Computer Faults in JPEG Compression and Decompression Systems

A proposal submitted in partial fulfillment of the requirements for the qualifying exam.

Cung Nguyen
Electrical and Computer Engineering
University of California, Davis
Davis, CA 95616
cunguyen@ece.ucdavis.edu

January 20, 2002

Abstract

This proposal carries out fault tolerance design methods in JPEG image compression computing structures so that temporary failures are detected, guaranteeing that no corrupted compressed data reaches the intended user without warnings or appropriate actions. The research analyzes the specialized effects of computer failure errors and provides the design methodologies to integrate fault tolerance in JPEG standard image compression algorithms. The progress in implementing the error detection schemes applied in different stages of JPEG data compression standard is discussed. The compression/decompression stages, include the fast algorithm for Discrete Cosine Transform, quantization, differential encoding, symbol mapping and entropy coding are analyzed with enough details. Protection levels are verified by computer simulations based on the proposed architectural and data level designs. The modification of the JPEG system for error detection purposes provides fault detection without changing the standard. The error detection algorithms are included in the decoding process. The ultimate goal of this research is to influence optional features in data compression standards that insure fault tolerance capabilities for critical applications.

Contents

List of Figures

List of Tables

1 Introduction

The ultimate goal of data compression techniques is to reduce transmission bit rate or storage capacity. The great achievements in information theory are realized in practical systems for source coding and channel coding. However, source and channel coding do not address issues arising in the computer system architectures that implement these functions. Currently, source and channel coding is a subject being addressed by computer technology. There are several issues arising in the computer systems that implement these functions. First, the computing resources that execute the source and channel coding operations are subject to temporary failures. For the important source coding applications, the impact of such errors can be significant. The collection of data in medical applications must faithfully represent the measurement throughout the compression/decompression processes. In these circumstances any soft fail effects on the compressed data cannot be detected until it is recognized by human perception or by a special check procedure. One of the tasks of remote-sensing satellites is acquiring and transmitting images to the earth. Radiation such as alpha particles and cosmic rays is one source of transient and permanent faults in electronics used in space environment. An example effect is a bit-flip a so called a soft error. A single particle can upset multiple bits. These error bits can be the elements of the intermediate data image stored in memory, or that being processed in the computer registers. In any circumstance, the image will be corrupted and cannot be used at the ground stations.

Failure of a device or component in a system might cause the system to function incorrectly and fail to provide the intended service. A fault models the effect of failure on digital signals. There are two approaches for avoiding system failures: fault avoidance and fault tolerance. Fault avoidance techniques try to reduce the probability of fault occurrence, while fault tolerance techniques try to keep the system operating despite the present of faults. Since faults cannot be completely eliminated, critical systems always employ fault tolerant techniques to guarantee high reliability and availability. Fault tolerant techniques are based on redundancy. Redundancy is provided by extra components (hardware redundancy), by extra execution time (time redundancy), or by combination of both.

In this work, the JPEG compression/decompression standard for still images will be examined from a fault tolerance viewpoint. The concerns in the JPEG environment are: errors due to computational noise or the temporary intermittent failures in the hardware that could corrupt current and the subsequent code words. Such failures can lead to a great degradation of the reconstructed image.

Furthermore, quantizing operations on individual DCT coefficients that are used in JPEG have an unpredictable effect result on modified coefficients. The nature of such operations makes it quite difficult to detect errors. Retrieving bad data can result from incorrect addressing, or errors in the lossless coding or the quantization tables. These types of errors can be detected by redundancy hardware or an error checking code.

In coding of the differential DC coefficients, another part of JPEG, if a small numerical error occurs that is coupled with the Huffman coding lookup operations, the result becomes very difficult to detect. The compressing of the block of AC coefficients containing inherent runs of zeros, involves either Huffman or arithmetic coding, is other major fault tolerance challenges. The JPEG mode also complicates the protection process. When the operating mode changes, the format of ultimate binary stream at the output is also changed, and the fault detection process must be adequately developed in order to handle errors occurring under these different modes of operations. Formatting the compressed bit stream with frame headers, and markers is extremely important. It allows the decoder to recognize and carry out the proper tasks for processing the incoming stream. Sending an improperly formatted image may not be detected until the decoder begins decompressing it.

There have been meaningful results on some of the fault tolerance challenges enumerated in the proposed work. The fault protection methods are applied to the DCT, the lookup operations in Huffman coding, quantization and markers generation. These topics are central to the JPEG standard's fault tolerance capabilities. Self-checking techniques are developed specifically targeted to each stage to protect its software routines or hardware from temporary faults. By the addition of information, resources, or time beyond what is needed for normal system, the benefits for these costs are the additional capabilities in the system: more reliable, better performance in critical operating environments.

Since the proposed research will develop the Fault Tolerant techniques for the JPEG data compression, it is important to understand the JPEG standards and algorithms, determine possible errors in each stage of the compression system. The development of the hardware and software redundancy intends to protect the system from error and minimize the change of the output stream format. For error detection at the decoder, the JPEG compressed data output may contain extra information in the form of error checking codes. This codes, if occur, will be inserted into proper locations of the data stream without violation of the compression standard.

To simplify the work, we will only solve the error detection problem for basedline JPEG standard, which is 8-bit sample precision gray scale sequential Discrete Cosine Transform with Huffman coding. Other JPEG modes contain similar fault tolerant challenges. The key aspects of the proposed research follow:

- Main steps that take place in an image and in an image compression process.
- The expected input/output data format, and all possible errors due to the faults in computation, memory, or data transfer existence in each stage.
- Reaction to the errors by realizing an algorithm or a hardware subsystem to detect and possibly process the errors

Objectives (Questions to be Answered)

1. What hardware redundancy models are used in the JPEG compression system?
2. What kind of algorithms are used?
3. How to calculate any potential of errors?
4. How the fault detection system interacts with the principle system?

Overview of Progress

We have conducted an extensive overview JPEG compression system and standards, fault detection schemes, focusing on the following topics:

- Discrete Cosine Transform: fast algorithm, matrix decomposition.
- Entropy coding: Constructing tables
- Quantization: Scalar quantizer, human perception about quantization error
- Marker segment structures.
- Modeling the errors in each stage
- Error detection schemes for each stage.

2 JPEG Image Compression Standard

The following subsections provide the important information about JPEG compression system and standard

2.1 Image Overview

2.1.1 Representation of Image

In most computer displays, the screen image is composed of discrete units called pixels. Each pixel occupies a small rectangular region on the screen and displays one color at a time. The pixels are arranged so that they form a 2-dimensional array.

As the amount of disk space and memory has increased along with the speed of processors, the use of bitmap images has expanded as well. The major draw back with bitmap images is the amount of data required to hold them. An 800×600 image with 24 bits per pixel requires $1{,}440{,}000$ bytes of memory o display or disk space to store. Compression is usually used to reduce the space an image file occupies on disk, allowing large number of images to be stored.

2.1.3 $YCbCr$ Color Model

In RGB, colors are composed of three component values that represent the relative intensities of red, green, and blue. RGB is not the only color model in use. JPEG images are almost always stored using a three-component color space known as $YCbCr$. The Y, or *luminance*, component represents the intensity of the image. Cb and Cr are the *chrominance* components. Cb specifies the blueness of the image and Cr give the redness. The relation between the

$YCbCr$ and RGB models as used in JPEG is represented in the equation

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ Cb &= -0.1687R - 0.3313G + 0.5B + 2^{Sample\ Precision/2} \\ Cr &= 0.5R - 0.4187G - 0.0813B + 2^{Sample\ Precision/2} \end{aligned} \tag{1}$$

$$\begin{aligned} R &= Y + 1.402(Cr - 2^{(Sample\ Precision)/2}) \\ G &= Y - 0.34414(Cb - 2^{Sample\ Precision/2}) - 0.71414(Cr - 2^{Sample\ Precision/2}) \\ B &= Y + 1.722(Cb - 2^{Sample\ Precision/2}) \end{aligned}$$

We can see the Y component contributes the most information to the image. Unlike the RGB color model, where all components are roughly equal, $YCbCr$ concentrates the most important information in one component. This makes it possible to get greater

compression by including more data from Y component than from Cb and Cr components.

2.2 JPEG

2.2.1 Overview

This section will covers the fundamentals of JPEG compression. JPEG is an acronym for "Join Photographic Experts Group." The JPEG standard is fairly complex because it defines a number of related image compression techniques.The power of the JPEG format is that, for photographic images, it gives the greatest (up to 20 times or more) image compression of any bitmap format in common use.

Many implementations of the JPEG standard support only the basic lossy compression algorithm. The elements of this algorithm are illustrated in Figure 2

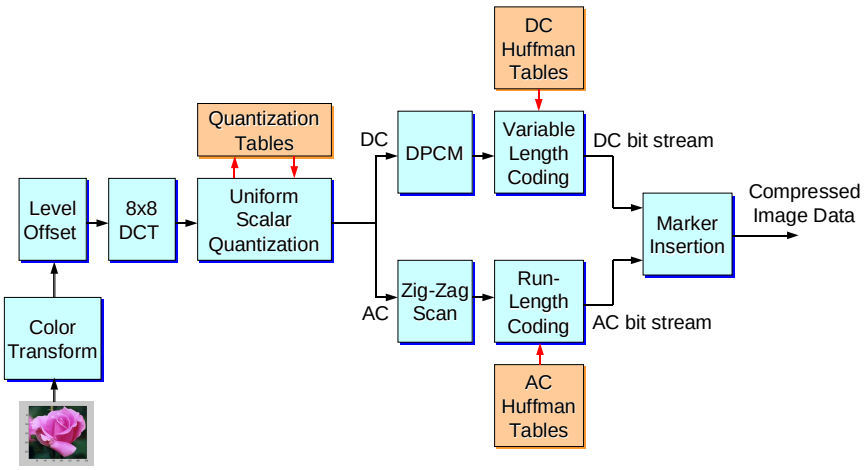


Figure 1 The basic parts of an encoder.

Unsigned image sample values are first offset to obtain a signed representation. These samples are collected into 8×8 blocks and subject to the DCT (Discrete Cosine Transform). The transform coefficients are individually quantized and then coded using variable length codes (also called Huffman codes). Various other techniques, such as DPCM and run-length coding are used to improve the compression efficiency.

The quantization and coding processes are controlled by parameter tables, which are compressor may choose to optimize for the image or application at hand. The tables must be explicitly recorded in the code-stream, regardless of whether any attempt is made to optimize these parameters. Many compressors simply use the example tables described in the standard, with no attempt at customization.

2.2.2 JPEG Compression Modes

The JPEG standard defined four compression modes: Hierarchical, Progressive, Sequential and lossless. Figure 1 shows the relationship of major JPEG compression modes and encoding processes.

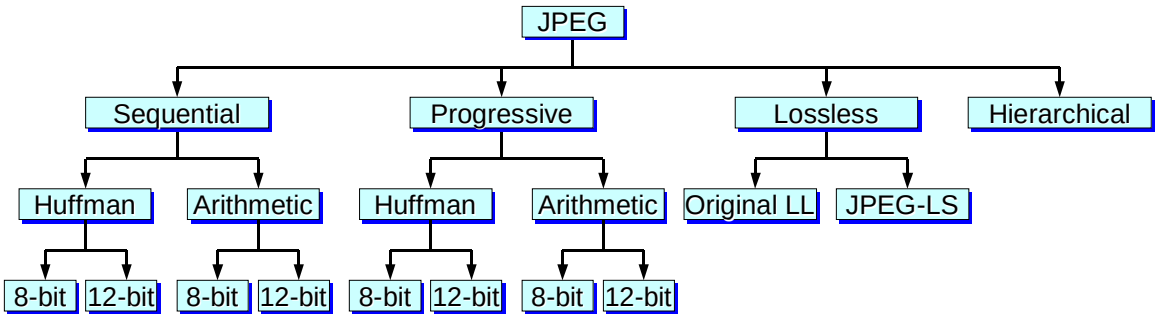


Figure 2 Elements of the basic JPEG algorithm

- Sequential
  Sequential-mode images are encoded from top to bottom. Sequential mode supports sample data with 8 and 12 bits of precision. In the sequential JPEG, each color component is completely encoded in single scan. Within sequential mode, two alternate entropy encoding processes are defined by the JPEG standard: one uses Huffman encoding; the other uses arithmetic coding.
- Progressive
  In progressive JPEG images, components are encoded in multiple scans. The compressed data for each component is placed in a minimum of 2 and as many as 896 scans. The initial scans create a rough version of the image, while subsequent scans refine it.
- Hierarchical
  Hierarchical JPEG is a super-progressive mode in which the image Is broken down into a number of subimages called frames. A frame is a collection of one or more scans. In hierarchical mode, the first frame creates a low-resolution version of image. The remaining frames refine the image by increasing the solution.

2.2.3 Baseline JPEG

Baseline JPEG decompressor supports a minimal set of features. It must be able to decompress image using *sequential DCT-based* mode. For baseline compression the bit depth must be $B=8$; however, it is convenient to describe a more general situation. The image samples are assumed to be unsigned quantities in the range $[0,\ 2^B-1]$. The level offset subtract 2^{B-1} from every sample value so as to produce signed quantities in the range $[-2^{B-1},\ 2^{B-1}-1]$. The purpose of this is to ensure that all the DCT coefficients will be signed quantities with a similar dynamic range.

The image is partitioned into blocks of size 8×8. Each block is then independently transformed using the 8×8 DCT. If the image dimensions are exact multiples of 8, the blocks on the lower and right hand boundaries may be only partially occupied. These boundary blocks must be padded to the full 8×8 block size and processed in an

identical fashion to every other block. This illustrate in Figure 3. The compressor is free to select the value used to pad partial boundary blocks.

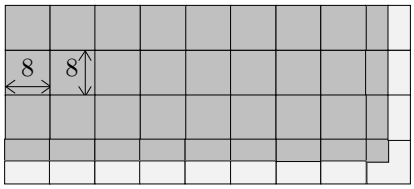


Figure 3 Image partition into 8×8 blocks for JPEG compression.

Usually, it is replicated the final row or column of actual image samples into the missing locations. Given $x_b[\mathbf{j}]\equiv x_b[j_1,j_2]$ denotes the array of level shifted sample values for the b^{th} block. Also, let $y_b[\mathbf{k}]\equiv y_b[k_1,k_2]$ denote the 8×8 array of DCT coefficients formed from these samples. The indices k_1 and k_2 represent vertical and horizontal frequency, respectively. The DCT coefficients are given by

$$y[\mathbf{k}]=\tfrac{2}{N}u(k_1)u(k_2)\sum_{j_1=0}^{N-1}\sum_{j_2=0}^{N-1}x_{j_1j_2}\cos[\tfrac{(2j_1+1)k_1}{2N}\pi]\cos[\tfrac{(2j_2+1)k_2}{2N}\pi] \tag{2}$$

where $$u(k)=\begin{cases}1/\sqrt{2}, & k=0\\ 1, & \text{otherwise}\end{cases}$$

For $N=8$, $\mathbf{k}=\mathbf{0}$, from (2) the "DC" coefficient for block b is

$$y_b[\mathbf{0}]=\frac{1}{8}\sum_{j_1=0}^{7}\sum_{j_2=0}^{7}x_b[\mathbf{j}] \tag{3}$$

so that $B+3$ bits are required to represent the integer part of the DC coefficients. The remaining 63 coefficients in each block are called "AC" coefficients.

In this proposal, we consider only one image component (monochrome image), and the blocks are processed in raster order, from left to right and from top to bottom. Each DCT coefficient is subjected to uniform scalar quantization. The quantization indices $q_b[\mathbf{k}]$, are give by.

$$q_b[\mathbf{k}]=\mathrm{sgn}(y_b[\mathbf{k}])round\left(\frac{|\,y_b[\mathbf{k}]\,|}{\Delta_{\mathbf{k}}}\right) \tag{4}$$

where $\Delta_{\mathbf{k}}$ is the quantization step size at spatial frequency $\mathbf{k}$ in block. They are collected in an 8×8 array known as a quantization table or Q-table, which is used for every DCT block. The Q-table entries must be integers in the range $[0,\ 255]$. The decompressor uses a mid-point reconstruction rule to recover approximate versions of the original DCT coefficients.

$$\hat{y}_b[\mathbf{k}]=\Delta_{\mathbf{k}}q_b[\mathbf{k}],\quad 0\le k_1,k_2\le 8 \tag{5}$$

At low bit-rates, most of the AC coefficients must be quantized to 0 since every non-zero coefficient requires at least 2 bits to code. JPEG uses a simple DPCM scheme to exploit some of the redundancy between the DC coefficients of adjacent blocks. The quantity which actually coded for block b is the difference, δ_b, between $q_b[\mathbf{0}]$ and $q_{b-1}[\mathbf{0}]$.

$$\delta_b = \begin{cases} q_b[\mathbf{0}] - q_{b-1}[\mathbf{0}], & \text{if } b > 0 \\ q_0[\mathbf{0}] & \text{if } b = 0 \end{cases} \tag{6}$$

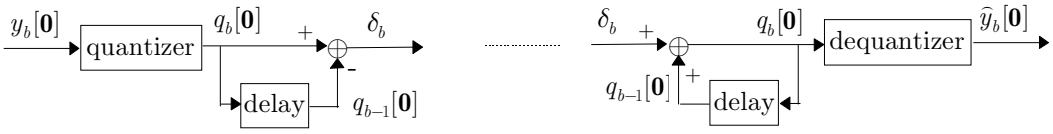


Figure 4 DPCM with quantizer outside the feedback loop

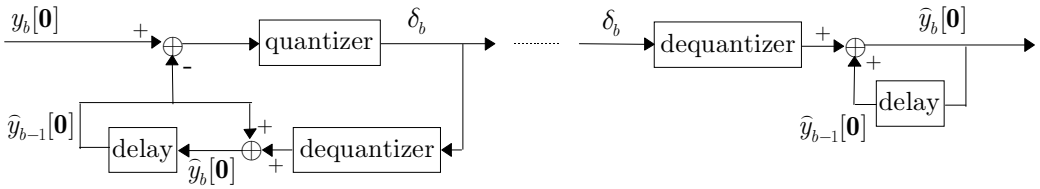


Figure 5 DPCM with quantizer inside the feedback loop

Figure 4 illustrates processing DC coefficients, $y_b[\mathbf{0}]$, at the compressor and decompressor. DPCM is a lossless tool for efficiently coding the quantization indices. Accordingly, the quantizer is not included inside the DPCM feedback loop.

The processing can be done also in a lossy DPCM structure, where the quantizer is include in the feedback loop as illustrated in Figure 5. It has the property that

$$\widehat{y}_b[\mathbf{0}] - \widehat{y}_{b-1}[\mathbf{0}] = \Delta_O \delta_b \tag{7}$$

Assume that at the initialize, $\widehat{y}_{-1}[\mathbf{0}] = 0$, we find that

$$\delta_b = round\left[\frac{y_b[\mathbf{0}] - \widehat{y}_{b-1}[\mathbf{0}]}{\Delta_o}\right] = round\left[\frac{y_b[\mathbf{0}]}{\Delta_o}\right] - \sum_{i=0}^{b-1}\delta_i \tag{8}$$

Note $\widehat{y}_{b-1}[\mathbf{0}] = \widehat{y}_{b-2}[\mathbf{0}] + \Delta_O\delta_{b-1} = \widehat{y}_{b-3}[\mathbf{0}] + \Delta_O(\delta_{b-1} + \delta_{b-2}) = \cdots = \widehat{y}_{-1}[\mathbf{0}] + \Delta_O\sum_{i=0}^{b-1}\delta_i$

It follows that

$$q_b[\mathbf{0}] = round\left[\frac{y_b[\mathbf{0}]}{\Delta_O}\right] = \sum_{i=0}^{b}\delta_i \tag{9}$$

or $\delta_b = q_b[\mathbf{0}] - q_{b-1}[\mathbf{0}]$. This shows the outside and inside feedback loops are equivalent.

2.2.4 Category Codes

As described by Figures 2, 4 and 5, the symbol δ_b, which represent the DC coefficients are to be coded using a variable length code (VLC). The number of possible symbols can be very large because the integer part of the DC coefficients are $B+3$ bit quantities and the smallest quantiation step size is 1. The range of possible values for the magnitude $|\delta_b|$ is

$$0 \le |\delta_b| \le 8(2^B - 1) \le 2^{B+3} \tag{10}$$

There are approximately 2^{B+4} possible different δ_b symbols. In similar, the total number of possible AC symbols is approximately 2^{B+3}. The DC or AC symbol, s, is mapped to an ordered pair, (c,u), which we call a "category code." The "size category," c, is subjected to variable length coding, and u contains exactly c bits appended to c without coding. The c is defined to be the smallest number of bits that can represent the magnitude of s. That is $|s| \le 2^c - 1$, or

$$c = \lceil \log_2(|s|+1) \rceil \tag{11}$$

When $c=0$, s must be zero and there is no need for u. When $s>0$, u contains a 1 followed by the $c-1$ least significant bits of $|s|$. When $s<0$, u consists of a 0 followed by the one's complement of the $c-1$ least significant bits of $|s|$. Table 2 identifies 2^{15} different symbols all have similar probabilities so that little is lost by leaving them uncoded. On the other hand, the size categories can be expected to have a high non-uniform.

Baseline Encoding Example

This section gives an example of Baseline compression and encoding of a single 8x8 sample block. For right now we omit the operation of a complete JPEG baseline encoder, including creation of interchange format information (parameters, headers, quantization and Huffman tables. This example should help to make concrete much of foregoing explanation. Figure **6**(a) is an 8×8 block of 8-bit samples. After subtracting 128 from each sample for the required level-shift, the 8×8 block is input to the FDCT. Figure **6**(b)

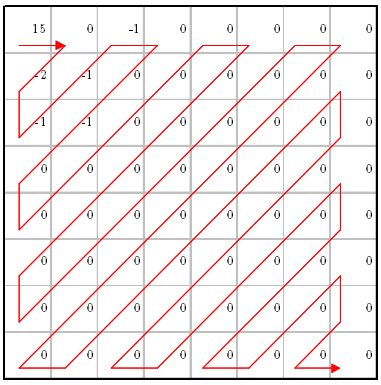
139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

235.6	-1.0	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	-1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	0.0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1.0	-1.0
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

(a) Source image samples

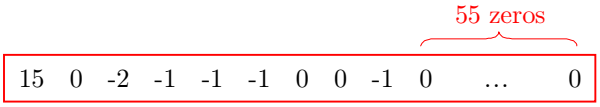
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

(b) Forward DCT coefficients

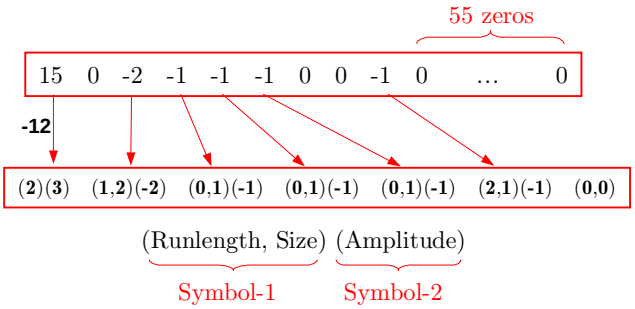


(c) Quantization table

(d) Quantized coefficients



(e) Quantized coefficients in zig-zag order



(f) Intermediate sequence of symbols

Figure 6 DCT and quantization example

shows the resulting DCT coefficients. Figure **6**(c) is the example quantization table for luminance components. Figure 6(d) shows the quantized DCT coefficients, normalized by their quantization table entries as specified in equation (**4). Assume t**he quantized DC term of the previous block is 12, then their difference is +3. Thus, the intermediate representation for the DC component is (2)(3) (use 2 bits and amplitude is 3). Figure 6(e) shows the quantized AC coefficients in zig-zag order. In Figure 6(f), the first nonzero coefficient is –2, preceded by a zero-run of 1. This yields a representation of (1,2)(-2). Next terms are the three consecutive nonzeros with amplitude –1. This means each is preceded by a zero-run of length zero. This yields the sequence (0,1)(-1) (0,1)(-1) (0,1)(-1). The last nonzero coefficient is –1 preceded by two zeros to be represented by (2,1)(-1). The final symbol representing this 8x8 block is EOB, or (0,0).

The sequence symbols is (2)(3), (1,2)(-2), (0,1)(-1), (0,1)(-1), (0,1)(-1), (2,1)(-1), (0,0)

The codes themselves must be assigned. From the Table 2, the differential-DC VLC is

(2) 110

From Table **1**, the AC luminance VLCs for this example are:

(0,0) 1010 (1,2) 11011

(0,1) 00 (2,1) 11100

The VLIs specified in section 2.2.4 are

(3) 11

(-2) 01

(-1) 0

Thus, the bit-stream for this 8x8 example block is as follows:

110 11 11011 01 00 0 00 0 00 0 11100 0 1010

We need 31 bits to represent 64 coefficients, which compression of 0.5 bits per sample

Table 1 Basedline entropy coding Symbol-1 Structure

	Runlength			
Size	Z0	Z1	Z2	Z3
C0	1010 (EOB)			
C1	00	1100	11100	111010
C2	01	11011	11111001	111110111
C3	100	1111001	1111110111	111111110101
C4	1011	111110110	111111110100	1111111110001111
C5	11010	11111110110	1111111110001001	1111111110010000
C6	1111000	1111111110000100	1111111110001010	1111111110010001
C7	11111000	1111111110000101	1111111110001011	1111111110010010
C8	1111110110	1111111110000110	1111111110001100	1111111110010011
C9	1111111110000010	1111111110000111	1111111110001101	1111111110010100
C10	1111111110000011	1111111110001000	1111111110001110	1111111110010101

	Runlength			
Size	Z4	Z5	Z6	Z7
C0				
C1	111011	1111010	1111011	11111010
C2	1111111000	11111110111	111111110110	111111110111
C3	1111111110010110	1111111110011110	1111111110100110	1111111110101110
C4	1111111110010111	1111111110011111	1111111110100111	1111111110101111
C5	1111111110011000	1111111110100000	1111111110101000	1111111110110000
C6	1111111110011001	1111111110100001	1111111110101001	1111111110110001
C7	1111111110011010	1111111110100010	1111111110101010	1111111110110010
C8	1111111110011011	1111111110100011	1111111110101011	1111111110110011
C9	1111111110011100	1111111110100100	1111111110101100	1111111110110100
C10	1111111110011101	1111111110100101	1111111110101101	1111111110110101

	Runlength			
Size	Z8	Z9	Z10	Z11
C0				
C1	111111000	111111001	111111010	1111111001
C2	111111111000000	1111111110111110	1111111111000111	1111111111010000
C3	1111111110110110	1111111110111111	1111111111001000	1111111111010001
C4	1111111110110111	1111111111000000	1111111111001001	1111111111010010
C5	1111111110111000	1111111111000001	1111111111001010	1111111111010011
C6	1111111110111001	1111111111000010	1111111111001011	1111111111010100
C7	1111111110111010	1111111111000011	1111111111001100	1111111111010101
C8	1111111110111011	1111111111000100	1111111111001101	1111111111010110
C9	1111111110111100	1111111111000101	1111111111001110	1111111111010111
C10	1111111110111101	1111111111000110	1111111111001111	1111111111011000

	Runlength			
Size	Z12	Z13	Z14	Z15
C0				11111111001 (run = 16)
C1	1111111010	11111111000	1111111111101011	1111111111110101
C2	1111111111011001	1111111111100010	1111111111101100	1111111111110110
C3	1111111111011010	1111111111100011	1111111111101101	1111111111110111
C4	1111111111011011	1111111111100100	1111111111101110	1111111111111000
C5	1111111111011100	1111111111100101	1111111111101111	1111111111111001
C6	1111111111011101	1111111111100110	1111111111110000	1111111111111010
C7	1111111111011110	1111111111100111	1111111111110001	1111111111111011
C8	1111111111011111	1111111111101000	1111111111110010	1111111111111100
C9	1111111111100000	1111111111101001	1111111111110011	1111111111111101
C10	1111111111100001	1111111111101010	1111111111110100	1111111111111110

Table 2 Based line entropy coding for Symbol s

Size c	Symbol s									Code
0	0									0
1	-1	1								10
2	-3	-2	2	3						110
3	-7	-6	-5	-4		4	5	6	7	1110
4	-15	-14	-13	-12	…	12	13	14	15	11110
5	-31	-30	-29	-28	…	28	29	30	31	111110
6	-63	-62	-61	-60	…	60	61	62	63	1111110
7	-127	-126	-125	-124	…	124	125	126	127	11111110
8	-255	-254	-253	-252	…	252	253	254	255	111111110
9	-511	-510	-509	-508	…	508	509	510	511	1111111110
10	-1023	-1022	-1021	-1020	…	1020	1021	1022	1023	11111111110
11	-2047	-2046	-2045	-2044	…	2044	2045	2046	2047	111111111110
12	-4095	-4094	-4093	-4092	…	4092	4093	4094	4095	1111111111110
13	-8191	-8190	-8189	-8188	…	8188	8189	8190	8191	11111111111110
14	-16383	-16382	-16381	-16380	…	16380	16381	16382	16383	111111111111110
15	-32767	-32766	-32765	-32764	…	32764	32765	32766	32767	1111111111111110
16	32768									1111111111111111

3 Fault Tolerant In Fast Discrete Cosine Transform

3.1 Overview of DCT

DCT based graphics compression usually employs an 8×8 DCT. For this reason, there has been extensive study of this particular DCT. Equation (12) is the one dimensional (1D) N-element DCT. Equation (13) is corresponding equation for the 1D N-element inverse DCT.

$$y_k = \alpha_k \sum_{n=0}^{N-1} x_n \cos\frac{\pi k(2n+1)}{2N}, \qquad k = 0,1,\ldots,N-1 \tag{12}$$

$$x_n = \sum_{k=0}^{N-1} \alpha_k y_k \cos\frac{\pi k(2n+1)}{2N}, \qquad n = 0,1,\ldots,N-1 \tag{13}$$

$$\alpha_k = \begin{cases} \sqrt{1/N}, & k = 0 \\ \sqrt{2/N}, & k = 1,2,\ldots,N-1 \end{cases}$$

The DCT has the property that, for a typical image, most of the visually significant information about the image is concentrated in just a few coefficients of the DCT. For this reason, the DCT is often used in image compression applications. The DCT is at the heart of the JPEG standard lossy image compression algorithm.
The two-dimensional DCT of an M-by-N matrix **X** is defined as follows.

$$y_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x_{mn} \cos\frac{\pi(2m+1)p}{2M}\cos\frac{\pi(2n+1)q}{2N}, \quad 0 \le p \le M-1,\ 0 \le q \le N-1 \tag{14}$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases}, \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

The values y_{pq} are called the *DCT coefficients* of **X**. The DCT is an invertible transform, and its inverse is given by

$$x_{mn} = \sum_{p=0}^{M-1}\sum_{q=0}^{N-1} \alpha_p \alpha_q y_{pq} \cos\frac{\pi(2m+1)p}{2M}\cos\frac{\pi(2n+1)q}{2N}, \quad 0 \le m \le M-1,\ 0 \le n \le N-1 \tag{15}$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases}, \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

The inverse DCT equation can be interpreted as meaning that any M-by-N matrix A can be written as a sum of functions of the form

$$\alpha_p \alpha_q \cos\frac{\pi(2m+1)p}{2M}\cos\frac{\pi(2n+1)q}{2N}, \qquad 0 \le p \le M-1,\ 0 \le q \le N-1 \tag{16}$$

These functions are called the *basis functions* of the DCT. The DCT coefficients, then, can be regarded as the *weights* applied to each basis function.

There were existing many fast algorithms of 1D and 2D DCTs [1]-[4]. Those algorithms achieved good improvements in computation complexities. Particularly, the algorithm [1] results arithmetic complexity of (N/2)log(N) multiplications and (3N/2)log(N)-N+1 additions for input sequence with radix 2 length ($N = 2^m$). It based on direct decomposition of the DCT. The derivation of the algorithm related to the application of the Kronecker matrix product as a construction tool. The sequential splitting method was used for proofing the correctness of the algorithm.

The approach in [3] processed the two-dimensional case directly, rather than treating it naively as a row-column implementation of the one-dimensional case. Whereas the method in [4] proposed the fast algorithm for the 2-D DCT, achieving a the purpose of reducing the hardware complexity for the parallel implementation, slight increase in time complexity. It requires N/2 1-D DCT modules and small number of multiplications.

The 2D forward and inverse transforms can be written as a of three items matrix product $\mathbf{Z}=\mathbf{AXA}^T$ and $\mathbf{X}=\mathbf{A}^T\mathbf{ZA}$, where $\mathbf{AA}^T = \mathbf{I}_N$. The decomposition to a triple matrix product requires $2N^3$ multiplications to be performed, which requires 2N multiplies to be computed per input sample.

The 2D DCT can be broken down into 1D DCT's (or IDCT's). The first computes $\mathbf{Y} = \mathbf{AX}$ (or $\mathbf{A}^T\mathbf{X}$) and the second computes $\mathbf{Z} = \mathbf{YA}^T$ (or $\mathbf{YA}$). The N×N matrix-matrix multiply has been separated into N matrix-vector products. Thus, the basic computation performed by the DCT (IDCT) is the evaluation of the product between the (N×N) matrix and the (N×1) vector. Each 1D DCT (or IDCT) unit must be capable of computing N multiplies per input sample to perform a matrix vector product. If the input block $\mathbf{X}$ is scanned column by column, the intermediate product $\mathbf{Y} = \mathbf{AX}$ (or $\mathbf{A}^T\mathbf{X}$) is also computed column by column. However, since the entire row of $\mathbf{Y}$ has to be computed prior to the evaluation of the next 1D DCT, the intermediate result $\mathbf{Y}$ must be stored in an on-chip buffer. Since columns are written into the buffer and rows are read from it, it is commonly called the *transposed memory*.

The first 1D DCT/IDCT unit operates on rows of $\mathbf{A}$ (or $\mathbf{A}^T$) and columns of $\mathbf{X}$, while the second 1D DCT unit operate on the rows of $\mathbf{A}^T$ (or $\mathbf{A}$) is equivalent to a row of $\mathbf{A}(\mathbf{A}^T)$, the second 1D DCT/IDCT is unnecessary if we can multiplex the first 1D DCT/IDCT unit between the column of $\mathbf{X}$ and the row of $\mathbf{Y}$. However, the 1D DCT/IDCT unit now processes samples at twice the input sample rate. Thus the 1D DCT/IDCT unit must be capable of computing 2N multiplies per input sample.

3.2 Fault Tolerant Model for the DCT

Fault tolerance has been a major issue for the VLSI based Discrete Cosine Transform (DCT) networks. The DCT is a unitary transform that can be expressed using matrix factorizations. The constituent factor matrices are sparse and represent simple single

computational patterns. In general $\mathbf{C}$ is an $N \times N$ unitary matrix and the input data are collected in a vector $\mathbf{x}$, the transform produces a vector $\mathbf{y}$

$$\mathbf{y} = \mathbf{C} \cdot \mathbf{x} \tag{17}$$

The unitary matrix $\mathbf{C}$ can be expanded as a product of factor matrices $\mathbf{A}_i$'s, which each corresponding to a stage of a fast algorithm. The modeling effects are achieved with a generic single error $\varepsilon_r^{(k)}$ inserted at the input of a stage. The output error pattern $v_r^{(k)}$ is a direct consequence of the specific features of the fast algorithm as encapsulated in the structure of the component matrices. We express a fast algorithm for the DCT on 8 points using the matrix factorization. By setting $C(k) = \cos(2\pi k/32)$, the 8-point DCT matrix can be simplified in the form

$$\mathbf{C}_8 = \frac{1}{2}\begin{bmatrix} C(4) & C(4) & C(4) & C(4) & C(4) & C(4) & C(4) & C(4) \\ C(1) & C(3) & C(5) & C(7) & -C(7) & -C(5) & -C(3) & -C(1) \\ C(2) & C(6) & -C(6) & -C(2) & -C(2) & -C(6) & C(6) & C(2) \\ C(3) & -C(7) & -C(1) & -C(5) & C(5) & C(1) & C(7) & -C(3) \\ C(4) & -C(4) & -C(4) & C(4) & C(4) & -C(4) & -C(4) & C(4) \\ C(5) & -C(1) & C(7) & C(3) & -C(3) & -C(7) & C(1) & -C(5) \\ C(6) & -C(2) & C(2) & -C(6) & -C(6) & C(2) & -C(2) & C(6) \\ C(7) & -C(5) & C(3) & -C(1) & C(1) & -C(3) & C(5) & -C(7) \end{bmatrix} \tag{18}$$

We will index rows of $\mathbf{C}_8$ from 0 to 7. Let $\mathbf{P}_{(8,1)}$ be the permutation matrix acting on the rows of $\mathbf{C}_8$ (or $\mathbf{P}_{(8,1)}\mathbf{C}_8$) reordering them as follows : 0, 2, 4, 6, 1, 3, 5, 7. Let $\mathbf{P}_{(8,2)}$ be the permutation matrix acting on the columns of $\mathbf{C}_8$ (or $\mathbf{C}_8\mathbf{P}_{(8,2)}$) keeping the first four columns fixed and reversing the order of the last four columns them as follows : 0, 1, 2, 3, 7, 6, 5, 4. These permutation which are given explicitly

$$\mathbf{P}_{(8,1)} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{bmatrix}, \qquad \mathbf{P}_{(8,2)} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \end{bmatrix} \tag{19}$$

$$\mathbf{P}_{(8,1)}\mathbf{C}_8\mathbf{P}_{(8,2)}\mathbf{R}_8 = \begin{bmatrix} C(4) & C(4) & C(4) & C(4) & & & & \\ C(2) & C(6) & -C(6) & -C(2) & & & & \\ C(4) & -C(4) & -C(4) & C(4) & & & & \\ C(6) & -C(2) & C(2) & -C(6) & & & & \\ & & & & C(4) & C(3) & C(5) & C(7) \\ & & & & C(3) & -C(7) & -C(1) & -C(5) \\ & & & & C(5) & -C(1) & C(7) & C(3) \\ & & & & C(7) & -C(5) & C(3) & -C(1) \end{bmatrix} \tag{20}$$

Let $\mathbf{I}_n$ be the $n \times n$ identity matrix. Let $F = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. Define $\mathbf{R}_8 = \mathbf{F} \otimes \mathbf{I}_4$ where $\otimes$ is the

Kronecker product symbol. Let $\tilde{G}_4$ be the bottom right 4×4 block

$$\text{Let } \mathbf{P}_{41} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & -\mathbf{1} \\ 0 & 0 & \mathbf{1} & 0 \end{bmatrix}, \quad \mathbf{R}_4 = \begin{bmatrix} 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 \end{bmatrix} \tag{21}$$

$$\text{we have } \mathbf{G}_4 = \mathbf{P}_{41}\tilde{\mathbf{G}}_4\mathbf{P}_{41}^{-1}\mathbf{R}_4 = \begin{bmatrix} C(3^3) & C(3^2) & C(3^1) & C(3^0) \\ -C(3^0) & C(3^3) & C(3^2) & C(3^1) \\ -C(3^1) & -C(3^0) & C(3^3) & C(3^2) \\ -C(3^2) & -C(3^1) & -C(3^0) & C(3^3) \end{bmatrix} \tag{22}$$

The matrix $\mathbf{G}_4$ can be viewed as an element in the regular representation of the matrix

G_4 can be viewed as an element in the regular representation of the polynomial ring in the variable u modulo u^4+1. The top left 4×4 block of (20) equals $\sqrt{2}$ times the 4 point DCT matrix $\mathbf{C}_4$. It yields a similar block diagonalization. Let $\mathbf{P}_{42}$ be the permutation matrix acting on the rows of $\mathbf{C}_4$ reordering them follows: 0,2,1,3.

Let $\mathbf{P}_{43}$ the permutation matrix acting on the columns of $\mathbf{C}_4$ by keeping the two first columns fixed and reversing the order of the last two columns. Let $\mathbf{R}_{41} = \mathbf{F} \otimes \mathbf{I}_4$

$$\text{We have } \mathbf{P}_{42} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} \end{bmatrix}, \quad \mathbf{P}_{43} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & \mathbf{1} & 0 \end{bmatrix}, \quad \mathbf{R}_{41} = \begin{bmatrix} \mathbf{1} & 0 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 & \mathbf{1} \\ \mathbf{1} & 0 & -\mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 & -\mathbf{1} \end{bmatrix} \tag{23}$$

$$\mathbf{P}_{42}\mathbf{C}_4\mathbf{P}_{43}\mathbf{R}_{41} = 2 \begin{bmatrix} C(4) & C(4) & & \\ C(4) & -C(4) & & \\ & & C(2) & C(6) \\ & & C(6) & -C(2) \end{bmatrix} \tag{24}$$

Let $\tilde{\mathbf{G}}_2$ be the bottom right 2×2 block of (24). We define $\mathbf{G}_2$ by reverse the order of the columns of $\tilde{\mathbf{G}}_2$

$$\mathbf{G}_2=\tilde{\mathbf{G}}_2\begin{bmatrix}0 & \mathbf{1}\\ \mathbf{1} & 0\end{bmatrix}=\begin{bmatrix}C(6) & C(2)\\ -C(2) & C(6)\end{bmatrix}\Rightarrow\tilde{\mathbf{G}}_2=\mathbf{G}_2\begin{bmatrix}0 & \mathbf{1}\\ \mathbf{1} & 0\end{bmatrix}=\mathbf{G}_2\mathbf{R}_2 \tag{25}$$

Also, the 2×2 subblock on the top left of (24) is the 2-point DCT matrix $\mathbf{C}_2$

$$\mathbf{C}_2\mathbf{F}=2\begin{bmatrix}C(4) & \\ & C(4)\end{bmatrix}\Rightarrow\mathbf{C}_2=2\mathbf{F}^{-1}\begin{bmatrix}\mathbf{G}_1 & \\ & \mathbf{G}_1\end{bmatrix} \tag{26}$$

Where $\mathbf{G}_1=[C(4)]$. Substitute $\tilde{\mathbf{G}}_2$ and $\mathbf{C}_2$ from (25) and (26) into (24) we obtain

$$\mathbf{P}_{42}\mathbf{C}_4\mathbf{P}_{43}\mathbf{R}_{41}=2\begin{bmatrix}\mathrm{C}_2 & \\ & \tilde{\mathrm{G}}_2\end{bmatrix}=2\begin{bmatrix}\mathbf{F}^{-1} & \\ & \mathbf{I}_2\end{bmatrix}\begin{bmatrix}2\mathbf{G}_1 & & \\ & 2\mathbf{G}_1 & \\ & & \mathbf{G}_2\end{bmatrix}\begin{bmatrix}\mathbf{I}_2 & \\ & \mathbf{R}_2\end{bmatrix} \tag{27}$$

We have $\begin{bmatrix}\mathbf{F}^{-1} & \\ & \mathbf{I}_2\end{bmatrix}=\begin{bmatrix}\frac{\mathbf{1}}{\mathbf{2}} & \frac{\mathbf{1}}{\mathbf{2}} & & \\ \frac{\mathbf{1}}{\mathbf{2}} & -\frac{\mathbf{1}}{\mathbf{2}} & & \\ & & \mathbf{1} & \\ & & & \mathbf{1}\end{bmatrix}$. Let $\mathbf{S}_4=\begin{bmatrix}\mathbf{1} & \mathbf{1} & & \\ \mathbf{1} & -\mathbf{1} & & \\ & & \mathbf{1} & \\ & & & \mathbf{1}\end{bmatrix}$; then (27) can be written

$$\mathbf{P}_{42}\mathbf{C}_4\mathbf{P}_{43}\mathbf{R}_{41}=2\mathbf{S}_4\begin{bmatrix}\mathbf{G}_1 & & \\ & \mathbf{G}_1 & \\ & & \mathbf{G}_2\end{bmatrix}\begin{bmatrix}\mathbf{I}_2 & \\ & \mathbf{R}_2\end{bmatrix} \tag{28}$$

From (28), solve for $\mathbf{C}_4$, we have:

$$\mathbf{C}_4=2\mathbf{P}_{42}^{-1}\mathbf{S}_4\begin{bmatrix}\mathbf{G}_1 & & \\ & \mathbf{G}_1 & \\ & & \mathbf{G}_2\end{bmatrix}\begin{bmatrix}\mathbf{I}_2 & \\ & \mathbf{R}_2\end{bmatrix}\mathbf{R}_{41}^{-1}\mathbf{P}_{43}^{-1} \tag{29}$$

Let $\mathbf{S}_{41}=\mathbf{P}_{42}^{-1}\mathbf{S}_4=\begin{bmatrix}\mathbf{1} & \mathbf{1} & 0 & 0\\ 0 & 0 & \mathbf{1} & 0\\ \mathbf{1} & -\mathbf{1} & 0 & 0\\ 0 & 0 & 0 & \mathbf{1}\end{bmatrix}$, $\mathbf{S}_{42}=2\begin{bmatrix}\mathbf{I}_2 & \\ & \mathbf{R}_2\end{bmatrix}\mathbf{R}_{41}^{-1}\mathbf{P}_{43}^{-1}=\begin{bmatrix}\mathbf{1} & 0 & 0 & \mathbf{1}\\ 0 & \mathbf{1} & \mathbf{1} & 0\\ 0 & \mathbf{1} & -\mathbf{1} & 0\\ \mathbf{1} & 0 & 0 & -\mathbf{1}\end{bmatrix}$

Substitute into (29), we have $\mathbf{C}_4=\mathbf{S}_{41}\begin{bmatrix}\mathbf{G}_1 & & \\ & \mathbf{G}_1 & \\ & & \mathbf{G}_2\end{bmatrix}\mathbf{S}_{42}$

From (22), solve for $\tilde{\mathbf{G}}_4$, we have

$$\tilde{\mathbf{G}}_4=\mathbf{P}_{41}^{-1}\mathbf{G}_4\mathbf{R}_4^{-1}\mathbf{P}_{41} \tag{30}$$

$$\mathbf{P}_{(8,1)}\mathbf{C}_8\mathbf{P}_{(8,2)}\mathbf{R}_8 = \begin{bmatrix}\mathbf{S}_{41} & \\ & \mathbf{P}_{41}^{-1}\end{bmatrix}\begin{bmatrix}\mathbf{G}_1 & & & \\ & \mathbf{G}_1 & & \\ & & \mathbf{G}_2 & \\ & & & \mathbf{G}_4\end{bmatrix}\begin{bmatrix}\mathbf{S}_{42} & \\ & \mathbf{R}_4^{-1}\mathbf{P}_{41}\end{bmatrix}$$

Thus the cosine transform matrix will be

$$\mathbf{C}_8 = \mathbf{P}_{(8,1)}^{-1}\begin{bmatrix}\mathbf{S}_{41} & \\ & \mathbf{P}_{41}^{-1}\end{bmatrix}\left(\frac{1}{2}\begin{bmatrix}\mathbf{G}_1 & & & \\ & \mathbf{G}_1 & & \\ & & \mathbf{G}_2 & \\ & & & \mathbf{G}_4\end{bmatrix}\right)\begin{bmatrix}\mathbf{S}_{42} & \\ & \mathbf{R}_4^{-1}\mathbf{P}_{41}\end{bmatrix}\left(2\mathbf{R}_8^{-1}\right)\mathbf{P}_{(8,2)}^{-1} \tag{31}$$

$$\text{Let } \mathbf{P}_8 = \mathbf{P}_{(8,1)}^{-1}\begin{bmatrix}\mathbf{S}_{41} & \\ & \mathbf{P}_{41}^{-1}\end{bmatrix} = \begin{bmatrix}\mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & \mathbf{-1} & 0 & 0 & 0\\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & \mathbf{-1} & 0 & 0\\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{-1}\\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0\end{bmatrix},\quad \mathbf{K}_8 = \frac{1}{2}\begin{bmatrix}\mathbf{G}_1 & & & \\ & \mathbf{G}_1 & & \\ & & \mathbf{G}_2 & \\ & & & \mathbf{G}_4\end{bmatrix} \tag{32}$$

$$\mathbf{B}_1 = \begin{bmatrix}\mathbf{S}_{42} & \\ & \mathbf{R}_4^{-1}\mathbf{P}_{41}\end{bmatrix} = \begin{bmatrix}\mathbf{1} & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0\\ 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0\\ 0 & \mathbf{1} & \mathbf{-1} & 0 & 0 & 0 & 0 & 0\\ \mathbf{1} & 0 & 0 & \mathbf{-1} & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{-1}\\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0\\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0\end{bmatrix} \tag{33}$$

$$\mathbf{B}_2 = 2\mathbf{R}_8^{-1} = \begin{bmatrix}\mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0\\ 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 & 0\\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0\\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1}\\ \mathbf{1} & 0 & 0 & 0 & \mathbf{-1} & 0 & 0 & 0\\ 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{-1} & 0 & 0\\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{-1} & 0\\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{-1}\end{bmatrix} \tag{34}$$

$$\mathbf{B}_3 = \mathbf{P}_{(8,2)}^{-1} = \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \end{bmatrix} \tag{35}$$

The expression (31) becomes $\boxed{\mathbf{C}_8 = \mathbf{P}_8\mathbf{K}_8\mathbf{B}_1\mathbf{B}_2\mathbf{B}_3}$

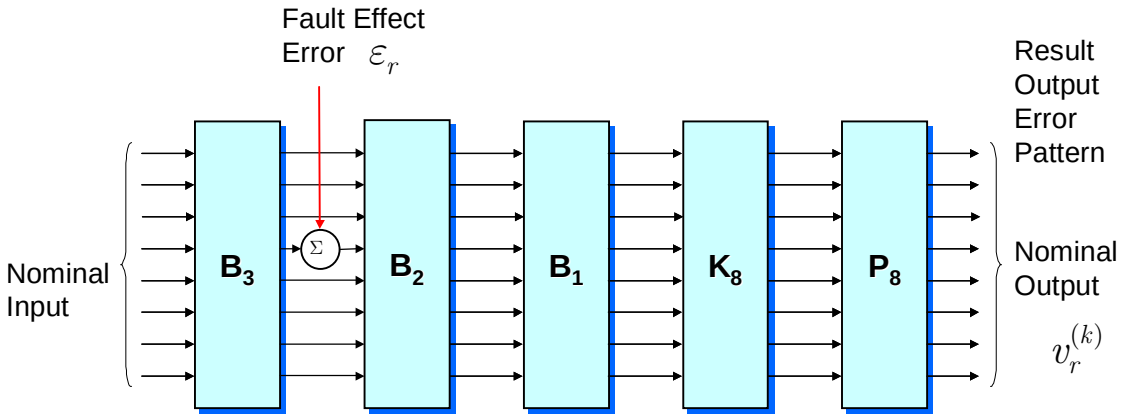


Figure 7 Modeling Fault Produced Errors in 8-point DCT

Based on the factorization of the transform matrix $\mathbf{C}_8$, the model of error is shown in Figure 7. As we see, a generic single error $\varepsilon_r^{(k)}$ is inserted at the input of a stage. The output err pattern $v_r^{(k)}$ is a direct consequence of the specific features of the fast algorithm as encapsulated in the structure of the component matrices.

3.3 Proposed Error Detection Algorithm for the DCT

This leads to a protection design methodology that uses real number parity values, similar to all previous FFT fault tolerance techniques [5-11]. Two comparable parity values are calculated, one from the input data vector and the other from the output transform coefficients in vector y. The parity values are both based on a weighting vector b, and calculated by taking the inner product of appropriate vectors and are

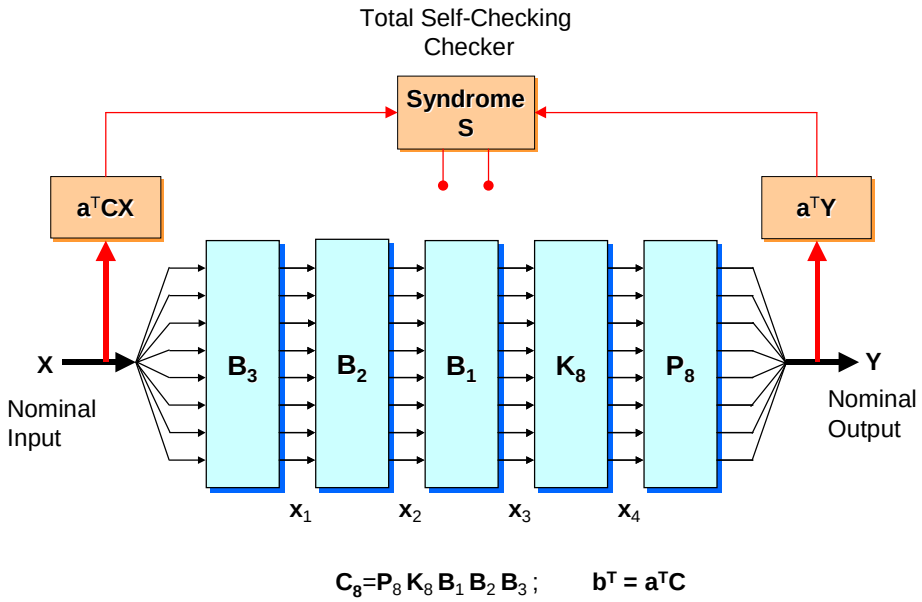


Figure 8 Protection in the 8-point DCT

comparable parity numbers that are subtracted to form a syndrome. The parity-checking operation involves a totally self-checking checker adapted for real number comparisons [12]. This is summarized in Figure 8. A design algorithm was developed using the concept of dual linear spaces that are affiliated naturally with the output error pattern viewed as a single-dimensional subspace. The code design method is diagrammed in Figure 8 and Matlab programs have been tested in designing protection for 2-D fast DCT algorithms. The preliminary results indicate that previous FFT protection methods have to be modified considerably.

Figure 9 describes the concurrent error detection scheme for the 1D-DCT network. From the definition of N-point DCT in equation (12), if we multiply the coefficient y_k with a weight factor F_k, then we have the equation

$$F_k y_k = \alpha_k \sum_{n=0}^{N-1} F_k x_n \cos\frac{\pi k(2n+1)}{2N}, \qquad k = 0,1,\ldots,N-1 \tag{36}$$

By constructing the sum $\sum_{k=0}^{N-1} F_k y_k$, we will have

$$\sum_{k=0}^{N-1} F_k y_k = \alpha_k \sum_{k=0}^{N-1}\sum_{n=0}^{N-1} F_k x_n \cos\frac{\pi k(2n+1)}{2N} \tag{37}$$

$$= \sum_{n=0}^{N-1} x_n \left[\sum_{k=0}^{N-1} \alpha_k F_k \cos\frac{\pi k(2n+1)}{2N}\right]$$

If F_k s are known, the expression inside the square bracket of (37) can be precomputed. Let $b_n = \sum_{k=0}^{N-1} \alpha_k F_k \cos\frac{\pi k(2n+1)}{2N}$ be the weight factor at the input line x_n of the DCT network. By selecting the weight factor $F_k = (k+1)/N$ (more theorems are found in [16]), the precomputed weight factors $b_0,...,b_7$ are shown below

By selecting the weight factor $F_k = (k+1)/N$ (more useful theorems are found in [16]), for $N = 8$ we have the following results

$$\begin{aligned} b0 &= 2.2864;\ b1 = -1.6865;\ b2 = 0.6945;\ b3 = -0.8128; \\ b4 &= 0.7754;\ b5 = -0.4605;\ b6 = 0.1848;\ b7 = 0.0188; \end{aligned} \quad (38)$$

At the output, the weight factors $F_k s$ can be decomposed into the sum of the powers of ½. For $N = 8$, the decompositions are shown below.

$$F_0 = \frac{1}{8} = \frac{1}{2^3},\ F_1 = \frac{2}{8} = \frac{1}{2^2},\ F_2 = \frac{3}{8} = \frac{1}{2^2} + \frac{1}{2^3},\ F_3 = \frac{4}{8} = \frac{1}{2},$$

$$F_4 = \frac{5}{8} = \frac{1}{2} + \frac{1}{2^3},\ F_5 = \frac{6}{8} = \frac{1}{2} + \frac{1}{2^2},\ F_6 = \frac{7}{8} = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} \quad (39)$$

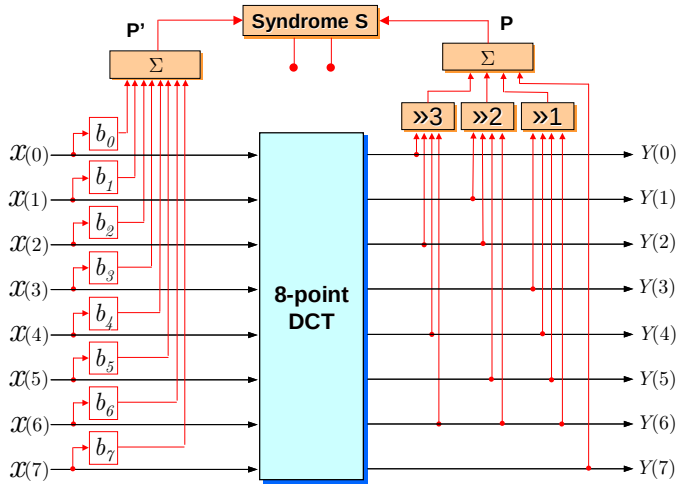


Figure 9 Concurrent Error Detection designed for 8-point DCT

In Figure 9, the block b_n s represent for the precomputed constants to be multiplied with data input $x(0),...,x(N-1)$. At the output of the DCT network, the lines connect between the coefficients X_k s with the 1, 2, 3 bits right shift determined by the decomposition as shown in (39)

3.4 Some Experiment Results

The experiment started by computing the DCT of a luminance image sample without the influence of error. This gives us the correct image of coefficients as shown in Figure

10. In the next step, a random noise with the variance equal to 10% of the image variance is injected at an arbitrary line and arbitrary stage of the DCT network shown in the Figure 8.

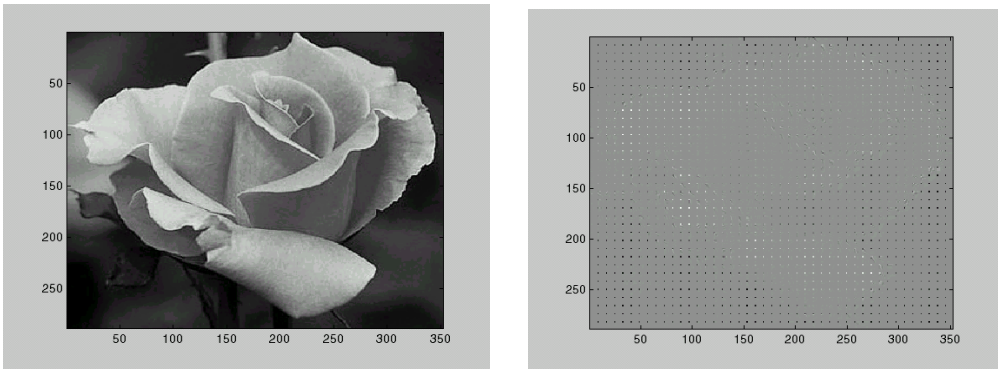


Figure 10 Luminance sample image Y (left) and its DCT coefficient image (right)

In our experiment, the noise in added into the coefficients in the fourth output line of the first stage. The noise levels are presented in Figure 11. The noise is assumed to be injected once per transform of a line in a 8x8 data block. The intermediate coefficients with error enter the input lines of the next stage. The next stage's intermediate output will have error spread to the more than one line. Finally, all coefficients X(0) to X(7) at the last stage's output very likely contain errors.

Figure 11 Noise injection pattern at stage 1 line 4 of the DCT networks.

It is hard to predict how the errors propagate from stage to stage. In this circumstance the weight factor scheme becomes a useful tool to check errors occur inside the DCT network. By collecting all error values injected into the system and comparing the differences between the weight sums P and P', we can show the checking system responding to error input.

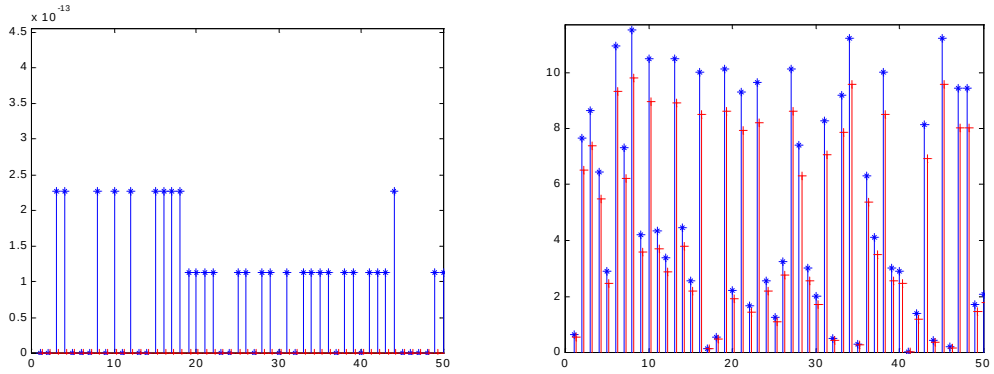


Figure 12 The responding to errors injected into the DCT network. Left: When no noise injection Right: The noise injection values (+) and the difference between P' and P (*)

Figure 12 shows how the errors are reflected at the checker output. The top figure shows a very small difference between the input and the output weights P and P'. The reason for the nonzero differences is round off errors due to the finite resolution of computing system. In the bottom figure, the values of $|P - P'|$ reflect errors occurred. Those signal can trigger the recomputing the input data.

If the error threshold is setup low enough, then most of the errors can be detected by the self-checking checker. However, if we set the threshold too low, the checker may pick up the round off errors an consider those to be the errors due to the system fault or the injected noise. Thus, we need to find a good threshold, which separates the errors due to computer resolution limited and the computer fault or noise.

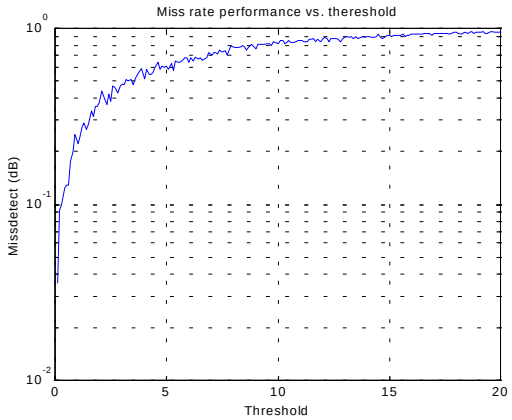


Figure 13 Detection performance of the checker vs. the setup threshold

Figure 13 gives the error detection performance versus the setup threshold. At the small setup threshold, the checker pick up most the errors occurred. The performance is getting worse when the threshold is getting larger. With a high resolution computing system, the errors due to fault and noise can be detected up to 100%

4 Fault Tolerant in Huffman Source coding Stage

4.1 Overview of Huffman Codes

Huffman codes are widely used and very effective technique for compression data; saving of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table of the frequencies of occurrence of characters to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. We observe that the characters in the file occur with frequencies given by Figure 14

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 14 A character coding problem.

The data file contains only the characters `a-f`, with the frequencies indicated. If each character is assigned as three bit codeword, the file can be encoded in 300,000 bits. Using the variable length code shown, the file can be encoded in 224,000 bits, which saves approximately 25%. In fact, this is an optimal character code for this file. Once alphabet symbols have been defined, the compression efficiency can be improved by using shorter code words for more probable symbols and longer codewords for the less probable symbols. This variable-length codewords belong to entropy coding scheme. Huffman coding is one of the entropy coding techniques that JPEG uses in its compression standard. We see here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix codes***. It is possible to show that the optimal data compression achievable by a character code can always be achieved with a prefix code.

Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with a variable length prefix code of Figure 14, we code the 3-character `abc` as $0 \cdot 101 \cdot 100 = 0101100$, where we use "$\cdot$" to denote concatenation.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off. A binary tree, whose leaves are the given characters provides one such presentation. We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child", and 1 means "go to the right child". Figure 15 shows the trees for the two codes of the example in Figure 14. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with a sum of the frequencies of the leaves in its subtree.

An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children. The fixed length code as shown in Figure 15(left), is not a full binary tree: there are codewords beginning 10...., but none beginning 11.... If we restrict to the full binary tree, we can say that if C is the alphabet from which the

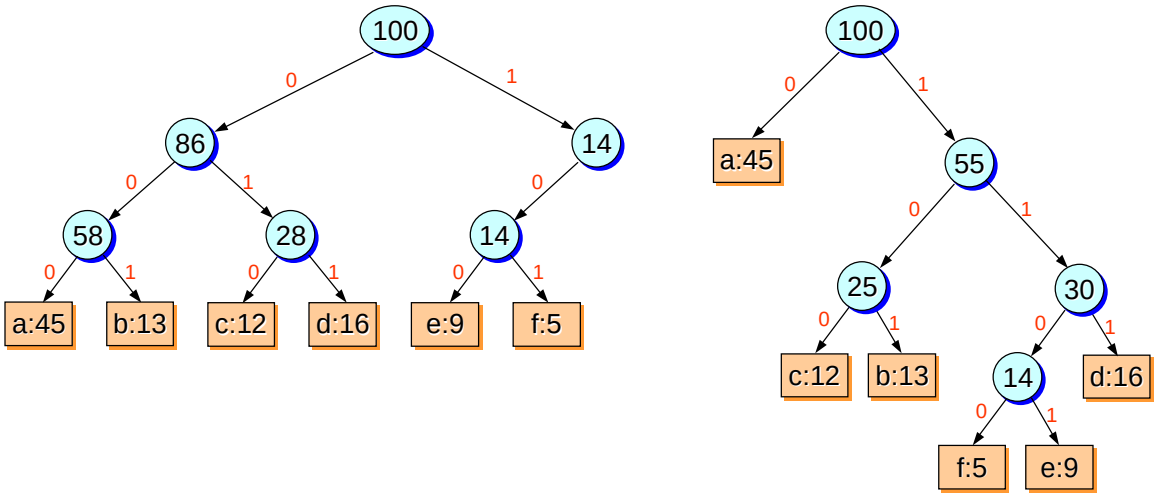


Figure 15 Trees corresponding to coding schemes in Fig.14. **Left:** The tree corresponding to the fixed-length code a=000,...,f=101. **Right:** The tree corresponding to the optimal prefix code a = 0, b = 101, c=110, d=111, e= 1101, f = 1100.

characters are drawn and all character are positive, then the tree for the optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C|$-1 internal nodes. Given a tree T corresponding to a prefix code, it is a simple matter to computer the number of bits required to code a file. For each character c in C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c) \tag{40}$$

4.2 Constructing a Huffman Code

Huffman invented a greedy algorithm that constructs an optimal prefix code, now called a ***Huffman code***. In the pseudocode that follows, we assume that C is an object with a defined frequency $f[c]$. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C|-1$ "merging" operations to create a final tree. A min-priority queue Q, keyed on f, is used to identify the two least frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

HUFFMAN(C)

1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. **for** $i \leftarrow 1$ **to** $n-1$

4. **do** allocate a new node z
5. $left[z] \leftarrow x \leftarrow ExtractMin(Q)$
6. $right[z] \leftarrow y \leftarrow ExtractMin(Q)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $INSERT(Q, z)$
9. **return** $ExtractMin(Q)$

Line 2 initializes the min-priority queue Q with the characters in C. The **for** loop in lines 3-8 repeatedly extract the two nodes x and y of lowest frequency from the queue, and replaces them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of frequencies of x and y in line 7. The node z has x as its left child and y as its right child. This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.) After $n-1$ mergers, the one node left in the queue – the root of the tree – is returned in line 9.

4.3 Fault Tolerant Issues in Huffman Codes

The problem with tree structures is that they are very sensitive to memory errors, lacking exploitable redundancy. As we have seen from the previous part, trees are constructed by using pointers to link nodes together and each node carries information about itself, its parent and children. Therefore, if one internal node disappears as a result of a memory error (pointer corruption, memory cell corruption) then all the nodes below that one are lost. Detecting and correcting errors can be done but cost additional computations. For example, an error-correcting code could be used to protect the content of each cell by parsing, checking each cell. This task can be done only if the tree structure is not corrupted by memory error or any other reasons. After a correction for an element, the updating the tree to maintain some properties usually requires a large of running time.

To overcome the difficulty of making the Huffman algorithm fault tolerant, table can be used instead of tree. Each table has a table head, which is the address of the first item. Table head allows us to access entire table's content by computing the actual address by the displacement from head to an item in the table. Protection of table head requires a small computation cost. With our design scheme, error detection must be available for the required table lookup procedures at the Huffman encoder. The codewords have variable lengths so the encoder cannot employ fixed-width memory.

A possible Huffman encoding structure is shown in Figure 16. The (*run*, *size*) input symbol accesses a pointer table that indicates the length of the corresponding Huffman codeword. This symbol is also used to indicate where the code for the RUN-SIZE symbol is located in memory storage. The variable length codeword is assembled in an output buffer available for other formatting operations. Figure 16 also indicates a protection

method. Parity bits associated with each selected codeword are stored and comparable with parity bits recalculated from the output buffer.

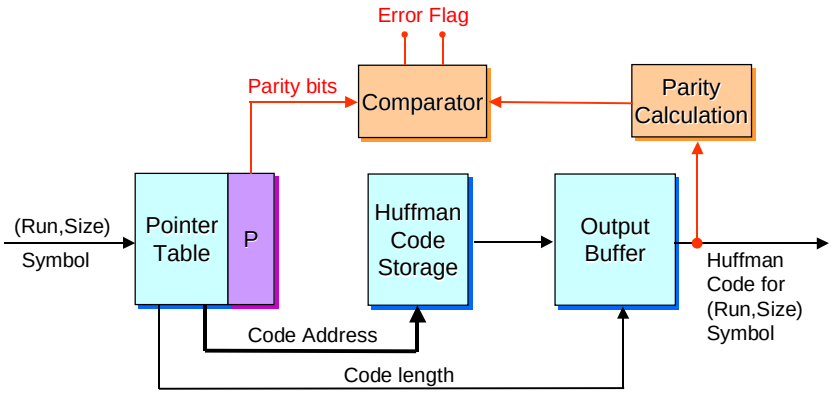


Figure 16 Huffman Encoding for Table 1 with Parity Checking

Note that Table 2 identify the size category for the DC Huffman differences. The extra bits to complete the coding of the sign and magnitude of the nonzero AC coefficients are exactly the same scheme as for different DC coding. When the AC coefficient s is positive, the extra bits contain a 1 followed by the $size-1$ least significant bits of $|s|$. When $s<0$, the extra bits consist of a 0 followed by the one's complement (inversion of bit pattern) $size-1$ least significant bits of $|s|$. The error detection for the additional bits can be completed in the Figure 17 must be available for table lookup procedures The integer AC coefficient s and the $size$ parameter access a pointer table that indicates the address of the *additional code bits*. The parity bits associated with each selected codeword are stored in the pointer table. The upper path of the diagram shows

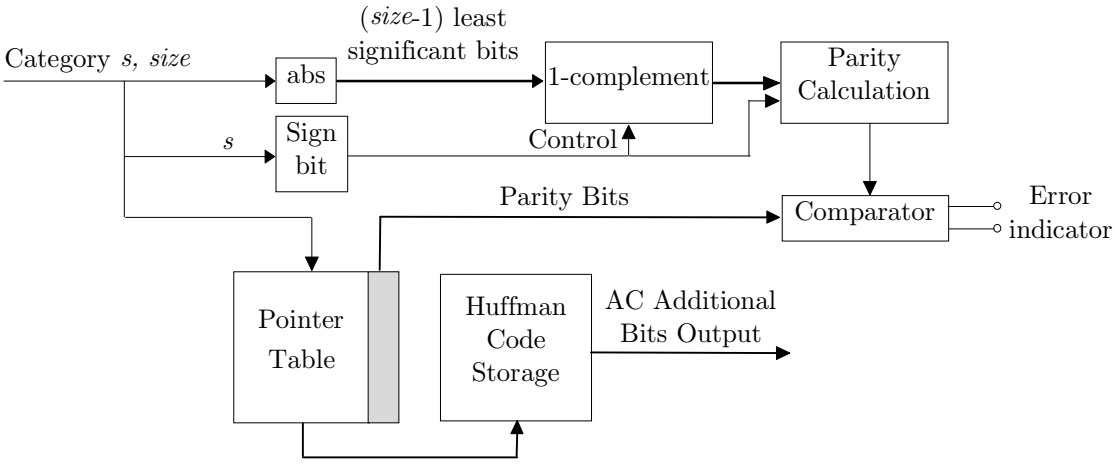


Figure 17 Huffman encoding for category code in Table 2 with parity checking

how the additional bits and the parity bits are generated. The comparator compares the generated parity and those stored in the pointer table.

4.4 Overview of Residue Codes

A residue code is a separable code created by appending the residue of a number to that number. The code word is constructed as $D|R$, where D is the original data and R is the residue of that data. The residue generation is determining the residue of the data. The residue of a number is the remainder generated when the number is divided by an integer. For example, suppose we have an integer N and we divide N by another integer m, N may be written as an integer multiple of m as

$$N = I \cdot m + r$$

where r is the remainder, sometimes called the residue, and I is the quotient. The quality m is called the check base, or the modulus. The primary advantage of the residue codes are they are invariant to the operation of addition, subtraction, and the residue can be handed separately from the data during the addition or subtraction

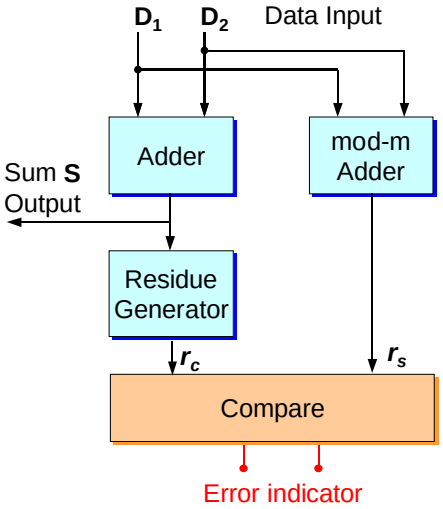


Figure 18 Structure of an adder designed using the separable residue code

Residue Code table

Input of residue Generator	Residue	Binary form
0	0	00
1	1	01
2	2	10

process. The structure of an adder that uses the separate residue code for error detection is shown in Figure 18. The two data D_1 and D_2 are added to form a sum word S. The residues r_1 and r_2 of D_1 and D_2, respectively, are also added using modulo-m adder, where m is modulus used to encode D_1 and D_2. If the operation is performed correctly, the modulo-m addition of r_1 and r_2 yield the residue r_s of the sum S. A separate circuit is then used to calculate the residue of S. If the calculated residue r_c differs from r_s, an error has occurred in one part of the process. For instance, given the modulus m = 3, the above table shows the residue output corresponding to the integer input. The detection of errors using the residue code is applied in the Huffman encoders for the DC and AC coefficients as shown in the following section.

4.5 Error Detection for Decoding DC Coefficients

Figure 19 describes error checking for the DC decoder. The DC Huffman code is divided into two binary code sections concatenate together: The first section consists of c number of consecutive 1's ending with a single 0 bit which indicates the "size category" code in the code Table 2. The second section contains exactly c bits that indicate the DC value. Let B be the decimal value of the second section. If the most significant bit of B is zero, then the DC value is negative with magnitude equal to $2^c - B - 1$. If the most significant bit of B is one, then the DC value is positive with magnitude equal to B.

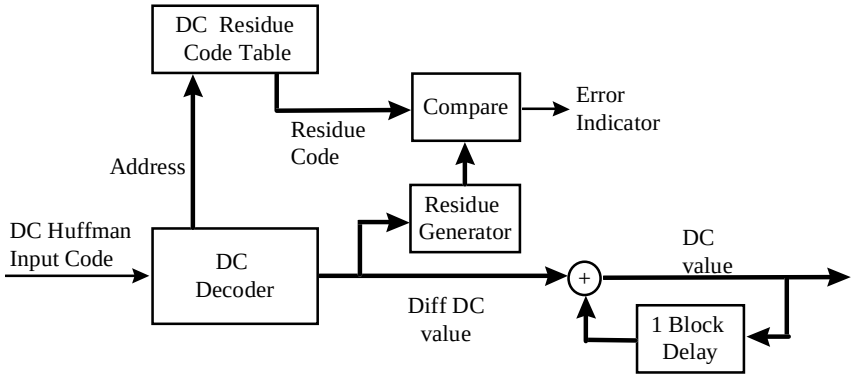


Figure 19 Error Detection System for DC Decoder

The error detection for the DC Huffman decoder is shown in Figure 19. The DC Huffman bit stream enters the DC decoder. The DC decoder compute the equivalent DC value, along with the address of its DC residue code table. The DC value at the output of the DC decoder enters to the residue generator. Both residue codes come out from the DC residue code table and the residue generator are compared in the comparator. If they are matched, no error is occurred. Otherwise, the error indicator is activated and send the error signal to decoder error handler for further process.

4.6 Error Detection for Decoding AC Coefficients

The error detection algorithm for the AC coefficients is more complicated than that for the DC coefficients because the AC code is a combination of two groups of bits. The first group, a Huffman codeword, indicates the size and zero run-length. If this group of bits is 1010, the end of block (EOB) is reached. If the group is 11111111001 then the number of zeros run-length is 15 and the nonzero AC coefficient will be determined form the next codeword. The second group indicates the AC value just like what has been done for the DC value. Note that Table 2 is used for both DC and AC Huffman decoders. Figure 20 shows error checking for AC Huffman code. The two groups of AC bit stream represent for (ZeroRun, Size)(Amp) enters the decoder. The first group is used to generate an appropriate address to access the ZeroRun storage. The output of the ZeroRun storage contains the number of zero run-length Z. The (Size, Amp) is used to issue an address to the AC Value Storage. At the same time, the (Size, Amp) also enters the AC Value Reconstruction unit to recompute the AC value. This AC

coefficient is saved in an AC Block Buffer and, at the same time, it is routed the Comparator along with the value comes from the AC value Storage . If they do not match, the output of the comparator will show an error signal.

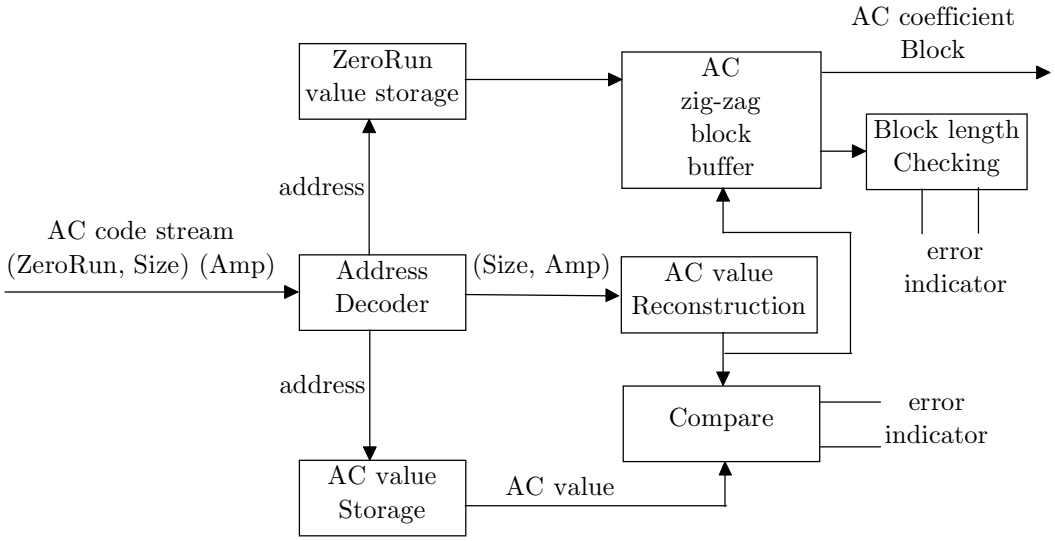


Figure 20. Error Detection System for AC Decoder

When the 'end of block' code stream is reached, the number of AC coefficients in the AC Block Buffer must be 63 (in the context of 8x8 2D-DCT image block). The block length checker verify the correctness of the block length. If the block length is incorrect(smaller or larger than 63), an error signal will occur. Otherwise, the content of the Buffer will be transferred to the de-quantization block for further processing.

5 Fault Tolerant in Quantization Stage

5.1 Error detection system for quantization stage

The quantization process discards information that is visually imperceptible. Every element in the 8×8 transform coefficients, is divided by a pre-selected integer, the *Quantum*(*i*,*j*), taken from the quantization table.

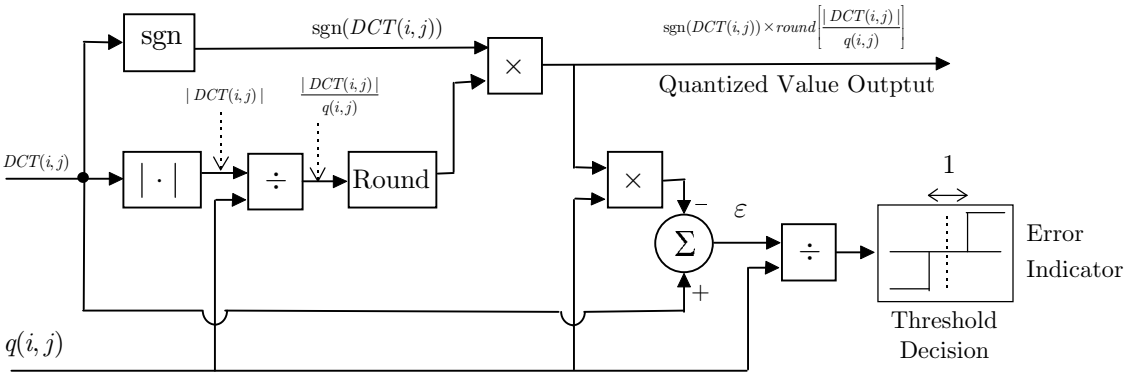


Figure 21 Error checking for the Quantization of a DCT coefficient

The elements that matter most to the picture will be divided by the smaller step size.

$$QuantizedValue(i,j) = \text{sgn}(DCT(i,j)) \times Round\left[\frac{|\,DCT(i,j)\,|}{Quantum(i,j)}\right] \tag{41}$$

Only if the high-frequency components get up to unusually large values will they be encoded as non-zero values. At the decoding, the de-quantization formula operates in reverse.

$$\widetilde{DCT}(i,j) = QuantizedValue(i,j) \times Quantum(i,j) \tag{42}$$

Distortion errors due to the quantization process are always present and such errors in a typical $DCT(i,j)$ coefficient may be modeled with an additive disturbance.

$$\frac{DCT(i,j)}{q} = \text{sgn}(DCT(i,j)) \times Round\left[\frac{|\,DCT(i,j)\,|}{q}\right] + \varepsilon_r \tag{43}$$

$\boldsymbol{\varepsilon}_r$ is a roundoff error an $q \triangleq q(i,j)$ is the quantum at the row i, column j. It is clearly satisfied $\varepsilon_r \le 1/2$. Therefore, the $DCT(i,j)$ may be expressed as

$$DCT(i,j) = q \times \text{sgn}(DCT(i,j)) \times Round\left[\frac{DCT(i,j)}{q}\right] + q \cdot \varepsilon_r \tag{44}$$

The quantity $\boldsymbol{\varepsilon} \triangleq \boldsymbol{q} \cdot \boldsymbol{\varepsilon}_r$ is the quantization error that degrades the image quality slightly. Error detection for quantization begins with determining if the quantization values are read properly from the quantization table storage. Assume the quantum $q(i,j)$ and the $DCT(i,j)$ are checked earlier. Figure 21 shows the error checking for the quantization operations. The $DCT(i,j)$ value taken absolute $|\,DCT(i,j)\,|$. This $|\,DCT(i,j)\,|$ is divided by the quantum factor $q(i,j)$. The quotient is rounded and then multiplied with sign of $DCT(i,j)$, which is 1 or -1. The quantized value output is transferred to the next coding block. The quantization error ε is determined by subtracting the quantized $\widetilde{DCT}(i,j) \triangleq q(i,j) \times QuantizedValue(i,j)$ from the $DCT(i,j)$ input. The ratio between ε and $q(i,j)$ is the subject to be checked. By using the $\pm 1/2$ thresholds we can decide if error occurred in the quatization process.

5.2 Error Protection for Dequantization

The critical task in dequantization is multiplication. Each coefficient in a DCT block is scaled up by the same factor as it was scaled down in the encoder. Hence, the data read from the quantization tables and the multiplication operations are need to be checked. Since the equation (42) represents for the dequantization operation, low-cost residue code is efficient to detect its errors from a computational resource viewpoint. In the low-cost codes, the number base is typically taken to be of the form $b = 2^a - 1$, where a is integer greater than 1. The residue of the code word (operand) A is obtained by performing a modulo $2^a - 1$ addition of the a bits in the code word. This procedure may

be simplified if the a bits are divided into k groups in which each group contain $l = a \div k$ bits. The quotient $a \div k$ must be an integer. If these groups are formed, the modulo $2^l - 1$ addition of the l bits for each of the k groups can be used to generate the residue of A [14,15]. By apply this method in multiplication of two residues, we only needs a-bit multiplication.

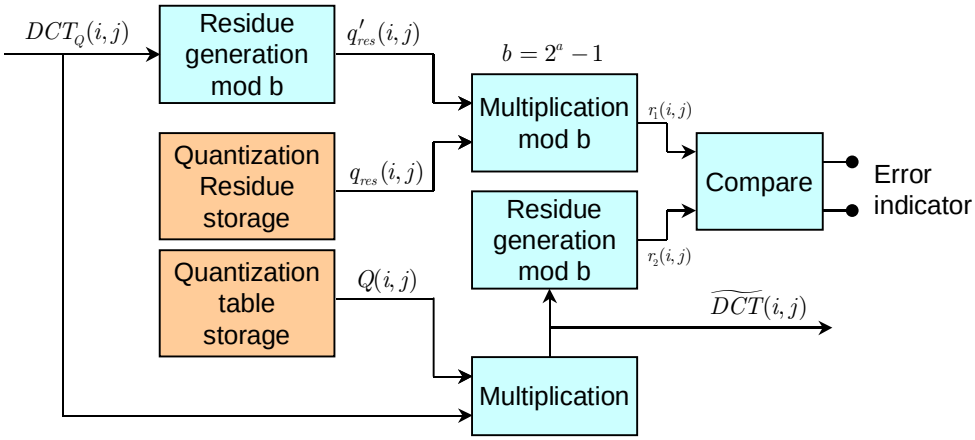


Figure 22 Residue Error Detection System for Multiplication Operations

Figure 22 shows the error check system for the dequantization. The residue codes of the quantization table are precalculated and stored in a table along with the quantization table content. When a $DCT_Q(i,j)$ is arrived from the previous stage, it is multiplied with an appropriate element $Q(i,j)$ in the quantization table yields the reconstructive coefficient $\widetilde{DCT}(i,j)$. In the other path, a residue code $q'_{res}(i,j)$ of $DCT_Q(i,j)$ is generated and then multiplied with the precalculated residue code $q_{res}(i,j)$. At the same time, the residue code $r_2(i,j)$ of $\widetilde{DCT}(i,j)$ is generated. The code $r_2(i,j)$ is compared with $r_1(i,j)$ at the output of modulo b multiplication. If no error in the process, $r_1(i,j)$ is equal to $r_2(i,j)$. The signal at the output of the comparator indicate the error status of the dequantization process.

6 Fault Tolerant in the JPEG Data Stream

In this section we will deal with the fault tolerant issues in the JPEG data stream. First of all, we need to review the information about baseline DCT process frames and extended sequential frames. These are identical except that extended sequential frames can use 12-bit or 8-bit samples and up to four DC or AC Huffman tables, whereas baseline DCT JPEG support only 8-bit samples and up to two DC and AC Huffman tables, these two frames are decoded in exactly the same manner.

6.1 Procedures and compressed data structure

JPEG compressed data contains two classes of segments: entropy coded segments and marker segments. Entropy coded segments contain the entropy coded data, whereas

marker segments contain header information, tables and other information required to interpret and decode the compressed image data. Marker segments are always begin with a marker, a unique two-byte code that identify the function of the segment.

A JPEG compression process consist of a single frame or a sequence of frames. Each frame is composed of one or more scans through the data, where a scan is a single pass through the data for one or more components of the image. At the beginning of each frame, the control procedures generate a frame header that contains parameter values needed for decoding the frame. Similarly, at the beginning of each scan, the control procedures generate a scan header that contains parameter values needed for decoding the scan. Each frame and scan header starts with a marker that can be located in the compressed data without decoding the entropy-coded segments. Marker segments defining quantization tables, entropy coding tables, and other parameters may precede the frame and scan headers. Figure 23 illustrate the structure of compressed image data stream.

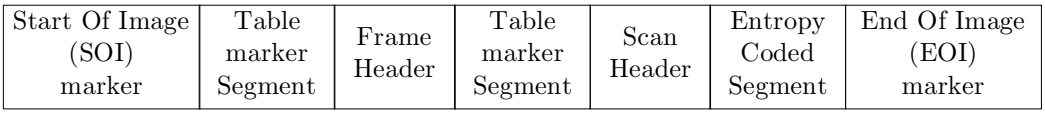


Figure 23 Structure of typical compressed image data stream

Scan header are always followed by one or more entropy-coded segments. The input data for the scan may be divided into fixed interval called restarted intervals, and the data in each restarted interval are compressed into a single entropy-coded segment. When a scan contains more than one entropy coded-segment, the segments are separated by special two-byte codes called a restarted markers that can be located in the compressed data without decoding. The encoding and decoding of each entropy-coded segment is done independently of the other entropy-coded segments in the scan.

6.2 Image ordering

One of the more complex of JPEG is the interleaving of data from image components during the compression process. This interleaving makes it possible to decompress the image, convert from a luminance-chrominance representation to a red-green-blue display output. A JPEG compressed image data stream contains a single image that can have up to 255 unique components. Each component of the image is represented internally as the rectangular array of samples. The rectangular arrays of samples are processed from left to right along rows, and from top row to bottom row. The data are decoded and returned to the application in the same order in which they were presented to the encoder.

6.3 Data units for DCT-based and lossless processes

A data unit is the smallest logical unit of source data that can be processed in a given JPEG mode of operation. In DCT-based modes, the component arrays are divided into

8×8 blocks. Therefore, the *data unit* for the DCT based processes is a *single* 8×8 *block of samples*. The block in the component are processed from left to right along block rows, and from the top block row to the bottom block row.

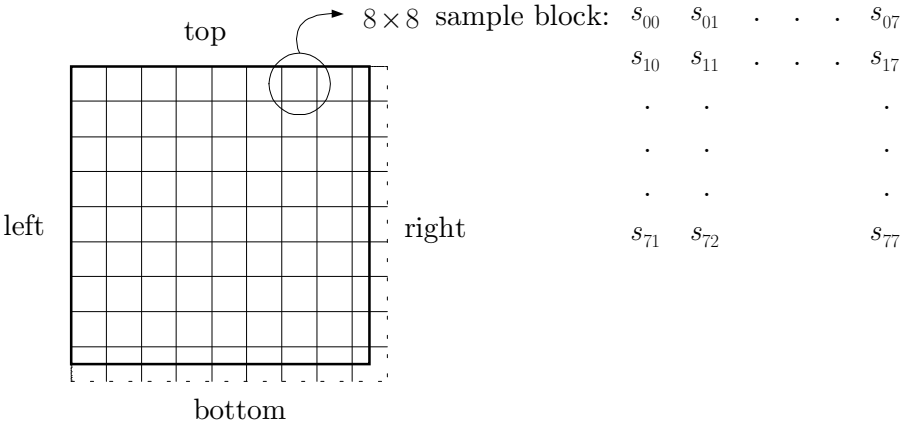


Figure 24 The orientation of samples for forward DCT computation

Figure 24 shows an image component which has been partitioned into 8×8 blocks for the DCT computations. It also shows the orientation of the samples. The definitions of block partitioning and sample orientation also apply to any DCT decoding process and the output reconstruction image.

For the coding processes, data units are assembled into groups called minimum coded units (MCU). In scans with only one component, the data is non-interleaved and the MCU is one data unit. In scans with more than one component, the data are interleaved and the MCU defines the repeating pattern of interleaved data units. The order and the number of data units in the MCU for interleaved data are determined from the horizontal and vertical sampling factors and the components in the scan. Let H_i and V_i be the number of samples in the i^{th} component relative to the other components in the frame. When the data are interleaved, the sampling factors define a two dimensional array of data units, $H_i \times V_i$ in size for the i^{th} component of the frame. If the $1,2,\ldots,N_s$ components in the scan correspond to frame components $c_1, c_2,\ldots, c_{N_s}$ each MCU in the scan is constructed by taking $H_{c_i} \times V_{c_i}$ data units from frame component c_i, for $i = 1,2,\ldots,N_s$. Within each $H_i \times V_i$ array of data units, the data units are ordered from left to right and from top row to bottom row. Processing always starts at the upper left corner of each component array. Figure 25(a) shows a small image with twice as many luminance blocks horizontally as chrominance component blocks. The sampling factors are therefore $(H = 2, V = 1)$ for the luminance Y, and $(H = 1, V = 1)$ for the Cb and Cr chrominance components. When the data units for all three components are interleaved in a single scan, the ordering of the data units would be:
$Y_1Y_2Cb_1Cr_1$, $Y_3Y_4Cb_2Cr_2$, $Y_5Y_6Cb_3Cr_3$, $Y_7Y_8Cb_4Cr_4$, where each group $YYCbCr$ forms a MCU. In the Figure 25(b), the sampling factors are $(H = 1, V = 2)$ for the luminance Y, and

$(H=1, V=1)$ for the Cb and Cr chrominance components. In this case, the ordering of the data units would be $Y_1Y_3Cb_1Cr_1$, $Y_2Y_4Cb_2Cr_2$, $Y_5Y_7Cb_3Cr_3$, $Y_6Y_8Cb_4Cr_4$.

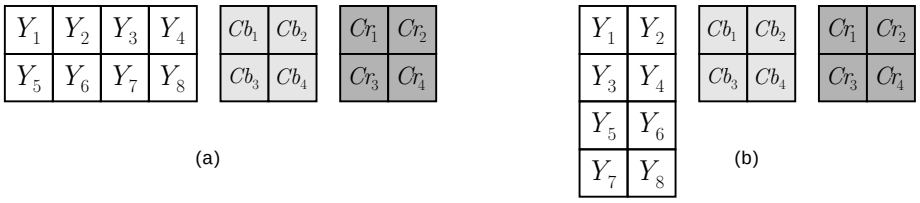


Figure 25 Three-component image with chrominance sub-sampled

6.4 Marker Overview

Markers serve to identify the various structural parts of the compressed data formats. Most markers start marker segments containing a related group of parameters; some markers stand alone. All markers are assigned two-byte codes: an X'FF' byte followed by a byte which is not equal to 0 or X'FF'. Any marker optionally be preceded by any number of X'FF'. Because of this special code-assignment structure, markers make it possible for a decoder to parse the interchange format and locate its various part without having to decode other segments of image data.

Marker segments consist of marker followed by a sequence of related parameters. The first parameter in a marker segment is the two-byte length parameter. This length parameter encodes the number of bytes in the marker segment, include the length parameter and excluding the two byte marker. The marker segments identified by the SOF and SOS marker codes are referred to as headers: the frame header and the scan header respectively.

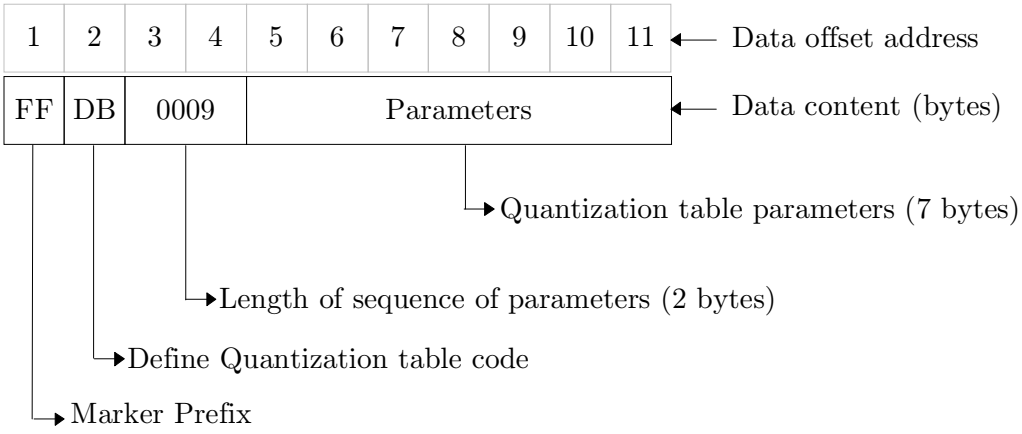


Figure 26 a marker with the following sequence of parameters

In generating the entropy-coded segments, 0xFF bytes are occasionally created. To prevent accidental generation of markers in the entropy coded segments, each occurrence of 0xFF byte is followed by a "stuffed" zero byte. All marker segments and entropy-

coded segments are followed by another marker. In Huffman coding, *one-bits* are used to pad the entropy-coded data to achieve byte alignment for the next marker. Because a leading one-bit sequence is always a prefix for a longer code, padding with one-bits cannot create valid Huffman codes that might be decoded before the marker is identified. The markers fall into two categories: those without parameters and those followed by a variable-length sequence of parameters of known structure. For the markers with parameters, the first parameter is a two- bytes parameter giving the length (in bytes) of the sequence of parameters. This length includes the length parameter itself, but excludes the marker that defines the segment

6.5 Application Markers (APP_n)

The $APP_1 - APP_{15}$ markers hold application-specific data. The hold information beyond what is specified in the JPEG standard. The format of these markers is application specific. The length field after the marker can be used to skip over the marker data. If an application need sot store information beyond the capabilities of JPEG, it can create APP_n markers to hold this information. An APP_n can appear anywhere within the JPEG data stream. An application that processes APP_n markers should check not only the marker identifier, but also the application name. Figure 27 specifies the marker segment structure for an application.

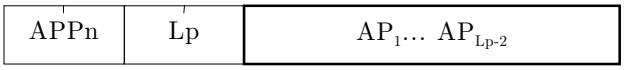


Figure 27 application data segment

6.6 Comment marker (COM)

Comment marker is used to hold comment strings. This marker should be used for plain text. It may appear anywhere within JPEG file.

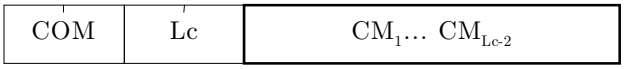


Figure 28 structure of a comment segment

6.7 Define Huffman Table (DHT)

The DHT marker defines (or redefines) Huffman tables. A single DHT marker can define multiple tables; however, baseline mode is limited to two of each type, and progressive and sequential modes are limited to four. The only restriction on the placement of DHT markers is that if a scan requires a specific table identifier and class, it must be defined by a DHT marker earlier. Figure 29 illustrates the structure of Huffman table segment. Each Huffman table is 17 bytes of fixed data followed by a variable filed of up to 256 additional bytes. The first fixed byte $TcTh$ contains the identifier for the table. The next 16, $L_1,\ldots,L_{16}$, form an array of unsigned of one-byte integers whose elements give the number of Huffman codes for each possible code length. The sum of the 16 code

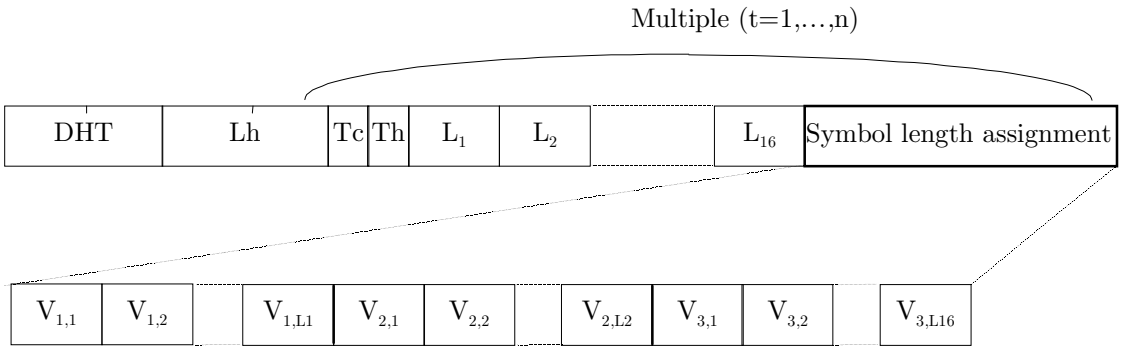


Figure 29 structure of Huffman table segment

lengths is the number of values in the Huffman table. The values are 1 byte each and follow the length counts. The value n in the Figure 29 is the number of Huffman tables in the marker segment. The value of Lh relates with n and the length of each Huffman table by the equality

$$Lh = 2 + \sum_{t=1}^{n}(17 + m_t) = 2 + 17n + \sum_{t=1}^{n} m_t \tag{45}$$

where m_t be the number of parameters, which follows the 16 $L_i^{(t)}$ parameters for Huffman table t and is given by $m_t = \sum_{i=1}^{16} L_i^{(t)}$ (46)

6.8 Error detection algorithm for Define Huffman Table (DHT)

The error checking for the DHT marker will check for the relation between parameters and the value range of parameters. Assume S DHT marker segment array that was created and stored in the memory. The algorithm for error checking DHT marker is shown in the Fig. 30. The algorithm is created based on the checksum results obtained from the equations (45) and (46). The $DHT\_ERR\_CHECKER(S)$ algorithm read and check the parameters as shown below

$DHT\_ERR\_CHECKER(S, n)$

1. $DHT \leftarrow Extract(S,2)$
2. **if** $DHT \neq FFC4$
3. **then return** *false*
4. $Lh \leftarrow Extract(S,2)$
5. $Q \leftarrow 2 + 17n$
6. for $t = 1$ to n
7. do $TcTh \leftarrow Extract(S,1)$
8. **if** $TcTh \& 0XEE \neq 0$
9. **then return** *false*
10. $L \leftarrow 0$

11. for $i \leftarrow 1$ to 16
12. do $L_i \leftarrow Extract(S,1)$
13. if $L_i \geq 2^i$
14. then return *false*
15. $L \leftarrow L + L_i$
16. $Q \leftarrow Q + L$
17. for $i \leftarrow 1$ to 16
18. do for $j = 1$ to L_i
19. do $V_{i,j} \leftarrow Extract(S,1)$
20. if $V_{i,j} \geq 2^i$
21. then return *false*
22. if $Q \neq Lh$
23. then return *false*
24. return *true*

The way the algorithm works is explained below with assumption that the number of Huffman tables n is known. Line 1 extracts the DHT marker. Line 2 check for the valid DHT marker, which is 0xFFC4. If it is invalid, return *false* in line 3. Line 4 extract length Lh. Line 5 stores the initial value as shown in the equality 45. Lines 6-21 extract parameters from the marker stream. Line 6 set up the number of for loops equal to n. Lines 7-9 extract and check for the value of Th and Tl. Since each of Tc and Th only contain value 0 or 1 for the baseline DCT, the and bit $TcTh\&0XEE$ must yield 0. We use this property to check for the validate of these parameters. Line 10 set a intermediate parameter L to zero. Lines 11-15 extract 16 parameters L_is,

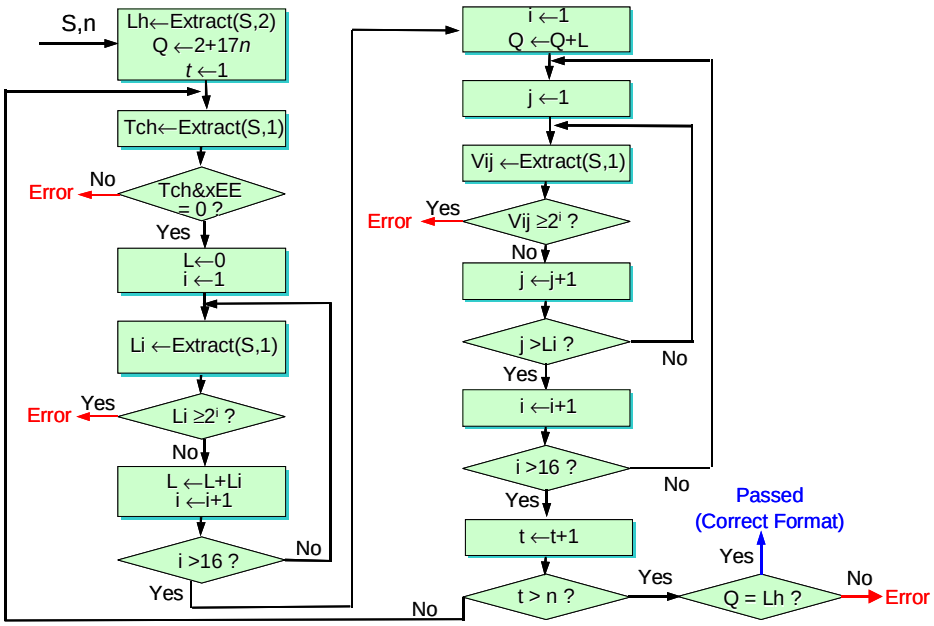


Figure 30 Algorithm for error checking DHT marker

each contains the number of DHT codes with length i. Since the number of code of length i cannot exceed $2^i - 1$, line 13 check for the value of L_i. Line 14 returns false if L_i's value is invalid. Line 15 add accumulate the value of L_i into L. Lines 16 extracts and checks the validate for the values of Huffman code. Since the value of an i-bit code $V_{i,j}$ cannot exceed 2^{i-1}, therefore line 20 checks for this condition and line 21 returns *false* if the code is invalid. Line 22 checks for the total length of the DHT marker segment have been computed and the parameter Lh records for that length. Line 23 returns *false* if the two values are not matching. Line 24 returns *true* to complete the algorithm.

6.9 Define Restart Interval

The DRI marker specifies the number of MCUs between restart markers within the compressed data. Figure 31 specifies the marker segment which defines the restart interval The value of the 2-byte length field Lr is always 4. There is only one data

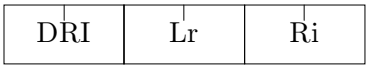


Figure 31 structure define start interval segment

field Ri in the marker define the restart data interval. If $Ri = 0$, then the restart marker are not used. A DRI marker with nonzero restart interval can be used to re-enable restart markers later in the image. A DRI marker may appear anywhere in the data stream to define or redefine the start interval, which remains in effect until the end of the image or until another DRI marker changes it. A DRI must appear somewhere in the file for a compressed data segment to include restart markers. Restart markers assist in error recovery. If the decoder finds corrupt scan data, it can use the restart marker ID and the restart interval to determine where in the image to resume decoding. The following formula can be used to determine the number of MCUs to skip:

$$\text{MCUs to skip} = \text{Restart Interval} \times ((8 + \text{CurrentMarkerID} - \text{LastMarkerID}) \text{ MOD } 8)$$

6.10 Restart Marker Processing

Restart markers are used to create independently encoded blocks of MCU. The restart interval specifies the number of MCUs between restart markers. A JPEG decoder needs to maintain a counter of MCUs processed between restart markers. When an image contains no restart markers, every MCU, except for the first, depend upon the previous MCU being decoded correctly. If the encoded data for an MCU get corrupted (e.g. while the data is being transmitted over a telephone line), each subsequence MCU will be incorrectly decoded.

Restart markers are placed on byte boundaries. Any fractional bits in the input stream that have not been used are discarded before the restart marker is read. The only processing that takes place after the reading of a restart marker is resetting the DC

difference values for each component in the scan to zero. If the decoder does not find a restart marker at the specified restart interval, the stream is corrupt. The valid restart markers are occurred in the order $RST_0, RST_2, \ldots, RST_7$ $RST_0, RST_2, \ldots, RST_7$.

If restart markers are used, decoder can use them to recover from corrupt compressed data. Since restart markers are placed in the output stream in sequence, decoders can compare the last one read before the corrupt data was encountered to the current one and use the restart interval to determine where in the image the decoding should resume. This works as long as the corruption in the compressed stream does not span eight restart markers. If it does, the decoder will not be able to match the remaining compressed data to its correct location within the image.

6.11 Define Quantization Table (DQT)

The DQT marker defines the quantization tables used in an image. A DQT marker can define multiple quantization tables (up to 4). Figure 32 specifies the marker segment which defines one or more quantization table. The quantization table definition follows the marker's length field. The value of the length field Lq is the sum of the sizes of the tables plus 2 (for the length field). The next 1 byte $PqTq$ contains information about the table. If the 4-bit number Pq is zero, the quantization table values are one byte each and entire table definition is 65 bytes long. If the value of Pq is 1, the size of each quantization value is 2 bytes and the table definition is 129 bytes long. Two byte quantization values may be used only 12-bit sample data.

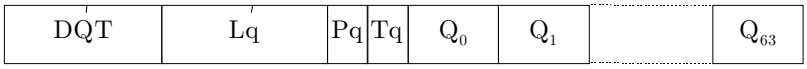


Figure 32 structure define quantization table segment

The 4 low-order bits Tq assign a numeric identifier to the table, which can be 0,1,2 or 3. The information byte is followed by 64 quantization values that are stored in JPEG zigzag order. DQT markers can appear anywhere in the compressed data stream. The only restriction is if a scan requires a quantization table, it must have been defined in a previous DQT marker.

6.12 Error Checking for Start Of Frame (SOF_n)

The SOF_n marker defines a frame. Although there are many frame types, all have the same format. The SOF marker consists of a fixed header after the marker length followed by a list of structures that define each component used by the frame. The structure of the fixed header is shown in Figure 33. Components are identified by an integer N_f in the range 0 to 255. The markers and parameters are defined below SOF_n starts of frame marker; marks the beginning of the frame parameters. The subscript n identifies whether the encoding process is basedline sequential, extended sequential, progressive, or lossless as well as coding procedure is used.

Lf: frame header length; specifies the length of the frame header

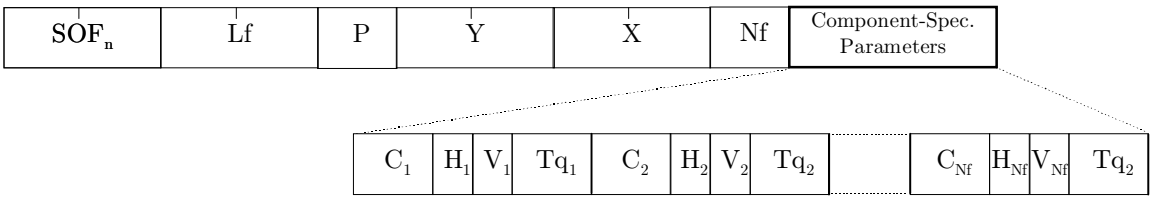


Figure 33 Structure of Frame Marker Segment

P: sample precision in bits for the samples of the components in the frame

Y: number of lines; specifies number of lines in the source image. This shall be equal to the number of lines in the component with the maximum number of vertical samples. Value 0 indicates that the number of lines shall be defined by the DNL markers and parameters marker and parameters at the end of the first scan.

X: number of samples per line; specifies the number of samples per line in the source image. This shall be equal to the number of samples per line in the component with the maximum number of horizontal samples

Nf: number of image components in frame; specifies number of sources image components in the frame. The value of Nf shall be equal to the number of sets of frame component specification parameters (C_i, H_i, V_i and Tq_i) present in the frame.

C_i component identifier; assigns a unique label to the i^{th} component in the sequence of frame component specification parameters. These values will be used in the scan headers to identify the components in the scan. The value of C_i shall be different from the values of C_1 through C_{i-1}

H_i, V_i horizontal/vertical sampling factors; specifies the number of horizontal/vertical units, respectively, of component C_i in each MCU when more than component is encoded in a scan.

Tq_i quantization table selector; select one of four possible quantization tables to use for dequantization of DCT coefficients of component C_i. If the decoding process uses the dequantization procedure, this table shall have been specified by the time the decoder is ready to decode the scan(s) containing component C_i, and shall not be re-specified until all scans containing C_i have been completed. The field sizes of frame header are summary in the table 3

Based on the above structure, the flow of SOF$_n$ marker segment is shown in Figure 34

We are going to design an error checking algorithm for the SOF$_n$ marker. The algorithm works based on the structure and value range of the fields we already know. Assume the

Table 3 Frame header field sizes assignment

Parameter	Symbol	Size (bits)
Marker (0xFFC0-3,5-7,9-B,D-F)	SOF_n	16
Frame header length	Lf	16
Sample precision	P	8
Number of lines	Y	16
Number of samples/line	X	16
Number of components in frame	Nf	8
Frame component specification (i=1..Nf)		
Component ID	C_i	8
Horizontal sampling factor	H_i	4
Vertical sampling factor	V_i	4
Q. table destination selector	Tq_i	8

Error checking Algorithm For SOF

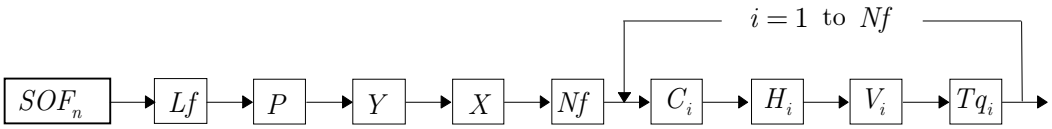


Figure 34 Structure of Frame Marker Segment

frame marker segment was created and stored in memory as an array S. The algorithm will check the structure of S and the valid values of its parameters. The procedure $SOF\_ERR\_CHECKER(S)$ accepts S as an input. To simplify the algorithm, we limit to checking for the SOF segment for the baseline DCT. In this mode, the marker SOF is 0XFFC0 (hexadecimal number), the value of P is 8; H_i, V_i are in the range 1-4, and Tq_i is in the range 0-3. The procedure $SOF\_ERR\_CHECKER(S)$ is shown below.

Line 1 extracts two bytes from the SOF segment. This is the SOF marker. Line 2 checks for the valid marker. If the invalid marker is found, it return the *false* flag to inform the error. Otherwise SOF is valid, the process is continue. Line 4 extract the Lf parameter. Line 5-7 extract and check the P parameter. If this P is invalid, it return

$SOF\_ERR\_CHECKER(S)$

1. $SOF \leftarrow Extract(S,2)$
2. **if** $SOF \neq FFC0$
3. **then return** *false*
4. $Lf \leftarrow Extract(S,2)$
5. $P \leftarrow Extract(S,1)$
6. **if** $P \neq 8$
7. **then return** *false*
8. $Skip(S,4)$

```
9.    Nf ← Extract(S,1)
10.   if  Lf ≠ 8 + 3Nf
11.         then return false
12.   for i = 1 to Nf
13.         do    skip(S,1)                    ▷ skip one byte for C_i
14.               tmp ← Extract(S,1)
15.               H_i ← (tmp >> 4) & 0x0F
16.               V_i ← tmp & 0x0F
17.               if  H_i ∉ [1,4] or V_i ∉ [1,4]
18.                     then return false
19.               Tq ← Extract(S,1)
20.               if Tq ∉ [0,3]
21.                     then return false
22.   return true
```

false to the caller. Line 8 skips four bytes contain Y and X parameters. Line 9 extracts 1 byte for the Nf parameter. Lines 10-11 check for the valid relationship between Lf and Nf. Each iteration in the loop lines 12-21 read 3 bytes from S and checks for the validate values of H_i, V_i, Tq_i parameters. Line 13 skips the parameter C_i because the range of this parameter is 0-255. Lines 14-16 extract H_i and V_i parameters. Lines 17-18 return *false* if either one of these H_i and V_i parameters contains an invalid value. Line 19-21 extract and check for the parameter Tp. Line 22 returns *true* indicates the structure SOF marker is correct.

6.13 Error Checking for Start Of Scan (SOS)

The SOS marker marks the beginning of compressed data for a scan in JPEG stream. Its structure is illustrated in Figure 35. The descriptor for each component is goes after the component count Ns. The component descriptor are ordered in the same sequence in which the component are ordered within MCUs in the scan data. While not all of the

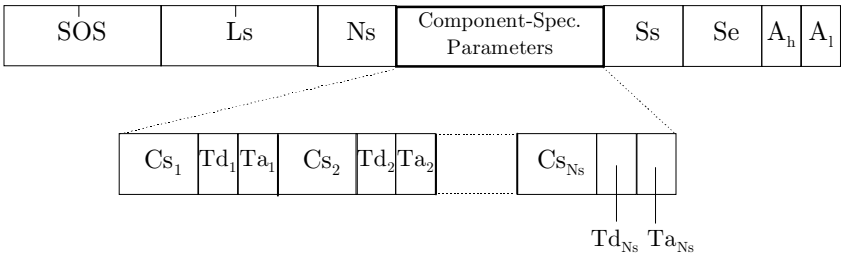


Figure 35 Structure of SOS marker segment

components from the SOF_n marker must be present, their order in the SOS marker and the SOF_n must match. The component identifier in the scan descriptor must match a

component identifier value defined in the SOF_n marker. The AC and DC Huffman table identifiers must match those of Huffman tables defined in a previous DHT marker. An SOS marker must occur after the SOF_n marker. It must be preceded by DHT markers that define all of the Huffman tables used by the scan and DQT markers that define allthe quantization tables used by the scan components. The marker and parameters shown in figure 35 are defined below.

SOS: start of scan marker; mark the beginning of the scan parameters.

Ls: scan header length; specifies the length of the scan header

Ns: number of image components in the scan; specifies the number of source image components in the scan. The value of Ns shall be equal to the number of sets of scan component specification parameters (Cs_j, Td_j and Ta_j) present in the scan.

Cs_j: scan component selector; selects which of the Nf image components specifies in the frame parameters shall be the j^{th} component in the scan. Each Cs_j shall match one of the C_i values specified in the frame header, and the ordering in the scan header shall follow the ordering in the frame header. If $Ns > 1$, the order of the interleaved components in the MCU is Cs_1 first, Cs_2 second, etc..

Td_j, Ta_j: DC/AC entropy coding table selector; selects one of four DC/AC entropy coding tables needed for decoding of the DC/AC coefficients of component Cs_j. The DC/AC entropy table selected shall have been specified by the time the decoder is ready to decode the current scan.

Ss: start of spectral or predictor selection; In the DCT modes of operation, this parameter specifies the first DCT coefficient in each block which shall be coded in the scan. This parameter shall be set to zero for the sequential DCT processes.

Se: end of spectral selection; specifies the last DCT coefficient in each block which shall be coded in the scan. This parameter shall be set to 63 for the sequential DCT processes

Ah: this parameter specifies the point transform used in the preceding scan for the band coefficients specified by Ss and Se. This parameter shall be set to zero for the first scan of each band of coefficients and no meaning in the lossless mode.

Al: successive approximation bit position low or point transform; in the DCT modes of operation this parameter specifies the point transform. This parameter shall be set to zero for the sequential DCT processes.

Error checking Algorithm

The error checking algorithm for the SOS marker segment is designed based on the SOS structure and its the valid value ranges of parameters. To simplify the algorithm, we limit to checking for the SOS segment for the baseline DCT. The size and value ranges

of the parameters in the SOS are described in the Table 4. We are going to develop an algorithm to check for the valid SOS marker. Figure 36 describes the flow of the SOS

Table 4 Start of scan header sizes and value range of parameters

Parameter	size(bits)	baseline DCT value
Ls	16	$6+2\times Ns$
Ns	8	1-4
Cs_j	8	0-255
Td_j	4	0-3
Ta_j	4	0-3
Ss	8	0
Se	8	63
Ah	4	0
Al	4	0

marker segment. The stream of SOS marker segment is generated and store in memory as shown in the Figure 37. The algorithm $SOS\_ERR\_CHECKER(S)$ extracts the data from the memory and step-by-step check for each parameter.

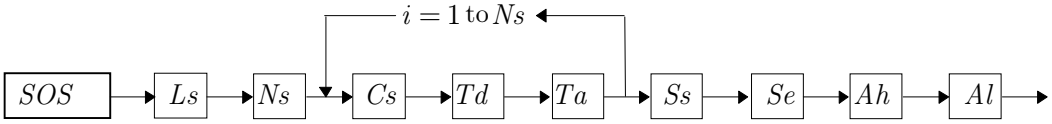


Figure 36 Flow of SOS marker segment

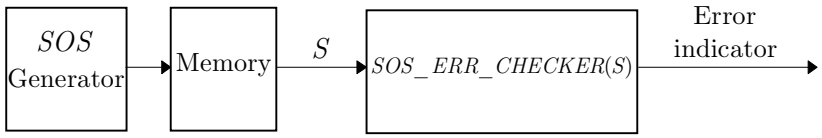


Figure 37 Error checking diagram for SOS marker segment

$SOS\_ERR\_CHECKER(S)$

1. $SOS \leftarrow Extract(S,2)$
2. **if** $SOS \neq FFDA$
3. **then return** *false*
4. $Ls \leftarrow Extract(S,2)$
5. $Ns \leftarrow Extract(S,1)$
6. **if** $Ls \neq 6+2Ns$
7. **then return** *false*
8. **for** $i = 1$ **to** Ns
9. **do** $skip(S,1)$ ▷ skip one byte for Cs

```
10.                 tmp ← Extract(S,1)
11.                 Td ← (tmp >> 4) & 0x0F
12.                 Ta ← tmp & 0x0F
13.                 if Td > 1 or Ta > 1
14.                     then return false
15.   Ss ← Extract(S,1)
16.   if Ss ≠ 0
17.         then return false
18.   Se ← Extract(S,1)
19.   if Se ≠ 63
20.         then return false
21.   Ahl ← Extract(S,1)
22.   if Ahl ≠ 0
23.         then return false
24.   return true
```

Line 1 of the $SOS\_ERR\_CHECKER(S)$ extracts two bytes from the SOS segment. Lines 2-3 check for the SOS marker code. If it is not valid, return $false$. Lines 4-5 extract Ls and Ns parameters. Line 6 checks the equality relation between Ls and Ns. If this equality is not satisfied, it returns $false$ in line 7. The loop in lines 8-14 extracts Cs, Td, Ta parameters from the memory. Since the range of the one-byte Cs is 0-255, line 9 skips this parameter. Lines 10-12 extract and compute Td and Ta. Lines 13-14 check for the validate of Td and Ta. If these parameters are not 0 or 1, it returns $false$. Lines 15-23 extract and check for the Ss, Se, Ah, Al parameters. If any of those is invalid, the procedure return $false$. Line 24 return $true$ indicates the SOS structure is correct.

6.14 Compressed Data Format

Figure 38 specifies the order of the high constituent parts of the interchange format, which begins by an SOI marker, a frame, and ends with EOI marker. The structure of frame begins with the frame header and shall contains one or more scans. A frame header may be preceded by one or more table-specification or simultaneous marker segments . If a DLN (Define Number of Lines) segment is present, it shall immediately follow the first scan. For DCT-based process each scan shall contain from one to to four image components. If 2 to 4 components are contained within the scan, they shall be interleaved within the scan.

The scan structure begins with a scan header and contains one or more entropy coded data segments. Each scan header may be preceded by one or more table specification or miscellaneous marker segments. If restarted is not enabled, there will be only one entropy coded segment. If restart is enabled, the number of entropy-coded segments is defined by the size of the image and the defined restart interval. In this case the restart marker shall follow each entropy coded segment except the last one.

To detect the error for the entire compressed data output, we break it into segments

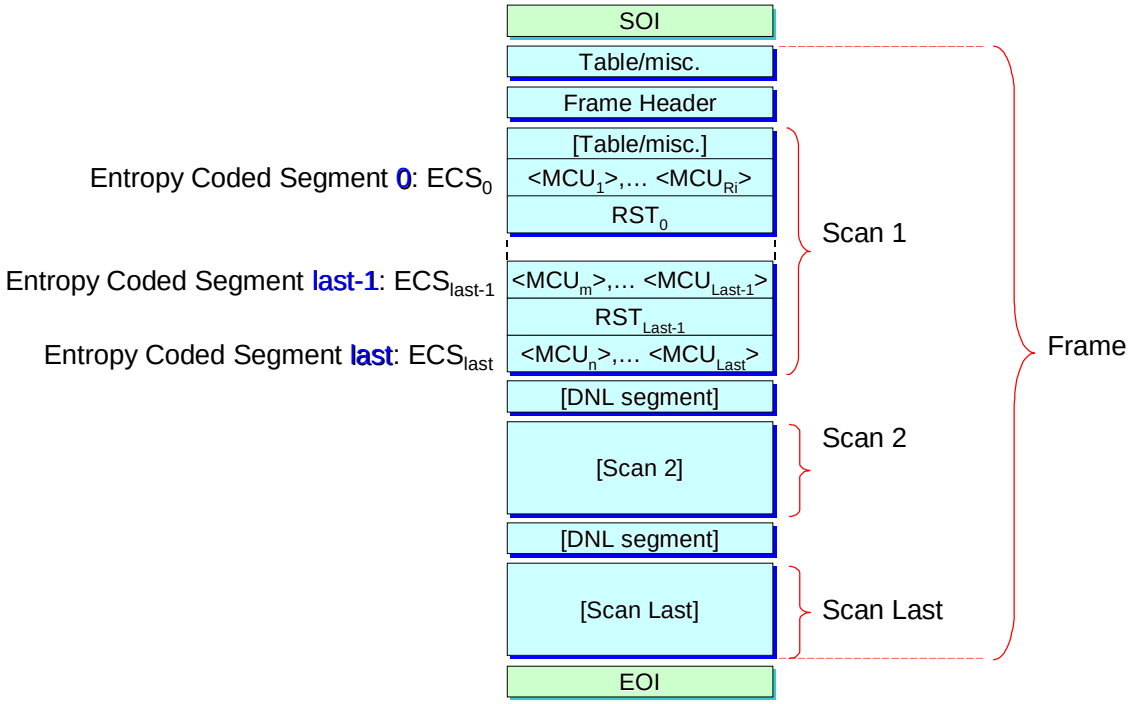


Figure 38 compressed image data

and perform the error checking for each segment, one at a time. If any errors are detected in a marker segment, this segment will be processed again. The error checking algorithm for each segment is described detail in section 6.

7 Proposal for Future Work

7.1 Fault Tolerant for JPEG 2000 Image Compression Standard

The JPEG committee has recently released its new image coding standard, JPEG 2000, which will serve as a supplement for the original JPEG standard introduced in 1992. Rather than incrementally improving on the original standard, JPEG 2000 implements an entirely new way of compressing images based on the wavelet transform, in contrast to the discrete cosine transform (DCT) used in the original JPEG standard. The significant change in coding methods between the two standards leads one to ask: What prompted the JPEG committee to adopt such a dramatic change? The answer to this question comes from considering the state of image coding at the time the original JPEG standard was being formed. At that time wavelet analysis and wavelet coding were still very new technologies, whereas DCT-based transform techniques were well established. Early wavelet coders had performance that was at best comparable to transform coding using the DCT. The comparable performance between the two methods, coupled with the considerable momentum already behind DCT-based transform coding, led the JPEG committee to adopt DCT-based transform coding as the foundation of the lossy JPEG standard.

Since the new JPEG standard is wavelet based, a much larger audience including hardware designers, software programmers, and systems designers will be interested in wavelet-based coding. One of the tasks in the future works is to comprehend the details and techniques of wavelet coding to better understand the JPEG 2000 standard. In particular, the work will focus on the fundamental principles of wavelet coding, try to clarify the design choices made in wavelet coders. The work will analyze the description of the tiling, multicomponent transformations, quantization and entropy coding (particularly is the arithmetic coding). Some of the most significant features of the standard are the region of interest coding, scalability, visual weighting, error resilience and file format aspects. Since the JPEG2000 compression standard will be largely used in the generations to come, the research and design techniques for improving system reliability become essentially tasks today. By using the redundancy techniques, we can provide the new compression system a better performance. The alternate data paths provided by parallel components (or subsystems) will be able to detect and process any computer faults happen.

7.2 Overview of the JPEG2000 Compression System

The JPEG 2000 compression engine is illustrated in block diagram form in Figure 39. At

the encoder, the discrete wavelet transform is applied on the source image data. The transform coefficients are then quantized and entropy coded (using arithmetic coding) before forming the output code stream (bit stream). The decoder is the reverse of the encoder. The code stream is first entropy decoded, dequantized, and inverse discrete wavelet transformed, thus resulting in the reconstructed image data. Although this general block diagram is similar to the one for the conventional JPEG, there are radical

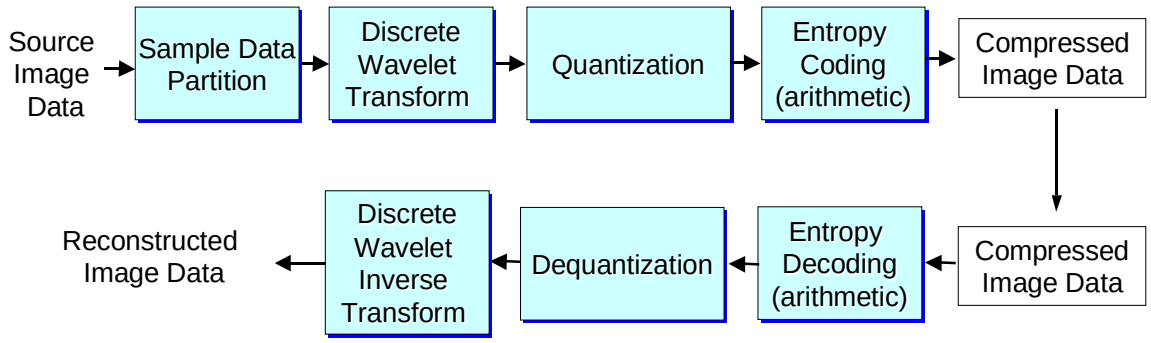


Figure 39 General block diagram of the JPEG2000 (a) encoder (b) decoder

differences in all of the processes of each block of the diagram. A quick overview of the whole system is as follows:

- The source image is decomposed into components.
- The image components are (optionally) decomposed into rectangular tiles. The tile component is the basic unit of the original or reconstructed image.

- A wavelet transform is applied on each tile. The tile is decomposed into different resolution levels.
- The decomposition levels are made up of subbands of coefficients that describe the frequency characteristics of local areas of the tile components, rather than across the entire image component.
- The subbands of coefficients are quantized and collected into rectangular arrays of "code blocks."
- The bit planes of the coefficients in a code block (i.e., the bits of equal significance across the coefficients in a code block) are entropy coded.
- The encoding can be done in such a way that certain regions of interest can be coded at a higher quality than the background.
- Markers are added to the bit stream to allow for error resilience.
  The code stream has a main header at the beginning that describes the original image and the various decomposition and coding styles that are used to locate, extract, decode and reconstruct the image with the desired resolution, fidelity, region of interest or other characteristics. For the clarity of presentation we have decomposed the whole compression engine into three parts: the preprocessing, the core processing, and the bit-stream formation part, although there exist high inter-relation between them. In the preprocessing part the image tiling, the dc level shifting and the component transformations are included. The core processing part consists of the discrete transform, the quantization and the entropy coding processes. Finally, the concepts of the precincts, code blocks, layers, and packets are included in the bit stream formation part.

7.3 Preprocessing and Component Transformations

Image Tiling The term "tiling" refers to the partition of the original (source) image into rectangular non-overlapping blocks (tiles), which are compressed independently, as though they were entirely distinct images (Fig. 40). All operations, including component mixing, wavelet transform, quantization and entropy coding are performed independently on the image tiles (Fig. 41). The tile component is the basic unit of the original or reconstructed image. Tiling reduces memory requirements, and since they are also reconstructed independently, they can be used for decoding specific parts of the image instead of the whole image. All tiles have exactly the same dimensions, except maybe those at the boundary of the image. Arbitrary tile sizes are allowed, up to and including the entire image (i.e., the whole image is regarded as one tile). Components with different subsampling factors are tiled with respect to a high resolution grid, which ensures spatial consistency on the resulting tile components. Smaller tiles create more tiling artifacts compared to larger tiles. In other words, larger tiles perform visually better than smaller tiles. Image degradation is more severe in the case of low bit rate than the case of high bit rate. It is seen, for example, that at 0.125 b/p there is a quality

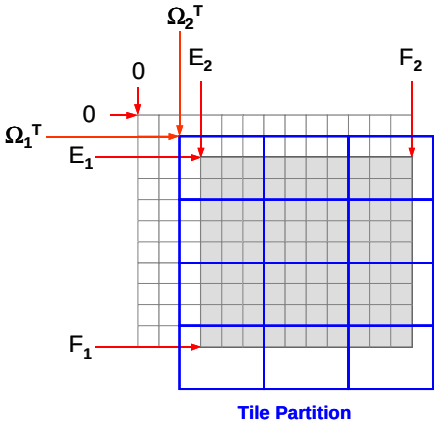


Figure 40 canvas, image region and tile partition areas

difference of more than 4.5 dB.

JPEG 2000 supports multiple component images. Different components need not have the same bit depths nor need to all be signed or unsigned. For reversible (i.e., lossless) systems, the only requirement is that the bit depth of each output image component must be identical to the bit depth of the corresponding input image component. Component transformations improve compression and allow for visually relevant quantization. The standard supports two different component transformations, one

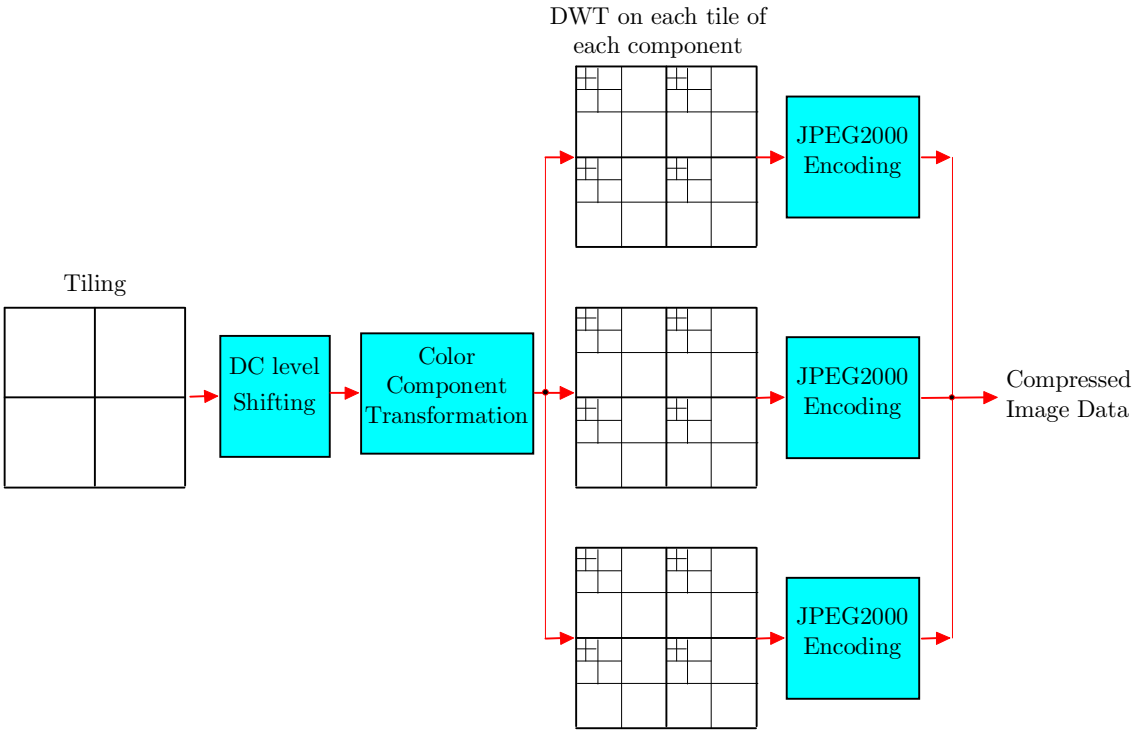


Figure 41 Tiling, dc-level shifting, color transformation and DWT of each image component

irreversible component transformation (ICT) that can be used for lossy coding and one reversible component transformation (RCT) that may be used for lossless or lossy coding, and all this in addition to encoding without color transformation. The block

diagram of the JPEG 2000 multicomponent encoder is depicted in Fig. 41 . (Without restricting the generality, only three components are shown in the figure. These components could correspond to the RGB of a color image.) Since the ICT may only be used for lossy coding, it may only be used with the 9/7 irreversible wavelet transform.

7.4 Embedded Image Coding Using Zerotrees of Wavelet (EZW) Coefficients

The EWZ algorithm is simple, yet remarkable effective, image compression algorithm. The algorithm is based on the parent-child dependencies of subband. The Figure 42 shows the tree structure of the coefficients in subbands of a tile component. The arrow points from the subband of the parents to the subband of the children. The lowest frequency subband is the top left, and the highest frequency subband is at the bottom right. The right figure shows the wavelet tree consisting of all of the descendents of a single coefficient in subband LL_3. The coefficient in LL_3 is a zerotree root if it is insignificant and all of its descendants are insignificant. The wavelet coefficients are sliced into K+1 binary arrays (or bit-planes). The first such bit-lane consists of the sign bit of each index. Each of the K magnitude bit-planes is coded into two passes. The first pass codes all refinement bits, while the second pass codes everything else (all bits not coded in the first pass). The refinement pass codes a bit for each coefficient that is significant (due to its significant bit having been coded in a previous bit-plane). The

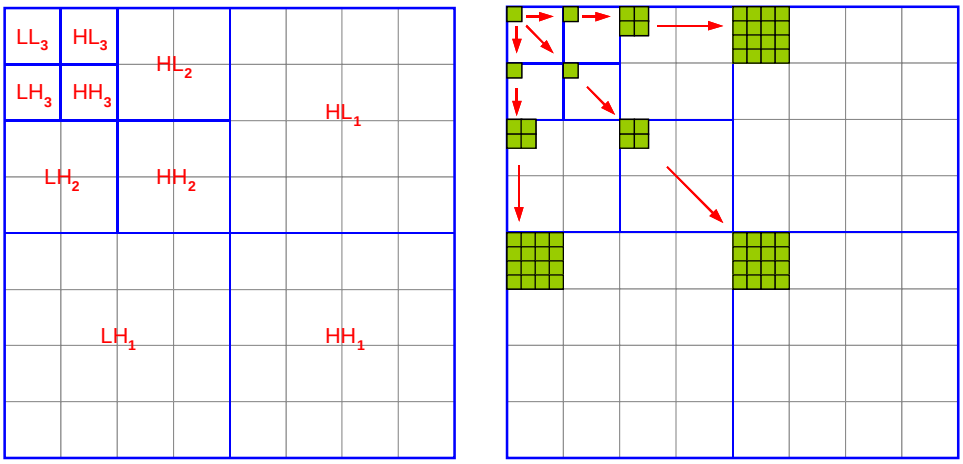


Figure 42 Parent-child dependency property of subbands

significant pass codes a bit for each coefficient that is not yet significant. In the Significant pass, each significant coefficient is visited in raster order from within LL_D, then within HL_D then LH_D, HH_D ,HL_{D-1} and so on, until it reaches HH_1. The coding is accomplished via a 4-ary alphabet

POS = Significant positive; **NEG** = Significant Negative
ZTR = Zero Tree Root; **IZ** = Isolate Zero
Z = for a zero when there are no children
As the scan of insignificant coefficients progress through subbands, any bit known

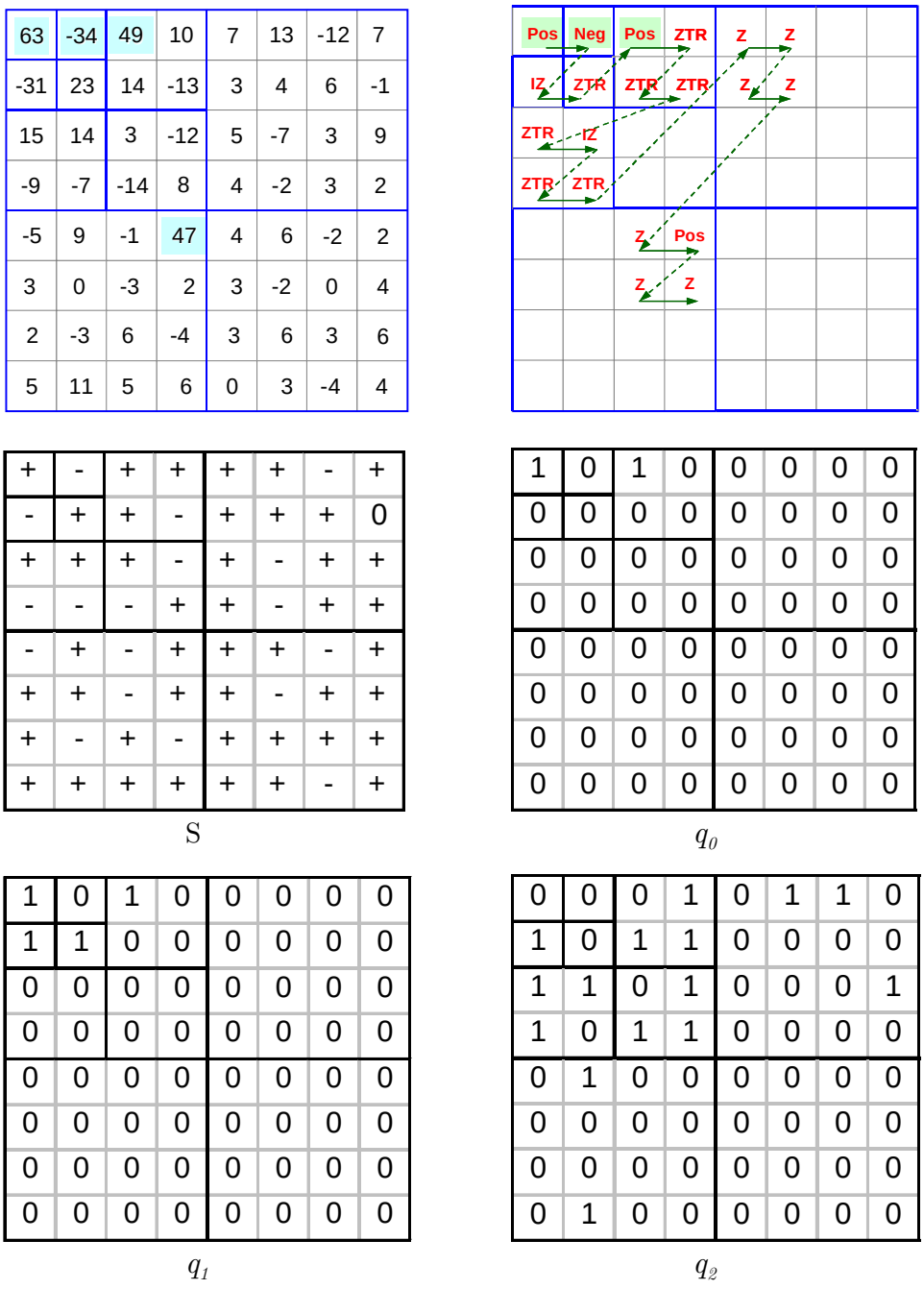


Figure 43 Example of 3-scale wavelet transform of a 8x8 image.

already to be zero is not coded again.

In the refinement pass, a refinement bit is coded for each significant coefficient. A coefficient is significant if it has been coded POS or NEG in a previous bit plane. Its current refinement bit is simply its corresponding bit in the current bit plane. The simple example in Figure 43 is used to highlight the order of operations in the EZW algorithm. The first threshold is determined by

$$T_0 = 2^{\left\lfloor \log_2(\max(|w_{i,j}|)) \right\rfloor} = 2^{\lfloor \log_2(63)) \rfloor} = 32$$

The following are the steps to be considered

1) The coefficient has magnitude 63 which is greater than threshold 32, and is positive so a positive symbol is generated. After decoding this symbol, the decoder knows the coefficient in the interval [32,64) whose center is 48.

2) Even though the coefficient 31 is insignificant with respect to the threshold 32, it has a significant descendant two generations down in subband LH1 with magnitude 47. This the symbol for an isolate zero is generated.

3) The magnitude 23 is less than 32 and all descendants in HH2 and HH1 are insignificant. A zero tree symbol is generated, and no symbol will be generated for any coefficient in subband HH2 and HH1 during the significant pass.

4) The magnitude 10 is less than 32 and all descendants (-12, 7, 6, -1) also have magnitudes less than 32. Thus the zerotree has a violation of the decaying spectrum hypothesis. The entire tree has magnitude less than threshold 32 so it is a zerotree.

5) The magnitude 14 in LH2 is insignificant with respect to 32. Its children are (-1, 47, -3, 2). Since its child with magnitude 47 is significant, an isolated zero symbol is generated.

6) No symbols were generated from HH2 which would ordinarily precede HL1 in the scan. Since HH1 has no descendants, the entropy coding can resume using a 3-symbol alphabet where IZ and ZTR symbols are merged into the Z (zero) symbol.

7) The magnitude 47 is significant with respect to 32. In the future dominant passes, this position will be replaced with the value 0, so that for the next dominant pass at the threshold 16, the parent of the coefficient, which has magnitude 14, can be coded using a zero tree coded symbol.

The code symbols for the significant pass of q_0 are given by

Pos Neg IZ ZTR

Pos ZTR ZTR ZTR ZTR IZ ZTR ZTR

Z Z Z Z Z Pos Z Z

The sign plane and first three magnitude bitplanes for the quantized coefficients are shown in the Figure 43. The significant pass contains four significant coefficients q[0,0], q[0,1], q[0,2], q[4,3]. The code symbols for the refinement pass q_1[0,0], q_1[0,1], q_1[0,2], q_1[4,3] = 1,0,1,0. The q_1 significance pass has added two more significant coefficients q[1,0] and q[1,1]. One bit must be coded for each of the (now six) significant coefficiens during the q_2 refinement pass. The order to code the coefficients such that the bits formed by q_0[i,j] q_1[i,j]=3, then q_0[i,j] q_1[i,j]=2, and q_0[i,j] q_1[i,j]=1. The symbol coded in the refinement pass are thus

q_2[0,0], q_2[0,2], q_2[0,1], q_2[4,3], q_2[1,0], q_2[1,1] = 1,0,0,1,1,0.

The coding continues with the q_2 significance pass, the q_3 refinement pass, and so on.

7.5 The Work Plan for the JPEG2000 Standard

The main goal of the future research are exploring the structures of the JPEG2000 compression system, modeling all possible hardware and software errors in its compression stages. The obvious technique for error detection and correction is adding redundancy hardware or software to the compression system. Although a redundancy subsystem itself can encounter some faults, the probability of both main compression stage and it error checker to be failed is quite small. One can further enhance the reliability for the error checking systems by providing repair for the failed system. The error checking and correcting subsystem for each stage will not affect the standard and the algorithm assigned to that stage. Modeling the number of software errors and the frequency with which they cause the system failures requires approaches that differ from hardware reliability.

Software is made more reliable by testing to find and remove errors, thereby lowering the error probability. Our main goal in the software error detection and correction for the JPEG2000 standard is to design the error detection/correction algorithms the intermediate data structures and the final data compression format as we have shown in the traditional JPEG compression system. As we processed the fault tolerant strategies for the conventional JPEG, we will design error detection models and correction strategies for the JPEG2000 image compression system, including the analysis of the JPEG2000 image compression standard, system, operations on different data types, data structures and fault models for each compression stage. After that, we will design the redundancy subsystem to detect and handle these error. The error performance, compression performance and all overheads will be evaluated via computer simulations. The analysis and design the error checking system for JPEG2000 image coding include the works on the following stages

- Data ordering stage. In this stage, we will analyze following techniques
  - Image division into components
  - Image division into tiles and tile-components
  - Tile-component division into resolution & sub-bands
  - Division of the sub-bands into code-blocks
  - Error checking model for the data ordering stage
  - Design a redundancy system to detect error that works with the proposed model
- Arithmetic entropy coding stage.
  - Description of the arithmetic encoder
  - Arithmetic decoding procedure
  - Binary encoding
  - Error checking model for the Arithmetic entropy coding stage
  - Design an error detection scheme that works with the proposed model

- Discrete wavelet transformation of tile components
  - Subband coding
  - Wavelet transform basics
  - Orthogonal and biorthogonal wavelet transform
  - Reversible and irreversible filtering
  - Zerotrees of wavelet coefficients
  - Inverse discrete wavelet transform
  - Error checking model for every operation in the DWT
  - Design an error detection scheme that works with the proposed model
- Coefficient bit modeling
  - Code-block scan pattern
  - Decoding passes over the bit-planes
  - Initialize and terminating
  - Error checking model for the coefficient bit processing stage
  - Design an error detection scheme that works with the proposed model
- Sample data partitions
  - tile partition
  - code block and precincts
  - packet construction
- JPEG2000 file format syntax
  - file format scope
  - grayscale/color/palettized/multi-component specification architecture
  - box definition
  - Design error checking algorithms for the JPEG2000 file format
- Code stream syntax
  - Header and marker segments
  - Construction of the code stream
  - Design error checking algorithms for header, marker segments and data stream

7.6 Evaluations the Fault models Performance

The performance of the designed fault models will be evaluated using the Matlab programming tool. With the convenient syntax and the build in data structure, we can be easily to generate the error detection system that is fitted with the our design models and specifications. The error checking performance is evaluated by generate different sources of errors. Then these errors are injected into the compression system at appropriate locations and rates. At the output, we will receive the corrupted data image, and the error indicator signals. The fault detection performance will based on the analysis of the corrupted data, the error injection and the error detection signals. Matlab

supports the timing for all operations. This tool will be used to evaluate the time cost due to the present of the redundancy error checking system. Although this time overhead is not applicable into practice, the percentage of time overhead can give us a meaningful quantity for the system we design. The compression performance will be evaluated using the size of the compressed data output (with or without the error correction codes involved) and the original data input.

Conclusions

As we have seen, the approach to fault tolerance design in JPEG compression and decompression systems is redundancy, that is, to have an additional element to detect and handle errors if an operating element fails. Fault-tolerant, in general, is necessary in the JPEG compression system in order to raise the level of safety. Since a failure could end up with a defective image and can jury or kill many people.

There are various ways to employ redundancy to enhance the reliability of the compression system. Beside that, we employed software redundancy. Although all copies of the same software will have the same fault and should process the same scenario identically. However, there are some errors due to the interactions of the inputs, the state of the hardware, and any residual faults. In this proposal we will not intend to build different independent versions of the software to provide software reliability. We will only design the redundancy algorithms to check for the valid data values and data structures. For example, we designed the error check algorithm for the definition Huffman table segment. This algorithm works with the assumption there is no existence of hardware errors.

Since JPEG 2000 implements an entirely new way of compressing images based on the wavelet transform, arithmetic coding, and many others different techniques, the fault tolerant schemes for the new compression system needed to be modified. I plan to work on fault tolerant designs for the new JPEG 2000 compression system. The tasks are organized in similar way to those have been done for the conventional JPEG.

Appendices

A. Mathematical Preliminaries for Lossless Compression

Compression schemes can be divided into two classes, lossy and lossless. Lossy compression schemes involve the loss of some information, and data that have been compressed using a lossy scheme generally cannot be recovered exactly. Lossless schemes compress data without loss of information, and the original data can be recovered exactly from the compressed data. In this appendix, some of the ideas in information theory that provide the framework for the development of lossless data compression schemes are briefly reviewed.

A.1 Introduction of Information Theory

The *self-information* quantity, by Shannon is determined based on the occurrence probability of occurrence of an event. If $P(A)$ is the probability that the event A will occur, then the self-information associated with A is given by

$$i(A) = -\log_b P(A) \tag{a1}$$

Thus, if the probability of information is low, the amount of self-information associated with it is high and vice-versa. The information obtained from the occurrence of two independent events is the sum of the information obtained from the occurrence of the individual events. Suppose A and B are two independent events. The self-information associated with the occurrence of both events A and event B is

$$i(AB) = -\log_b P(AB) = -\log_b P(A)P(B) = -\log_b P(A) - \log_b P(B) = i(A) + i(B) \tag{a2}$$

The unit of information depends on the base of the log. If we use log base 2, the units is *bits*; if we use the log base e, the unit is *nats*; and if we use the log base 10, the unit is *hartleys*. If we have a set of independent events A_i, which are set of outcomes of some experiment $\mathcal{S}$, such that

$$\bigcup A_i = S \tag{a3}$$

where S is the sample space, then the self-information associated with the random experiment is given by

$$H = \sum P(A_i) i(A_i) = -\sum P(A_i) \log_b P(A_i) \tag{a4}$$

This quantity is called the entropy associated with the experiment. One of the many contributions of Shannon was he showed that if the experiment is a source that puts out symbols A_i from a set $\mathcal{A}$, then the entropy is a measure of the average of the number of binary symbols needed to code the output of the source. Shannon showed that the best that the lossless compression scheme can do is to encode the output of a source with an

average number of bits equal to the entropy of the source. For general source $\mathcal{S}$ with alphabet $\mathcal{A} = \{1,2,\ldots,m\}$ that generates a sequence $\{X_1, X_2, \ldots\}$, the entropy is given by

$$H(S) = \lim_{n\to\infty} \frac{1}{n} G_n \tag{a5}$$

where

$$G_n = -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\cdots\sum_{i_n=1}^{m} P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n)\log P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n) \tag{a6}$$

and $\{X_1, X_2, \ldots\}$ is a sequence of length n from the source. If each element in the sequence is independent and identically distributed (*iid*), then we can show that

$$G_n = -n\sum_{i_1=1}^{m} P(X_1 = i_1)\log P(X_1 = i_1) \tag{a7}$$

and (a5) becomes

$$H(S) = -\sum P(X_1)\log P(X_1) \tag{a8}$$

For most sources Equations (a5) and (a8) are not identical. If we need to distinguish between the two, we will call the quantity computed in (a8) the *first-order entropy* of the source, while the quantity in (a5) will be referred to as the *entropy* of the source. For example, consider a source $\mathcal{S}_1 = (S_1, \mathcal{P}_1)$ for which each source symbol is equally likely to occur, that is, for which $\mathcal{P}_1(s_i) = p_i = 1/q$, for all $i = 1,2,\ldots,q$. Then

$$H(S_1) = H\left(\tfrac{1}{q},\ldots,\tfrac{1}{q}\right) = \sum_{i=1}^{q} p_i \lg\frac{1}{p_i} = \lg q \tag{a9}$$

On the other hand, for a source $\mathcal{S}_2 = (S_2, \mathcal{P}_2)$,where $p_1 = 1$ and $p_i = 0$ for all $i = 2,3,\ldots,q$, we have

$$H(S_2) = H(1,0,\ldots,0) = p_1 \lg\frac{1}{p_1} = 0 \tag{a10}$$

The previous example illustrates why the entropy of a source is often thought of as a measure of the amount of uncertainty in the source. The source $\mathcal{S}_1$, which emits all symbols with equal probability, is in a much greater state uncertainty than the source $\mathcal{S}_2$, which always emits the same symbol. Thus, the greater entropy, the greater the uncertainty in each sample and the more information is obtained from the sample.

A.2 Probability Model

Good models for sources lead to more efficient compression algorithms. In general, in order to develop techniques that manipulate data using mathematical operations, we

need to have a mathematical model for the data. The better the model, the more likely it is that we will come up with a satisfactory technique.

The simplest statistical model for the source is to assume that each letter that is generated by the source is independent of every other letter, and each occurs with the same probability. This is so called the ignorance model, as it would generally be useful only when we know nothing about the source. For a source that generates letters from an alphabet $\mathcal{A} = \{a_1, a_2, ..., a_M\}$, we can have a probability model $\{P(a_1), P(a_2), ..., P(a_M)\}$. Given a probability model with the independent assumption, we can compute the entropy of the source using the equation (a8). By using the probability model, we can also construct some very efficient codes to represent the letters in $\mathcal{A}$. Of course, these codes are only efficient in our mathematical assumptions are in accord with reality. If the assumption of independence does not fit with the observation of the data, we can find better compression schemes if we discard this assumption. We then come up with a way to describe the dependence of elements of the data sequence on each other.

A.3 Markov Models

One of the most popular ways of representing dependence in the data is through the use of Markov models. For models used in lossless compression, we use a specific type of Markov process called a *discrete time Markov chain*. Let $\{x_n\}$ be a sequence of observations. This sequence is said to follow a kth-order Markov model if

$$P(x_n|x_{n-1},...,x_{n-k}) = P(x_n|x_{n-1},...,x_{n-k},...) \tag{a11}$$

Thus, knowledge of the past k symbols is equivalent to the knowledge of the entire past history of the process. The values taken on by the set $\{x_1, x_2, ..., x_{n-k}\}$ are called the *states* of the process. If the size of the source alphabet is l, then the number of states is l^k. The most commonly used Markov model is the first-order Markov model, for which

$$P(x_n|x_{n-1}) = P(x_n|x_{n-1}, x_{n-2}, ...) \tag{a12}$$

Equations (a11) and (a12) indicate the existence of dependence between samples. If we assumed that the dependence was introduced in a linear manner, we could view the data sequence as the output of a linear filter driven by white noise. The output of such a filter can be given by the difference equation

$$x_n = \rho x_{n-1} + \varepsilon_n \tag{a13}$$

where ε_n I a white noise process. This model is often used when developing coding algorithms for speech and images. The entropy of a finite state process with states S_i is simply the average value of the entropy at each state:

$$H = \sum_{i=1}^{M} P(S_i) H(S_i) \tag{a14}$$

B. Fast Discrete Cosine Transform

The Discrete Cosine Transform (DCT) is a crucial part of modern image and sound compression. It is used in JPEG, MPEG video, MPEG audio, and Digital VCR, to name just a few. Mathematically, the DCT provides a way to break a complex signal down into separate components, and does so in a way that's nearly optimal for compression.

The DCT converts a single block of data into a collection of DCT coefficients. Intuitively, these coefficients can be though as the representing different frequency components. The first coefficient (the DC coefficient) is simply the average of the entire block. Later coefficients (the AC coefficients) represent successively higher frequencies. For image compression, higher frequency roughly corresponds to finer detail.

A typical compression algorithm starts by breaking the data into small blocks. A DCT is applied to each block. Each coefficients is then multiplied by a fixed weight; higher frequency values coefficients typically use smaller weights. The result is that the higher frequency values become small, zeros usually predominate. Finally, standard compression techniques such as Huffman, arithmetic coding, or simple run-length coding are used to pack the coefficients into a small number of bits. Decompression works in reverse. First, the bits are encoded to yields a series of weighted coefficients. Then, each coefficient is divided by a corresponding weight and an Invert DCT is used to recover the final values.

However, the computational overhead of the DCT is an obstacle to efficient implementation of these compression algorithms. In a typical DCT-based encoder/decoder, DCT calculations alone can easily take up 50 percent of the CPU time for the entire program. Since the DCT has been recognized as one of the standard techniques in image compression, an algorithm which rapidly computes DCT has become a key component in image compression.

This project will explore several fast algorithms for computing the 8-point DCT and IDCT. These algorithm will be presented by the diagrams and then translating into matlab and C codes.

B.1 Overview The 1D and 2D DCTs

DCT based graphics compression usually employs an 8×8 DCT. For this reason, there has been extensive study of this particular DCT. Equation (1) is the one dimensional (1D) N-element DCT. Equation (2) is corresponding equation for the 1D N-element invert DCT.

$$y_k = \alpha_k \sum_{n=0}^{N-1} x_n \cos\frac{\pi k(2n+1)}{2N}, \qquad k = 0,1,\ldots,N-1 \tag{1}$$

$$x_n = \sum_{k=0}^{N-1} \alpha_k y_k \cos\frac{\pi k(2n+1)}{2N}, \qquad n = 0,1,\ldots,N-1 \tag{2}$$

$$\alpha_k = \begin{cases} \sqrt{1/N}, & k = 0 \\ \sqrt{2/N}, & k = 1,2,\ldots,N-1 \end{cases}$$

The DCT has the property that, for a typical image, most of the visually significant information about the image is concentrated in just a few coefficients of the DCT. For this reason, the DCT is often used in image compression applications. The DCT is at the heart of the JPEG standard lossy image compression algorithm.

The two-dimensional DCT of an M-by-N matrix **X** is defined as follows.

$$y_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{mn} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad 0 \le p \le M-1,\ 0 \le q \le N-1 \qquad (3)$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases}, \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

The values y_{pq} are called the *DCT coefficients* of **X**. (Note that matrix indices in MATLAB always start at 1 rather than 0; therefore, the MATLAB matrix elements $X(1,1)$ and $Y(1,1)$ correspond to the mathematical quantities x_{00} and y_{00}, respectively.)

The DCT is an invertible transform, and its inverse is given by

$$x_{mn} = \sum_{p=0}^{M-1} \sum_{q=0}^{N-1} \alpha_p \alpha_q y_{pq} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad 0 \le m \le M-1,\ 0 \le n \le N-1 \qquad (4)$$

$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p = 0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases}, \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q = 0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases}$$

The inverse DCT equation can be interpreted as meaning that any M-by-N matrix A can be written as a sum of functions of the form

$$\alpha_p \alpha_q \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \qquad 0 \le p \le M-1,\ 0 \le q \le N-1 \qquad (5)$$

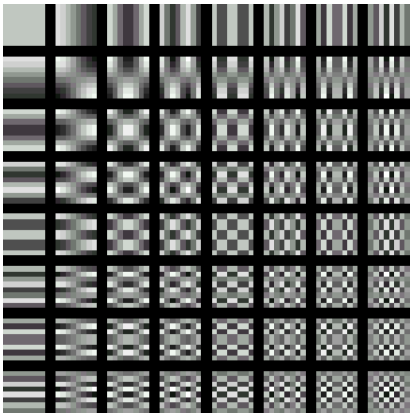


Figure 44: The 64 Basis Functions of an 8-by-8 Matrix

These functions are called the *basis functions* of the DCT. The DCT coefficients, then, can be regarded as the *weights* applied to each basis function. For 8-by-8 matrices, the 64 basis functions are illustrated by the image as show on Figure 44.

For an M-by-M matrix **X**, **T***X is an M-by-M matrix whose columns contain the one-dimensional DCT of the columns of **X**. The two-dimensional DCT of **X** can be computed as **Y**=**T***X***T**$^{\mathbf{T}}$. Since **T** is a real orthonormal matrix, its inverse is the same as its transpose. Therefore, the inverse two-dimensional DCT of **Y** is given by **T**$^{\mathbf{T}}$***Y***T**.

B.2 Fast Algorithms of The 4 and 8-point DCTs

There were existing many fast algorithms of 1D and 2D DCTs achieved good improvements in computation complexities. Particularly, the algorithm. results arithmetic complexity of (N/2)log(N) multiplications and (3N/2)log(N)-N+1 additions for input sequence with radix 2 length (N = 2^m). It based on direct decomposition of the DCT. The derivation of the algorithm related to the application of the Kronecker matrix product as a construction tool. The sequential splitting method was used for proofing the correctness of the algorithm.

The 2D forward and inverse transforms can be written as a triple matrix product **Z**=**AXA**^{T} and **X**=**A**^{T}**ZA**, where **AA**^{T} = **I**$_N$. The decomposition to a triple matrix product requires $2N^3$ multiplications to be performed, which requires 2N multiplies to be computed per input sample. The 2D DCT can be broken down into 1D DCT's (or IDCT's). The first computes **Y** = **AX** (or **A**^{T}**X**) and the second computes **Z** = **YA**^{T} (or **YA**). The N×N matrix-matrix multiply has been separated into N matrix-vector products.. Thus, the basic computation performed by the DCT (IDCT) is the evaluation of the product between the (N×N) matrix and the (N×1) vector. Each 1D DCT (or IDCT) unit must be capable of computing N multiplies per input sample to perform a matrix vector product. If the input block **X** is scanned column by column, the intermediate product **Y** = **AX** (or **A**^{T}**X**) is also computed column by column. However, since the entire row of **Y** has to be computed prior to the evaluation of the next 1D DCT, the intermediate result **Y** must be stored in an on-chip buffer. Since columns are written into the buffer and rows are read from it, it is commonly called the *transposed memory*.

The first 1D DCT/IDCT unit operates on rows of **A** (or **A**^{T}) and columns of **X**, while the second 1D DCT unit operate on the rows of **A**^{T} (or **A**) is equivalent to a row of **A**(**A**^{T}), the second 1D DCT/IDCT is unnecessary if we can multiplex the first 1D DCT/IDCT unit between the column of **X** and the row of **Y**. However, the 1D DCT/IDCT unit must now process samples at twice the input sample rate. Thus the 1D DCT/IDCT unit must be capable of computing 2N multiplies per input sample. The following section exploits the features of the 1D DCT/IDCT to reduce the computation overhead of the basic building block. Consider for the case N = 8, the 8×8 basic DCT matrix, can be written as

$$\mathbf{A}=\begin{bmatrix} a & a & a & a & a & a & a & a \\ b & d & e & g & -g & -e & -d & -b \\ c & f & -f & -c & -c & -f & f & c \\ d & -g & -b & -e & e & b & g & -d \\ a & -a & -a & a & a & -a & -a & a \\ e & -b & g & d & -d & -g & b & -e \\ f & -c & c & -f & -f & c & -c & f \\ g & -e & d & -b & b & -d & e & -g \end{bmatrix} \tag{6}$$

where $a=\frac{1}{2}\cos\frac{\pi}{4},\quad b=\frac{1}{2}\cos\frac{\pi}{16},\quad c=\frac{1}{2}\cos\frac{\pi}{8},\quad d=\frac{1}{2}\cos\frac{3\pi}{16},\quad e=\frac{1}{2}\cos\frac{5\pi}{16},\quad f=\frac{1}{2}\cos\frac{3\pi}{8},\quad g=\frac{1}{2}\cos\frac{7\pi}{16}.$

Even rows of A are symmetric and odd rows are anti symmetric. By exploiting this symmetry in the rows of an **A**, and separating the even and odd rows, we get

$$\begin{bmatrix} Y(0) \\ Y(2) \\ Y(4) \\ Y(6) \end{bmatrix}=\begin{bmatrix} a & a & a & a \\ c & f & -f & -c \\ a & -a & -a & a \\ f & -c & c & -f \end{bmatrix}\begin{bmatrix} X(0)+X(7) \\ X(1)+X(6) \\ X(2)+X(5) \\ X(3)+X(4) \end{bmatrix} \tag{7a}$$

$$\begin{bmatrix} Y(1) \\ Y(3) \\ Y(5) \\ Y(7) \end{bmatrix}=\begin{bmatrix} b & d & e & g \\ d & -g & -b & -e \\ e & -b & g & d \\ g & -e & d & -b \end{bmatrix}\begin{bmatrix} X(0)-X(7) \\ X(1)-X(6) \\ X(2)-X(5) \\ X(3)-X(4) \end{bmatrix} \tag{7b}$$

After we pulled out the constant out off the square matrices and the replaced $a, b..., g$ by the following symbols

$$a\to 1,\ b\to c_1,\ c\to c_2,\ d\to c_3,\ e\to s_3,\ f\to s_2,\ g\to s_1 \tag{8}$$

where $$c_1=\cos\frac{\pi}{16},\ c_2=\cos\frac{2\pi}{16},\ c_3=\cos\frac{3\pi}{16},\ s_1=\sin\frac{\pi}{16},\ s_2=\sin\frac{2\pi}{16},\ s_3=\sin\frac{3\pi}{16} \tag{9}$$

We have

$$\begin{bmatrix} Y(0) \\ Y(2) \\ Y(4) \\ Y(6) \end{bmatrix}=\frac{1}{2\sqrt{2}}\begin{bmatrix} 1 & 1 & 1 & 1 \\ \sqrt{2}c_2 & \sqrt{2}s_2 & -\sqrt{2}s_2 & -\sqrt{2}c_2 \\ 1 & -1 & -1 & 1 \\ \sqrt{2}s_2 & -\sqrt{2}c_2 & \sqrt{2}c_2 & -\sqrt{2}s_2 \end{bmatrix}\begin{bmatrix} X(0)+X(7) \\ X(1)+X(6) \\ X(2)+X(5) \\ X(3)+X(4) \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} Y(1) \\ Y(3) \\ Y(5) \\ Y(7) \end{bmatrix}=\frac{1}{2\sqrt{2}}\sqrt{2}\begin{bmatrix} c_1 & c_3 & s_3 & s_1 \\ c_3 & -s_1 & -c_1 & -s_3 \\ s_3 & -c_1 & s_1 & c_3 \\ s_1 & -s_3 & c_3 & -c_1 \end{bmatrix}\begin{bmatrix} X(0)-X(7) \\ X(1)-X(6) \\ X(2)-X(5) \\ X(3)-X(4) \end{bmatrix} \tag{11}$$

Without loss of general, we can omit the scale constants $1/2\sqrt{2}$ in the equations (10) and (11). Equation (10) can be split into two equations:

$$\begin{bmatrix} Y(0) \\ Y(4) \end{bmatrix}=\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} X(0)+X(7)+X(3)+X(4) \\ X(1)+X(6)+X(2)+X(5) \end{bmatrix} \tag{12a}$$

$$\begin{bmatrix} Y(2) \\ Y(6) \end{bmatrix}=\sqrt{2}\begin{bmatrix} c_6 & s_6 \\ -s_6 & c_6 \end{bmatrix}\begin{bmatrix} X(1)+X(6)-X(2)-X(5) \\ X(0)+X(7)-X(3)-X(4) \end{bmatrix} \tag{12b}$$

Similar, the equation (11) can be split into two equations

$$\begin{bmatrix} Y(5) \\ Y(3) \end{bmatrix} = -\sqrt{2} \begin{bmatrix} -s_1 & c_1 \\ c_1 & s_1 \end{bmatrix} \begin{bmatrix} X(2)-X(5) \\ X(1)-X(6) \end{bmatrix} + \begin{bmatrix} c_3 & s_3 \\ -s_3 & c_3 \end{bmatrix} \begin{bmatrix} X(3)-X(4) \\ X(0)-X(7) \end{bmatrix} \tag{13a}$$

$$\begin{bmatrix} Y(7) \\ Y(1) \end{bmatrix} = \sqrt{2} \begin{bmatrix} -c_1 & s_1 \\ s_1 & c_1 \end{bmatrix} \begin{bmatrix} X(3)-X(4) \\ X(0)-X(7) \end{bmatrix} + \begin{bmatrix} c_3 & -s_3 \\ s_3 & c_3 \end{bmatrix} \begin{bmatrix} X(2)-X(5) \\ X(1)-X(6) \end{bmatrix} \tag{13b}$$

The signal flow graph for the equations (12) and (13) is shown on Figure 45, where the outputs named 0, 4, 2, 6, 3, 5 are obviously identical to their equations in (12) and (13a). The outputs 1,7 need to be verified equivalent to the equation (13b). From the Figure 45, we get

$$\begin{aligned} Y(7) &= -[-s_1(X(2)-X(5)) + c_1(X(1)-X(6)) + c_3(X(3)-X(4)) + s_3(X(0)-X(7))] + \\ &\quad + [c_1(X(2)-X(5)) + s_1(X(1)-X(6)) - s_3(X(3)-X(4)) + c_3(X(0)-X(7))] \\ &= -[X(3)-X(4)](c_3+s_3) + [X(0)-X(7)](c_3-s_3) + [X(2)-X(5)](c_1+s_1) \\ &\quad -[X(1)-X(6)](c_1-s_1) \\ &= \sqrt{2}[-(X(3)-X(4))c_1 + (X(0)-X(7))s_1 + (X(2)-X(5))c_3 - (X(1)-X(6))s_3] \end{aligned} \tag{14a}$$

$$\begin{aligned} Y(1) &= [c_3(X(3)-X(4)) + s_3(X(0)-X(7)) + c_1(X(2)-X(5)) + s_1(X(1)-X(6))] + \\ &\quad + [-s_3(X(3)-X(4)) + c_3(X(0)-X(7)) - s_1(X(2)-X(5)) + c_1(X(1)-X(6))] \\ &= -[X(3)-X(4)](c_3-s_3) + [X(0)-X(7)](c_3+s_3) + [X(2)-X(5)](c_1-s_1) \\ &\quad + [X(1)-X(6)](c_1+s_1) \\ &= \sqrt{2}[-(X(3)-X(4))s_1 + (X(0)-X(7))c_1 + (X(2)-X(5))s_3 - (X(1)-X(6))c_3] \end{aligned} \tag{14b}$$

The last lines of the equations (14a) and (14b) are identical to equation (13b).

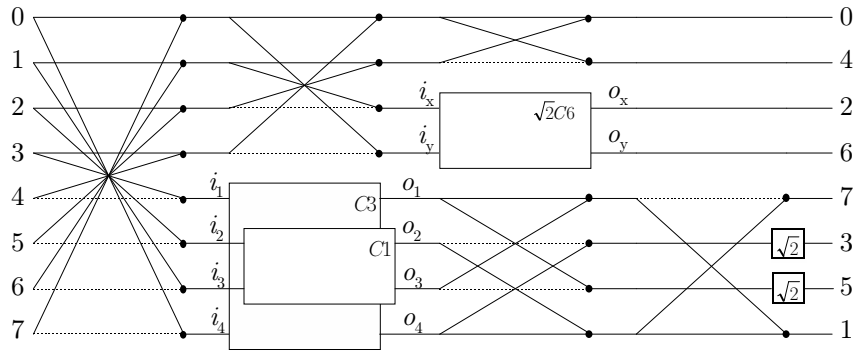


Figure 45 Fast 8-element 1D DCT

Note that in the Figure 45, the boxes C1 and C3 perform the rotation operations

$$o_x = \sqrt{2}i_x\cos(6\pi/16) + \sqrt{2}i_y\sin(6\pi/16), \quad o_y = -\sqrt{2}i_x\sin(6\pi/16) + \sqrt{2}i_y\cos(6\pi/16)$$

$$o_1 = i_1\cos(3\pi/16) + i_4\sin(3\pi/16), \quad o_4 = -i_1\sin(3\pi/16) + i_4\cos(3\pi/16)$$

$$o_2 = i_2\cos(\pi/16) + i_3\sin(3/16), \quad o_3 = -i_2\sin(\pi/16) + i_3\cos(\pi/16)$$

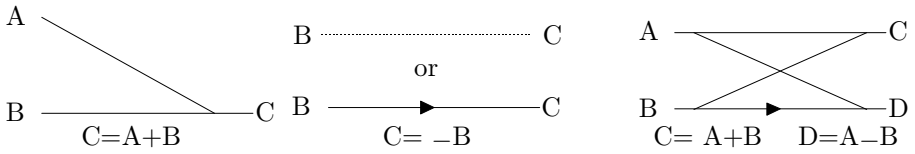


Figure 46 Flow diagram notation

B.3 Implementation the Fast Algorithm and its C Code

The starting point for a 2D DCT is a fast 1D DCT. Figure 45 is a flow diagram for our fast algorithm. If we step through the diagram, it is remarkably routine to convert this diagram into efficient code. To demonstrate the process, it is convenience to consider the converting of four-point DCT diagram in Figure 47 into C code. The input Figure 47(a) are labeled $x0$ through $x3$, and $x4$ is an additional variable we need for our computations. In Figure 47(b), the sum $x0+x3$ is stored in $x4$. In Figure 47(c), the difference $x0$-$x3$ is stored in $x0$, and $x3$ is now available for Figure 47(d). The remaining add/subtract pairs are calculated following the same pattern. This strategy will optimize the memory that used for variables.

For the rotation part, we can use a temporary variable to reduce the calculations to only three multiplications. By applying the formula from Figure 47(f) into the four-point DCT, we let

$$temp = (A + B)\sqrt{2}\cos(3\pi/8) \tag{15a}$$

$$C = temp + B\sqrt{2}(\sin 3\pi/8 - \cos 3\pi/8) \tag{15b}$$

$$D = temp + A\sqrt{2}(-\sin 3\pi/8 - \cos 3\pi/8)$$

From 15(a) and 15(b), we conduct the output C and D by using only three

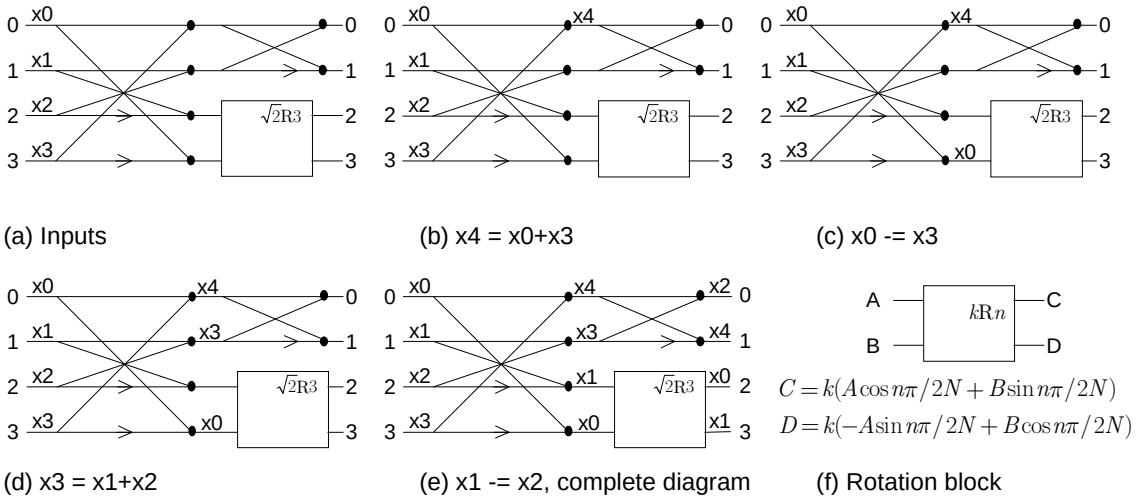


Figure 47 Flow diagram for 4 element DCT

multiplications instead of four as shown in Figure 47(f). Assume we used 10-bit fixed point for the constants in the rotation. This means, at the end of algorithm, we shift the

outputs of the rotation to the right by 10 bits. Before doing so, we add the constants with 512, which is ½ in 10-bit fixed-point arithmetic. This properly rounds the result, and significantly improves the accuracy.

B.4 Performance

Although optimizing a program for speed often requires sacrificing accuracy, fast DCT algorithms can actually be more accurate than slower algorithms. The reason is each multiplication involves some loss of accuracy due to round off. Algorithms with fewer multiplications are therefore not only faster, but also more accurate. The only place that practical DCT implementations sacrifice accuracy is by using fix-point rather than floating-point arithmetic. With careful attention to rounding, however, even fixed-point implementations can be extremely accurate.

The DCT testing program computes the DCT in two different ways, and compares the results. The reference version is a direct implementation of the formulas using double precision floating-point. The measurements of interest are: How much error outputs are? How fast the algorithms achieved compare to the direct computation using the formula? Another way to measure the error is to compute the mean square error.

Simulation were carried out on the set of 10,000 8×8 block. These blocks were generated by the random number generator that is shown in the code. Simulation carried out for the different lower and upper bounds on the random numbers: (L= -256,H=255); (L=H=5) and (L=H=300). Figure 48 plots the peak and the overall mean square error for the case (L=-256,H=255). An internal wordlength of 20 bits has been found to be adequate.

The error characteristics of the process are summarized in Table 5-7

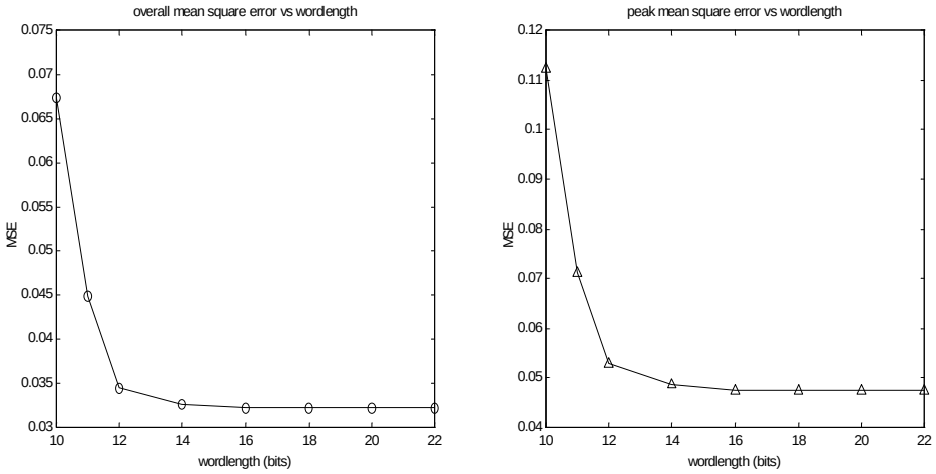


Figure 48 Peak (left) and Overall (right) mean square error vs. wordlength

Table 5 Characteristics of the process

	L, H = -256, 256	L, H = 300	L, H = 5
Peak MSE	0.1822	3.2312e-027	7.8886e-031
Overall MSE	0.1182	3.2312e-027	7.8886e-031
Peak pixel error	1.5989	5.6843e-014	8.8818e-016
Peak pixel mean error	0.0625	5.6843e-014	8.8818e-016
Overall mean error	0.00063	-5.6843e-014	-8.8818e-016

Table 6 : Testing 4-Element 1-D Forward DCT Implementation:

Probability of error	error value > 0: 0.039445	error value >1: 0	error value >2: 0	error value >3: 0
Overall mean square error	0.039445			
Maximum mean square error	0.500000			
Fast algorithm Speed	0.248μs (based on 10000000 iterations)			
Reference algorithm Speed	7.100μs (based on 100000 iterations)			

Table 7 : Testing 4-Element 1-D IDCT Implementation

Probability of error	error value > 0: 0.012555	error value >1: 0	error value >2: 0	error value >3: 0
Overall mean square error	0.012555			
Maximum mean square error	0.500000			
Fast algorithm Speed	6.000μs (based on 10000000 iterations)			
Reference algorithm Speed	0.247μs (based on 100000 iterations)			

Table 8 : Testing 8-Element 1-D DCT Implementation

Probability of error	error value > 0: 0.065125	error value >1: 0	error value >2: 0	error value >3: 0
Overall mean square error	0.065125			
Maximum mean square error	0.500000			
Fast algorithm Speed	0.770μs (based on 1000000 iterations)			
Reference algorithm Speed	33.000μs (based on 10000 iterations)			

Table 9 : Testing 8-Element 1-D IDCT Implementation

Probability of error	error value > 0: 0.04505	error value >1: 0	error value >2: 0	error value >3: 0
Overall mean square error	0.045050			
Maximum mean square error	0.500000			
Fast algorithm Speed	0.440μs (based on 1000000 iterations)			
Reference algorithm Speed	16.000μs (based on 10000 iterations)			

Table 10 : Testing 8x8-Element 2-D DCT Implementation

Probability of error	error value > 0: 0.0282125	error value >1: 0	error value >2: 0	error value >3: 0
Overall mean square error	0.028213			
Maximum mean square error	0.109375			
Fast algorithm Speed	3.900000μs (based on 100000 iterations)			
Separable method	270.0000μs (based on 1000 iterations)			
Reference algorithm Speed	2200.000μs (based on 100 iterations)			

Table 11 : Testing 8x8-Element 2-D IDCT Implementation

Probability of error	error value > 0: 0.18025	error value >1: 0	error value >2: 0	error value >3: 0
Overall mean square error	0.180250			
Maximum mean square error	0.375000			
Fast algorithm Speed	4.400000μs (based on 100000 iterations)			
Separable method	280.0000μs (based on 1000 iterations)			
Reference algorithm Speed	2200.000μs (based on 100 iterations)			

In this proposed algorithm, the number of multiplications are halved since the (N×N) ×(N×1) matrix-vector multiply has been replaced by two [(N/2) ×(N/2)] ×[(N/2) ×1] matrix-vector multiplies. Moreover, these two can be computed in parallel.

B.6 FAST DCT Source Code

```
// cdct8x8test.cpp : Defines the entry point for the console application.

#include "stdafx.h"
static const double PI=3.14159265358979323;

/**************************************************************************/
void InitRandom() { srand(time(0)); }
int Random() { return rand(); }

/*
 * 2-d Forward DCT implemented directly from the formulas.
 * accurate, but very slow.
 * The ouput is 1/4 the definition of 2D DCT
 */
static void
dct2dRef(int (*data)[8]) {
   double output[8][8] = {{0}};
   short x,y,n,m;

   for(y=0;y<8;y++) {
      for(x=0;x<8;x++) {
         for(n=0;n<8;n++) {
            for(m=0;m<8;m++) {
               output[y][x] += data[n][m]
               *cos(PI * x * (2*m+1)/16.0) * cos(PI * y * (2*n+1)/16.0);
            }
         }
      }
   }

   for(y=0;y<8;y++) {
      for(x=0;x<8;x++) {
         if(x==0) output[y][x] /= sqrt(2);
         if(y==0) output[y][x] /= sqrt(2);
         data[y][x] = (int)floor(output[y][x]/16 + 0.5);
      }
   }
}

/**************************************************************************/

/*
 * 2-d Forward DCT implemented in terms of 1-D DCT
 */
static void
dct2dSeparable(int (*data)[8]) { /* pointers to 8 element arrays */
   double work[8][8] = {{0}};
   int row,col;
   for(row=0;row<8;row++) {
      short x,n;
      for(x=0;x<8;x++) {
         for(n=0;n<8;n++)
            work[row][x] += data[row][n] * cos(PI * x * (2*n+1)/16.0);
         work[row][x] /= 4.0; /* typical weighting */
         if(x == 0) work[row][x] /= sqrt(2.0);
      }
   }

   for(col=0;col<8;col++) {
      short x,n;
      for(x=0;x<8;x++) {
         double result=0;
         for(n=0;n<8;n++)
```

```
          result += work[n][col] * cos(PI * x * (2*n+1)/16.0);
        if(x==0) result /= sqrt(2.0);
        result /= 4.0;
        /* Assign final result back into data */
        data[x][col] = (int)floor(result + 0.5); /* Round correctly */
      }
    }
}

/*************************************************************************/
/* Note that the 1-D DCT algorithm in LL&M results in the output
 * scaled by 4*sqrt(2) (i.e., 2 1/2 bits).  After two passes,
 * I need to scale the output by 32 (>>5). The output is 1/4 of definition
 */
static void
dct2dTest(int (*dctBlock)[8]) {
  static const int c1=1004 /*cos(pi/16)<<10*/, s1=200 /*sin(pi/16)<<10*/;
  static const int c3=851 /*cos(3pi/16)<<10*/, s3=569 /*sin(3pi/16)<<10*/;
  static const int r2c6=554 /*sqrt(2)*cos(6pi/16)<<10*/, r2s6=1337;
  static const int r2=181; /* sqrt(2)<<7 */
  int row,col;

  for(row=0;row<8;row++) {
    int x0=dctBlock[row][0], x1=dctBlock[row][1], x2=dctBlock[row][2],
      x3=dctBlock[row][3], x4=dctBlock[row][4], x5=dctBlock[row][5],
      x6=dctBlock[row][6], x7=dctBlock[row][7], x8;

    /* Stage 1 */
    x8=x7+x0; x0-=x7; x7=x1+x6; x1-=x6; x6=x2+x5; x2-=x5; x5=x3+x4; x3-=x4;

    /* Stage 2 */
    x4=x8+x5; x8-=x5; x5=x7+x6; x7-=x6;
    x6=c1*(x1+x2); x2=(-s1-c1)*x2+x6; x1=(s1-c1)*x1+x6;
    x6=c3*(x0+x3); x3=(-s3-c3)*x3+x6; x0=(s3-c3)*x0+x6;

    /* Stage 3 */
    x6=x4+x5; x4-=x5; x5=x0+x2;x0-=x2; x2=x3+x1; x3-=x1;
    x1=r2c6*(x7+x8); x7=(-r2s6-r2c6)*x7+x1; x8=(r2s6-r2c6)*x8+x1;

    /* Stage 4 and output */
    dctBlock[row][0]=x6;  dctBlock[row][4]=x4;
    dctBlock[row][2]=x8>>10; dctBlock[row][6] = x7>>10;
    dctBlock[row][7]=(x2-x5)>>10; dctBlock[row][1]=(x2+x5)>>10;
    dctBlock[row][3]=(x3*r2)>>17; dctBlock[row][5]=(x0*r2)>>17;
  }

  for(col=0;col<8;col++) {
    int x0=dctBlock[0][col], x1=dctBlock[1][col], x2=dctBlock[2][col],
      x3=dctBlock[3][col], x4=dctBlock[4][col], x5=dctBlock[5][col],
      x6=dctBlock[6][col], x7=dctBlock[7][col], x8;

    /* Stage 1 */
    x8=x7+x0; x0-=x7; x7=x1+x6; x1-=x6; x6=x2+x5; x2-=x5; x5=x3+x4; x3-=x4;

    /* Stage 2 */
    x4=x8+x5; x8-=x5; x5=x7+x6; x7-=x6;
    x6=c1*(x1+x2); x2=(-s1-c1)*x2+x6; x1=(s1-c1)*x1+x6;
    x6=c3*(x0+x3); x3=(-s3-c3)*x3+x6; x0=(s3-c3)*x0+x6;

    /* Stage 3 */
    x6=x4+x5; x4-=x5; x5=x0+x2;x0-=x2; x2=x3+x1; x3-=x1;
    x1=r2c6*(x7+x8); x7=(-r2s6-r2c6)*x7+x1; x8=(r2s6-r2c6)*x8+x1;

    /* Stage 4 and output */
    dctBlock[0][col]=(x6+16)>>5;  dctBlock[4][col]=(x4+16)>>5;
    dctBlock[2][col]=(x8+16384)>>15; dctBlock[6][col] = (x7+16384)>>15;
    dctBlock[7][col]=(x2-x5+16384)>>15; dctBlock[1][col]=(x2+x5+16384)>>15;
```

```
    dctBlock[3][col]=((x3>>8)*r2+8192)>>14;
    dctBlock[5][col]=((x0>>8)*r2+8192)>>14;
  }
}

/*************************************************************************/
/*
 * 2-d IDCT implemented directly from the formulas.
 * Very accurate, very slow. ouput is not scaled compared to formula
 */
static void
idct2dRef(int (*data)[8]) {
   double output[8][8] = {{0}};
   short x,y,m,n;
   for(y=0;y<8;y++) {
      for(x=0;x<8;x++) {
         output[y][x]=0.0;
         for(n=0;n<8;n++)
            for(m=0;m<8;m++) {
               double term = data[n][m]
                    * cos(PI * m * (2*x+1)/16.0) * cos(PI * n * (2*y+1)/16.0);
               if(n==0) term /= sqrt(2);
               if(m==0) term /= sqrt(2);
               output[y][x] += term;
            }
      }
  }
  for(y=0;y<8;y++) {
    for(x=0;x<8;x++) {
      output[y][x] /= 4.0;
      data[y][x] = (int)floor(output[y][x] + 0.5); /* Round accurately */
    }
  }
}

/*************************************************************************/

static void
idct2dSeparable(int (*data)[8]) {
  double work[8][8] = {{0}};
  int row,col;
  for(row=0;row<8;row++) {
    short x,n;
    for(x =0;x<8;x++){
      work[row][x]   = data[row][0] / sqrt(2.0);
      for(n=1;n<8;n++)
        work[row][x] += data[row][n] * cos(PI * n * (2*x+1)/16.0);
    }
  }

  for(col=0;col<8;col++) {
    short x,n;
    for(x=0;x<8;x++) {
      double result = work[0][col] / sqrt(2.0);
      for(n=1;n<8;n++)
        result += work[n][col] * cos(PI * n * (2*x+1)/16.0);

      /* Assign final result back into data */
      result /= 4.0;
      data[x][col] = (int)floor(result + 0.5); /* Round correctly */
    }
  }
}
/*************************************************************************/

static void
idct2dTest(int (*dctBlock)[8]) {
```

```
  int row,col;

  for(row=0;row<8;row++) {
    static const int c1=251 /*cos(pi/16)<<8*/, s1=50 /*sin(pi/16)<<8*/;
    static const int c3=213 /*cos(3pi/16)<<8*/, s3=142 /*sin(3pi/16)<<8*/;
    static const int r2c6=277 /*cos(6pi/16)*sqrt(2)<<9*/, r2s6=669;
    static const int r2=181; /* sqrt(2)<<7 */

    /* Stage 4 */
    int x0=dctBlock[row][0]<<9, x1=dctBlock[row][1]<<7, x2=dctBlock[row][2],
      x3=dctBlock[row][3]*r2, x4=dctBlock[row][4]<<9, x5=dctBlock[row][5]*r2,
      x6=dctBlock[row][6], x7=dctBlock[row][7]<<7;
    int x8=x7+x1; x1 -= x7;

    /* Stage 3 */
    x7=x0+x4; x0-=x4; x4=x1+x5; x1-=x5; x5=x3+x8; x8-=x3;
    x3=r2c6*(x2+x6);x6=x3+(-r2c6-r2s6)*x6;x2=x3+(-r2c6+r2s6)*x2;

    /* Stage 2 */
    x3=x7+x2; x7-=x2; x2=x0+x6; x0-= x6;
    x6=c3*(x4+x5);x5=(x6+(-c3-s3)*x5)>>6;x4=(x6+(-c3+s3)*x4)>>6;
    x6=c1*(x1+x8);x1=(x6+(-c1-s1)*x1)>>6;x8=(x6+(-c1+s1)*x8)>>6;

    /* Stage 1 and output */
    x7+=512; x2+=512; x0+=512; x3+=512;
    dctBlock[row][0]=(x3+x4)>>10;  dctBlock[row][1]=(x2+x8)>>10;
    dctBlock[row][2]=(x0+x1)>>10;  dctBlock[row][3]=(x7+x5)>>10;
    dctBlock[row][4]=(x7-x5)>>10;  dctBlock[row][5]=(x0-x1)>>10;
    dctBlock[row][6]=(x2-x8)>>10;  dctBlock[row][7]=(x3-x4)>>10;
  }

  for(col=0;col<8;col++) {
    static const int c1=251 /*cos(pi/16)<<8*/, s1=50 /*sin(pi/16)<<8*/;
    static const int c3=213 /*cos(3pi/16)<<8*/, s3=142 /*sin(3pi/16)<<8*/;
    static const int r2c6=277 /*cos(6pi/16)*sqrt(2)<<9*/, r2s6=669;
    static const int r2=181; /* sqrt(2)<<7 */

    /* Stage 4 */
    int x0=dctBlock[0][col]<<9, x1=dctBlock[1][col]<<7, x2=dctBlock[2][col],
      x3=((dctBlock[3][col]))*r2, x4=dctBlock[4][col]<<9,
      x5=((dctBlock[5][col]))*r2, x6=dctBlock[6][col],
      x7=dctBlock[7][col]<<7;
    int x8=x7+x1; x1 -= x7;

    /* Stage 3 */
    x7=x0+x4; x0-=x4; x4=x1+x5; x1-=x5; x5=x3+x8; x8-=x3;
    x3=r2c6*(x2+x6);x6=x3+(-r2c6-r2s6)*x6;x2=x3+(-r2c6+r2s6)*x2;

    /* Stage 2 */
    x3=x7+x2; x7-=x2; x2=x0+x6; x0-= x6;
    x4>>=6;x5>>=6;x1>>=6;x8>>=6;
    x6=c3*(x4+x5);x5=(x6+(-c3-s3)*x5);x4=(x6+(-c3+s3)*x4);
    x6=c1*(x1+x8);x1=(x6+(-c1-s1)*x1);x8=(x6+(-c1+s1)*x8);

    /* Stage 1, rounding and output */
    x7+=1024; x2+=1024;x0+=1024;x3+=1024; /* For correct rounding */
    dctBlock[0][col]=(x3+x4)>>11;  dctBlock[1][col]=(x2+x8)>>11;
    dctBlock[2][col]=(x0+x1)>>11;  dctBlock[3][col]=(x7+x5)>>11;
    dctBlock[4][col]=(x7-x5)>>11;  dctBlock[5][col]=(x0-x1)>>11;
    dctBlock[6][col]=(x2-x8)>>11;  dctBlock[7][col]=(x3-x4)>>11;
  }
}
/*************************************************************************/

void test2dAccuracy(int maxIterations,
         void (*testFunc)(int (*)[8]),
         char *testFuncName,
```

```
       void (*referenceFunc)(int (*)[8]),
       char *referenceFuncName) {
int input[8][8], reference[8][8], test[8][8];
int iteration;
int totalCoefficients=0; /* Total number of coefficients tested */
int errorCoefficients[4]={0}; /* # coefficients out of range */
double squareError=0; /* Total squared error over all coefficients */
double maxSquareError=0; /* Largest squared error for any block */
int i,j;
printf("Testing Accuracy: %s (%d iterations, comparing to %s)\n",
  testFuncName,maxIterations,referenceFuncName);

for(iteration=0;iteration<maxIterations;iteration++) {
  double thisSquareError = 0.0;
  /* Build random input values in range -128...127 */
  for(i=0;i<8;i++) {
    for(j=0;j<8;j++) {
 int t = Random() & 0xff;
 if(t > 127) t-= 256;
 input[i][j] = t;
    }
  }

  /* Compute reference version */
  memcpy(reference,input,sizeof(input));
  (*referenceFunc)(reference);

  /* Compute test version */
  memcpy(test,input,sizeof(input));
  (*testFunc)(test);

  /* Count number of errors exceeding one */
  totalCoefficients += 64;
  for(i=0;i<8;i++) {
    for(j=0;j<8;j++) {
 int err = test[i][j] - reference[i][j];
 double err2 = (double)err * (double)err;
 if(err < 0) err = -err;
 {
   int k;
   for(k=0;k<4;k++)
     if(err > k) errorCoefficients[k]++;
 }
 squareError += err2;
 thisSquareError += err2;
    }
  }
  if(thisSquareError > maxSquareError)
    maxSquareError = thisSquareError;
  if(thisSquareError > 100) {
    int i,j=0;
    printf("Bad Example: mean square error = %f\n",thisSquareError/64);
    printf("Input: ");  for(i=0;i<8;i++) printf("  %4d",input[i][j]);
    printf("\nRef:   ");  for(i=0;i<8;i++) printf("  %4d",reference[i][j]);
    printf("\nTest:  ");  for(i=0;i<8;i++) printf("  %4d",test[i][j]);
    printf("\n\n");
  }
}

{
  int k;
  printf("   Probability of error > 0: %g",
    (double)errorCoefficients[0] / (double)totalCoefficients);
  for(k=1;k<4;k++)
    printf(",  > %d: %g",k,
      (double)errorCoefficients[k] / (double)totalCoefficients);
  printf("\n");
```

```
  }
  printf("   Overall mean square error: %f\n", squareError/totalCoefficients);
  printf("   Maximum mean square error: %f\n", maxSquareError / 64);
}

/*****************************************************************************/
/*
 * Since the Random() function might not be infinitely fast,
 * I choose one set of random values for every hundred calls
 * to the test function.  That way, my time measures the function being
 * tested, not the random number generator.
 */
static void
test2dSpeed(int maxIterations, void (*testFunc)(int (*)[8]), char *funcName) {
   int i,j,iterations;
   static const int incr = 100;
   int input[8][8],work[8][8];
   clock_t start, finish;
   double  duration;
   start = clock();
    printf("   %s: ",funcName); fflush(stdout);
    for(iterations = 0; iterations < maxIterations; iterations+=incr) {
     /* Build random input values in range -128...127 */
      for(i=0;i<8;i++) {
         for(j=0;j<8;j++) {
            int t = Random() & 0xff;
            if(t > 127) t-= 256;
            input[i][j] = t;
         }
      }
      for(i=0;i<incr;i++) {
         memcpy(work,input,sizeof(input));
         (*testFunc)(work);
      }
   }
   finish = clock();
   duration = (double)(finish - start) / CLOCKS_PER_SEC;
   printf("%f microseconds (based on %d iterations)\n",
   duration/maxIterations * 1000000, maxIterations);
}

/*****************************************************************************/
int main(int argc, char **argv) {
  InitRandom();

  printf("\nTesting 8x8-Element 2-D Forward DCT Implementation\n\n");
  {
    /* Double-check that Separable and Reference versions agree. */
    test2dAccuracy(100,dct2dSeparable,"dct2dSeparable",
        dct2dRef,"dct2dRef");
    /* Use faster separable version as reference now */
    test2dAccuracy(5000,dct2dTest,"dct2dTest",
        dct2dSeparable,"dct2dSeparable");

    printf("Measuring Speed\n");
    test2dSpeed(100,dct2dRef,"2d Ref");
    test2dSpeed(1000,dct2dSeparable,"2d Separable");
    test2dSpeed(100000,dct2dTest,"2d Test");
  }
  printf("\n\nTesting 8x8-Element 2-D IDCT Implementation\n\n");
  {
    test2dAccuracy(100,idct2dSeparable,"idct2dSeparable",
        idct2dRef,"idct2dRef");
    /* Use faster separable version as reference now */
    test2dAccuracy(5000,idct2dTest,"idct2dTest",
        idct2dSeparable,"idct2dSeparable");
```

```
    printf("Measuring Speed\n");
    test2dSpeed(100,idct2dRef,"2d Reference");
    test2dSpeed(1000,idct2dSeparable,"2d Separable");
    test2dSpeed(100000,idct2dTest,"2d Test");
  }
  return 0;
}
```

B.7 JPEG Image Compression Source Code

```
/************************* Start of BITIO.C ************************
 *
 * This utility file contains all of the routines needed to impement
 * bit oriented routines under either ANSI or K&R C.  It needs to be
 * linked with every program used in the entire book.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "bitio.h"
#include "errhand.h"

#define PACIFIER_COUNT 2047

BIT_FILE *OpenOutputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "wb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

BIT_FILE *OpenInputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
   return( bit_file );
    bit_file->file = fopen( name, "rb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

void CloseOutputBitFile( bit_file )
BIT_FILE *bit_file;
{
    if ( bit_file->mask != 0x80 )
        if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
            fatal_error( "Fatal error in CloseBitFile!\n" );
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void CloseInputBitFile( bit_file )
BIT_FILE *bit_file;
{
    fclose( bit_file->file );
```

```
    free( (char *) bit_file );
}

void OutputBit( bit_file, bit )
BIT_FILE *bit_file;
int bit;
{
    if ( bit )
        bit_file->rack |= bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 ) {
   if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
       fatal_error( "Fatal error in OutputBit!\n" );
   else
        if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
     putc( '.', stdout );
   bit_file->rack = 0;
   bit_file->mask = 0x80;
    }
}

void OutputBits( bit_file, code, count )
BIT_FILE *bit_file;
unsigned long code;
int count;
{
    unsigned long mask;

    mask = 1L << ( count - 1 );
    while ( mask != 0) {
        if ( mask & code )
            bit_file->rack |= bit_file->mask;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 ) {
      if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
     fatal_error( "Fatal error in OutputBit!\n" );
        else if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
     putc( '.', stdout );
      bit_file->rack = 0;
            bit_file->mask = 0x80;
        }
        mask >>= 1;
    }
}

int InputBit( bit_file )
BIT_FILE *bit_file;
{
    int value;

    if ( bit_file->mask == 0x80 ) {
        bit_file->rack = getc( bit_file->file );
        if ( bit_file->rack == EOF )
            fatal_error( "Fatal error in InputBit!\n" );
    if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
      putc( '.', stdout );
    }
    value = bit_file->rack & bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 )
   bit_file->mask = 0x80;
```

```
    return( value ? 1 : 0 );
}

unsigned long InputBits( bit_file, bit_count )
BIT_FILE *bit_file;
int bit_count;
{
    unsigned long mask;
    unsigned long return_value;

    mask = 1L << ( bit_count - 1 );
    return_value = 0;
    while ( mask != 0) {
   if ( bit_file->mask == 0x80 ) {
       bit_file->rack = getc( bit_file->file );
       if ( bit_file->rack == EOF )
     fatal_error( "Fatal error in InputBit!\n" );
        if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
     putc( '.', stdout );
   }
   if ( bit_file->rack & bit_file->mask )
            return_value |= mask;
        mask >>= 1;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 )
            bit_file->mask = 0x80;
    }
    return( return_value );
}

void FilePrintBinary( file, code, bits )
FILE *file;
unsigned int code;
int bits;
{
    unsigned int mask;

    mask = 1 << ( bits - 1 );
    while ( mask != 0 ) {
        if ( code & mask )
            fputc( '1', file );
        else
            fputc( '0', file );
        mask >>= 1;
    }
}

/************************** End of BITIO.C *************************/
```

Huffman Coding Source Code

```
/************************** Start of HUFF.C ************************
 *
 * This is the Huffman coding module used in Chapter 3.
 * Compile with BITIO.C, ERRHAND.C, and either MAIN-C.C or MAIN-E.C
 */


#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <ctype.h>
#include "bitio.h"
#include "errhand.h"
#include "main.h"

/*
 * The NODE structure is a node in the Huffman decoding tree.  It has a
 * count, which is its weight in the tree, and the node numbers of its
 * two children.  The saved_count member of the structure is only
 * there for debugging purposes, and can be safely taken out at any
 * time.  It just holds the intial count for each of the symbols, since
 * the count member is continually being modified as the tree grows.
 */
typedef struct tree_node {
    unsigned int count;
    unsigned int saved_count;
    int child_0;
    int child_1;
} NODE;

/*
 * A Huffman tree is set up for decoding, not encoding.  When encoding,
 * I first walk through the tree and build up a table of codes for
 * each symbol.  The codes are stored in this CODE structure.
 */
typedef struct code {
    unsigned int code;
    int code_bits;
} CODE;

/*
 * The special EOS symbol is 256, the first available symbol after all
 * of the possible bytes.  When decoding, reading this symbols
 * indicates that all of the data has been read in.
 */
#define END_OF_STREAM 256

/*
 * Local function prototypes, defined with or without ANSI prototypes.
 */
void count_bytes( FILE *input, unsigned long *long_counts );
void scale_counts( unsigned long *long_counts, NODE *nodes );
int build_tree( NODE *nodes );
void convert_tree_to_code( NODE *nodes,
                           CODE *codes,
                           unsigned int code_so_far,
                           int bits,
                           int node );
void output_counts( BIT_FILE *output, NODE *nodes );
void input_counts( BIT_FILE *input, NODE *nodes );
void print_model( NODE *nodes, CODE *codes );
void compress_data( FILE *input, BIT_FILE *output, CODE *codes );
void expand_data( BIT_FILE *input, FILE *output, NODE *nodes,
                  int root_node );
void print_char( int c );


/*
 * These two strings are used by MAIN-C.C and MAIN-E.C to print
 * messages of importance to the user of the program.
```

```
 */
char *CompressionName = "static order 0 model with Huffman coding";
char *Usage = "infile outfile [-d]\n\nSpecifying -d will dump the modeling
data\n";

/*
 * CompressFile is the compression routine called by MAIN-C.C.  It
 * looks for a single additional argument to be passed to it from
 * the command line:  "-d".  If a "-d" is present, it means the
 * user wants to see the model data dumped out for debugging
 * purposes.
 *
 * This routine works in a fairly straightforward manner.  First,
 * it has to allocate storage for three different arrays of data.
 * Next, it counts all the bytes in the input file.  The counts
 * are all stored in long int, so the next step is scale them down
 * to single byte counts in the NODE array.  After the counts are
 * scaled, the Huffman decoding tree is built on top of the NODE
 * array.  Another routine walks through the tree to build a table
 * of codes, one per symbol.  Finally, when the codes are all ready,
 * compressing the file is a simple matter.  After the file is
 * compressed, the storage is freed up, and the routine returns.
 *
 */
void CompressFile(hWnd, input, output )
HWND hWnd;
FILE *input;
BIT_FILE *output;
{
    unsigned long *counts;
    NODE *nodes;
    CODE *codes;
    int root_node;

    counts = (unsigned long *) calloc( 256, sizeof( unsigned long ) );
    if ( counts == NULL )
//        fatal_error( "Error allocating counts array\n" );
     MessageBox(hWnd , "Error allocating counts array", "Message 3", MB_OK);
    if ( ( nodes = (NODE *) calloc( 514, sizeof( NODE ) ) ) == NULL )
// fatal_error( "Error allocating nodes array\n" );
   MessageBox(hWnd , "Error allocating nodes array", "Message 4", MB_OK);

    if ( ( codes = (CODE *) calloc( 257, sizeof( CODE ) ) ) == NULL )
   fatal_error( "Error allocating codes array\n" );
    count_bytes( input, counts );
    scale_counts( counts, nodes );
    output_counts( output, nodes );
    root_node = build_tree( nodes );
    convert_tree_to_code( nodes, codes, 0, 0, root_node );
    compress_data( input, output, codes );
    free( (char *) counts );
    free( (char *) nodes );
    free( (char *) codes );
}

/*
 * ExpandFile is the routine called by MAIN-E.C to expand a file that
 * has been compressed with order 0 Huffman coding.  This routine has
 * a simpler job than that of the Compression routine.  All it has to
 * do is read in the counts that have been stored in the compressed
 * file, then build the Huffman tree.  The data can then be expanded
```

```
 * by reading in a bit at a time from the compressed file.  Finally,
 * the node array is freed and the routine returns.
 *
 */
void ExpandFile(hWnd, input, output )
HWND hWnd;
BIT_FILE *input;
FILE *output;
{
    NODE *nodes;
    int root_node;

    if ( ( nodes = (NODE *) calloc( 514, sizeof( NODE ) ) ) == NULL )
     MessageBox(hWnd , "Error allocating nodes array", "Expand File", MB_OK);
//      fatal_error( "Error allocating nodes array\n" );
    input_counts( input, nodes );
    root_node = build_tree( nodes );
    expand_data( input, output, nodes, root_node );
    free( (char *) nodes );
}

/*
 * In order for the compressor to build the same model, I have to store
 * the symbol counts in the compressed file so the expander can read
 * them in.  In order to save space, I don't save all 256 symbols
 * unconditionally.  The format used to store counts looks like this:
 *
 *  start, stop, counts, start, stop, counts, ... 0
 *
 * This means that I store runs of counts, until all the non-zero
 * counts have been stored.  At this time the list is terminated by
 * storing a start value of 0.  Note that at least 1 run of counts has
 * to be stored, so even if the first start value is 0, I read it in.
 * It also means that even in an empty file that has no counts, I have
 * to pass at least one count.
 *
 * In order to efficiently use this format, I have to identify runs of
 * non-zero counts.  Because of the format used, I don't want to stop a
 * run because of just one or two zeros in the count stream.  So I have
 * to sit in a loop looking for strings of three or more zero values in
 * a row.
 *
 * This is simple in concept, but it ends up being one of the most
 * complicated routines in the whole program.  A routine that just
 * writes out 256 values without attempting to optimize would be much
 * simpler, but would hurt compression quite a bit on small files.
 *
 */
void output_counts( output, nodes )
BIT_FILE *output;
NODE *nodes;
{
    int first;
    int last;
    int next;
    int i;

    first = 0;
    while ( first < 255 && nodes[ first ].count == 0 )
       first++;
/*
```

```
 * Each time I hit the start of the loop, I assume that first is the
 * number for a run of non-zero values.  The rest of the loop is
 * concerned with finding the value for last, which is the end of the
 * run, and the value of next, which is the start of the next run.
 * At the end of the loop, I assign next to first, so it starts in on
 * the next run.
 */
   for ( ; first < 256 ; first = next ) {
  last = first + 1;
  for ( ; ; ) {
      for ( ; last < 256 ; last++ )
    if ( nodes[ last ].count == 0 )
        break;
      last--;
      for ( next = last + 1; next < 256 ; next++ )
    if ( nodes[ next ].count != 0 )
        break;
      if ( next > 255 )
    break;
      if ( ( next - last ) > 3 )
    break;
      last = next;
  };
/*
 * Here is where I output first, last, and all the counts in between.
 */
  if ( putc( first, output->file ) != first )
      fatal_error( "Error writing byte counts\n" );
  if ( putc( last, output->file ) != last )
      fatal_error( "Error writing byte counts\n" );
  for ( i = first ; i <= last ; i++ ) {
            if ( putc( nodes[ i ].count, output->file ) !=
                 (int) nodes[ i ].count )
    fatal_error( "Error writing byte counts\n" );
  }
   }
   if ( putc( 0, output->file ) != 0 )
      fatal_error( "Error writing byte counts\n" );
}

/*
 * When expanding, I have to read in the same set of counts.  This is
 * quite a bit easier that the process of writing them out, since no
 * decision making needs to be done.  All I do is read in first, check
 * to see if I am all done, and if not, read in last and a string of
 * counts.
 */

void input_counts( input, nodes )
BIT_FILE *input;
NODE *nodes;
{
    int first;
    int last;
    int i;
    int c;

    for ( i = 0 ; i < 256 ; i++ )
  nodes[ i ].count = 0;
    if ( ( first = getc( input->file ) ) == EOF )
  fatal_error( "Error reading byte counts\n" );
```

```
    if ( ( last = getc( input->file ) ) == EOF )
   fatal_error( "Error reading byte counts\n" );
    for ( ; ; ) {
   for ( i = first ; i <= last ; i++ )
       if ( ( c = getc( input->file ) ) == EOF )
     fatal_error( "Error reading byte counts\n" );
       else
     nodes[ i ].count = (unsigned int) c;
   if ( ( first = getc( input->file ) ) == EOF )
       fatal_error( "Error reading byte counts\n" );
   if ( first == 0 )
       break;
   if ( ( last = getc( input->file ) ) == EOF )
       fatal_error( "Error reading byte counts\n" );
    }
    nodes[ END_OF_STREAM ].count = 1;
}

/*
 * This routine counts the frequency of occurence of every byte in
 * the input file.  It marks the place in the input stream where it
 * started, counts up all the bytes, then returns to the place where
 * it started.  In most C implementations, the length of a file
 * cannot exceed an unsigned long, so this routine should always
 * work.
 */


void count_bytes( input, counts )
FILE *input;
unsigned long *counts;
{
    long input_marker;
    int c;

    input_marker = ftell( input );
    while ( ( c = getc( input )) != EOF )
   counts[ c ]++;
    fseek( input, input_marker, SEEK_SET );
}

/*
 * In order to limit the size of my Huffman codes to 16 bits, I scale
 * my counts down so they fit in an unsigned char, and then store them
 * all as initial weights in my NODE array.  The only thing to be
 * careful of is to make sure that a node with a non-zero count doesn't
 * get scaled down to 0.  Nodes with values of 0 don't get codes.
 */
void scale_counts( counts, nodes )
unsigned long *counts;
NODE *nodes;
{
    unsigned long max_count;
    int i;

    max_count = 0;
    for ( i = 0 ; i < 256 ; i++ )
       if ( counts[ i ] > max_count )
      max_count = counts[ i ];
    if ( max_count == 0 ) {
   counts[ 0 ] = 1;
```

```
   max_count = 1;
    }
    max_count = max_count / 255;
    max_count = max_count + 1;
    for ( i = 0 ; i < 256 ; i++ ) {
   nodes[ i ].count = (unsigned int) ( counts[ i ] / max_count );
   if ( nodes[ i ].count == 0 && counts[ i ] != 0 )
       nodes[ i ].count = 1;
    }
    nodes[ END_OF_STREAM ].count = 1;
}
/*
 * Building the Huffman tree is fairly simple.  All of the active nodes
 * are scanned in order to locate the two nodes with the minimum
 * weights.  These two weights are added together and assigned to a new
 * node.  The new node makes the two minimum nodes into its 0 child
 * and 1 child.  The two minimum nodes are then marked as inactive.
 * This process repeats until their is only one node left, which is the
 * root node.  The tree is done, and the root node is passed back
 * to the calling routine.
 *
 * Node 513 is used here to arbitratily provide a node with a guaranteed
 * maximum value.  It starts off being min_1 and min_2.  After all active
 * nodes have been scanned, I can tell if there is only one active node
 * left by checking to see if min_1 is still 513.
 */
int build_tree( nodes )
NODE *nodes;
{
    int next_free;
    int i;
    int min_1;
    int min_2;

    nodes[ 513 ].count = 0xffff;
    for ( next_free = END_OF_STREAM + 1 ; ; next_free++ ) {
   min_1 = 513;
   min_2 = 513;
   for ( i = 0 ; i < next_free ; i++ )
             if ( nodes[ i ].count != 0 ) {
                 if ( nodes[ i ].count < nodes[ min_1 ].count ) {
                     min_2 = min_1;
                     min_1 = i;
                 } else if ( nodes[ i ].count < nodes[ min_2 ].count )
                     min_2 = i;
             }
   if ( min_2 == 513 )
       break;
   nodes[ next_free ].count = nodes[ min_1 ].count
                            + nodes[ min_2 ].count;
        nodes[ min_1 ].saved_count = nodes[ min_1 ].count;
        nodes[ min_1 ].count = 0;
        nodes[ min_2 ].saved_count =  nodes[ min_2 ].count;
        nodes[ min_2 ].count = 0;
   nodes[ next_free ].child_0 = min_1;
   nodes[ next_free ].child_1 = min_2;
    }
    next_free--;
    nodes[ next_free ].saved_count = nodes[ next_free ].count;
    return( next_free );
}
```

```
/*
 * Since the Huffman tree is built as a decoding tree, there is
 * no simple way to get the encoding values for each symbol out of
 * it.  This routine recursively walks through the tree, adding the
 * child bits to each code until it gets to a leaf.  When it gets
 * to a leaf, it stores the code value in the CODE element, and
 * returns.
 */
void convert_tree_to_code( nodes, codes, code_so_far, bits, node )
NODE *nodes;
CODE *codes;
unsigned int code_so_far;
int bits;
int node;
{
    if ( node <= END_OF_STREAM ) {
   codes[ node ].code = code_so_far;
   codes[ node ].code_bits = bits;
   return;
    }
    code_so_far <<= 1;
    bits++;
    convert_tree_to_code( nodes, codes, code_so_far, bits,
                          nodes[ node ].child_0 );
    convert_tree_to_code( nodes, codes, code_so_far | 1,
                          bits, nodes[ node ].child_1 );
}
/*
 * If the -d command line option is specified, this routine is called
 * to print out some of the model information after the tree is built.
 * Note that this is the only place that the saved_count NODE element
 * is used for anything at all, and in this case it is just for
 * diagnostic information.  By the time I get here, and the tree has
 * been built, every active element will have 0 in its count.
 */
void print_model( nodes, codes )
NODE *nodes;
CODE *codes;
{
    int i;

    for ( i = 0 ; i < 513 ; i++ ) {
   if ( nodes[ i ].saved_count != 0 ) {
            printf( "node=" );
            print_char( i );
            printf( "  count=%3d", nodes[ i ].saved_count );
            printf( "  child_0=" );
            print_char( nodes[ i ].child_0 );
            printf( "  child_1=" );
            print_char( nodes[ i ].child_1 );
      if ( codes && i <= END_OF_STREAM ) {
     printf( "  Huffman code=" );
                FilePrintBinary( stdout, codes[ i ].code, codes[ i ].code_bits
);
      }
      printf( "\n" );
   }
    }
}
```

```
/*
 * The print_model routine uses this function to print out node numbers.
 * The catch is, if it is a printable character, it gets printed out
 * as a character.  Makes the debug output a little easier to read.
 */
void print_char( c )
int c;
{
    if ( c >= 0x20 && c < 127 )
        printf( "'%c'", c );
    else
        printf( "%3d", c );
}

/*
 * Once the tree gets built, and the CODE table is built, compressing
 * the data is a breeze.  Each byte is read in, and its corresponding
 * Huffman code is sent out.
 */
void compress_data( input, output, codes )
FILE *input;
BIT_FILE *output;
CODE *codes;
{
    int c;

    while ( ( c = getc( input ) ) != EOF )
        OutputBits( output, (unsigned long) codes[ c ].code,
                    codes[ c ].code_bits );
    OutputBits( output, (unsigned long) codes[ END_OF_STREAM ].code,
     codes[ END_OF_STREAM ].code_bits );
}

/*
 * Expanding compressed data is a little harder than the compression
 * phase.  As each new symbol is decoded, the tree is traversed,
 * starting at the root node, reading a bit in, and taking either the
 * child_0 or child_1 path.  Eventually, the tree winds down to a
 * leaf node, and the corresponding symbol is output.  If the symbol
 * is the END_OF_STREAM symbol, it doesn't get written out, and
 * instead the whole process terminates.
 */
void expand_data( input, output, nodes, root_node )
BIT_FILE *input;
FILE *output;
NODE *nodes;
int root_node;
{
    int node;

    for ( ; ; ) {
        node = root_node;
        do {
            if ( InputBit( input ) )
                node = nodes[ node ].child_1;
            else
                node = nodes[ node ].child_0;
        } while ( node > END_OF_STREAM );
  if ( node == END_OF_STREAM )
            break;
        if ( ( putc( node, output ) ) != node )
```

```
            fatal_error( "Error trying to write expanded byte to output" );
    }
}
/*************************** End of HUFF.C *************************/
```

DCT error Detection Matlab Program

```
close all;
clear all;
A = imread('rose','jpg');
[M,N,Z] = size(A);
N = floor(N/8);
M = floor(M/8);
A = double(A(1:8*M,1:8*N,:));
Y  = 0.299*A(:,:,1)+0.587*A(:,:,2)+0.114*A(:,:,3);
Cb = -0.1687*A(:,:,1)-0.3313*A(:,:,2)+0.5*A(:,:,3)+128;
Cr = 0.5*A(:,:,1)-0.4187*A(:,:,2)-0.0813*A(:,:,3)+128;
Y = Y-128;
[C8, P8, K8, B1, B2, B3, b] = dct8matrix;
F = [2.2864 -1.6865 0.6945 -0.8128 0.7754 -0.4605 0.1848 0.0188];
[Yf1, Errin,eDetect] = dct8x8(Y, M, N, P8, K8, B1, B2, B3, F*C8,F);

figure;
imagesc(Y);
colormap('gray');

figure;
imagesc(Yf1);
colormap('gray');

figure;
imagesc(abs(Errin));
colormap('gray');

figure;
imagesc(abs(eDetect));
colormap('gray');

figure;
stem(1:size(eDetect,2),abs(eDetect(4,:)),'b*');hold on;
stem(1.25:1:size(eDetect,2)+.25,abs(Errin(4,:)),'r+');hold off;
axis([0,50,0,max(max(eDetect(4,:)),max(Errin(4,:)))]);
%-------------------------------------------------------------
% DCT8 factorizes the 8-point transform matrix C8
% into five factors P8, K8, B1, B2, B3
% C8 = original DCT matrix
% P8,K8,B1,B2,B3 = matrix factor
% b = weight matrix

function [C8, P8, K8, B1, B2, B3] = dct8matrix
   C8 =[C(4)  C(4)  C(4)  C(4)  C(4)  C(4)  C(4)  C(4);
        C(1)  C(3)  C(5)  C(7) -C(7) -C(5) -C(3) -C(1);
        C(2)  C(6) -C(6) -C(2) -C(2) -C(6)  C(6)  C(2);
        C(3) -C(7) -C(1) -C(5)  C(5)  C(1)  C(7) -C(3);
        C(4) -C(4) -C(4)  C(4)  C(4) -C(4) -C(4)  C(4);
        C(5) -C(1)  C(7)  C(3) -C(3) -C(7)  C(1) -C(5);
        C(6) -C(2)  C(2) -C(6) -C(6)  C(2) -C(2)  C(6);
        C(7) -C(5)  C(3) -C(1)  C(1) -C(3)  C(5) -C(7)];

   P81=[1 0 0 0 0 0 0 0;
        0 0 1 0 0 0 0 0;
```

```
      0 0 0 0 1 0 0 0;
      0 0 0 0 0 0 1 0;
      0 1 0 0 0 0 0 0;
      0 0 0 1 0 0 0 0;
      0 0 0 0 0 1 0 0;
      0 0 0 0 0 0 0 1];

P82 =[1 0 0 0 0 0 0 0;
      0 1 0 0 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 1;
      0 0 0 0 0 0 1 0;
      0 0 0 0 0 1 0 0;
      0 0 0 0 1 0 0 0];
F   =  [1 1; 1 -1];

R8  = kron(F,eye(4));

P41 = [ 1 0 0  0;
        0 1 0  0;
        0 0 0 -1;
        0 0 1  0];

P42 = [ 1 0 0 0;
        0 0 1 0;
        0 1 0 0;
        0 0 0 1]; % P42*C4

P43 = [ 1 0 0 0;
        0 1 0 0;
        0 0 0 1;
        0 0 1 0];   % C4*P43
R41 = kron(F,eye(2));
R4  = [0 0 0 1;
       0 0 1 0;
       0 1 0 0;
       1 0 0 0];

R2  = [0 1;1 0];
S4  = [1 1 0 0;
       1 -1 0 0;
       0 0 1 0;
       0 0 0 1];
S41 = inv(P42)*S4;
S42 = 2*[eye(2) zeros(2);zeros(2) R2]*inv(R41)*inv(P43);
P8  = inv(P81)*[S41 zeros(4);zeros(4) inv(P41)];
B1  = [S42 zeros(4);zeros(4) inv(R4)*P41];
B2  = 2*inv(R8);
B3  = inv(P82);
G1  = C(4);
G2  = [C(6) C(2);-C(2) C(6)];
G4  = [C(27) C(9)    C(3)    C(1);
      -C(1)   C(27)   C(9)    C(3);
      -C(3)   -C(1)   C(27)   C(9);
      -C(9)   -C(3)   -C(1)   C(27)];

K8  = [G1 0;0 G1];
K8  = [K8 zeros(2);zeros(2) G2];
K8  = [K8 zeros(4);zeros(4) G4];
%-------------------------------------------------------------
```

```
function c = C(n)
   c = cos(2*pi*n/32);
%-------------------------------------------------------------
function [Y, W, eDetect] = dct8x8(X, M, N, P8, K8, B1, B2, B3, b, F)
   Y = 0*X;
   W = Y;
   eDetect = W;
   line = 4;
   stage = 1;
   for m = 0:M-1
      rs = 8*m+1:8*(m+1);
      for n = 0:N-1
         cs = 8*n+1:8*(n+1);
         [Y(rs,cs), W(rs,cs),eDetect(rs,cs)] = ...
         dct2trans(X(rs,cs), P8, K8, B1, B2, B3, b,line,stage,F);
      end
   end
%-------------------------------------------------------------
function [Y, W, Flag] = dct2trans(X, P8, K8, B1, B2, B3, b,line,stage,F)
   W = 0*X;
   Flag = W;
   for k = 1:8
      noi       = 0;%10*rand(1);
      W(line,k) = noi;
      Tmp(:,k)  = dctnoise(X(:,k),P8,K8,B1,B2,B3,noi,stage,line);
      %Tmp(:,k) = P8*(K8*(B1*(B2*(B3*X(:,k)))));
      Flag(line,k) =  F*Tmp(:,k)-b*X(:,k);
   end
   Tmp = Tmp';
   for k = 1:8
      noi     = W(line,k);
      Y(:,k)  = dctnoise(Tmp(:,k),P8,K8,B1,B2,B3,noi,stage,line);
      %Y(:,k) = P8*(K8*(B1*(B2*(B3*Tmp(:,k)))));
   end
   Y = Y';
%-------------------------------------------------------------
function Ye = dctnoise(X,P8,K8,B1,B2,B3,e,stage,line)
   if (line < 1 | line >8)
      disp('invalid line number');
      Ye = '';
      return;
   end
   if (stage < 1 | stage > 5)
      disp('invalid stage number');
      Ye = '';
      return;
   end

   X1 = B3*X;
   if stage == 1
      X1(line) = X1(line)+e;
   end

   X2 = B2*X1;
   if stage == 2
      X2(line) = X2(line)+e;
   end
   X3 = B1*X2;
   if stage == 3
      X3(line) = X3(line)+e;
   end
```

```
   X4 = K8*X3;
   if stage == 4
      X4(line) = X4(line)+e;
   end

   X5 = P8*X4;
   if stage == 5
      X5(line) = X5(line)+e;
   end
   Ye = X5;
```

```
/************************* Start of BITIO.H ************************/

#ifndef _BITIO_H

#define _BITIO_H

#include <stdio.h>

typedef struct bit_file {

    FILE *file;

    unsigned char mask;

    int rack;

    int pacifier_counter;

} BIT_FILE;

BIT_FILE     *OpenInputBitFile( char *name );

BIT_FILE     *OpenOutputBitFile( char *name );

void          OutputBit( BIT_FILE *bit_file, int bit );

void          OutputBits( BIT_FILE *bit_file,

                          unsigned long code, int count );

int           InputBit( BIT_FILE *bit_file );

unsigned long InputBits( BIT_FILE *bit_file, int bit_count );

void          CloseInputBitFile( BIT_FILE *bit_file );

void          CloseOutputBitFile( BIT_FILE *bit_file );

void          FilePrintBinary( FILE *file, unsigned int code, int bits );

#endif  /* _BITIO_H */


/************************** End of BITIO.H *************************/
```

References

[1] C.W. Kok , "Fast Algorithm for Computing Discrete Cosine Transform". *IEEE Transactions on Signal Processing*, Vol. 45, No. 3, pp. 757-760, March 1997.

[2] W. Pennebaker and J. Michell, *JPEG Still Image Data Compression Standard.* Van Nostrand Reinhold, 1994.

[3] Ephraim Feig and Shmuel Winograd, "Fast Algorithm for the Discrete Cosine Transform". *IEEE Transactions on Signal Processing*, Vol. 40, No. 9, September 1992.

[4] Nam Ik Cho and sang Uk Lee, "Fast Algorithm and implementation of 2-D Discrete Cosine Transform". *IEEE Transactions on Circuit and Systems*, Vol. 38, NO.3, pp. 297-305, March 1991.

[5] J.Y. Jou and J.A. Abraham, "Fault-Tolerant FFT Networks," *IEEE Transactions on Computers,* Vol. 37, pp. 548-561, May 1988.

[6] D.L. Tao, C.R.P. Hartmann and Y.S. Chen, "A Novel Concurrent Error Detection Scheme for FFT Networks," Digest of Papers, The 20th International Symposium on Fault-Tolerant Computing (FTCS-20), Newcastle Upon Tyne, UK, pp. 114-121, June 1990.

[7] D.L. Tao, C.R.P. Hartmann and Y.S. Chen, "A Novel Concurrent Error Detection Scheme for FFT Networks," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 4, pp. 198-221, Feb. 1993.

[8] F. Lombardi and J.C. Muzio, "Concurrent Error Detection and Fault Location in an FFT Architecture," *IEEE Journal of Solid-State Circuits,* Vol. 27, pp. 728-736, May 1992.

[9] C.G. Oh and H.Y. Youn, "On Concurrent Error Detection, Location and Correction of FFT Networks," Digest of Papers, The 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, pp. 596-605, 1993.

[10] C.G. Oh, H.Y. Youn and V.K. Raj, "An Efficient Algorithm-Based Concurrent Error Detection for FFT Networks," *IEEE Transactions on Computers,* Vol. 44, pp. 1157-1162, Sept. 1995.

[11] S.J. Wang and N.K. Jha, "Algorithm-Based Fault Tolerance for FFT Networks," *IEEE Transactions on Computers,* Vol. 43, pp. 849-854, July 1994.

[12] G.R. Redinbo, "Optimum Kalman Detector/Corrector for Fault-Tolerant Linear Processing, "*Digest of Papers, The Twenty-third International Symposium on Fault Tolerant Computing,* Vol. FTCS-23, Toulouse, France, pp. 299-308, June 1993.

[13] Choong Gun Oh and Hee Yong Youn, "On Current Error Detection, Location, and Correction of FFT Networks,"

[14] P. K. Lala, *Self-Checking and Fault-Tolerant Digital Design.* San Francisco:Morgan Kaufmann Publishers, 2001.

[15] George D. Kraft and Wing N. Toy, *Microprogrammed Control and Reliable Design of Small Computers.* Bell Telephone Laboratories, Inc., Englewood Cliffs, New Jersey 1981.

[16] Matteo Frigo, "A Fast Fourier Transform Compiler". *Procedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta GA, May 1999.