & -0.5 & -0.5 & 0.5 \\ 0.271 & -0.653 & -0.653 & 0.271 \end{pmatrix} \tag{6.8}$$

Sometimes DCT is referred to as Forward DCT (FDCT) in order to distinguish it from Inverse DCT (IDCT).

6.3 Floating-point Implementation of DCT and IDCT

A direct floating-point implementation of DCT and IDCT of an $N \times N$ block is straight forward and simple. All we need to do is to use two for-loops to do summations. Program Listing 6-1 shows the implementation. In the program, the functions **dct_direct**() and **idct_direct**() do the actual work of DCT, and IDCT respectively; they use floating point (double) in calculations. The functions **dct**() and **idct**() simply cast short values to double and call **dct_direct**() or **idct_direct**() to do the transformations. In the functions, the values of *(f+i*N+j) and *(F+u*N+v) refer to the values of f[i][j] and F[u][v], corresponding to f_{ij} and F_{uv} in equations (6.1) and (6.3) respectively; a[u] and a[v] correspond to N_u and N_v.

Program Listing 6-1 Floating Point Implementation of DCT and IDCT

```
//---------------------------------------------------------------------
/*
  dct_direct.cpp
  A straight forward implementation of DCT and IDCT for the purpose
  of learning and testing. Floating-point arithmetic is used.  Such
  an implementation should not be used in practical applications.

  Compile: g++ -o dct_direct dct_direct.cpp -lm
  Execute: ./dct_direct
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
using namespace std;
#define PI 3.141592653589

//input: f, N; output: F
short dct_direct( short N, double *f, double *F )
{
  double a[32], sum, coef;
  short i, j, u, v;

  if ( N > 32 || N <= 0 ) {
    printf ("\ninappropriate N\n");
    return -1;
  }
  a[0] = sqrt ( 1.0 / N );
  for ( i = 1; i < N; ++i ) {
    a[i] = sqrt ( 2.0 / N );
  }
  for ( u = 0; u < N; ++u ) {
    for ( v = 0; v < N; ++v ) {
      sum = 0.0;
      for ( i = 0; i < N; ++i ) {
        for ( j = 0; j < N; ++j ) {
          coef = cos((2*i+1)*u*PI/(2*N))*cos((2*j+1)*v*PI/(2*N));
          sum += *(f+i*N+j) * coef;            //f[i][j] * coef
        } //for j
        *(F+u*N+v) = a[u] * a[v] * sum;
      } //for i
    } //for u
  } //for v

  return 1;
}

//input: N, F; output f
short idct_direct( short N, double *F, double *f )
{
  double a[32], sum, coef;
  short i, j, u, v;

  if ( N > 32 || N <= 0 ) {
    printf ("\ninappropriate N\n");
    return -1;
  }
  a[0] = sqrt ( 1.0 / N );
  for ( i = 1; i < N; ++i ) {
    a[i] = sqrt ( 2.0 / N );
  }
  for ( i = 0; i < N; ++i ) {
    for ( j = 0; j < N; ++j ) {
      sum = 0.0;
      for ( u = 0; u < N; ++u ) {
        for ( v = 0; v < N; ++v ) {
          coef =  cos((2*j+1)*v*PI/(2*N))*cos ((2*i+1)*u*PI/(2*N));
          sum+=a[u]*a[v]*(*(F+u*N+v))*coef;//a[u]*a[v]*F[u][v]*coef
        } //for j
```

```
        *(f+i*N+j) =  sum;
      } //for i
    } //for u
  } //for v

  return 1;
}

/*
  change values from short to double and vice versa.
*/
short dct ( short N, short *f, short *F )
{
  double  tempx[1024], tempy[1024];
  int  total, i;

  if ( N > 32 || N <= 0 ) {
    printf ("\ninappropriate N\n");
    return -1;
  }
  total = N * N;
  for ( i = 0; i < total; ++i ) {
    tempx[i] = (double) *(f+i);
  }
  dct_direct ( N, tempx, tempy );               //DCT operation
  for ( i = 0; i < total; ++i ) {
    *(F+i) = (short ) ( floor (tempy[i]+0.5) ); //rounding
  }

  return 1;
}

/*
  change values from short to doulbe, and vice versa.
*/
short idct ( short N, short *F, short *f )
{
  double  tempx[1024], tempy[1024];
  int  total, i;

  if ( N > 32 || N <= 0 ) {
    printf ("\ninappropriate N\n");
    return -1;
  }
  total = N * N;
  for ( i = 0; i < total; ++i ) {
    tempy[i] = (double) *(F+i);
  }
  idct_direct ( N, tempy, tempx );              //IDCT operation
  for ( i = 0; i < total; ++i ) {
    *(f+i) = (short ) ( floor (tempx[i]+0.5) ); //rounding
  }

  return 1;
}
```

```
void print_elements ( short N,  short *f )
{
  short i, j;

  for ( i = 0; i < N; ++i ){
    printf("\n");
    for ( j = 0; j < N; ++j ) {
      printf ("%4d, ", *(f+N*i+j) );
    }
  }
}

int main()
{
  short f[8][8], F[8][8];
  int i, j, N;
  char temp[8][8];
  N = 8;

  //try some values for testing
  for ( i = 0; i < N; ++i ) {
    for ( j = 0; j < N; ++j ) {
        f[i][j] = i + j;
    }
  }

  printf("\nOriginal sample values");
  print_elements ( N, &f[0][0] );
  printf("\n--------------------\n");

  dct ( N, &f[0][0], &F[0][0] );                //performing DCT
  printf("\nCoefficients of DCT:");
  print_elements ( N, &F[0][0] );
  printf("\n--------------------\n");

  idct ( N, &F[0][0], &f[0][0] );               //performing IDCT
  printf("\nValues recovered by IDCT:");
  print_elements ( N, &f[0][0] );
  printf("\n");
}
-------------------------------------------------------------------------
```

The implementation shown in Listing 6-1 is inefficient and impractical in image and video compression, which requires numerous operations of DCT and IDCT. Also, it is not a good programming practice to print out messages inside a function which is designed for other purposes. However, this program can be used for checking purposes when we later implement DCT and IDCT using more efficient methods. When the program is executed, it prints the following outputs, where $N = 8$ has been considered.

```
-------------------------------------------------------------------------
Original sample values
   0,    1,    2,    3,    4,    5,    6,    7,
   1,    2,    3,    4,    5,    6,    7,    8,
   2,    3,    4,    5,    6,    7,    8,    9,
```

```
   3,    4,    5,    6,    7,    8,    9,   10,
   4,    5,    6,    7,    8,    9,   10,   11,
   5,    6,    7,    8,    9,   10,   11,   12,
   6,    7,    8,    9,   10,   11,   12,   13,
   7,    8,    9,   10,   11,   12,   13,   14,
--------------------

Coefficients of DCT:
  56,  -18,    0,   -2,    0,   -1,    0,    0,
 -18,    0,    0,    0,    0,    0,    0,    0,
   0,    0,    0,    0,    0,    0,    0,    0,
  -2,    0,    0,    0,    0,    0,    0,    0,
   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,
--------------------

Values recovered by IDCT:
   0,    1,    2,    3,    4,    5,    6,    7,
   1,    2,    3,    4,    5,    6,    7,    8,
   2,    3,    4,    5,    6,    7,    8,    9,
   3,    4,    5,    6,    7,    8,    9,   10,
   4,    5,    6,    7,    8,    9,   10,   11,
   5,    6,    7,    8,    9,   10,   11,   12,
   6,    7,    8,    9,   10,   11,   12,   13,
   7,    8,    9,   10,   11,   12,   13,   14,
------------------------------------------------------------------------
```

We can see from the output that there are only a few nonzero coefficient values after DCT and they are clustered at the upper left corner. So after DCT, it becomes clear to us that the sample block **f** actually does not contain as much information as it appears and it is a lot easier to carry out data compression in the transformed domain.

The value of the DCT coefficient F(0, 0) at position (0, 0) is in general referred to as the DC value and others are referred to as AC values. This is because F(0, 0) is essentially a scaled average of all the sample values. We can easily see this if we write down the formula for calculating its value by setting $u = 0, v = 0$, and $N = M$ in Equation (6.1). That is,

$$F_{00} = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f_{ij} \tag{6.9}$$

where we have used the fact that $N_0 N_0 = \sqrt{\frac{1}{N}}\sqrt{\frac{1}{N}} = \frac{1}{N}$, and $cos(0) = 1$. (F_{00}/N is the exact average of all the values.) In Chapter 2, we discussed that the average of a set of values could give us the most crucial information of the set if we are only allowed to know one single value. Therefore, the DC value of the DCT coefficients of a block of samples is the most important single value and we want to retain its value.

6.4 Fast DCT

As DCT is so important in signal processing, a lot of research has been done to speed up its calculations. One main idea behind speeding up DCT is to break down the summation into stages and in each stage, intermediate sums of two quantities are formed. The intermediate sums will be used in

later stages to obtain the final sum. In this way, the number of calculations grows with $NlogN$ rather than N^2 as in the case of direct DCT for an $N \times N$ sample block. This kind of DCT is referred to as Fast DCT. In this section, we only discuss the speeding up of Forward DCT. The techniques also apply to Inverse DCT that we shall discuss later.

If we consider only small values of N in the form of $N = 2^n$, it is not difficult to understand how the stage break-down is done. For instance, consider $N = 8$, which is what we need in our video compression. (In Chapter 5, we discussed that a macroblock consists of 8×8 sample blocks.) We can rewrite (6.1) as follows.

$$F_{uv} = a_u a_v \sum_{i=0}^{7} \sum_{j=0}^{7} f_{ij} cos\frac{(2j+1)v\pi}{16} cos\frac{(2i+1)u\pi}{16} \tag{6.10}$$

with

$$\begin{aligned} a_0 &= \sqrt{\frac{1}{8}} = \frac{1}{2\sqrt{2}} \\ a_k &= \sqrt{\frac{2}{8}} = \frac{1}{2} \qquad \text{for } k > 0 \end{aligned} \tag{6.11}$$

Equation (6.10) implies that we can express a two dimensional (2D) DCT as two one-dimensional (1D) DCT. Equation (6.10) can be rewritten as follows.

$$F_{uv} = a_u a_v \sum_{i=0}^{7} \overline{F}_{iv} cos\frac{(2i+1)u\pi}{16} \tag{6.12}$$

where

$$\overline{F}_{iv} = \sum_{j=0}^{7} f_{ij} cos\frac{(2j+1)v\pi}{16} \tag{6.13}$$

Aside from a multiplicative constant, Equation (6.13) can be interpreted as a 1D DCT or the DCT of one row (the i-th row) of samples of an 8×8 sample block. For convenience of writing, we shall suppress writing the index i; it is understood that we consider one row of samples. Also, we let

$$\begin{aligned} x_j &= f_{ij} \\ y_v &= \overline{F}_{iv} \end{aligned} \tag{6.14}$$

Equation (6.13) becomes

$$y_k = \sum_{j=0}^{7} x_j cos\frac{(2j+1)k\pi}{16}, \qquad k = 0, 1, ..., 7 \tag{6.15}$$

If we let $\theta = \frac{\pi}{16}$, we can list all the coefficients of y_k (k = 0, 1, .. 7) in a table as shown below.

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
y_0	1	1	1	1	1	1	1	1
y_1	$cos(1\theta)$	$cos(3\theta)$	$cos(5\theta)$	$cos(9\theta)$	$cos(11\theta)$	$cos(13\theta)$	$cos(15\theta)$	$cos(15\theta)$
y_2	$cos(2\theta)$	$cos(6\theta)$	$cos(10\theta)$	$cos(14\theta)$	$cos(18\theta)$	$cos(22\theta)$	$cos(26\theta)$	$cos(30\theta)$
y_3	$cos(3\theta)$	$cos(9\theta)$	$cos(15\theta)$	$cos(21\theta)$	$cos(27\theta)$	$cos(33\theta)$	$cos(39\theta)$	$cos(45\theta)$
y_4	$cos(4\theta)$	$cos(12\theta)$	$cos(20\theta)$	$cos(28\theta)$	$cos(36\theta)$	$cos(44\theta)$	$cos(52\theta)$	$cos(60\theta)$
y_5	$cos(5\theta)$	$cos(15\theta)$	$cos(25\theta)$	$cos(35\theta)$	$cos(45\theta)$	$cos(55\theta)$	$cos(65\theta)$	$cos(75\theta)$
y_6	$cos(6\theta)$	$cos(18\theta)$	$cos(30\theta)$	$cos(42\theta)$	$cos(54\theta)$	$cos(66\theta)$	$cos(78\theta)$	$cos(90\theta)$
y_7	$cos(7\theta)$	$cos(21\theta)$	$cos(35\theta)$	$cos(49\theta)$	$cos(63\theta)$	$cos(77\theta)$	$cos(91\theta)$	$cos(105\theta)$

Table 6-1

Since $\theta = \frac{\pi}{16}$, we have $16\theta = \pi$. We can simplify the above table by making use of some basic cosine properties such as

$$\begin{array}{l} cos(\pi - \alpha) = -cos(\alpha) \\ cos(2\pi - \alpha) = cos(\alpha) \end{array} \tag{6.16}$$

Using (6.16), we can reduce $cos(n\theta)$ with $n \leq 8$ to a form of $\pm cos(k\theta)$ with $k \leq 7$. For example,

$$\begin{array}{llll} cos(9\theta) & = cos(16\theta - 7\theta) & = cos(\pi - 7\theta) & = -cos(7\theta) \\ cos(35\theta) & = cos(32\theta + 3\theta) & = cos(2\pi + 3\theta) & = cos(3\theta) \end{array}$$

Applying these, we can simplify Table 6-1 to Table 6-2 as shown below, where cs represents $cosine$:

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
y_0	1	1	1	1	1	1	1	1
y_1	$cs(1\theta)$	$cs(3\theta)$	$cs(5\theta)$	$cs(7\theta)$	$-cs(7\theta)$	$-cs(5\theta)$	$-cs(3\theta)$	$-cs(1\theta)$
y_2	$cs(2\theta)$	$cs(6\theta)$	$-cs(6\theta)$	$-cs(2\theta)$	$-cs(2\theta)$	$-cs(6\theta)$	$cs(6\theta)$	$cs(2\theta)$
y_3	$cs(3\theta)$	$-cs(7\theta)$	$-cs(1\theta)$	$-cs(5\theta)$	$cs(5\theta)$	$cs(1\theta)$	$cs(7\theta)$	$-cs(3\theta)$
y_4	$cs(4\theta)$	$-cs(4\theta)$	$-cs(4\theta)$	$cs(4\theta)$	$cs(4\theta)$	$-cs(4\theta)$	$-cs(4\theta)$	$cs(4\theta)$
y_5	$cs(5\theta)$	$-cs(1\theta)$	$cs(7\theta)$	$cs(3\theta)$	$-cs(3\theta)$	$-cs(7\theta)$	$cs(1\theta)$	$-cs(5\theta)$
y_6	$cs(6\theta)$	$-cs(2\theta)$	$cs(2\theta)$	$-cs(6\theta)$	$-cs(6\theta)$	$cs(2\theta)$	$-cs(2\theta)$	$cs(6\theta)$
y_7	$cs(7\theta)$	$-cs(5\theta)$	$cs(3\theta)$	$-cs(1\theta)$	$cs(1\theta)$	$-cs(3\theta)$	$cs(5\theta)$	$-cs(7\theta)$

Table 6-2

In Table 6-2, each y_i is equal to the sum over k of x_k times the coefficient at column k and row i. We observe that the columns possess certain symmetries; besides the first row, whenever $cos(k\theta)$ appears in an i-th column, it also appears in another j-th column when $i + j = 7$. This implies that we can always group the i-th and j-th columns together in our summing operations to compute $(x_i \pm x_j)cos(k\theta)$ provided $i + j = 7$. For example, we can rewrite y_0 and y_2 as follows:

$$\begin{array}{l} y_0 = (x_0 + x_7) + (x_1 + x_6) + (x_2 + x_5) + (x_3 + x_4) \\ y_2 = (x_0 + x_7)c_2 + (x_1 + x_6)c_6 - (x_2 + x_5)c_6 - (x_3 + x_4)c_2 \end{array} \tag{6.17}$$

where

$$c_k = cos(k\theta)$$

Once we have computed $(x_i + x_j)$ with $i + j = 7$, we can use this intermediate result in the calculations of both of y_0 and y_2. Moreover, we can continue this process recursively until we obtain the final values. For instance, let

$$\begin{array}{ll} x'_i = (x_i + x_j), & i + j = 7 \\ x'_j = (x_i - x_j), & i + j = 7 \\ x''_i = (x'_i + x'_j), & i + j = 7/2 = 3 \\ x''_j = (x'_i - x'_j), & i + j = 7/2 = 3 \end{array} \tag{6.18}$$

We can rewrite y_0 and y_2 in (6.17) as follows:

$$\begin{array}{lll} y_0 = & (x'_0 + x'_3) + (x'_1 + x'_2) & = (x''_0 + x''_1) \\ y_2 = & (x'_0 - x'_3)c_2 + (x'_1 - x'_2)c_6 & = x''_3 c_2 + x''_2 c_6 \end{array} \tag{6.19}$$

We can decompose y_4 and y_6 in a similar way as y_4 only uses c_4 and y_6 uses only c_2 and c_6 of the cosine functions in the calculations. Equations of (6.20) shows the decomposition of y_4 and y_6:

$$\begin{array}{lll} y_4 = & (x'_0 + x'_3)c_4 - (x'_1 + x'_2)c_4 = & (x''_0 - x''_1)c_4 \\ y_6 = & (x'_0 - x'_3)c_6 + (x'_2 - x'_1)c_2 = & x''_3 c_6 - x''_1 c_2 \end{array} \tag{6.20}$$

The calucation of the other $y'_k s(y_1, y_3, y_5,$ and $y_7)$ involves four cosine functions $(c_1, c_3, c_5$ and $c_7)$ and they appear to be more difficult to decompose. It turns out that these cosine functions can be expressed in terms of each other by making use of some basic cosine properties like those expressed in (6.21):

$$\begin{aligned}
&cos(\alpha-\beta) = cos(\alpha)cos(\beta) + sin(\alpha)sin(\beta)\\
&8\theta = \frac{8\pi}{16} = \frac{\pi}{2}\\
&c_4 = cos(4\theta) = sin(4\theta) = cos(\frac{\pi}{4}) = sin(\frac{\pi}{4}) = \frac{1}{\sqrt{2}}\\
&c_1 = cos(1\theta) = cos(8\theta - 7\theta) = cos(\frac{\pi}{2} - 7\theta) = sin(7\theta)\\
&c_3 = cos(3\theta) = cos(7\theta - 4\theta) = cos(7\theta)cos(4\theta) + sin(7\theta)sin(4\theta) = \frac{c_7 + c_1}{\sqrt{2}}\\
&c_5 = cos(5\theta) = cos(1\theta + 4\theta) = cos(1\theta)cos(4\theta) - sin(1\theta)sin(4\theta) = \frac{c_1 - c_7}{\sqrt{2}}
\end{aligned} \tag{6.21}$$

Making use of the identities of (6.21), we can apply the decomposition steps to all the $y'_i s$. For example, we can express y_1 as

$$\begin{aligned}
y_1 &= (x_0 - x_7)c_1 + (x_1 - x_6)c_3 + (x_2 - x_5)c_5 + (x_3 - x_4)c_7\\
&= x'_7 c_1 + x'_6 c_3 + x'_5 c_5 + x'_4 c_7\\
&= x'_7 c_1 + \frac{x'_6(c_1 + c_7)}{\sqrt{2}} + \frac{x'_5(c_1 - c_7)}{\sqrt{2}} + x'_4 c_7\\
&= [x'_7 + \frac{x'_6 + x'_5}{\sqrt{2}}]c_1 + [x'_4 + \frac{x'_6 - x'_5}{\sqrt{2}}]c_7\\
&= [x'_7 + x''_6]c_1 + [x'_4 + x''_5]c_7\\
&= x'''_7 c_1 + x'''_4 c_7
\end{aligned} \tag{6.22}$$

In (6.22), the subscripts in the third and fourth recursive stages are not well-defined but their meanings should be clear. We observe that in the calculations, we often have expressions in the the following form:

$$\begin{aligned}
c &= (a + b)\\
d &= (a - b)
\end{aligned} \tag{6.23}$$

The computations of (6.23) can be represented by a diagram shown in Figure 6-2, which is referred to as a butterfly computation because of its appearance. It is also a simple flow graph.

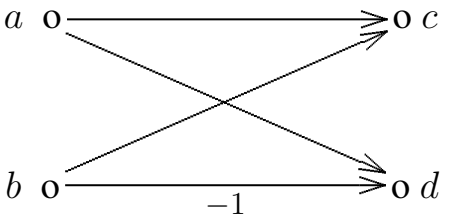

Figure 6-2. Flow Graph of Butterfly Computation

If we include the constants a_k in our calculation of y_k and let

$$C_k = c_k a_k = \frac{c_k}{2} = \frac{cos(\theta)}{2} \qquad k \geq 1, \; and$$
$$C_0 = \frac{1}{\sqrt{2}}$$

we arrive at the following flow graph. (Note that $a_0 = \frac{1}{2\sqrt{2}} = C_4$.)

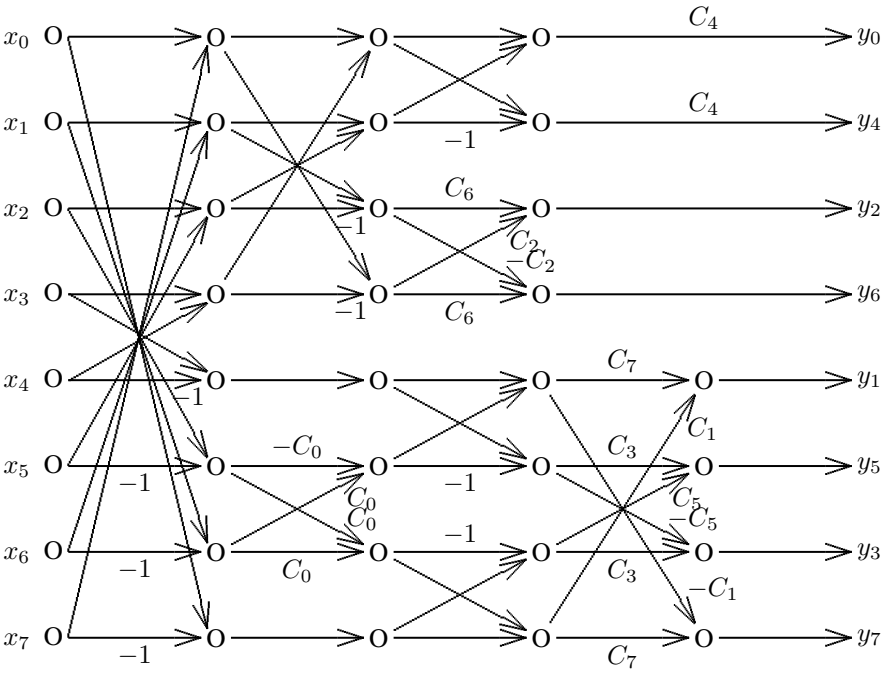

Figure 6-3. Flow Graph of 8×8 DCT using Butterfly Computation

We can obtain y_k by tracing the paths of getting to it. For example, y_2 is given by

$$\begin{aligned} y_2 &= x''_2 C_6 + x''_3 C_2 \\ &= [x'_1 - x'_2]C_6 + [x'_0 - x'_3]C_2 \\ &= [(x_1 + x_6) - (x_2 + x_5]C_6 + [(x_0 + x_7) - (x_3 + x_4)]C_2 \end{aligned}$$

The following piece of C/C++ code shows how we can compute $y'_k s$ from $x'_k s$. The first for-loop does the butterfly computations $(x_i \pm x_j)$, which are the first stage operations of the flow graph shown in Figure 6-3. It then performs the eight operations of the second stage of Figure 6-3. Next, the code does the calculations of the upper-half third stage, which is also the final stage of the upper-half flow graph. The lower-half has a total of four stages; the remaining code carries out the transforms of the third and fourth stages of the lower-half flow graph of Figure 6-3:

```
-------------------------------------------------------------------------
   for (j = 0; j < 4; j++) {  //1st stage transform, see flow-graph
        j1 = 7 - j;
```

```
        x1[j] = x[j] + x[j1];
        x1[j1] = x[j] - x[j1];
    }
    x[0] = x1[0] + x1[3];       //second stage transform
    x[1] = x1[1] + x1[2];
    x[2] = x1[1] - x1[2];
    x[3] = x1[0] - x1[3];
    x[4] = x1[4];
    x[5] = ( x1[6] - x1[5] ) * C0;
    x[6] = ( x1[6] + x1[5]) * C0
    x[7] = x1[7];

    y[0] = ( x[0] + x[1] )*C4; //upper-half of 3rd (final) stage,
    y[4] = ( x[0] - x[1] )*C4; //  see flow-graph
    y[2] = x[2] * C6 + x[3] * C2;
    y[6] = x[3] * C6 - x[2] * C2;

    x1[4] = x[4] + x[5];        //lower-half of third stage
    x1[5] = x[4] - x[5];
    x1[6] = x[7] - x[6];
    x1[7] = x[7] + x[6];

    y[1] = x1[4] * C7 + x1[7] * C1;//lower-half of 4th (final) stage
    y[7] = x1[7] * C7 - x1[4] * C1;
    y[5] = x1[5] * C3 + x1[6] * C5;
    y[3] = x1[6] * C3 - x1[5] * C5;
-------------------------------------------------------------------------
```

This code works a lot faster than the direct implementation of DCT; most practical video coders use a similar implementation. However, for the coder to be commercially competitive, we need to go one step further. We have to use integer-arithmetic, which makes further significant speed improvement of the calculations.

6.5 Integer Arithmetic

In practice, we use integer arithmetic to implement a Fast DCT. To see how this works, without loss of generality, let us consider 16-bit integers and consider a simple example with a number $x = 2.75$. We cannot express this number as an integer directly, but we can imagine that there is a binary point at the right side of bit 0 of a 16-bit integer. In this imaginary format, x can be represented as a bit vector with imaginary digits (i.e. $2.75 = 2^1 + 2^{-1} + 2^{-2}$):

$$\begin{array}{lcccccccccccccccccc} \textit{bit position}: & 15 & & & & & & & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ x = & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & . & 1 & 1 \end{array} \tag{6.24}$$

In (6.24), the 16-bit integer does not contain the two '1' bits to the right of the binary point, which is the fractional part of x. To retain the information of the fractional part, we can shift the whole number left by 8 bits. Shifting a number left by 8 bits is the same as multiplying it by $2^8 (= 256)$. After this shift, the binary point will be at the position between bit 8 and bit 7; the number $x (= 2.75)$

becomes an integer x' as shown below:

$$\begin{array}{rcccccccccccccccc} \textit{bit position}: & 15 & & & & & & 9 & 8 & & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ x' = & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & . & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{6.25}$$

The value of x now becomes

$$x' = 2^9 + 2^7 + 2^6 = 704$$

which is the same as $2.75 \times 256 = 704$. If we now divide x' by 256, or right-shift it by 8 bits, we obtain the integer 2; that is, we have truncated the fraction part of 2.75. In many cases, we would prefer a round operation than a truncate. Rounding 2.75 gives us 3. The rounding result can be achieved by first adding 0.5 to 2.75 before the truncation. If we express 0.5 in our imaginary format and left-shift it by 8 bits, we obtain the value $128(= 2^7)$. So adding 0.5 to x corresponds to the operation of adding 2^7 to x'. This is illustrated in the following equations, where $x'' = x' + 0.5 \times 2^7$:

$$\begin{array}{rcccccccccccccccc} \textit{bit position} & : & 15 & & & & & & 9 & 8 & & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ x' & = & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & . & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ +0.5 \times 2^7 & = & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & . & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x'' & = & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & . & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{6.26}$$

Obviously, when we right-shift x'' by 8 bits, we obtain our desired value 3. In general, we can transform real positive numbers to integers by multiplying the real numbers by 2^n; rounding effect is achieved by adding the value 2^{n-1} to the results before shifting them right n bits. This is true for positive numbers but *will it be also true for negative numbers? Should we still add 0.5 to the negative number or should we subtract 0.5 from it before the truncation?* The following example sheds light on what we should do.

Most modern-day computers use two's complement to represent negative numbers. Binary numbers that can have negative values are referred to as signed numbers, otherwise they are referred to as unsigned numbers. In two's complement representation, if we right-shift an unsigned number one bit, a 0 is always shifted into the leftmost bit position. However, if we right-shift a negative signed-number, a 1 is shifted in. For example, consider the following piece of code:

```
-------------------------------------------------------------------------
int main()
{
  char sc = 0x82;         //8-bit signed number
  unsigned uc = 0x82;     //8-bit unsigned number

  sc >>= 1;               //right-shift one bit
  uc >>= 1;               //right-shift one bit

  printf("\nsigned shift:   0x%x", sc );
  printf("\nunsigned shift: 0x%x", uc );
}
-------------------------------------------------------------------------
```

When executed, the program will produce the following outputs:

```
signed shift:   0xc1
unsigned shift: 0x41
```

The outputs show that a 1 has been shifted into *sc* (signed char) and a 0 has been shifted into *uc* (unsigned char). Because of this property, in C/C++ programming, integer-division of a negative signed-number by 2^n is different from right-shifting it by n bits. For example,

$$-1/2 = 0 \qquad but \;\; -1 >> 1 \;\; yields \;\; -1$$

$$-3/2 = -1 \qquad but \;\; -3 >> 1 \;\; yields \;\; -2$$

Now consider the negative signed-number $y = -2.75$. Its 2s complement representation can be obtained by complementing all bits of the corresponding positive number (2.75) shown in (6.24) and adding 1 to the rightmost bit of the complemented number. Thus it has the following binary form.

$$\begin{array}{rccccccccccccccccccc} bit\ position: & 15 & & & & & & & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & & \\ y = & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & . & 0 & 1 \end{array} \tag{6.27}$$

When we round -2.75, we would like to obtain a value of -3 rather than -2 as the former is closer to its real value. Suppose we perform the same operations that we did to positive numbers discussed above, shifting it left 8 bits, adding 0.5×2^7 and then right shifting 8 bits. This situation is illustrated by following equations where $y'' = y' + 0.5 \times 2^7$.

$$\begin{array}{rcccccccccccccccc} bit\ position & : & 15 & & & & & & 9 & 8 & & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\ y' & = & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & . & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ +0.5 \times 2^7 & = & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & . & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ y'' & = & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & . & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{6.28}$$

When we right-shift y'' 8 bits, we obtain the following.

$$\begin{array}{rcccccccccccccccccc} bit\ position & : & 15 & & & & & & & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & & \\ y'' >> 8 & = & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & . & 1 & 1 \end{array} \tag{6.29}$$

In (6.29), $y'' >> 8$ has a value of -3 (in 2s complement representation) and is what we want. Therefore, we conclude that to obtain the rounding effect, we always add 0.5 (= 0.1 in binary) to the number before truncating (right-shifting) for both positive and negative numbers.

When we add two numbers using integer arithmetic, we must align the binary points of the two numbers before addition. If the first number has been left-shifted by n bits, the second one must also be shifted by the same amount. In our implementation of Fast DCT, we use 32-bit integers; all the coefficients C_i are pre-multiplied by $1024 (= 2^{10})$. After the calculations, we right-shift the integers by the same number of bits to obtain the final results. The code of the implementation of Fast DCT is listed below:

Program Listing 6-2 . Integer-arithmetic Implementation of Fast DCT

```
//-------------------------------------------------------------------------
#include <string.h>
#include <stdio.h>
#include <math.h>

#define PI 3.141592653589
const short shift = 10; //10 bits precision: fixed-point arithmetic
```

```
const short shift1 = 2*shift;//at final stage,values shifted twice
const int fac = 1 << shift;  //multiply all constants by 2^10(=1024)
const int delta = 1 << (shift-1);//rounding adjustment ~ 0.5x2^10
const int delta1=1<<(shift1-1);//final rounding adjustment~0.5x2^20
const double a = PI / 16.0;    //angle theta

//DCT constants; use integer-arithmetic.
const int c0 = (int) ( 1 / sqrt ( 2 ) * fac );
const int c1 = (int) ( cos ( a ) / 2 * fac );
const int c2 = (int) ( cos ( 2*a ) / 2 * fac );
const int c3 = (int) ( cos ( 3*a ) / 2 * fac );
const int c4 = (int) ( cos ( 4*a ) / 2 * fac );
const int c5 = (int) ( cos ( 5*a ) / 2 * fac );
const int c6 = (int) ( cos ( 6*a ) / 2 * fac );
const int c7 = (int) ( cos ( 7*a ) / 2 * fac );

/*
  DCT function.
  Input: X, array of 8x8, containing data with values in [0, 255].
  Ouput: Y, array of 8x8 DCT coefficients.
*/
void dct(int *X, int *Y)
{
   int   i, j, j1, k;
   int x[8], x1[8], m[8][8];


   /*
     Row transform
     i-th row, k-th element
   */
   for (i = 0, k = 0; i < 8; i++, k += 8) {
      for (j = 0; j < 8; j++)
        x[j] = X[k+j];      //data for one row

      for (j=0; j < 4; j++){//first stage transform, see flow-graph
        j1 = 7 - j;
        x1[j] = x[j] + x[j1];
        x1[j1] = x[j] - x[j1];
      }
      x[0] = x1[0] + x1[3]; //second stage transform
      x[1] = x1[1] + x1[2];
      x[2] = x1[1] - x1[2];
      x[3] = x1[0] - x1[3];
      x[4] = x1[4];   //after multiplication,add delta for rounding
      //shift-right to undo 'x fac' to line up binary
      //  points of all x[i]
      x[5] = ((x1[6] - x1[5]) * c0 + delta) >> shift;
      x[6] = ((x1[6] + x1[5]) * c0 + delta ) >> shift;
      x[7] = x1[7];

      m[i][0] = (x[0] + x[1])*c4;//upper-half of 3rd (final) stage,
      m[i][4] = (x[0] - x[1]) * c4;    //  see flow-graph
      m[i][2] = x[2] * c6 + x[3] * c2;
      m[i][6] = x[3] * c6 - x[2] * c2;
```

```
    x1[4] = x[4] + x[5];             //lower-half of 3rd stage
    x1[5] = x[4] - x[5];
    x1[6] = x[7] - x[6];
    x1[7] = x[7] + x[6];

    m[i][1] = x1[4] * c7 + x1[7] * c1;//lower-half of 4th stage
    m[i][7] = x1[7] * c7 - x1[4] * c1;
    m[i][5] = x1[5] * c3 + x1[6] * c5;
    m[i][3] = x1[6] * c3 - x1[5] * c5;
} //for i

/*
   At this point, coefficients of each row (m[i][j]) has been
   multiplied by 2^10. We can undo the multiplication by <<10
   here before doing the column transform. However, as we are
   using int variables, which are 32-bit to do multiplications,
   we can tolerate another multiplication of 2^10. So we delay
   our undoing until the end of the vertical transform and we
   undo all left-shift operations by shifting the results right
   20 bits ( i.e. << 2 * 10 ).
*/
// Column transform
for (i = 0; i < 8; i++) {              //eight columns

   //consider one column
   for (j = 0; j < 4; j++) {           //first-stage operation
     j1 = 7 - j;
     x1[j] = m[j][i] + m[j1][i];
     x1[j1] = m[j][i] - m[j1][i];
   }
                                       //second-stage operation
   x[0] = x1[0] + x1[3];
   x[1] = x1[1] + x1[2];
   x[2] = x1[1] - x1[2];
   x[3] = x1[0] - x1[3];
   x[4] = x1[4];
   //undo one shift for x[5], x[6] to avoid overflow
   x1[5] = (x1[5] + delta) >> shift;
   x1[6] = (x1[6] + delta) >> shift;

   x[5] = (x1[6] - x1[5]) * c0;
   x[6] = (x1[6] + x1[5]) * c0;
   x[7] = x1[7];

   m[0][i] = (x[0] + x[1])*c4;//upper-half of 3rd (final) stage
   m[4][i] = (x[0] - x[1])*c4;//  see flow-graph
   m[2][i] = ( x[2] * c6 + x[3] * c2 );
   m[6][i] = ( x[3] * c6 - x[2] * c2 );

   x1[4] = x[4] + x[5];       //lower-half of third stage
   x1[7] = x[7] + x[6];
   x1[5] = x[4] - x[5];
   x1[6] = x[7] - x[6];
```

```
      m[1][i] = x1[4] * c7 + x1[7] * c1;//lower-half of 4th stage
      m[5][i] = x1[5] * c3 + x1[6] * c5;
      m[3][i] = x1[6] * c3 - x1[5] * c5;
      m[7][i] = x1[7] * c7 - x1[4] * c1;
    } // for i

  //we have left-shift (multiplying constants) twice
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      *Y++ =  (m[i][j] + delta1) >> shift1;//round by adding delta
    }
  }
}
-------------------------------------------------------------------------
```

In Listing 6-2, constant *shift* has value 10 and *shift1* has value 20 which are used for shifting values. Constant *fac* is obtained by left-shift 1 by 10 and thus has a value of 1024. Integer constants $c0, c1, c2, c3, c4, c5, c6, c7$ correspond to coefficients $C'_k s$ in (6.23) multiplied by 1024. Constants *delta* and *delta1* are used for rounding adjustments as explained above. In the second stage transform, in calculating $x[5]$ and $x[6]$, we have multiplied $x1[5]$ and $x1[6]$ by $c0$. However, no multiplication of any ci is involved in other $x[i]$'s. Therefore, to line up the binary point of all $x[i]$'s including $x[5]$ and $x[6]$, we have to right-shift the intermediate results of $x[5]$ and $x[6]$ by 10 bits, i.e.

$$\begin{aligned} x1[5] &= (x1[5] + delta) >> shift; \\ x1[6] &= (x1[6] + delta) >> shift; \end{aligned} \tag{6.30}$$

At the end, because we have multiplied the constants ci's twice in the intermediate calculations, we must undo the corresponding shifting operations by right-shifting the intermediate results by $shift1$ (= 20). Shifting an integer 20 bits is equivalent to shifting it 10 bits twice.

The code in Listing 6-2 can be used to do Fast DCT of a practical video compression application.

6.6 Inverse DCT (IDCT) Implementation

Once we have written the code for DCT, the implementation of IDCT becomes easy. We just need to reverse the steps in the DCT program. In the flow graph shown in Figure 6-3, when we traverse from left to right, we obtain the DCT; if we traverse from right to left, we obtain the IDCT. To understand why this is so, lets examine a butterfly computation, where we obtain (c, d) from (a, b). If we reverse the direction of the butterfly flow-graph, going from right to left, we recover (a, b) from (c, d) by

$$\begin{aligned} a &= (c + d)/2 \\ b &= (c - d)/2 \end{aligned} \tag{6.31}$$

Figure 6-4 shows the reversed butterfly except that we have suppressed writing the constant $\frac{1}{2}$ in the diagram.

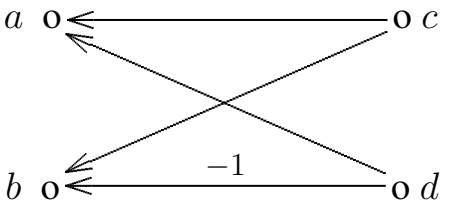

Figure 6-4. Flow Graph of Reversed Butterfly Computation

Basically, (6.23) and (6.31) have the same form. If we had multiplied the right side of (6.23) by $\frac{1}{\sqrt{2}}$ and solved for c, and d to obtain (6.31), then the forms of (6.23) and (6.31) would become identical; there's no difference in going forward (from left to right) and going backward (from right to left) in the flow graph.

Another case shown in the flow graph of Figure 6-3 is of the form

$$\begin{aligned} c &= aC_i - bC_j \\ d &= aC_j + bC_i \end{aligned} \tag{6.32}$$

where $C_k = cos(k\theta)$, and $\theta = \frac{\pi}{16}$. (If the "minus" operation occurs in the second rather than the first equation of (6.32), we can simply interchange the roles of c and d to make it look the same as (6.32).) It turns out that in (6.32) we always have $i + j = 8$. Therefore,

$$C_j = cos(j\theta) = cos(\frac{j\pi}{16}) = cos(\frac{(8-i)\pi}{16}) = cos(\frac{\pi}{2} - i\theta) = sin(i\theta) \tag{6.33}$$

and we can express the equations of (6.32) in a matrix form as shown below.

$$\begin{pmatrix} c \\ \\ d \end{pmatrix} = \begin{pmatrix} cos(i\theta) - sin(i\theta) \\ \\ sin(i\theta) + cos(i\theta) \end{pmatrix} \begin{pmatrix} a \\ \\ b \end{pmatrix} \tag{6.34}$$

You may now recognize that the equations in (6.34) represent a rotation of $i\theta$ on a plane about the origin, which rotates the point (a, b) to the point (c, d). The inverse of such a transformation is a rotation of $-i\theta$ which will bring the point (c, d) back to (a, b). That is,

$$\begin{pmatrix} a \\ \\ b \end{pmatrix} = \begin{pmatrix} cos(i\theta) + sin(i\theta) \\ \\ -sin(i\theta) + cos(i\theta) \end{pmatrix} \begin{pmatrix} c \\ \\ d \end{pmatrix} \tag{6.35}$$

where we have applied $sin(-i\theta) = -sin(i\theta)$ and $cos(-i\theta) = cos(i\theta)$ in the evaluation. Thus, we can obtain the inverse of (6.32) by simply changing the sign of C_j. That is, we replace any $-C_j$ by C_j and the corresponding C_j by $-C_j$. So the inverse of (6.32) is given by

$$\begin{aligned} a &= cC_i + dC_j \\ d &= -cC_j + dC_i \end{aligned} \tag{6.36}$$

By doing this substitution for all $-C_j's$ in Figure 6-3, we can obtain the flow graph for IDCT. Correspondingly, in developing the IDCT program, we start with the DCT code, starting from the bottom

of the DCT function, and working our way up with the replacing strategy; the resulted code is the IDCT function. We list the complete program, **dct_video.cpp** that implements both DCT and IDCT below; they are straightforward implementations of what we have discussed. The functions are ready for use for video compression.

Program Listing 6-3 DCT and IDCT Integer Implementation

```
-------------------------------------------------------------------------
/*
  dct_video.cpp
  An implementation of 8x8 DCT and IDCT for video compression.
  32-bit integer-arithmetic is assumed.
  For DCT operation:
     dct ( int *X, int *Y );
     X is the input pointing to an 8x8 array of samples; values
       must be within [0, 255]
     Y is the output pointing to an 8x8 array of DCT coefficients.
  For IDCT operation:
     idct ( int *Y, int *X );
     Y is the input pointing to an 8x8 array of DCT coefficients.
     X is the output pointing to an 8x8 array of sample values.

  compile: g++ -c dct_video.cpp
  To use them, include the the following header statements in your
  application.
      void dct(int *X, int *Y);
      void idct(int *Y, int *X);
*/

#include <string.h>
#include <stdio.h>
#include <math.h>

#define PI 3.141592653589
const short shift = 10;               //10 bits precision
//at the final stage,values have been shifted twice
const short shift1 = 2 * shift;
const int fac = 1 << shift ;     //multiply all constants by 2^10
const int delta = 1 << (shift-1);//for rounding adjustment~0.5x2^10
const int delta1 = 1 << (shift1-1);//final rounding adjust~0.5x2^20
const double a = PI / 16.0;        //angle theta

//DCT constants; use integer-arithmetic.
const int c0 = (int) ( 1 / sqrt ( 2 ) * fac );
const int c1 = (int) ( cos ( a ) / 2 * fac );
const int c2 = (int) ( cos ( 2*a ) / 2 * fac );
const int c3 = (int) ( cos ( 3*a ) / 2 * fac );
const int c4 = (int) ( cos ( 4*a ) / 2 * fac );
const int c5 = (int) ( cos ( 5*a ) / 2 * fac );
const int c6 = (int) ( cos ( 6*a ) / 2 * fac );
const int c7 = (int) ( cos ( 7*a ) / 2 * fac );

/*
  DCT function.
  Input: X, array of 8x8, containing data with values in [0, 255].
```

```
  Ouput: Y, array of 8x8 DCT coefficients.
*/
void dct(int *X, int *Y)
{
   int   i, j, j1, k;
   int x[8], x1[8], m[8][8];

   /*
     Row transform
     i-th row, k-th element
   */
   for (i = 0, k = 0; i < 8; i++, k += 8) {
      for (j = 0; j < 8; j++)
        x[j] = X[k+j];                    //data for one row

      for (j = 0; j < 4; j++) {           //first stage transform
        j1 = 7 - j;
        x1[j] = x[j] + x[j1];
        x1[j1] = x[j] - x[j1];
      }
      x[0] = x1[0] + x1[3];               //second stage transform
      x[1] = x1[1] + x1[2];
      x[2] = x1[1] - x1[2];
      x[3] = x1[0] - x1[3];
      x[4] = x1[4];  //after multiplication,add delta for rounding
      //shift to line up binary points
      x[5] = ((x1[6] - x1[5]) * c0 + delta ) >> shift;
      x[6] = ((x1[6] + x1[5]) * c0 + delta ) >> shift;
      x[7] = x1[7];

      m[i][0] = (x[0] + x[1])*c4; //upper-half of 3rd (final) stage
      m[i][4] = (x[0] - x[1]) * c4;
      m[i][2] = x[2] * c6 + x[3] * c2;
      m[i][6] = x[3] * c6 - x[2] * c2;

      x1[4] = x[4] + x[5];        //lower-half of third stage
      x1[5] = x[4] - x[5];
      x1[6] = x[7] - x[6];
      x1[7] = x[7] + x[6];

      m[i][1] = x1[4] * c7 + x1[7] * c1; //lower-half of 4th stage
      m[i][7] = x1[7] * c7 - x1[4] * c1;
      m[i][5] = x1[5] * c3 + x1[6] * c5;
      m[i][3] = x1[6] * c3 - x1[5] * c5;
   } //for i

   /*
      At this point, coefficients of each row ( m[i][j] ) has
      been multiplied by 2^10. We can undo the multiplication
      by << 10 here before doing the vertical transform. However,
      as we are using int variables, which are 32-bit to do
      multiplications, we can tolerate another multiplication of
      2^10. So we delay our undoing until the end of the vertical
      transform and we undo all left-shift operations by shifting
      the results right 20 bits ( i.e. << 2 * 10 ).
```

```
   */
   // Column transform
   for (i = 0; i < 8; i++) {              //eight columns

      //consider one column
      for (j = 0; j < 4; j++) {           //first-stage operation
        j1 = 7 - j;
        x1[j] = m[j][i] + m[j1][i];
        x1[j1] = m[j][i] - m[j1][i];
      }
                                          //second-stage operation
      x[0] = x1[0] + x1[3];
      x[1] = x1[1] + x1[2];
      x[2] = x1[1] - x1[2];
      x[3] = x1[0] - x1[3];
      x[4] = x1[4];
      //undo one shift for x[5], x[6] to avoid overflow
      x1[5] = (x1[5] + delta) >> shift;
      x1[6] = (x1[6] + delta) >> shift;

      x[5] = (x1[6] - x1[5]) * c0;
      x[6] = (x1[6] + x1[5]) * c0;
      x[7] = x1[7];

      m[0][i] = (x[0] + x[1])*c4;//upper-half of 3rd (final) stage,
      m[4][i] = (x[0] - x[1])*c4;//  see flow-graph
      m[2][i] = ( x[2] * c6 + x[3] * c2 );
      m[6][i] = ( x[3] * c6 - x[2] * c2 );

      x1[4] = x[4] + x[5];       //lower-half of third stage
      x1[7] = x[7] + x[6];
      x1[5] = x[4] - x[5];
      x1[6] = x[7] - x[6];

      m[1][i] = x1[4] * c7 + x1[7] * c1;//lower-half of 4th stage
      m[5][i] = x1[5] * c3 + x1[6] * c5;
      m[3][i] = x1[6] * c3 - x1[5] * c5;
      m[7][i] = x1[7] * c7 - x1[4] * c1;
   } // for i

  //we have left-shift (multiplying constants) twice
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      *Y++ = (m[i][j] + delta1) >> shift1;//round by adding delta
    }
  }
}

/*
  Implementation of idct() is to reverse the operations of dct().
  We first do vertical transform and then horizontal;this is easier
  for debugging as the operations are just the reverse of those in
  dct(); of course, it works just as well if you do the horizontal
  transform first. So in this implementation, the first stage of
  idct() is the final stage of dct() and the final stage of idct()
```

```
  is the first stage of dct().
*/

void idct(int *Y, int *X)
{
   int   j1, i, j, k;
   int x[8], x1[8], m[8][8], y[8];
   k = i = 0;

  //column transform
  for ( i = 0; i < 8; ++i ) {
    for (j = 0; j < 8; j++)
        y[j] = Y[i+8*j];

    x1[4] = y[1] * c7 - y[7] * c1;  //lower-half final stage of dct
    x1[7] = y[1] * c1 + y[7] * c7;
    x1[6] = y[3] * c3 + y[5] * c5;
    x1[5] = -y[3] * c5 + y[5] * c3;

    x[4] = ( x1[4] + x1[5] );      //lower-half of 3rd stage of dct
    x[5] = ( x1[4] - x1[5] );
    x[6] = ( x1[7] - x1[6] );
    x[7] = ( x1[7] + x1[6] );

    x1[0]=(y[0] + y[4])*c4;//upper-half of 3rd (final) stage of dct
    x1[1] = ( y[0] - y[4] ) * c4;
    x1[2] = y[2] * c6 - y[6] * c2;
    x1[3] = y[2] * c2 + y[6] * c6;


    x[0] = ( x1[0] + x1[3] );             //second stage of dct
    x[1] = ( x1[1] + x1[2] );
    x[2] = ( x1[1] - x1[2] );
    x[3] = ( x1[0] - x1[3] );


    //x[4], x[7] no change
    //after multiplication, add delta for rounding
    // shift-right to undo 'x fac'  to line up x[]s
    x1[5] = ((x[6] - x[5]) * c0 + delta ) >> shift;
    x1[6] = ((x[6] + x[5]) * c0 + delta ) >> shift;
    x[5] = x1[5];
    x[6] = x1[6];

    for (j = 0; j < 4; j++) {    //first stage transform of dct
        j1 = 7 - j;
        m[j][i] = (x[j] + x[j1] );
        m[j1][i] = (x[j] - x[j1] );
    }
  } //for i

   //row transform
  for ( i = 0; i < 8; i++ ) {
    for (j = 0; j < 8; j++)
        y[j] = m[i][j] ;         //data for one row
```

```
    x1[4] = y[1] * c7 - y[7] * c1;
    x1[7] = y[1] * c1 + y[7] * c7;
    x1[6] = y[3] * c3 + y[5] * c5;
    x1[5] = -y[3] * c5 + y[5] * c3;

    x[4] = ( x1[4] + x1[5] );    //lower-half of third stage
    x[5] = ( x1[4] - x1[5] );
    x[6] = ( x1[7] - x1[6] );
    x[7] = ( x1[7] + x1[6] );

    x1[0] = ( y[0] + y[4] ) * c4;
    x1[1] = ( y[0] - y[4] ) * c4;
    x1[2] = y[2] * c6 - y[6] * c2;
    x1[3] = y[2] * c2 + y[6] * c6;

    //undo one shift for x[5], x[6] to avoid overflow
    x1[5] = (x[5] + delta) >> shift;
    x1[6] = (x[6] + delta) >> shift;
    x[5] = (x1[6] - x1[5]) * c0;
    x[6] = (x1[6] + x1[5]) * c0;
      //x[4], x[7] no change
    x[0] = ( x1[0] + x1[3] );    //second stage transform
    x[1] = ( x1[1] + x1[2] );
    x[2] = ( x1[1] - x1[2] );
    x[3] = ( x1[0] - x1[3] );

    for (j = 0; j < 4; j++) {    //first stage transform
        j1 = 7 - j;
        m[i][j] = (x[j] + x[j1]);
        m[i][j1] = (x[j] - x[j1]);
      }
    }
    //we have left-shift (multiplying constants) twice
    for ( i = 0; i < 8; ++i ) {
      for ( j = 0; j < 8; ++j ) {
        *X++ = (m[i][j] + delta1) >> shift1;//round by adding delta
      }
    }
}
-------------------------------------------------------------------------
```

We can compile **dct_video.cpp** of Listing 6-3 using the command,

```
g++ -c dct_video.cpp
```

which generates the object module **dct_video.o** that can be linked to an application. We provide **test_dct.cpp** that can be downloaded from the web site of this book for you to test the DCT and IDCT routines of **dct_video.cpp**. Note that the sample values used in testing must be within the range [0, 255].

The following command links **dct_video.o** with **test_dct.o**, the object module of **test_dct.cpp** to generate the executable **test_dct**; the option "-lm" means to link with the standard maths library.

```
g++ -o test_dct dct_video.o test_dct.cpp -lm
```

The following is what will be displayed when we execute **test_dct**.

```
-------------------------------------------------------------------------
Original Data:
------------------
1, 2, 3, 4, 5, 6, 7, 8,
1, 5, 9, 13, 17, 21, 25, 29,
1, 8, 15, 22, 29, 36, 43, 50,
1, 11, 21, 31, 41, 51, 61, 71,
1, 14, 27, 40, 53, 66, 79, 92,
1, 17, 33, 49, 65, 81, 97, 113,
1, 20, 39, 58, 77, 96, 115, 134,
1, 23, 45, 67, 89, 111, 133, 155,

Data after dct:
330, -210, 0, -22, 0, -6, 0, -2,
-191, 124, 0, 13, 0, 4, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0,
-20, 13, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
-6, 4, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0,
-1, 1, 0, 1, 0, 0, 0, 0,

Data recovered by idct:
------------------
1, 2, 3, 4, 5, 6, 7, 8,
1, 5, 9, 13, 17, 21, 25, 29,
1, 8, 15, 22, 29, 36, 43, 50,
1, 11, 21, 31, 41, 51, 61, 71,
1, 14, 27, 40, 53, 66, 79, 92,
1, 17, 33, 49, 65, 81, 97, 113,
1, 20, 39, 58, 77, 96, 115, 134,
1, 23, 45, 67, 89, 111, 133, 155,
-------------------------------------------------------------------------
```

6.7 Applying DCT and IDCT to YCbCr Macroblocks

In Chapter 5, we have discussed the down sampling of an RGB image to 4:2:0 YCbCr macroblocks. A 4:2:0 YCbCr macroblock conists of four 8x8 Y sample blocks , one 8x8 Cb sample block and one 8x8 Cr sample block. It is natural to apply the Fast DCT with N = 8 discussed above to each of these sample blocks. In Chapter 5, we developed a test program ("test_encode_ppm.cpp") that reads an RGB image in PPM format, decomposes and converts it to 4:2:0 YCbCr macroblocks, and saves the YCbCr data in a ".ycc" file. Here, we go one step further. We want to develop a test program that reads the RGB data from a PPM file, converts them to YCbCr macroblocks, applies DCT to the sample blocks, and saves the DCT coefficients in a file with extension ".dct". Our ".dct" file has a format similar to that of a ".ycc" file; the first 8 bytes consists of the header text "DCT4:2:0"; the next two bytes contains the image width followed by another two bytes of image height; data start from the thirteenth byte. The test program can also reverse the process. The reversed process consists of reading a ".dct" file, applying IDCT to the data to recover the YCbCr macroblocks, converting YCbCr back to RGB and saving the RGB data in a PPM file.

To accomplish these, we need to add a few more functions in the programs **encode.cpp** and **decode.cpp** discussed in Chapter 5. Suppose we have put all functions that read and write ".ppm" and ".ycc" files presented in "test_encode_ppm.cpp" in the file "ppm-ycc.cpp" and their prototypes in

"ppm-ycc.h". The following functions are added to **encode.cpp**:

```
void save_one_dctblock ( int *Y, FILE *fpo );
void save_dct_yccblocks( YCbCr_MACRO *ycbcr_macro, FILE *fpo );
void encode_dct ( RGBImage *image, FILE *fpo );
```

These functions are shown in Listing 6-4. Their meanings are self-explained by the code.

Program Listing 6-4 Encoding an RGB Frame to DCT Coefficients

```
-------------------------------------------------------------------------
/*
  encode.cpp
  Contains functions to convert an RGB frame to YCbCr and from YCbCr
  to 16-bit DCT coefficients. It also contains functions to save
  the converted data.
  Compile: g++ -c encode.cpp
*/
#include <stdio.h>
#include <stdlib.h>
#include "../5/common.h"
#include "../5/rgb_ycc.h"

void dct(int *X, int *Y);
void idct(int *Y, int *X);

void print_dct_block ( int Y[] )
{

   for ( int i = 0; i < 64; ++i ){
      if ( i % 8 == 0 ) printf("\n");
      printf("%d,\t", Y[i] );
    }
    printf("\n-----------------");
}

void save_one_dctblock ( int Y[], FILE *fpo )
{
    short Ys[64];

    for ( int i = 0; i < 64; ++i ) //change to short for saving
      Ys[i] = ( short ) Y[i];
    fwrite ( Ys, 2, 64, fpo );    //save DCT coefficients of the
                                  //  block
}

/*
 * Apply DCT to six 8x8 sample blocks of a 4:2:0 YCbCr macroblock
 * and save the coefficients in a file pointed by fpo
 */
void save_dct_yccblocks( YCbCr_MACRO *ycbcr_macro, FILE *fpo )
{
  short block, i, j, k;
  unsigned char *py;
```

```
  int  X[64], Y[64];               //for dct transform

  //save DCT of Y
  for ( block = 0; block < 4; block++){//Y has 4 8x8 sample blocks
    if ( block < 2 )
      py=(unsigned char *)&ycbcr_macro->Y+8*block;//begin of block
    else //points to beginning of block
      py = (unsigned char *)&ycbcr_macro->Y+128+8*(block-2);
    k = 0;
    for ( i = 0; i < 8; i++ ) {    //one sample-block
      if ( i > 0 ) py += 16;       //multiply i by 16(one row)
      for ( j = 0; j < 8; j++ ) {
        X[k++] = ( int ) *( py + j );
      }
    }
    dct (  X, Y );                 //DCT transform of 8x8 block
    save_one_dctblock ( Y, fpo );  //save DCT coeffs of 8x8 block
  }
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      X[k] = ycbcr_macro->Cb[k];
      k++;
    }
  }
  dct (  X, Y );                   //DCT of Cb 8x8 sample block
  save_one_dctblock( Y, fpo );
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      X[k] = ycbcr_macro->Cr[k];
      k++;
    }
  }
  dct ( X, Y );                    //DCT of Cr 8x8 sample block
  save_one_dctblock ( Y, fpo );
}

/*
 * Convert RGB  data to YCbCr, then to DCT coeffs and
 * save DCT coeffs.
 */
void encode_dct ( RGBImage *image, FILE *fpo )
{
  short row, col, i, j, r;
  RGB *p;                          //pointer to a pixel
  RGB macro16x16[256];             //16x16 macroblock;24-bit RGB pixel
  YCbCr_MACRO ycbcr_macro;         //macroblock for YCbCr samples
  static int nframe = 0;

  for ( row = 0; row < image->height; row += 16 ) {
    for ( col = 0; col < image->width; col += 16 ) {
      //points to beginning of macroblock
      p = (RGB *)image->ibuf+(row * image->width + col);
      r = 0;                       //note pointer arithmetic
```

```
      for ( i = 0; i < 16; ++i ) {
        for ( j = 0; j < 16; ++j ) {
          macro16x16[r++] = (RGB) *p++;
        }
        p += (image->width-16); //next row within macroblock
      }
      macroblock2ycbcr ( macro16x16,  &ycbcr_macro );
      save_dct_yccblocks( &ycbcr_macro, fpo );
    } //for col
  } //for row
}
.......
( Functions already presented in Listing 5-2 are omitted here. )
-------------------------------------------------------------------------
```

In Listing 6-4, the function **encode_dct**() declares *p* as a pointer variable pointing to an RGB object (i.e., RGB *p), which contains the red, green, blue values of one pixel for a total of 3 bytes. This pointer is used to access the pixels of a 16×16 macroblock. Since the image width and height are divisible by 16, the starting position of each macroblock at row *row* and column *col* is given by:

$$\text{macroblock address} = imageAddress + row \times imagewidth + col$$

The 'address' is counted relative to the 'imageAddress' in terms of pixels. It is not measured in bytes. Therefore, when we increment *p* (i.e. ++p), it automatically advances 3 bytes (or one pixel) and points to the next pixel.

The corresponding functions we need to add to **decode.cpp** of Listing 5-2 of Chapter 5 include the following:

```
int get_one_dctblock ( int *Y, FILE *fpi );
int get_dct_yccblocks( YCbCr_MACRO *ycbcr_macro, FILE *fpi );
int decode_dct ( RGBImage &image, FILE *fpi );
```

These functions are shown in Listing 6-5. Again, their meanings are self-explained by the code.

Program Listing 6-5 Decoding DCT Coefficients to an RGB Frame

```
-------------------------------------------------------------------------
/*
  decode.cpp
  Contains functions to:
     read DCT data from a file,
     carries out IDCT to obtain YCbCr macroblocks from
       DCT coefficients,
     convert YCbCr data to RGB, and
     read YCbCr data from a file
  Compile: g++ -c decode.cpp
*/
#include <stdio.h>
#include <stdlib.h>
#include "../5/common.h"
#include "../5/rgb_ycc.h" //header developed in Chapter 5

void dct(int *X, int *Y);
```

```
void idct(int *Y, int *X);

int get_one_dctblock ( int Y[], FILE *fpi )
{
  short Ys[64];
  if ( !fread ( Ys, 2, 64, fpi ) )
        return 0;
  for ( int i = 0; i < 64; ++i )
    Y[i] = Ys[i];
  return 1;
}

/*
 * Get DCT coefficients from file, apply IDCT to obtain the
 * YCbCr macroblocks.
 */
int get_dct_yccblocks( YCbCr_MACRO *ycbcr_macro, FILE *fpi )
{
  short r, row, col, i, j, k, n, block;
  short c;
  unsigned char *py;
  int Y[64], X[64];

  n = 0;
  //read data from file and put them in four 8x8 Y sample blocks
  for ( block = 0; block < 4; block++ ) {
    if ( !get_one_dctblock( Y, fpi ) )
      return 0;
    idct ( Y, X );
    k = 0;
    if ( block < 2 )
      py = ( unsigned char * ) &ycbcr_macro->Y + 8*block;
    else
      py = (unsigned char *)&ycbcr_macro->Y + 128 + 8*(block-2);
    for ( i = 0; i < 8; i++ ){//one sample-block
      if ( i > 0 )py+=16;  //advance py by 16 (1 row of macroblock)
      for ( j = 0; j < 8; j++ ) {
        *( py + j ) = X[k++];
         n++;
      } //for j
    } //for i
  } //for block
  //now do that for 8x8 Cb block
  k = 0;
  if ( !get_one_dctblock( Y, fpi ) )
    return 0;
  idct ( Y, X );
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
       ycbcr_macro->Cb[k] = X[k];
       k++;    n++;
    }
  }

  //now do that for 8x8 Cr block
```

```
  k = 0;
  if ( !get_one_dctblock( Y, fpi ) )
    return 0;
  idct ( Y, X );
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      ycbcr_macro->Cr[k] = X[k];
      k++;    n++;
    }
  }

  return n;                       //number of bytes read
}

/*
 *   Decode DCT coeffs to a YCbCr frame and then to RGB.
 */
int decode_dct ( RGBImage &image, FILE *fpi )
{
  short r, row, col, i, j, k, block;
  int n = 0;
  RGB macro16x16[256];     //16x16 macroblock; 24-bit RGB pixel
  YCbCr_MACRO ycbcr_macro; //macroblock for YCbCr samples
  RGB *rgbp;               //pointer to an RGB pixel
  int fpos = ftell ( fpi );
  for ( row = 0; row < image.height; row += 16 ) {
    for ( col = 0; col < image.width; col += 16 ) {
      int m = get_dct_yccblocks( &ycbcr_macro, fpi );
      if ( m <= 0 ) { printf("\nout of dct data\n"); return m;}
      n += m;
      ycbcr2macroblock( &ycbcr_macro, macro16x16 );
      rgbp = (RGB *) (image.ibuf) + (row * image.width + col);
      r = 0;
      for ( i = 0; i < 16; ++i ) {
        for ( j = 0; j < 16; ++j ) {
          *rgbp++ =  macro16x16[r++];
        }
        rgbp += ( image.width - 16 ); //next row within macroblock
      }
    } //for col
  }  //for row
  return n;
}
..............
(Functions presented in Listing 5-2 are omitted here.)
-------------------------------------------------------------------------
```

Finally, the program **test_dct_ppm.cpp** of Listing 6-6 performs the tasks of DCT testing on an image file. It first reads RGB data from the testing PPM file "beach.ppm" and uses the function **encode_dct** () to convert the RGB data to 4:2:0 YCbCr macroblocks and then to 16-bit DCT coefficients, saving them in the file "beach.dct". Secondly, it uses the function **decode_dct**() to read the DCT data back from the file "beach.dct" and recover the RGB data. The recovered RGB data are saved in the PPM file "beach2.ppm". To generate an executable, we may link this file to the other object files discussed before using the following command:

```
$g++ -o test_dct_ppm test_dct_ppm.cpp ppm_ycc_dct.o \
```

```
                ../5/rgb_ycc.o encode.o decode.o dct_video.o
```

After executing the program, we can check the recovered data by issuing the command "display ../data/beach2.ppm". One can observe that the image of "beach2.ppm" is essentially identical to that of the original file, "beach.ppm". The images of these two files are shown at the end of this Chapter (Figure 6-6).

Program Listing 6-6 Testing Integer Implementation of DCT and IDCT Using a PPM Image

```
//------------------------------------------------------------------------
/*
 * test_dct_ppm.cpp
 * Program to test integer implementations of DCT, IDCT, and
 * RGB-YCbCr conversions using macroblocks. PPM files are used
 * for testing. It reads "../data/beach.ppm" RGB data, converts
 * them to 4:2:0 YCbCr macroblocks, and then to 16-bit DCT
 * coefficients which will be saved in file "../data/beach.dct".
 * The program then reads back the DCT coefficients from
 * "../data/beach.dct", performs IDCT, converts them to YCbCr
 * macroblocks and the to RGB data. The  recovered RGB data are
 * saved in "../data/beach2.ppm".  PPM files can be viewed
 * using "display".
 * Compile:  g++ -o test_dct_ppm test_dct_ppm.cpp ppm_ycc_dct.o \
 *             ../5/rgb_ycc.o encode.o decode.o dct_video.o
 * Execute: ./test_dct_ppm
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../5/common.h"
#include "ppm_ycc_dct.h"

void encode_dct ( RGBImage *image, FILE *fpo );
int decode_dct ( RGBImage &image, FILE *fpi );

int main()
{
  FILE *fp;
  int c;
  int isize;        //image size

  fp = fopen ("../data/beach.ppm", "rb");//PPM file for testing
  RGBImage image;

  ppm_read_comments ( fp );              //read comments
  char temp[100];
  fscanf ( fp, "%2s", temp );
  temp[3] = 0;
  if ( strncmp ( temp, "P6", 2 ) )
    throw ppm_error();
  ppm_read_comments ( fp );
  fscanf ( fp, "%d", &image.width );
  ppm_read_comments ( fp );
  fscanf ( fp, "%d", &image.height );
```

```
  ppm_read_comments ( fp );
  int colorlevels;
  fscanf ( fp, "%d", &colorlevels );
  ppm_read_comments ( fp );
  while ((c = getc ( fp )) == '\n');//get rid of extra line returns
  ungetc ( c ,fp );

  if ( image.width % 16 != 0 || image.height % 16 != 0 ) {
    printf("\nProgram only works for image dimensions divisible
                by 16.\n
            Use 'convert' utility to change image dimension.!\n");
    return 1;
  }

  //convert RGB data to YCbCr 4:2:0 and then to DCT coeffs; save
  //  data in "beach.dct"
  isize = image.width * image.height;
  //allocate memory to hold RGB data
  image.ibuf = ( unsigned char *) malloc (3 * isize);
  fread ( image.ibuf,  3, isize, fp );
  fclose ( fp );

  fp = fopen ( "../data/beach.dct", "wb" );
  write_dct_header ( image.width, image.height, fp );
  encode_dct ( &image, fp );   //convert to 16-bit DCT coeffs,
                               // save data in "beach.dct"
  delete image.ibuf;           //remove the image buffer
  fclose ( fp );

  //read the DCT data back from "beach.dct" and convert to RGB
  fp = fopen ( "../data/beach.dct", "rb" );
  if ( read_dct_header ( image.width, image.height, fp ) == -1 ){
    printf("\nNot DCT File\n");
    return 1;
  }
  isize = image.width * image.height;
  image.ibuf = ( unsigned char *) malloc ( 3 * isize );
  decode_dct ( image, fp );//perform IDCT, convert data back to RGB
  fclose ( fp );

  //now save the decoded data in ppm again
  fp = fopen ("../data/beach2.ppm", "wb");      //output PPM file
  int ppmh[20];                                 //PPM header*
  make_ppm_header ( ppmh, image.width, image.height );
  for ( int i = 0; i < 15; ++i )                //save PPM header
    putc ( ppmh[i], fp );
  save_ppmdata ( image, fp );                   //save RGB data
  printf("\nRecovered RGB data saved in ../data/beach2.ppm\n");
  fclose ( fp );
  delete image.ibuf;                            //deallocate memory

  return 0;
```

We may use the command “ls -l ../data/beach*” to list the sizes of the *beach* files. If you do so,

you may see something similar to the following, where the left column shows the file sizes:

```
73743  ../data/beach.ppm
73743  ../data/beach2.ppm
73740  ../data/beach.dct
36876  ../data/beach.ycc
```

We see that the file size of the DCT data is twice as large as that of the YCbCr data. This is because we have saved each DCT coefficient as a 16-bit number but each YCbCr sample value is only 8-bit. It seems that we have done something that have expanded rather than compressed the image data. Actually, the DCT is only an intermediate process. We do not really need to save any DCT coefficients. We do so here only for the purpose of testing and learning DCT. In the next chapter, we shall discuss what we shall do after the DCT step. At the moment, let us summarize what we have discussed, the encoding and decoding processes up to this point. The encoding stage consists of the following steps:

1. Converts RGB data to 4:2:0 YCbCr macroblocks. Each macroblock consists of four 8×8 Y sample blocks, one 8×8 Cb sample block, and one 8×8 Cr sample block. This step compresses the data by a factor of 2.
2. Applies DCT to each 8x8 sample block which gives an 8×8 array of 16-bit DCT coefficients. This step expands the data by a factor of 2.

On the other hand, the decoding stage consists of the following steps:

1. Applies IDCT to each 8×8 arrary of DCT coefficients to recover an 8×8 YCbCr 8-bit sample block. Reconstruct 4:2:0 YCbCr macroblocks from the sample blocks.
2. Converts YCbCr macroblocks to RGB data.

The encoding steps are shown in Figure 6-5.

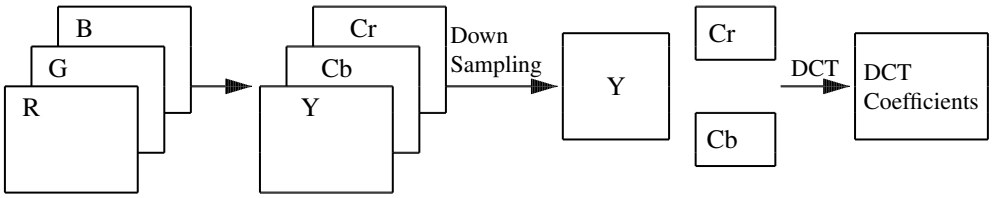

Figure 6-5. Encoding of RGB Data

Figure 6-6 Effect of down-sampling and DCT: The Original RGB Image (left) and The Restored Image (right)


../c


t.p