1-1988

Variable size block truncation coding with adaptive bit plane omission for image compression

Hieu T. Pham

Rochester Institute of Technology
School of Computer Science

Variable Size Block Truncation Coding with Adaptive Bit Plane Omission for Image Compression

Hieu T. Pham

A Thesis, submitted to
The Faculty of the School of Computer Science,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

December, 1988

Approved by:

S. Radziszowski Jan 23/1989
Professor Stanislaw P. Radziszowski (Chairman)

Peter G. Anderson 23 Jan 89
Professor Peter G. Anderson

James E. Heliotis 1/23/89
Professor James E. Heliotis

Title of Thesis:

Variable Size Block Truncation Coding
with Adaptive Bit Plane Omission
for Image Compression

I, Hieu T. Pham, prefer to be contacted each time a request for reproduction of this thesis is made. I can be reached at the following address:

Date:

Abstract

A modified version of the Block Truncation Coding (BTC), which is a non-information preserving image compression technique, is studied. The first modification is the introduction of variable block sizes to the standard BTC technique. The second modification is the adaptive omission of bit planes. Threshold selections for this modified BTC technique are analyzed in the context of the human visual system. Modified BTC techniques are compared against the standard technique from the point of view of visual image quality and compresion efficiency.

KEYWORDS: Image compression, Block Truncation Coding, adaptive threshold, image quality factor, subjective quality response.

ACKNOWLEDGMENTS

I would like to thank my advisers, Professor P. Anderson and Professor S. Radziszowski, for their help and advices in preparing this thesis. I also would like to extend my sincere gratitude to those at Eastman Kodak Company who supported my graduate work and who made possible the work presented here. I express my deepest appreciation to Scott Chang who gave me the original idea and to E. Zeise, H. T. Tai, D. Schwarz and T. Flynn who offered valuable suggestions and advices.

Finally, I wish to thank my wife who supported and encouraged my thesis work with patience and understanding.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Why is there a need for image compression?

Since the beginning of the use of digital computers in the area of image processing, there have been efforts to compress images for efficient transmission and/or archival storage. In recent years, this desire has been accentuated due to the popularity of commercially available electronic sources such as still electronic cameras and video electronic recorders. These images are typically composed of at least 512×512 picture elements (pixels). Assuming that each pixel is represented by eight bits, corresponding to grey level variations between 0 and 255, this represents approximately two megabits (or 256 Kbytes). Thus, it is very desirable to be able to represent the information in the image with fewer bits and to be able to reconstruct an image that is as close to the original image as possible. Consequently, image compression continues to be one of the major areas of research in image processing.

1.2 An overview of some popular compression techniques

Image compression can usually be grouped into two general methods, information preserving and non-preserving codings. Information preserving coding often takes the form of entropy coding. One performs histograms of the image pixel grey levels and uses optimum code words, such as Huffman or Shannon-Fano codes, for the distribution of the grey levels. A good review of

information preserving coding is presented in [12]. Two major disadvantages of entropy coding are their poor performance in the presence of channel errors and their small compression rate (typical compressed data rate is about 3 bits per pixel; the pixel in the original image is assumed to be 8 bits).

Information non-preserving coding techniques are traditionally subdivided into transform coding and spatial coding. In transform coding, one first transforms the image using a linear orthogonal discrete transform such as the Karhunen-Loeve, Fourier, Cosine, Slant, Walsh, or Haar transform [25], [3], [1], [19], 18], [2]. Once the desired transform is obtained, a subset of the transformed coefficients is retained, quantized and then stored or transmitted. The receiver reconstructs the image by applying the inverse transformation. The Cosine transform seems to be a robust transform for a large class of images, and in some cases, its performance resembles that of the Karhunen-Loeve transform [6].

In cases of high resolution images, transform coding has a tendency to blur the image. However, one of the advantages of transform coding is that it is quite easy to obtain a data rate below 1 bit per pixel. A disadvantage is the large amount of computations to obtain the transform and then operate on the coefficients.

In spatial coding techniques, the basic idea is to generate an array of uncorrelated variables from the given image by using an invertible transformation. The structure of the transformation is defined by using an appropriate image model [11]. By fitting a model to the given image, one generates the so-called residuals, which are less correlated than the original image pixels, and hence can be represented with fewer bits. Given the

quantized residuals and parameters of the model, an image close to the original can be reconstructed.

Hybrid image compression coding is also possible by combining the transform and spatial coding schemes. The coding artifacts of all the non-information preserving techniques are indeed different; in some cases, this prevents accurate comparisons between the methods.

2. Block Truncation Coding

2.1 Introduction

Block Truncation Coding (BTC) was developed at Purdue University by Mitchell and Delp [16]. BTC is an image compression technique which uses a two-level (one-bit) adaptive non-parametric moment-preserving quantizer to represent non-overlapping blocks of pixels. In this chapter, performances of the basic BTC quantizer as well as several other non-parametric quantizers are presented.

2.2 Basic BTC algorithm

The image is first segmented into $n \times n$ non-overlapping blocks ($n = 4$ is a popular choice for still electronic video images). Each block is then coded individually into a two-level signal. The levels for each block are chosen to preserve the first two sample moments.

Let $m = n^2$, and let $X_1, X_2, .. , X_m$ be the values of the pixels in the block of the original image.

Let

$$\overline{X} = \frac{1}{m}\sum_{i=1}^{m} X_i \qquad \text{be the first moment} \tag{2.1}$$

$$\overline{X^2} = \frac{1}{m}\sum_{i=1}^{m} X_i^2 \qquad \text{be the second moment} \tag{2.2}$$

Variance σ is given by:

$$\sigma^2 = \overline{X^2} - \overline{X}^2 \tag{2.3}$$

Thus, if $\overline{X^2}$ and $\overline{X}^2$ are preserved, then the variance is also preserved.

The bit plane is chosen such that all pixel values above some threshold value are set to one, and all other values are set to zero. For the basic non-parametric moment-preserving quantizer, the threshold, X_{th}, is set to $\overline{X}$. Thus each block is described by the pair ($\overline{X}$, σ), plus a bit plane of $n \times n$ 0's and 1's, indicating whether a given pixel is below or above the threshold $\overline{X}$.

At the receiver site, when each image block is reconstructed, it is necessary to find two reconstructed levels, a and b, such that :

if $X_i < \overline{X}$, output = a

if $X_i \geq \overline{X}$, output = b

for i = 1, 2, .. , m

Since the first two sample moments are preserved, the reconstructed levels, a and b, can be found as follows:

Let q be the number of X_i's $\geq X_{th} (= \overline{X})$

Then, the first and second moments of the reconstructed block will satisfy equations (2.4) and (2.5) below:

$$m\overline{X} = (m - q)\,a + qb \tag{2.4}$$

$$m\overline{X^2} = (m - q)\,a^2 + qb^2 \tag{2.5}$$

solving for a, and b:

$$a = \overline{X} - \sigma\sqrt{\frac{q}{m - q}} \tag{2.6}$$

$$b = \overline{X} + \sigma\sqrt{\frac{m - q}{q}} \tag{2.7}$$

A block diagram of the basic BTC algorithm is shown in figure 2.1.

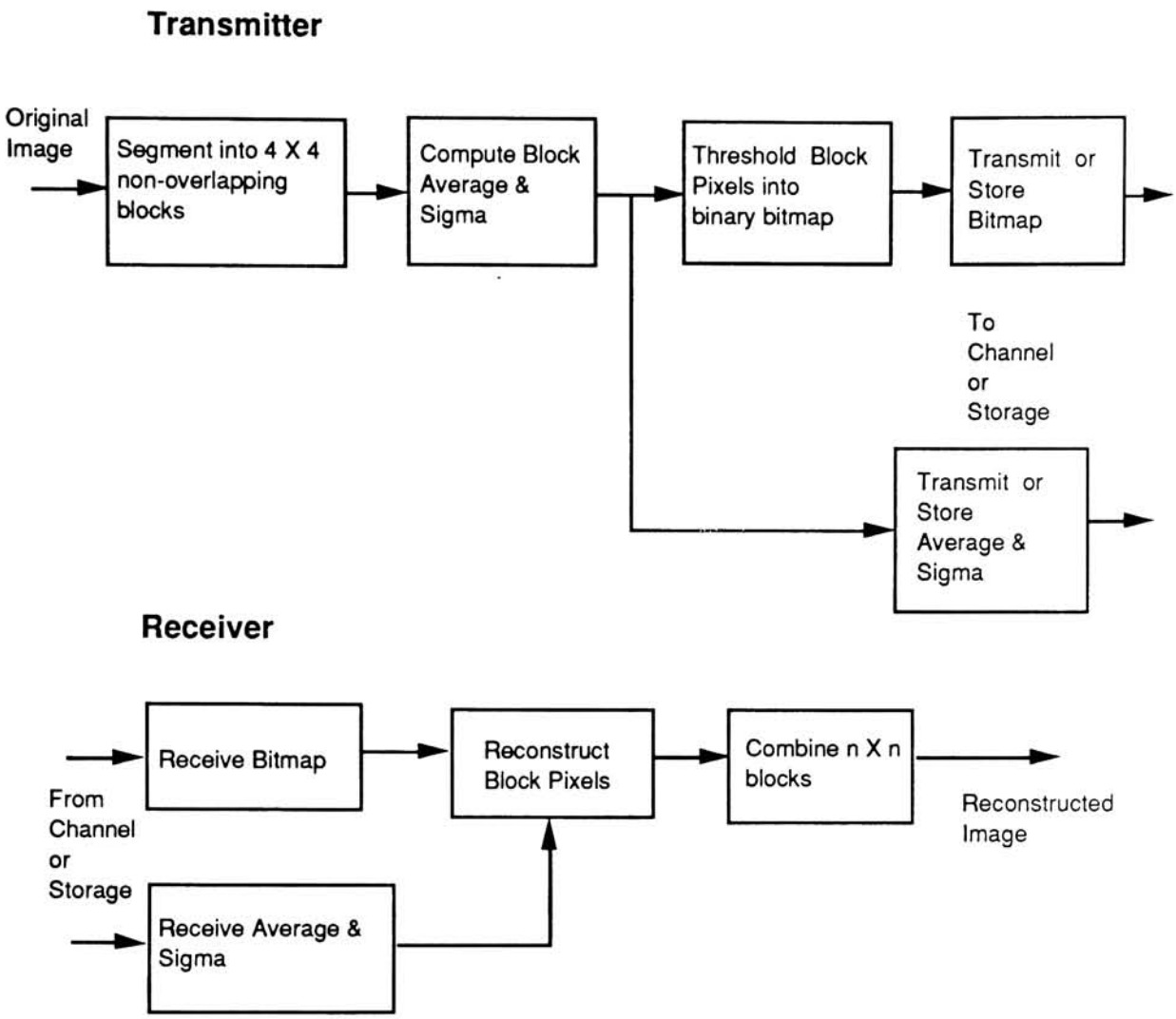


Figure 2.1: Block diagram of the basic BTC algorithm

The bit rate of a compression technique is defined as the number of bits needed to represent a compressed pixel. The bit rate of BTC coding scheme is:

$$\text{Bit Rate} = \frac{m \times n + l_{\bar{X}} + l_{\sigma}}{m \times n} \qquad (2.8)$$

where:

m, n are the block dimensions;

$l_{\bar{X}}$ is the number of bits needed to represent $\bar{X}$;

l_{σ} is the number of bits needed to represent σ;

Thus, for a block size of 4×4, and eight bits are used to represent each of $\bar{X}$ and σ, the bit rate is 2 bits per pixel.

The compression ratio is defined as the size of the original image divided by the size of the compressed image. The compression ratio of the basic BTC algorithm is :

$$\text{Compression Ratio} = \frac{8 \times m \times n}{m \times n + l_{\sigma} + l_{\bar{X}}} \qquad (2.9)$$

where all parameters are the same as those of equation (2.8), and each pixel of the original image is 8 bits.

Note that the product $m \times n$ represents the bit plane overhead of BTC. The compression ratio of standard BTC with 4×4 block size is 4.

Below is an example of BTC coding scheme:

Suppose the block to be encoded is :

$$X = \begin{bmatrix} 121 & 114 & 56 & 47 \\ 37 & 200 & 247 & 255 \\ 16 & 0 & 12 & 169 \\ 42 & 5 & 7 & 251 \end{bmatrix}$$

so :

$$\overline{X} = 98.75$$

$$\sigma = 92.95$$

$$q = 7$$

and the bit plane is :

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

At the receiver site, the levels a and b are found to be:

$$a = 16.7 \approx 17$$

$$b = 204.2 \approx 204$$

and the reconstructed block becomes:

$$\begin{bmatrix} 204 & 204 & 17 & 17 \\ 17 & 204 & 204 & 204 \\ 17 & 17 & 17 & 204 \\ 17 & 17 & 17 & 204 \end{bmatrix}$$

BTC is a lossy compression technique and introduces certain artifacts, but its advantages are simple implementation, low bit rates, and accurate reconstruction of sharp edges. Because the calculations involved are relatively simple and the storage of data is small, BTC is fairy easy to implement on an integrated circuit [8].

2.3 Quantizer design

The original BTC scheme discussed above uses moment preserving quantizers, that is, quantizers which preserve the block mean and variance. The rationale for this approach is that by preserving moments, important visual information is maintained, namely, the central tendency (mean) and the dispersion of busyness about the mean (variance).

Since the quantizer used in BTC is adaptive, the criterion used to design the quantizer greatly impacts the BTC performance in terms of computation time and reconstructed image quality. Consequently, several

other quantizers have been investigated and reported since the introduction of the original BTC.

2.3.1 Preservation of first, second and third moments

The image is segmented into non-overlapping $n \times n$ blocks. Let $m = n^2$, and let $X_1, X_2, \ldots, X_m$ be the pixels values in a given block. The first two moments were described in equations (2.1) and (2.2), the third moment is:

$$\overline{X^3} = \frac{1}{m}\sum_{i=1}^{m} X_i^3 \qquad (2.10)$$

The problem is to find two reconstruction levels and q, where q is the number of X_i's $\geq X_{th}$ (Note that q defines the threshold when X_i's are ordered). To preserve the first three sample moments, the following equations must be satisfied:

$$m\overline{X} = (m - q)\,a + qb \qquad (2.11)$$

$$m\overline{X^2} = (m - q)\,a^2 + qb^2 \qquad (2.12)$$

$$m\overline{X^3} = (m - q)\,a^3 + qb^3 \qquad (2.13)$$

solving for a, b and q:

$$a = \overline{X} - \sigma\sqrt{\frac{q}{m-q}} \tag{2.14}$$

$$b = \overline{X} + \sigma\sqrt{\frac{m-q}{q}} \tag{2.15}$$

$$q = \frac{m}{2}\left[1 + A\sqrt{\frac{1}{A^2+4}}\right] \tag{2.16}$$

where:

$$A = \frac{3\overline{X}\overline{X^2} - \overline{X^3} - 2(\overline{X})^3}{\sigma^3} \tag{2.17}$$

for $\sigma \neq 0$;

if $\sigma = 0$, then $a = b = \overline{X}$.

An interpretation of (2.16) is that the threshold is nominally set to the median of the block ($q = \frac{m}{2}$). The threshold is then biased higher or lower depending upon the value of the third moment which is a measure of the skewness of the ordered X_i's. In general, the value q computed from (2.16) will not be an integer, and in practice, it must be rounded to the nearest integer to determine the bit map assignment. However, the reconstruction

values, a and b, can be computed using either the non-integer value of q or the rounded integer value of q. If a and b are calculated using the rounded value of q, then the first and second moments are preserved but the third moment may not be preserved. Reported results indicated the reconstructed image quality produced by this scheme is comparable or slightly better than that produced by preserving only the first two moments [7].

2.3.2 Preservation of the first and second absolute moments

The idea behind moment preserving quantizers is to preserve some of the sample moments; it is not required that these moments be the mean and variance of the block. An alternative is to preserve the mean (central tendency) and the first absolute central moment (dispersion about the mean). This technique is called Absolute Moment Block Truncation Coding (AMBTC) [13] and is described below:

The sample mean and sample first absolute central moment of a $n \times n$ ($m = n^2$) block are defined as:

$$\overline{X} = \frac{1}{m}\sum_{i=1}^{m} X_i \tag{2.18}$$

$$\alpha = \frac{1}{m}\sum_{i=1}^{m} \left| X_i - \overline{X} \right| \tag{2.19}$$

The mean value, $\overline{X}$, contains information about central tendency; this is the same central tendency information used in the original BTC. On the other hand, the sample absolute central moment α , contains information about dispersion about the mean. The corresponding value used in the original BTC is the sample standard deviation σ.

When reconstructing the image, the quantizer threshold is set to $\overline{X}$, and the problem is to find two reconstruction levels, a and b, such that:

$$m\overline{X} = (m - q)\,a + qb \tag{2.20}$$

$$m\alpha = (m - q) \times (\overline{X} - a) + q \times (b - \overline{X}) \tag{2.21}$$

where q is the number of X_i's $\geq \overline{X}$.

Solving for a and b:

$$a = \overline{X} - \frac{m\alpha}{2(m - q)} \tag{2.22}$$

$$b = \overline{X} + \frac{m\alpha}{2q} \tag{2.23}$$

Each reconstruction level is found by biasing the mean by an amount which depends directly on the dispersion about the mean and inversely with the number of pixels assigned to that reconstructed level.

To process each block, calculations of $\overline{X}$ and α , using (2.18) and (2.19) are required at the transmitter site, and calculations of a and b using (2.22) and (2.23) are required at the receiver site. These equations are easier to compute than the corresponding equations for the mean and variance of the original BTC. Thus, the AMBTC is computationally more efficient, with reported performance comparable to or slightly better than that of original BTC [13].

2.3.3 Error minimizing quantizers

Another approach to design a two-level quantizer for use in a BTC algorithm is to minimize the reconstruction error, given certain constraints on the design and an appropriate error metric. In this section we will discuss two standard error metrics: minimum mean square error (MMSE), and minimum mean absolute error (MMAE) [7].

To use the MMSE fidelity criterion, one proceeds by first constructing a histogram of the X_i's (i = 1,2, ... , m; m = n^2) of the n x n block. Let Y_1, Y_2, .. , Y_m be the sorted X_i's, i.e. $Y_1 < Y_2 .. < Y_m$. Again, let q be the number of $X_i \geq X_{th}$. Then a and b are found by minimizing:

$$J_{MMSE} = \sum_{i=1}^{m-q-1} (Y_i - a)^2 + \sum_{i=m-q}^{m} (Y_i - b)^2 \qquad (2.24)$$

It can be shown that the two reconstructed levels, a and b, required to minimize (2.24) are:

$$a = \frac{1}{m - q} \sum_{i=1}^{m-q-1} Y_i \tag{2.25}$$

$$b = \frac{1}{q} \sum_{i=m-q}^{m} Y_i \tag{2.26}$$

Unfortunately, there is no closed form expression to find X_{th}, so a search has to be done to find the threshold (here are at most m-1 thresholds) which minimizes J_{MMSE}.

The problem of using the MMAE fidelity criterion is similar to the MMSE. The values a and b are found by minimizing :

$$J_{MMAE} = \sum_{i=1}^{m-q-1} \left| Y_i - a \right| + \sum_{i=m-q}^{m} \left| Y_i - b \right| \tag{2.27}$$

where

a = median of $(Y_1, Y_2, \ldots, Y_{m-q-1})$

b = median of $(Y_{m-q}, \ldots, Y_m)$.

Again, the non-parametric quantizer is arrived at by an exhaustive search.

As anticipated, the reconstructed images using the MMSE and MMAE criteria produced smallest computed mean square error and mean absolute

error, respectively. However, it has been reported that the mean square error and mean absolute error measures can not be easily correlated with photo analysts' evaluations [7]. Moreover, the MMSE and MMAE approaches require a large number of computations.

3. Block Truncation Coding with bit plane compression

3.1 Rationale

Equation 2.8 described the BTC bit rate as a function of block size and the number of bits needed to represent the block average and block standard deviation. For a block size of 4×4, the bit plane overhead is 50%, and this overhead increases as the size of the block increases, as shown in table 3.1 below. Thus, compressing the bit plane will enhance the BTC compression efficiency. We will examine several bit plane compression techniques in this chapter.

Block Size	Bit Plane Overhead
4 X 4	50 %
8 X 4 or 4 X 8	67 %
8 X 8	80 %
16 X 8 or 8 X 16	89 %
16 X 16	94 %

Table 3.1: Bit plane overhead as a function of block size

3.2 Various types of models

3.2.1 Bit plane reduction by independent/dependent bits

One approach to bit plane encoding is to denote certain locations in the bit map as independent bits and others as dependent bits [17], [15]. Only the independent bits are transmitted; the dependent bits are determined at the receiver site using predefined logic. One possible choice of independent/dependent bit is shown in figure 3.1 where the independent bits are underlined [15].

A B C D

E F G H

I J K L

M N O P

Figure 3.1: Independent/dependent bits

Rules used to determine the dependent bits are:

B = 1 iff [(A and D) = 1] or [(F and J) = 1]

C = 1 iff [(A and D) = 1] or [(G and K) = 1]

E = 1 iff [(A and M) = 1] or [(F and G) = 1]

H = 1 iff [(D and P) = 1] or [(F and G) = 1]

I = 1 iff [(A and M) = 1] or [(J and K) = 1]

L = 1 iff [(D and P) = 1] or [(J and K) = 1]

N = 1 iff [(M and P) = 1] or [(F and J) = 1]

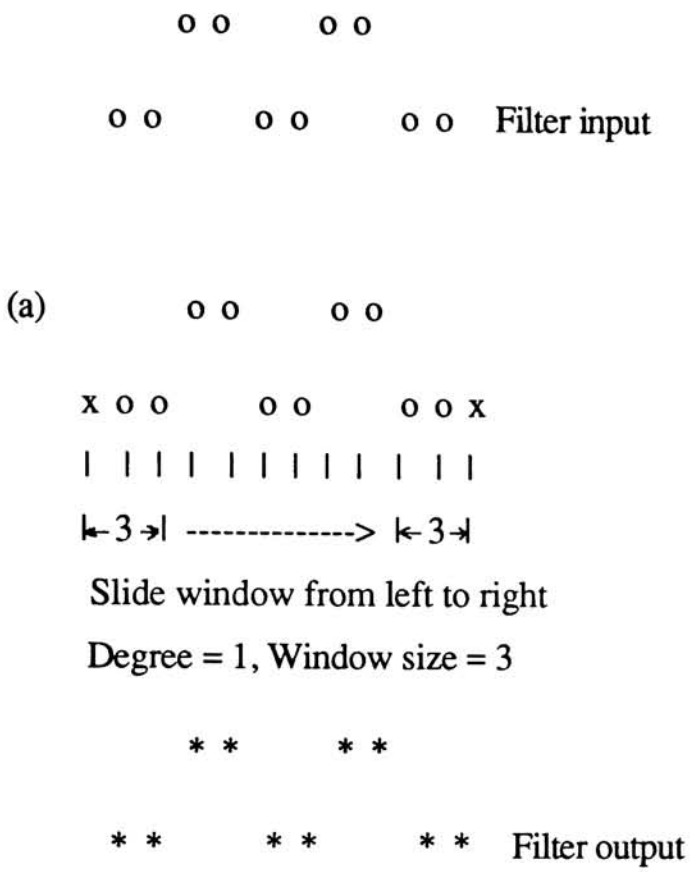
O = 1 iff [(M and P) = 1] or [(G and K) = 1]

In practice, a look-up table would be used to determine the dependent bits. This particular choice of logic is reported to preserve all horizontal and vertical edges and some diagonal edges [15]. However, the reconstructed images, in general, exhibit poor image quality. It seems possible that acceptable quality might be obtained with a different choice of independent bits or logical expressions, but the results will be highly image dependent

3.2.2 Bit plane encoding using Median Filter [4]

3.2.2.1 Median filter [5]

A main characteristic of the median filter compression technique is the mapping of the input signal into a root signal, where a root signal is an invariant signal to the filter. The root signal retains the spatial characteristic of the input signal such as edges, but deletes redundant impulses and oscillations. The implementation of a median filter requires simple, nonlinear digital operations. With a sampled input signal of length L, we slide a window that spans 2N+1 points, where N is the degree of the root filter. The filter operation is to replace the center bit of the window. The filter output is set to the median value of these 2N+1 signal samples. N samples are appended to the beginning and to the end of the sampled signal to account for startup and end effects. Thus the length of the sampled signal becomes L+2N. The value of the appended samples at the beginning is the value of the first sample, and the value of the appended samples at the end of the signal is the value of the last sample. The example below, figure 3.2, shows a binary signal of length 10 being filtered by filters with degrees 1, 2, and 3. The input bits are shown with o's (o), the appended bits are shown as crosses (x), the filter output or root signal is shown with asterisks (*).


o o o o
o o o o o o Filter input
(a) o o o o
x o o o o o o x
| | | | | | | | | | | | |
|← 3 →| ----------> |← 3 →|
Slide window from left to right
Degree = 1, Window size = 3
* * * *
* * * * * * Filter output
(b) o o o o
x x o o o o o o x x
| | | | | | | | | | | | |
|<-- 5 -->| ---------> |<-- 5 -->|
Slide window from left to right
Degree = 2, Window size = 5
* *
* * * * * * * * Filter output

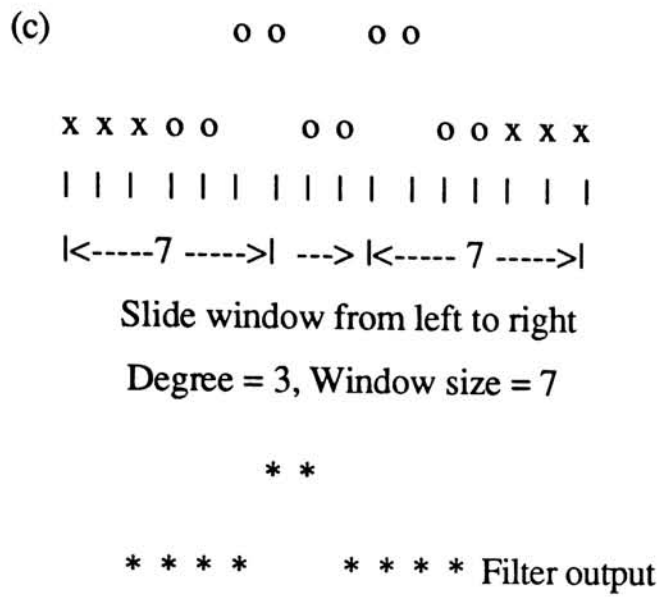


Figure 3.2: Example of median filter

Many variations on the implementation of the median filter can be constructed, but the principal idea is to collect an odd number of samples from the signal, rank them by value, and select the median. This filter has the property that if a signal is repeatedly median filtered, the output after each filter pass converges to a signal invariant to further median filtering. The signal set invariant to the filter is called "the root signal set". The next example illustrates the one dimensional filtering process, where o's denote the input samples and x's denote the appended samples.

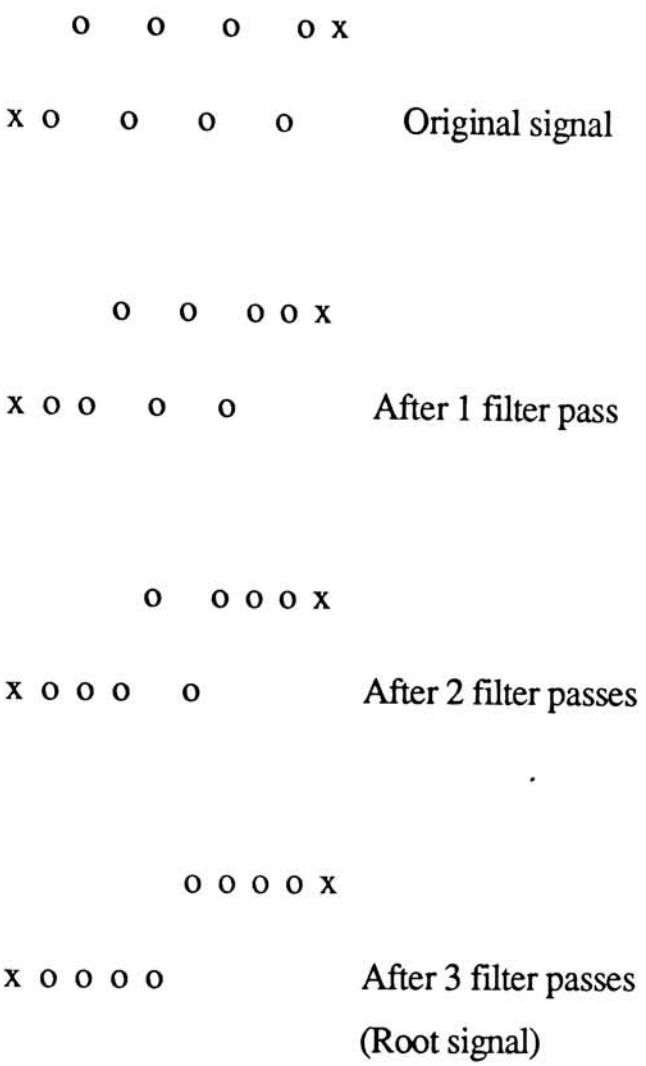


Figure 3.3: Illustration of 1-dimensional median filtering process

The number of binary root signals R_n for a filter of window size 3 and signal length n are related to the Fibonacci numbers, f_n, by:

$$R_n = 2 f_{n+1} \qquad (3.1)$$

where

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

3.2.2.2 Root coding

The approach is to median filter the bit plane into a root signal and, instead of transmitting the bit plane in the binary domain, send a code specifying the root. It has been shown that the bit plane signals have a high probability of belonging to the root signal set [4]. Since the root signal set is smaller than the binary bit plane, we need fewer bit to represent a block in a root domain. In the process of median filtering the bit plane, we have the option of filtering the block in two dimensions, obtaining a single root for the entire block, or of filtering in one dimension, obtaining one root for each row of the bit plane. The next example shows the one-dimensional median filtering process for an BTC bit plane.

$$\begin{bmatrix} 10 & 10 & 11 & 12 \\ 10 & 9 & 12 & 15 \\ 3 & 4 & 12 & 15 \\ 2 & 4 & 10 & 15 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 12 & 12 & 12 & 12 \\ 12 & 12 & 12 & 12 \\ 3 & 3 & 12 & 12 \\ 3 & 3 & 12 & 12 \end{bmatrix}$$

(a) **(b)** **(c)** **(d)**

(a) = Input block, applying BTC, we have $\overline{X} = 9.6$; $\sigma = 4.2$.
two reconstructed values: a = 3, b = 12.

(b) = BTC bit plane

(c) = Filtered block

(d) = Reconstructed block using BTC with bit plane median filtered.

Figure 3.4 : One-dimensional median filtering process on a 4 × 4 BTC block

Two-dimensional filtering is very complicated and will not be discussed here. Based on equation (3.1) above, there are 10 roots for each row of the bit plane (n=4, $R_4 = 10$). Two of the ten roots are much less likely to occur than the rest of the root set [4], [23]. Thus, if we delete two roots from the root set, we only need three bits to specify eight possible roots of each row of the bit plane, and the bit plane can be encoded with twelve bits instead of

sixteen bits used by the standard BTC approach, a 25% reduction. It has been reported that, in general, the reconstructed image produced by this technique is slightly better than that of the independent/dependent bits discussed in section 3.2.1 [4], [23].

3.2.3 Bit Plane Omission

Another method of compressing the bit plane is to omit the transmission of the bit plane if the variance of the block is small. The overall compression ratio is highly image dependent. This technique can increase the compression ratio significantly if the variance threshold is set properly. The threshold selection analysis, as well as the reconstructed image quality for the bit plane omission technique are the subject of discussion in chapter 5.

4. Image Quality based on properties of human visual system

The ultimate judges of the quality of reproduced images are human observers. Thus, visual judgement is probably the natural criterion. However, the root mean square error (RMSE) method, described in equation 4.1 below, has been often used as a criterion for measuring the quality of reconstructed images. The reason for its popularity is its mathematical simplicity, and also because the human visual judgement is subjective and difficult to model mathematically.

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(X_i - Y_i)^2} \qquad (4.1)$$

where:

m: number of pixels.

X_i: the value of i^{th} pixel of the original image.

Y_i: the value of the i^{th} pixel of the reconstructed image.

RMSE describes the numerical difference, as far as pixel values are concerned, between the original and the reconstructed images without taking into account the psychovisual properties of the human visual system. It appears that an observer bases his or her estimate of picture quality on the sensitivity of the eye to different spatial and temporal frequencies. Other attributes associated with the picture such as sharpness, graininess and

contrast also affect how human beings judge picture image quality. Sakrison [22] surveyed the characteristics of the human visual system and discussed several ways to determine the properties of the human observer. Even though the human visual system has been studied and modeled for decades to evaluate the image quality of photographic prints, the proposed models are mathematically too complex to be used on a routine basis. With the recent advances of digital computers, it is now possible to apply some of those techniques to evaluate image quality.

In this chapter, we discuss a model developed by Hunt [10] and modify it for our purpose. We begin the discussion by a brief functional description of the human visual system. A detailed description of the visual system appears in references [20] and [21].

4.1 The human visual system

Figure 4.1 below illustrates the principle components of the human eye. Light is focused by the cornea and lens onto the retina at the back of the eyeball. The retina consists of a layer of photoreceptors and connecting nerve cells. The photoreceptors are of two kinds: rods and cones. The visual system can operate at levels of illumination varying over 10 orders of magnitude, i.e. 10 log units or a range of 10^{10}. The rods take part in reception for the lower several orders of magnitude. This is termed the scotopic region and goes from darkness to dimly lighted surroundings. The cones take part in reception for the higher 5-6 orders of magnitude. This is termed the photopic region and goes from well lighted to bright sunlight surroundings. In the region in between, termed mesopic, both the rods and

cones are active. Our interest is associated with the photopic region since observation of pictures are normally take place in well lighted surroundings. The cones are densely packed in the center of the retina, in an area called the fovea. Cones in the fovia are spaced about 2 μm or 0.5 minutes of arc apart in the human retina. This density drops outside a circle of roughly 0.5° radius but is still high within a circle of 1° radius. The iris of the eye acts as an aperture, or spatial filter, very close to the plane of the lens. In bright light, the iris is about 2 mm in diameter, and for green light, acts as a low pass filter with pass band of about 60 cycles per degree of arc subtended in the field of vision. Thus, the human eye eliminates aliasing errors in the sampled image formed by the cones spaced at a density of 120 cones per degree. This explains why a human observer is sensitive toward certain spatial frequencies in the image.

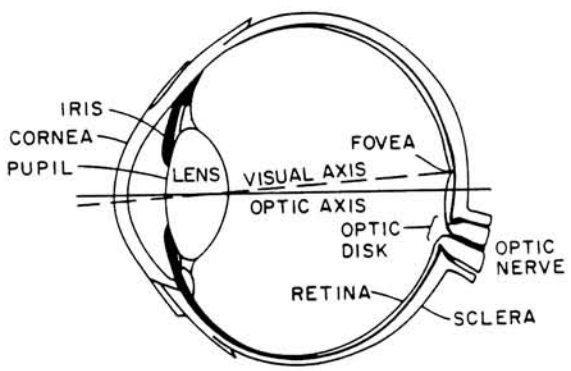


Figure 4.1: Diagram of cross section of the human eye

4.2 Power-Law Stimulus-Response model for measuring image quality

In this section, we describe a model called the Power-Law Stimulus-Response, proposed by Hunt and Sera [10] for measuring image quality.

There are two distinct environments in which human observers interact with images. The first is referred to as performance or task oriented environment. In this situation, the human observer must use the image to perform a task. For example, a physician must detect a disease condition from a radiograph, or an air traffic controller must find the radar image of the airliner he or she is directing to the airport. The image is a tool employed in achieving a particular goal. The second situation in which humans interact with images can be described as non-performance environments. It is characterized by the existence of recreational, entertainment, or aesthetic purposes. Broadcast television, amateur photography and motion pictures are examples of the second case. Although image quality is important in both environments, they are distinctly different, and we should not expect the same quality measure to apply to both cases. The Power-Law Stimulus-Response model is used to evaluate images in the non-performance environments. The measure is based on a common principle, scaling of subjective response from physical stimuli. There is extensive work available describing the response perceived by an individual to the stimulus. This response is expressed by an equation of the form:

$$SQR = k\,(s - s_0)^{\gamma} \qquad (4.2)$$

where:

SQR is the Subjective Quality Response assigned to the image by the observer.

k is a scaling constant.

s_o is a threshold of the stimulus, i.e. for $s < s_o$, no stimulus is perceived.

γ is a power coefficient which determines the non-linear transformation of stimulus to response.

s is the computed stimulus of the observed image whose computational procedure will be discussed in section 4.2.1.

The development and application of power-law stimulus-response, described in equation 4.2 above, is due to the work of Stevens [24] and represents the most important advance in stimulus response modelings since the earlier logarithmic relations postulated by Fechner.

4.2.1 Stimulus calculation procedure

Computation of the stimulus s, mentioned in equation 4.2 above, is carried out in the following steps:

(1) The image f(x,y) is transformed by a non-linear function L which represents the transformation of incident light to perceived brightness. This is because human visual system responds linearly in the brightness domain. Thus, g(x,y) = L(f(x,y)), where :

$$L(X) = 24.98 \times (100\,X + 0.2631)^{0.3333} - 16 \qquad (4.3)$$

(2) Compute the Fourier transform of g(x,y), G(x,y).

(3) Find the magnitude of G(x,y) and move the zeroth lag to the center of the spectrum, for ease of successive computations. We now have G1(x,y).

(4) Calculate the Modulation Transfer Function (MTF) matrix for a particular viewing condition based on the human visual response transfer function [9], a function V(wx,wy) shown in figure 4.2.

(5) Multiply G1(x,y) by the MTF matrix, point by point, and sum all the products to obtain the image stimulus.

The image stimulus is essentially the integrated image signal weighted by the brightness response and human visual MTF.

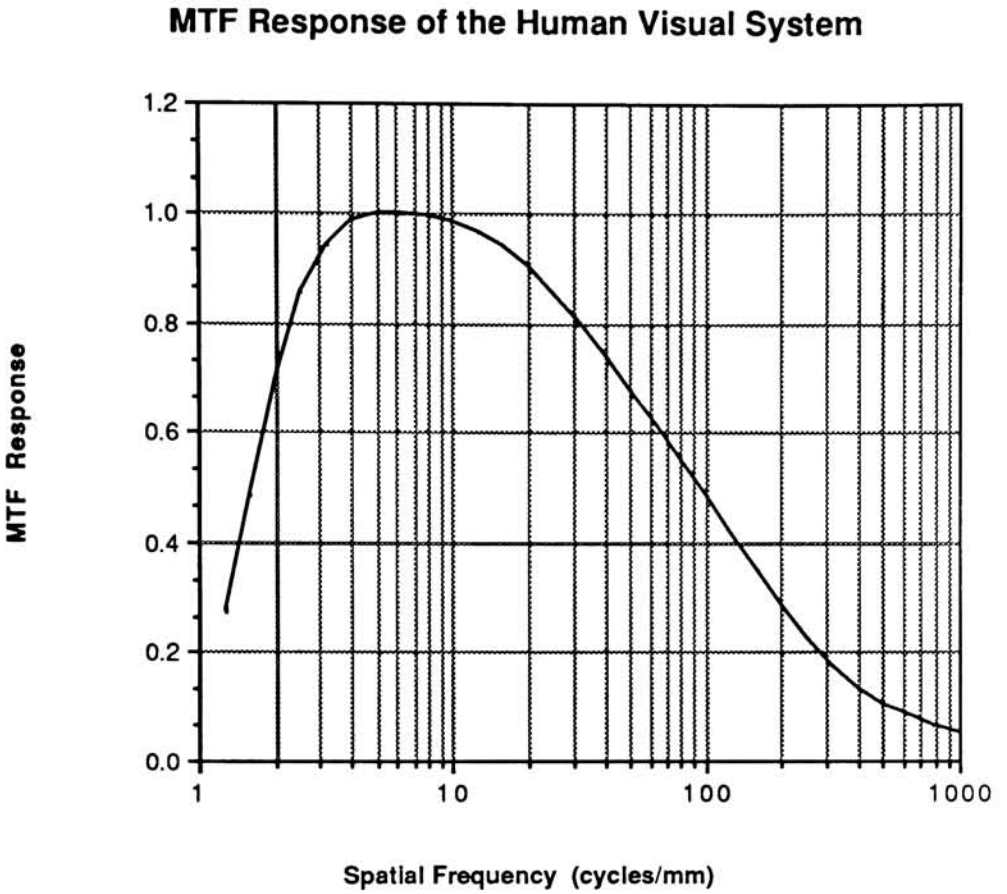


Figure 4.2
Modulation Transfer Function response of a typical observer at a viewing distance of 30 cm

4.2.2 Subjective Quality Response (SQR) calculation

Hunt and Sera reported that parameters k and s_o are not constants. They are functions of the signal to noise ratio (SNR) of the observed image. This is expected, since SNR causes a change in perceived quality response. Thus, equation 4.2 should be replaced by a model of the form:

$$SQR = k(SNR) \times (s - s_o(SNR))^{\gamma} \qquad (4.3)$$

where :

$$k(SNR) = -0.1231\,(SNR)^2 + 4.1866\,(SNR) + 62.4374$$

$$s_o(SNR) = (0.778)^{SNR + 11}$$

$$\gamma = 0.2059$$

The SNR calculation will be discussed in the next chapter.

5. Variable size Block Truncation Coding with adaptive bit plane elimination, approach and results

5.1 Statement of the problem and rationale

In previous chapters, we have described the basic BTC algorithm, have examined various quantizers which can improve the reconstructed image quality, and have reviewed several bit plane compression techniques to increase the overall compression ratio. In this chapter, we discuss two modifications to the basic BTC algorithm which will increase the compression ratio with little degradation in quality of the reconstructed image. The first modification allows the block size to change dynamically while the image is compressed. The second modification conditionally eliminates the encoding of bit planes. Encoding of high activity regions requires small blocks for accurate image reconstruction, while low activity regions can be coded with large blocks. By allowing the block size to vary in response to local signal characteristics, the compression ratio can be improved at the possible expense of some reconstructed image quality degradation. The second modification extends the above concept to the encoding of bit planes. In the BTC algorithm, bit planes contain detailed structures of the image. Thus, high activity regions require the coding of bit planes so that those detailed structures can be faithfully reconstructed. Low activity regions, on the other hand, do not contains much details. Consequently, eliminating bit planes at these regions will result in higher compression ratio with little loss in image quality. Going back to equation 2.9 in chapter 2, which describes the compression ratio of the BTC

algorithm, we see that as the block size increases, the compression ratio increases. Secondly, table 3.1 in chapter 3 indicates that the bit plane overhead (the portion of the compressed image devoted to the bit plane) of the BTC algorithm is quite significant. This overhead is 50% for a 4×4 block size and increases to 94% as block size grows to 16×16. These two observations suggest that in order to increase the compression ratio of the BTC algorithm, the block size should be increased and bit plane encoding should be eliminated whenever possible. The combination of these two factors can increase the compression ratio dramatically, as depicted in figure 5.1.

Obviously, compression ratio is not the only criterion in image compression. Reconstructed image quality is also important. After all, if one can not recognize the reconstructed image, then one should not do the compression in the first place. Thus, one can not choose any arbitrary block size, nor eliminate the bit plane encoding at will. It is the objective of this thesis to define and evaluate the following:

a. Find a parameter which serves as an effective threshold to vary the block size as well as to determine whether bit plane encoding should take place.

b. Investigate an appropriate method to quantify image quality. The method should produce results which comply with human visual perception of image quality.

c. Define proper values of the threshold to maximize compression ratio with little degradation of reconstructed image, whose quality rating is calculated using the method mentioned in the preceding paragraph.

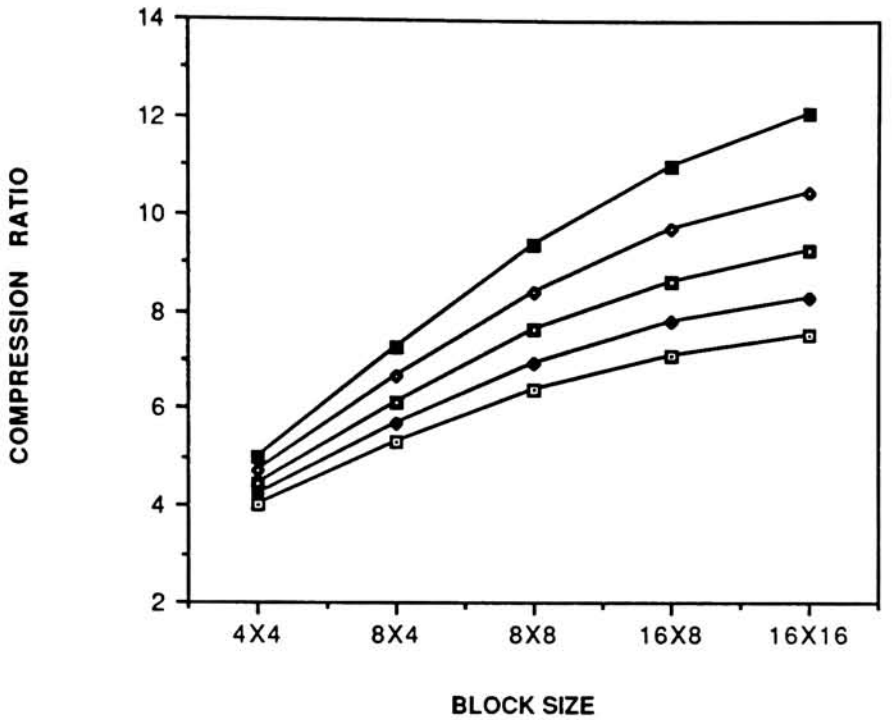


Figure 5.1

Compression Ratio vs. Block Size and % Bit Plane Reduction
Five curves, from top to bottom, represent: 40%, 30%, 20%, 10% and 0% bit plane reductions.

5.2 Threshold selection

There are different image properties of importance to image quality. The importance of these properties may depend on the observer's experience and preference, the type of tasks he or she would like to perform with the image, the nature of the information sought from the image, and so on.

Consequently, image quality is likely to be the result of contributions from a number of different perceptual dimensions. Marmonlin and his colleagues at the Swedish Defense Research Institute attempted to identify the independent dimensions of image quality by using multidimensional scaling techniques [14]. The factors Marmonlin identified are summarized below and listed in order of relative importance:

1) The most important quality dimension is that which relates to image sharpness. "Sharpness" is a subjective property that must be related to an objective measurement on the image. Marmonlin found it is most strongly related to the area under the Modulation Transfer Function (MTF).

2) The second most important quality dimension is noise. Marmonlin found that signal to noise ratio was a physical property that could be related to the perceived effects of noise on image quality. Marmonlin also reported that the effects of noise on image quality were weak. This finding agrees with the intuitions of experienced users of imagery. The capabilities of the human visual system to "see through" noise in the image are often quite profound.

3) Image contrast is the third important quality dimension. The weak relation between image quality and contrast differs from other studies which found contrast to be a main factor in degrading the image quality. One explanation may lie in the role of system MTF, as pointed out by Granger and Cupery [9]. A decrease in image contrast can be incorporated in the MTF, by decreasing the MTF at all frequencies and multiplying the MTF by a common factor less than one. Thus, MTF response and contrast become coupled together.

4) Marmonlin indicated that texture could be the physical image property to describe the fourth dimension of image quality.

Marmonlin's work indicated that sharpness and noise are of great importance to image quality. In choosing an appropriate parameter for sizing blocks and conditionally encoding bit planes during block truncation coding, one should not consider sharpness, however. This is because BTC reconstructed images normally have sharp edges due to its block-oriented operations [7]. Contrast is not a very good choice either, because BTC conserves the block averages throughout the image. As with all non-information preserving image codings, coding artifacts are produced in the reconstructed image. BTC does produce sharp edges, but these edges have a tendency to be ragged. This is because BTC only preserves image structures at the block level, not at the pixel level. Moreover, false contours may exist in the low contrast regions of the image. This is due to inherent quantization noise in the one bit quantizer. These two problems can be classified as image noises. Thus, noise is selected as the threshold parameter for varying the block size and adaptably eliminating the encoding of bit planes.

5.3 Approach

As stated above, one of the objectives of this thesis is to verify the feasibility of varying the block size and conditionally excluding the bit plane codings to improve the compression efficiency with little degradation in the reconstructed image quality. All programming was done with Microsoft C 5.0 language, except for the two-dimensional Fast-Fourier-Transform (2D-FFT) operation. The 2D-FFT operation of images was done with a software

package named MathLab. All program listings are included in the appendix. The images were displayed on a monochrome video monitor using the TARGA8 graphic display board. The modified BTC approach is described below.

5.3.1 Quantizer selection

From the discussion of Quantizer Design in chapter 2, it is apparent that Absolute Moment Block Truncation Coding (AMBTC), section 2.3.2, offers the best performance with a minimum number of calculations. Thus, it is selected over the standard BTC.

5.3.2 Threshold calculations

As stated above, image noise has been selected as the threshold parameter. However, it is not possible to calculate the amount of noises present in the image while compressing it, because image noise represents the differences between the original and the reconstructed images. Thus, one can only "predict" the image noise during compressing. The AMBTC requires the calculations of the block average, $\overline{X}$, and the block first absolute central moment, α. The ratio DF= $\frac{\alpha}{\overline{X}}$, can be interpreted as the relative dispersion from the sample mean the block pixels exhibit within a given block. A small value of DF means all block pixels values are very close together. Thus, the difference between these pixels and the reconstructed block pixels will be small, or low image noise. Conversely, when DF is large, the difference between the original block pixels and those from the reconstructed block is large, or more image noise. Consequently, DF, which stands for distortion

factor, is used to represent the image noise of the reconstructed image, and is expressed in percent for convenience, i.e. $DF = 100 \times \frac{\alpha}{\overline{X}}$.

5.3.3 Block size types and coding mechanism

Four different block sizes are selected, 4×4, 8×4, 8×8, and 16×8. It has been reported that AMBTC using 6 bits for $\overline{X}$ and 4 bits for α results in very little quality loss. Thus, is it decided that the modified BTC uses 8 bits for $\overline{X}$ and only 6 bits for α. The remaining two bits are used to indicate the type of block size. For each block, the compressed data is organized as :

(1) Byte #1: $\overline{X}$, 8 bits.

(2) Byte #2: α, 6 bits and 2 bits, (i, j), for block size coding.

Bit	7	6	5	4	3	2	1	0
	<----		α		---->		i	j

where

i	j	Block size type
0	0	4×4
0	1	8×4
1	0	8×8
1	1	16×8

(3) Bit plane data if the bit plane encoding takes place, otherwise none:

2 bytes for 4×4,

4 bytes for 8×4,

8 bytes for 8×8 and

16 bytes 16×8.

5.3.4 Strategy for elimination of bit plane encodings

Since we are going to vary the block size and conditionally eliminate the encoding of the bit plane at the same time, there will be one threshold for each action. The first threshold, T1, is used to decide if the block size should be varied. The second threshold, T2, is used to decide if the bit plane of a given block should be encoded. It has been observed, visually as well as analytically, that BTC with large block size results in less image noise than BTC without bit plane encoding. This is because bit planes contain image structure. Without bit planes, there is no image. Whereas, with large block size, the image structure is still preserved, but distorted. Thus, the following strategy is employed:

One starts out with the largest block size, 16×8.

(1) Calculate the distortion factor, DF. Go to (2).

(2) If (DF $\geq$ T1) & (block size $\neq 4 \times 4$), select next smaller block size then go back to (1), else transmit $\overline{X}$, α and the bit plane code bits, then go to (3).

(3) If DF $\geq$ T2, encode and transmit the bit plane, else done.

Also, T2 is assumed to be less than or equal to T1.

5.4 Program description & results

Program *mbtc.c* (Modified Block Truncation Coding) implements the modified BTC described above. The program queries the user for image size and values of two thresholds. While compressing, the program keeps track of the block size type usages for compression efficiency calculation and

reports these data afterward. The image used is the Building picture. It is a matrix of 256 × 256 pixels, 8 bits each, video image. This image was processed with T1 varying from 3 to 20 and T2 varying from 3 to 8. It is observed that with values of T1 above 20, the reconstructed image quality is subjectively unacceptable. The same thing is true for T2 greater than 8. Table 5.1 below summarizes the compression ratio results.

As expected, the compression ratio increases as T1 and T2 increase. The original as well as all the reconstructed images were displayed on a monochrome video monitor. These images were captured using photographic film, reproduced using a high quality electrophotographic process, and are included at the end chapter 6.

Image No.	Image Type	Compression Ratio
4	5T0	4.8223
5	5T3	6.0170
6	5T5	6.7454
7	8T0	5.0490
8	8T3	6.3114
9	8T5	6.8727
10	8T8	7.9673
11	10T0	5.2078
12	10T3	6.5446
13	10T5	7.1001
14	10T8	7.8453
15	12T0	5.3525
16	12T3	6.7606
17	12T5	7.3375
18	12T8	8.0043
19	15T0	5.6390
20	15T3	7.0532
21	15T5	7.6510
22	15T8	8.2752
23	20T0	6.0772
24	20T3	7.6492
25	20T5	8.3129
26	20T8	8.9049

Table 5.1

Under Image Type column, the first number represents the first threshold , the second number after T represents the second threshold. Thus 5T3 means the image was compressed with T1 = 5 and T2 = 3.

5.5 Subjective Quality Response results

The Subjective Quality Response (SQR) of all 23 images listed above were calculated using the procedure described in section 4.2 of chapter 4. At first, for each reconstructed image, the image stimulus was calculated as described in section 4.2.1. Program *convert.c* transforms image pixel values from the reflectance domain to the lightness domain. This program produces output data format compatible with that of MatLab software. The Modulation Transfer Function (MTF) matrix was calculated using program *mtfcal.c*. It generates a 256×256 matrix, whose elements represent the weighted MTF response of a typical human observer under a particular viewing condition. This matrix also has data format compatible with that of MatLab software. Note that this matrix had to be generated only once because the viewing condition is the same for all reconstructed images. The 2D-FFT of reconstructed image, whose values are expressed in the lightness domain, was computed using program *stimulus.m*. This program runs inside the MatLab environment. The image stimulus was calculated by performing point to point multiply-and-accumulate between the MTF matrix and the image 2D-FFT matrix.

Next, the program *snr.c* calculates the signal to noise ratio (SNR) value of the reconstructed image. The SNR of the whole image is the sum of the SNR of its individual pixels, as described in equation 5.1 below:

$$SNR_{image} = 10 \times \log_{10}\left(\frac{1}{m}\sum_{i=1}^{m}\left|\frac{X_i}{X_i - Y_i}\right|\right) \qquad (5.1)$$

where:

m is the number of pixels

X_i the value of pixel i^{th} of the original image

Y_i the value of pixel i^{th} of the reconstructed image

Table 5.2 below lists the stimulus, SNR and SQR results.

Image No.	Image Type	Stimulus	SNR(db)	SQR
4	5T0	4.7821	16.3831	135.2317
5	5T3	4.7759	15.6715	134.9581
6	5T5	4.7649	15.5016	134.8119
7	8T0	4.777	16.3155	135.1869
8	8T3	4.7696	15.6488	134.9110
9	8T5	4.7629	15.5077	134.8034
10	8T8	4.7272	15.3978	134.5366
11	10T0	4.7738	16.3036	135.1654
12	10T3	4.7675	15.6470	134.8980
13	10T5	4.7613	15.5022	134.7913
14	10T8	4.7399	15.4074	134.6162
15	12T0	4.7646	16.2784	135.1056
16	12T3	4.7633	15.6265	134.8639
17	12T5	4.7564	15.4817	134.7512
18	12T8	4.7380	15.4038	134.6032
19	15T0	4.7595	16.2478	135.0680
20	15T3	4.7542	15.6152	134.8055
21	15T5	4.7483	15.4767	134.7022
22	15T8	4.7325	15.4092	134.5739
23	20T0	4.7439	16.1409	134.9472
24	20T3	4.6766	15.5261	134.3060
25	20T5	4.6705	15.3881	134.1974
26	20T8	4.6571	15.3337	134.0877

Table 5.2

One can see that as T1 and T2 increase, the SNR decreases, and the stimulus and SQR decrease. One might notice that stimulus and SNR values are different by small amounts. This is because the human visual system is very sensitive toward sharpness and very tolerable toward noise. So does the stimulus value, because it is modeled after the human perception system. The BTC algorithm, as has been stated, reproduces sharp images because of the block-oriented operations. SQR calculation represents the amount of preference above some threshold value an observer has toward an image. Since the unit of SQR is arbitrary, an image which has a larger SQR value only means that it is preferred over another image which has a smaller SQR value. The viewer is free to scale the "raw" SQR value linearly to his or her preference. For example, if one thinks that image 5T0 is rated 100 points and image 20T8 is rated 50 points, then the relative SQR values of all other images can be calculated as shown below:

$$\text{Relative SQR}_i = \frac{\text{SQR}_i - 134.0877}{135.2317 - 134.0877} \times (100 - 50) + 50 \qquad (5.2)$$

where:

SQR_i is the "raw" SQR value of image i

135.2317 is the "raw" SQR value of image 5T0

134.0877 is the "raw" SQR value of image 20T8

Thus, the relative SQR value of image 20T3, under the above assumption is 59.54.

6. BTC vs. Modified BTC

In this chapter, performances, i.e. the compression ratio and the reconstructed image quality, of the modified BTC (MBTC) are compared against those of AMBTC. Program *btc.c* implements the AMBTC algorithm. It allows the user to specify the image size as well as the desired block size. With three different block sizes, 4×4, 8×4, and 8×8, the AMBTC results of the Building picture are listed in table 6.1 below.

Image No.	Image Type	Compression Ratio
1	4×4	4
2	8×4	5.333
3	8×8	6.40

Table 6.1

Results of the SQR calculation are listed in table 6.2 below.

Image No.	Image Type	Stimulus	SNR(db)	SQR
1	4×4	4.8310	17.4205	135.5532
2	8×4	4.7770	16.0163	135.1011
3	8×8	4.6740	15.2646	134.1417

Table 6.2

Since MBTC employs 4 different block sizes (4×4, 8×4, 8×8, and 16×8), the quality of its reconstructed images is at best equal to that of BTC processed with 4×4 block size, i.e. when T1 and T2 are set to zero. Thus, the reader should compare MBTC images against BTC images processed with block sizes 8×4 and 8×8. Figure 6.1 and 6.2 and 6.3 compare compression ratios, stimuli and SQR values of all 26 images. Note that the image numbers of the horizontal axis are the same as those mentioned in tables 5.1, 5.2, 6.1 and 6.2.

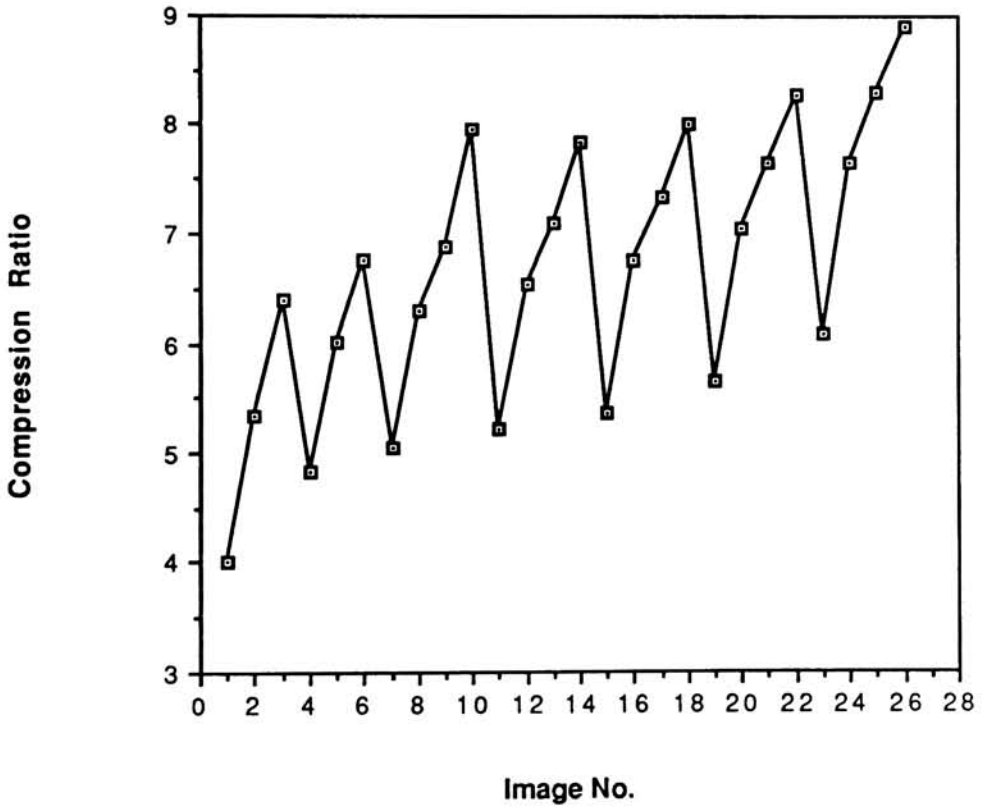


Figure 6.1

Compression Ratios of MBTC vs. BTC images

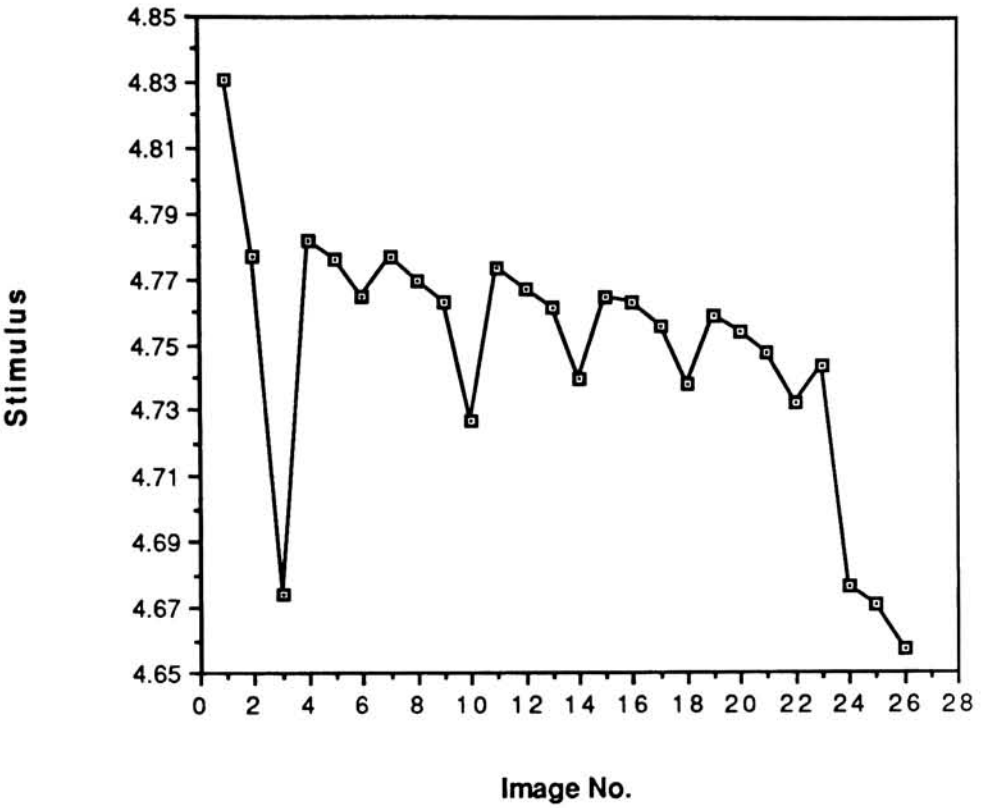


Figure 6.2

Stimulus Values of MBTC vs. BTC images

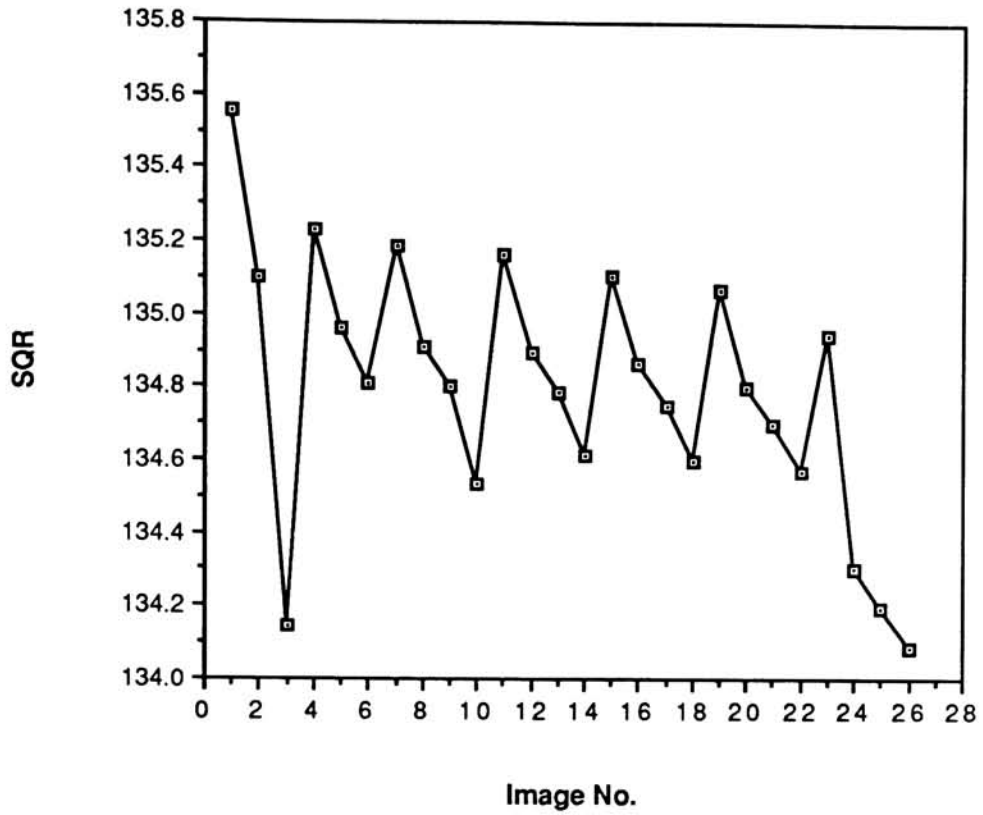


Figure 6.3
SQR values of MBTC vs. BTC images

As expected, MTBC images performs quite well a far as compression ratio and Subjective Quality Response value are concerned. Without bit plane omission, MBTC with threshold T1 equal to 12% or greater offers more compression than BTC with 8×4 block size. With bit plane omission, MBTC with threshold T2 equal to 5% delivers more compression than BTC with 8×8 block size. As far as reconstructed image quality is concerned, without bit plane reduction, MBTC with T1 $\leq$ 12% produces higher quality

images than BTC with 8 × 4 block size. The reconstructed image quality deteriorates sharply when BTC block size changed to 8 × 8. MBTC images, on the other hand, degrade slowly until T1 > 15%. The same observation holds true for the overall performance, which is the product of compression ratio and SQR value, figure 6.4, of MBTC versus BTC.

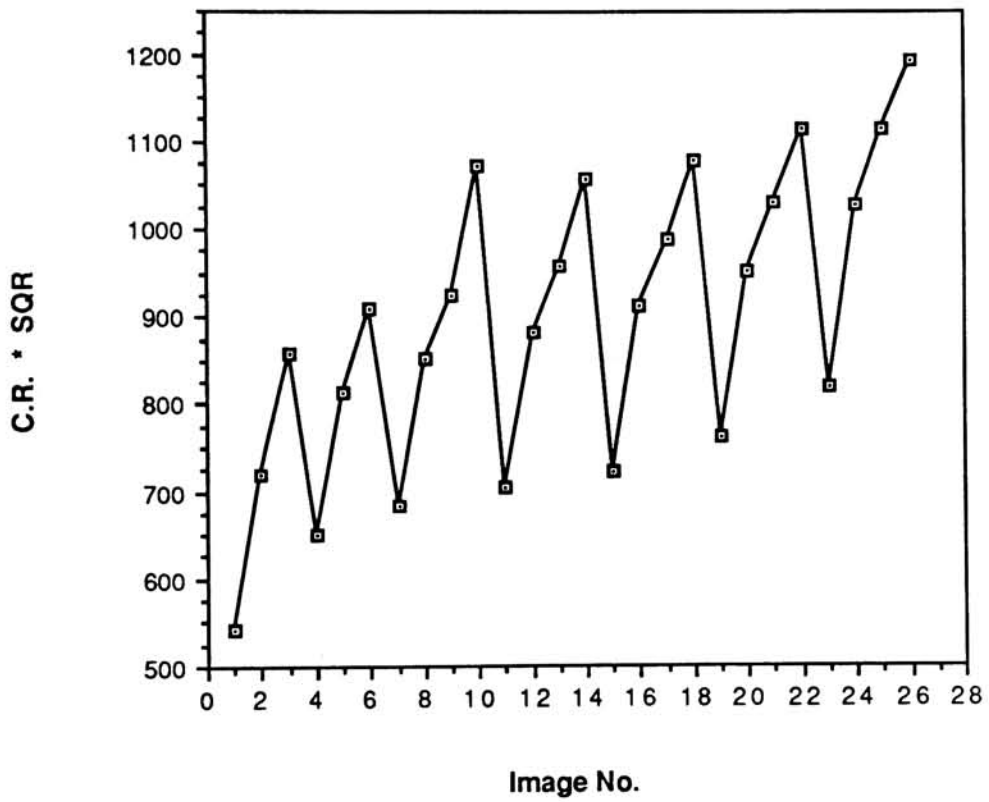
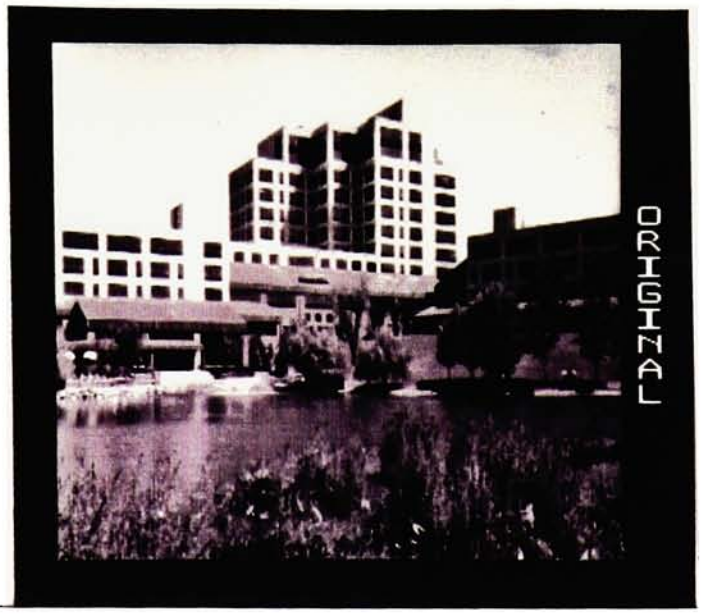


Figure 6.4

Overall performance of MBTC vs. BTC images



Building Picture

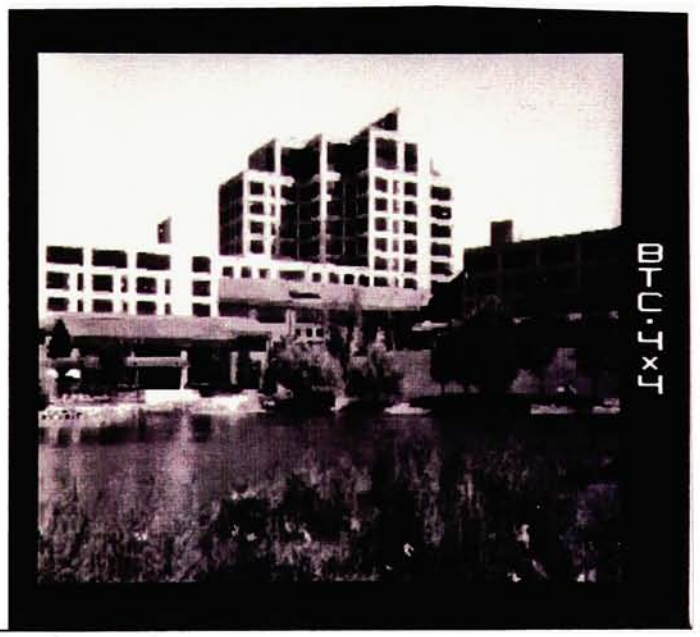


Image 1

Compression Ratio: 4.000

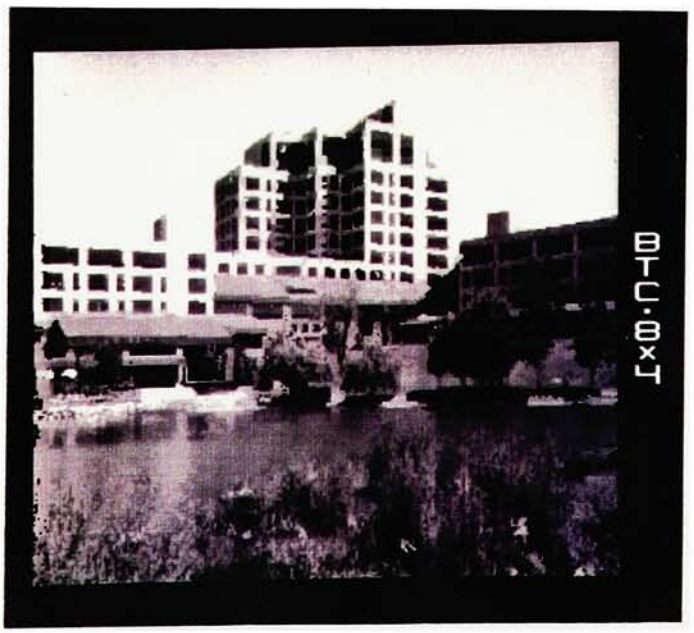


Image 2

Compression Ratio: 5.333

Image 3

Compression Ratio: 6.400

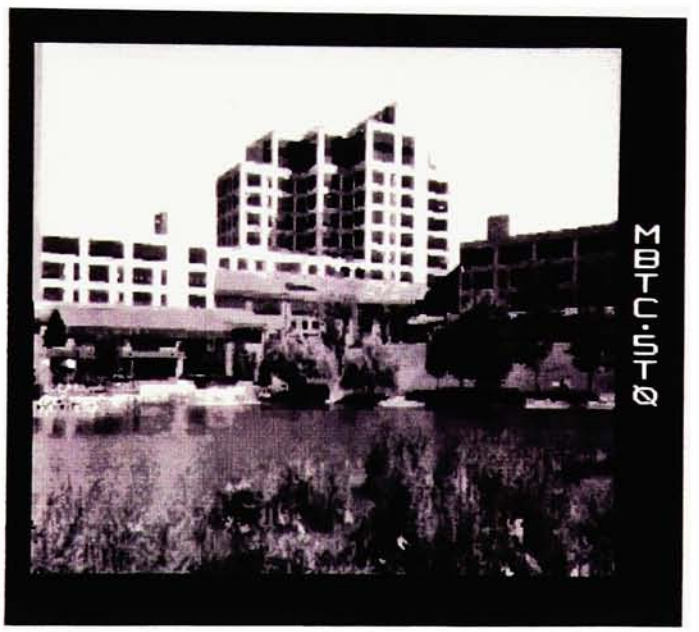


Image 4

Compression Ratio: 4.822

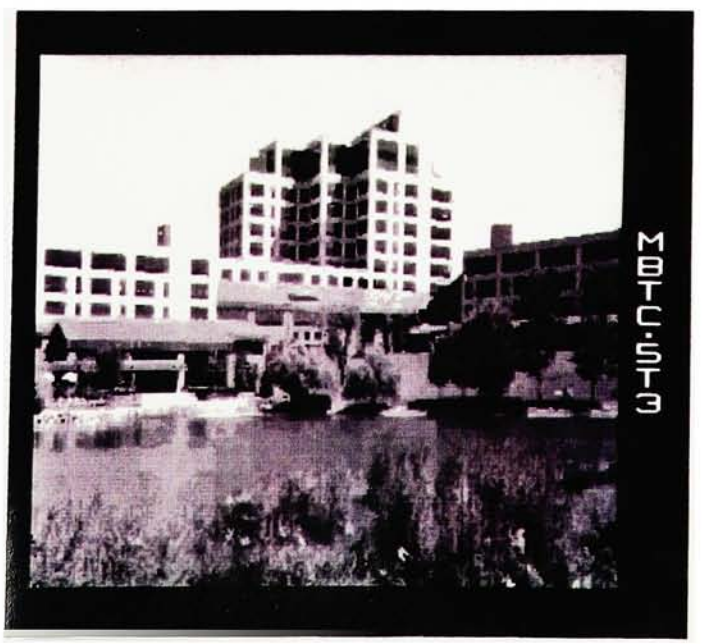


Image 5

Compression Ratio: 6.017

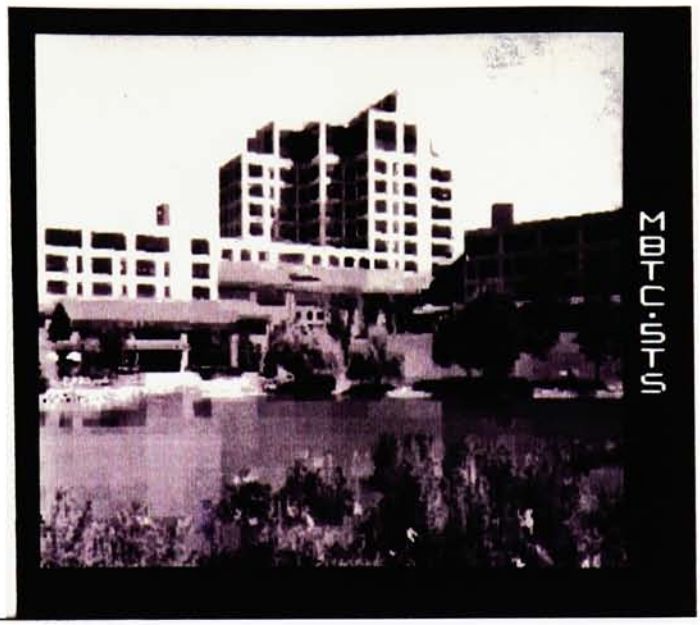


Image 6

Compression Ratio: 6.745

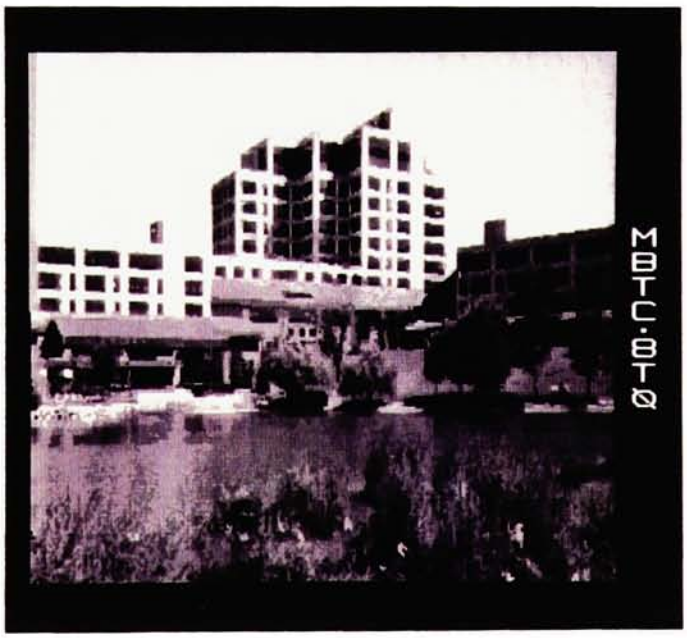


Image 7

Compression Ratio: 5.049

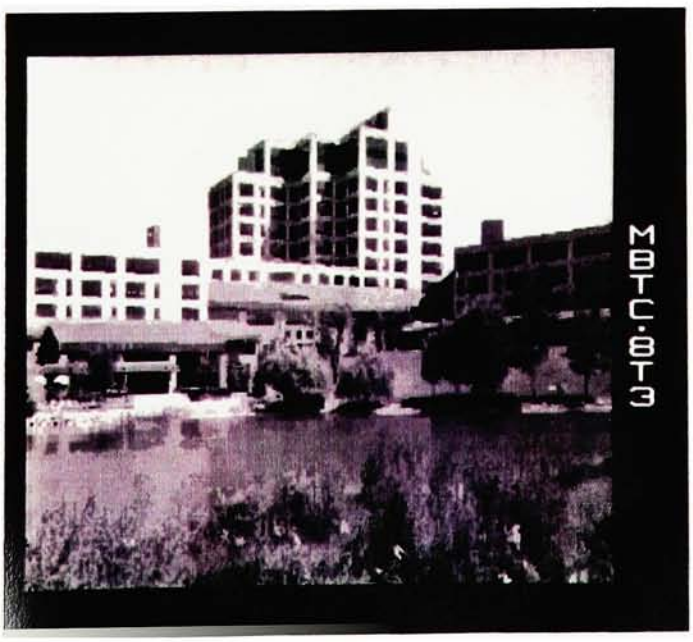


Image 8

Compression Ratio: 6.311

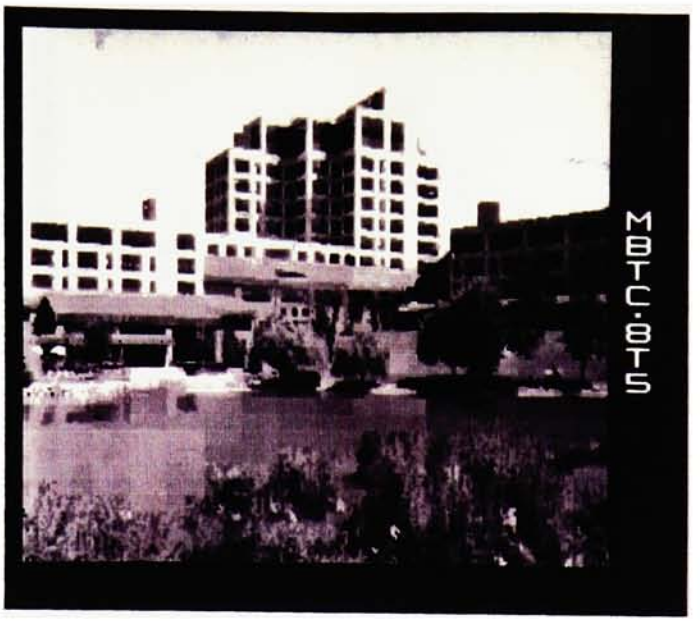


Image 9

Compression Ratio: 6.872

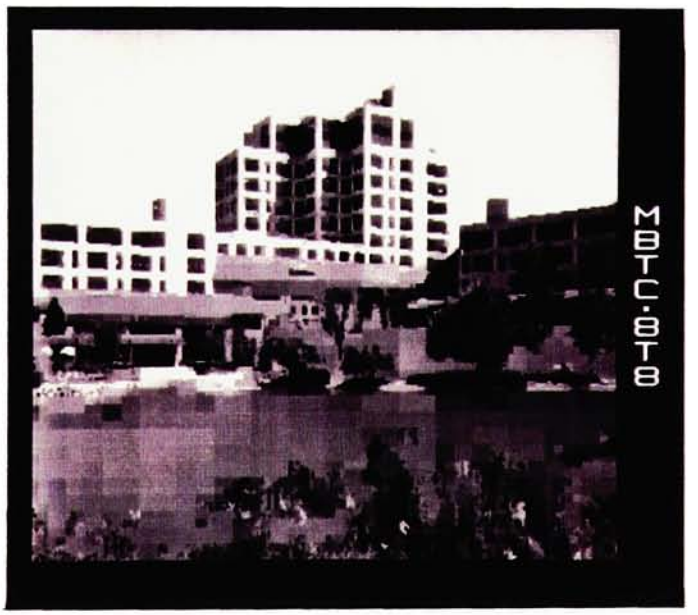


Image 10

Compression Ratio: 7.967

Image 11

Compression Ratio: 5.208

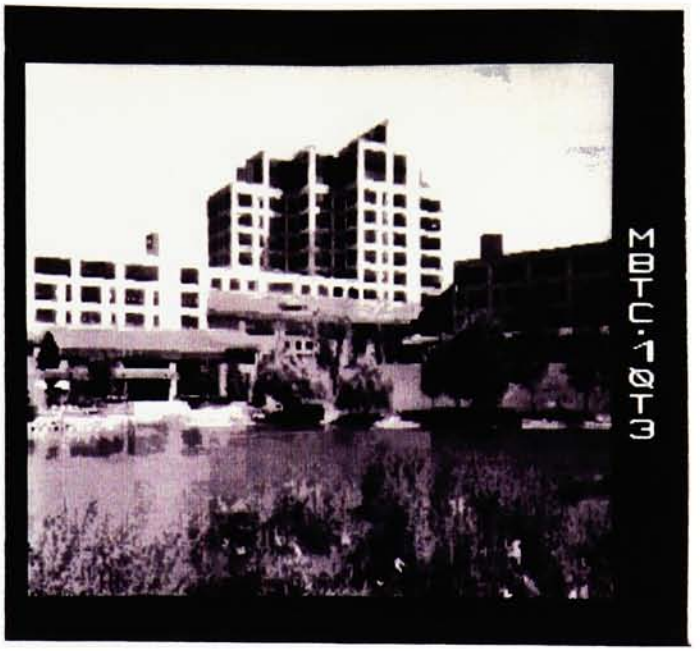


Image 12

Compression Ratio: 6.545

Image 13

Compression Ratio: 7.100

Image 14

Compression Ratio: 7.845

Image 15

Compression Ratio: 5.352

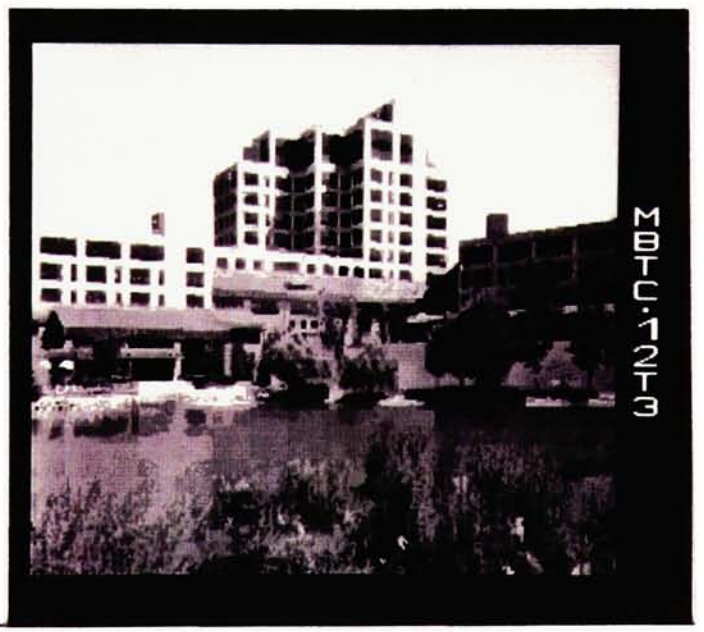


Image 16

Compression Ratio: 6.760

Image 17

Compression Ratio: 7.337

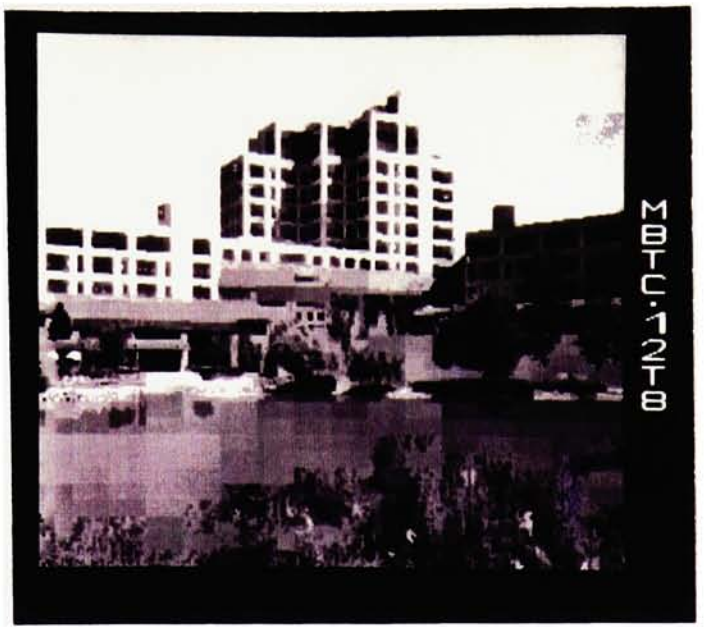


Image 18

Compression Ratio: 8.004

Image 19

Compression Ratio: 5.639

Image 20

Compression Ratio: 7.053

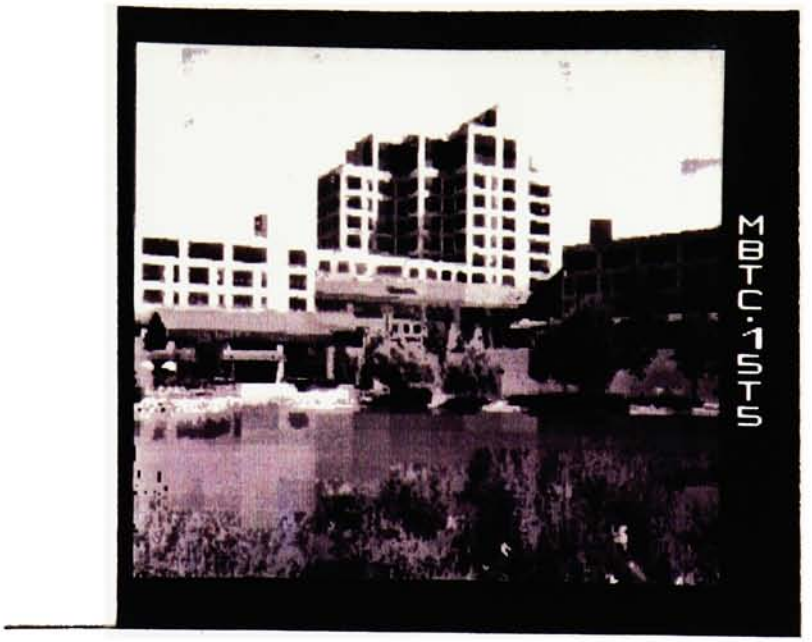


Image 21

Compression Ratio: 7.651

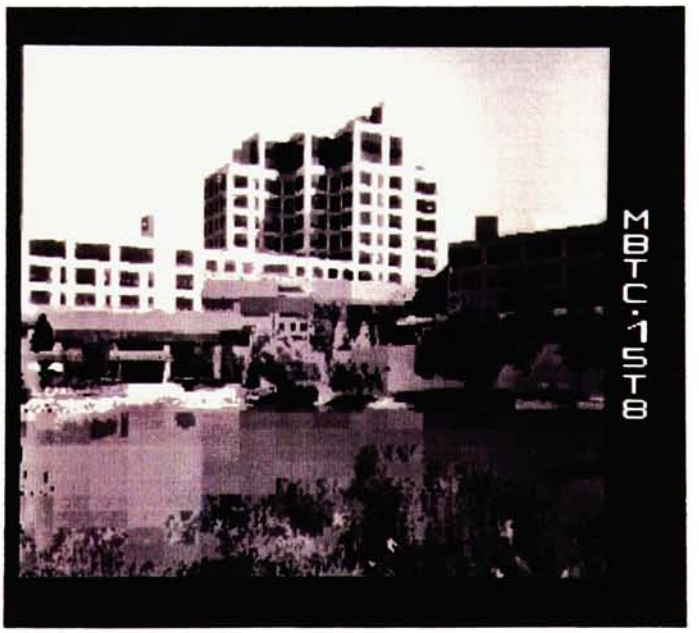


Image 22

Compression Ratio: 8.275

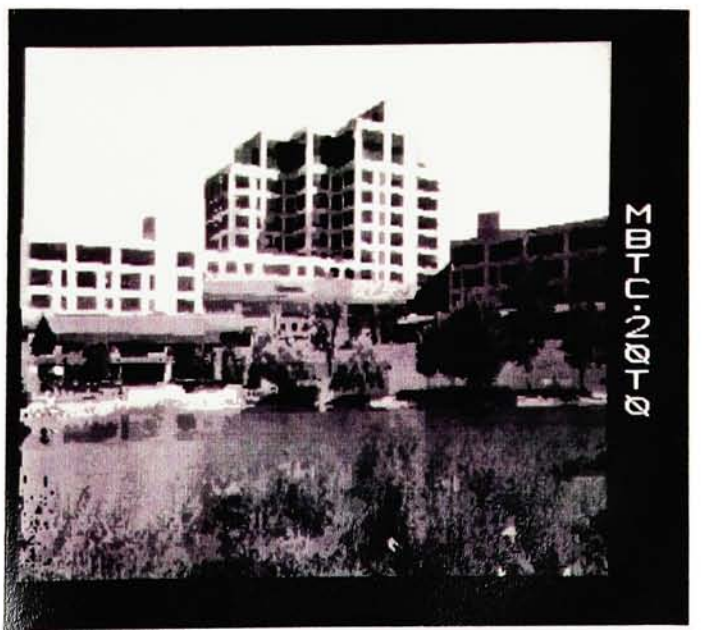


Image 23

Compression Ratio: 6.077

Image 25

Compression Ratio: 8.313

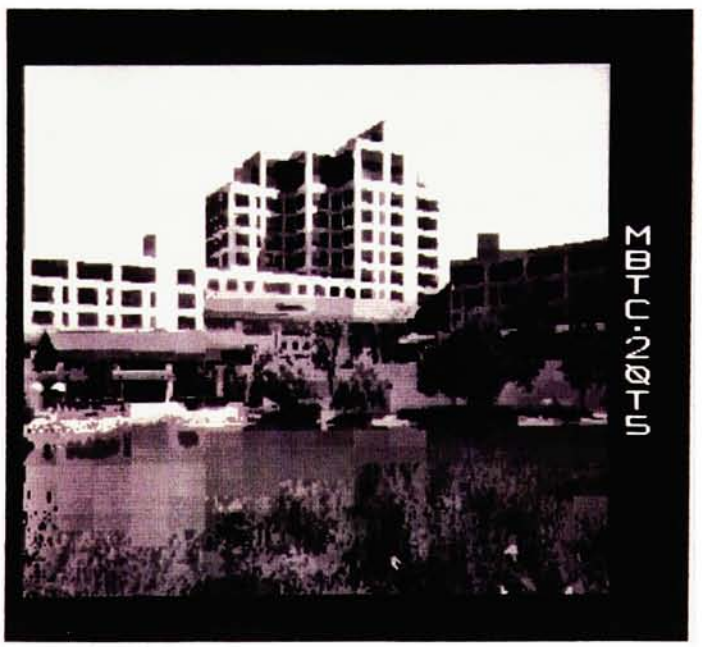


Image 24

Compression Ratio: 7.649

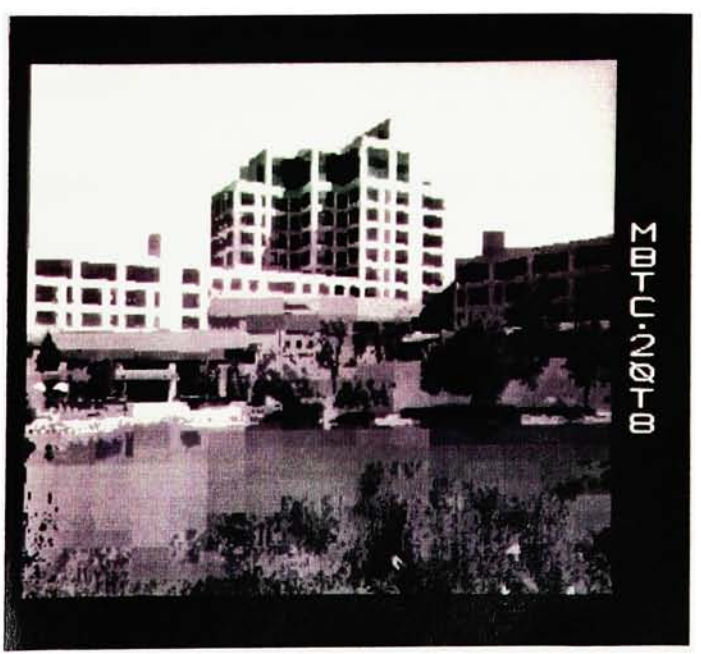


Image 26

Compression Ratio: 8.905

7. Conclusion

What has been discussed here are modifications to the standard Block Truncation Coding algorithm and a method to evaluate the visual image quality. The modifications involve varying the block size and conditionally eliminating the bit plane encoding while compressing the image. A guideline for selecting a threshold parameter is based on different perception dimensions of image. From that discussion, image noise was selected as the threshold parameter. The Power-Law Stimulus-Response model was used to calculate the Subjective Quality Response of reconstructed images. Comparing against the standard BTC algorithm, the modified BTC algorithm offers better compression efficiency with little degradation in visual image quality, as pictures at the end of chapter 6 indicate.

The following is a brief list of possible further work in this area:

1. Optimize the proposed algorithm for possible integrated circuit implementation. What has been done here was illustration of the concept feasibility. No optimization was attempted.

2. Modify the proposed concept to improve image quality rather than compression efficiency.

3. Implement the proposed concept with wider sets of values for different parameters, such as:

- Number of grey levels.
- Other block sizes.
- Other bit plane encodings...

4. Investigate other approaches to calculate the image noise during compression.

5. Modify the image evaluation model to improve the image quality factor. The numerical responses of the model discussed in this thesis, even though accurately predict image quality, do not provide enough differential resolutions.

6. Investigate the feasibility of applying the proposed concept to other types of data such as computer graphic, speech ...

7. Investigate the possibility of applying the proposed concept to color images.

List of References:

1. Ahmed, N., Natarajan, T. and Rao, K. R., " Discrete Cosine Transform", IEEE Trans. on Computers, pp. 90-93, Jan 1974.

2. Ahmed, N. and Rao, K. R., "Orthogonal Transform for Signal Processing", New York: Springer-Verlag, 1975.

3. Anderson, G. B. and Huang, T. S., "Piecewise Fourier Transformation for Picture Bandwidth Compression", IEEE Trans. on Communication Technology, Vol. COM-19, pp. 133-140, April, 1971

4. Arce, G. R. and Gallagher, N. C. Jr., "BTC Image Coding Using Median Filter Roots", IEEE Trans. on Communications, Vol. COM-31, No. 6, pp. 784-793, June 1983.

5. Arce, G. R. and Gallagher, N. C. Jr.,"State Description for the Root-Signal Set of Median Filters", IEEE Trans. on Speech, and Signal Processing, Vol. ASSP-30, No. 6, December 1982.

6. Campanella, S. J. and Robinson, G. S., "A Comparison of Orthogonal Transformations", IEEE Trans. on Communication Technology, Vol. COM19, pp. 1045-1050, Dec. 1971.

7. Delp, E. and Mitchell, O. R., "Image Compression Using Block Truncation Coding", IEEE Trans. on Communications, Vol. COM-27, No. 9, pp. 1335-1342, September 1979.

8. Eversole, W. L., Mayer, D. J., Frazee, F. B. and Cheek, T. F., "Investigation of VLSI Technologies for Image Processing", Proceedings: Image Understanding Workshop, pp. 191-195, Pittsburgh, PA, Nov. 14-15, 1978.

9. Granger, E. and Cupery, K., "An Optical Merit Function which Correlates with Subjective Image Judgments", Photo Science and Engineer, Vol. 16, pp. 221-230, 1972.

10. Hunt, B. R. and Sera, G.F., "Power-Law Stimulus-Response Models for Measures of Image Quality in Non-performance Environments", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-8, No. 11, pp. 781-791, November 1978.

11. Jain, A. K., "Advances in Mathematical Models for Image Processing", Proceedings of the IEEE, May 1981, pp. 502-528.

12. Jain, A. K., "Image Data Compression: A Review", Proceedings of the IEEE, Vol. 69, No. 3, pp. 349-389, March 1981.

13. Lema, M. D. and Mitchell, O. R., "Absolute Moment Block Truncation Coding and Its Applications to Color Images", IEEE Trans. on Communications, Vol. COM-32, pp. 1148-1157, 1984.

14. Marmolin, H. and Nyberg, S., "Multidimensional Scaling of Image Quality", FOA Rep. C-30039-H9, Swedish National Defense Research Institute, Stockholm, 1975.

15. Mitchell, O. R. and Delp, E., "Multilevel Graphics Representation Using Block Truncation Coding", Proc. IEEE, 68, pp. 868-873, 1980.

16. Mitchell, O. R., Delp, E. and Carlton, S. G., "Block Truncation Coding: A New Approach to Image Compression", Conference Record, 1978 IEEE International Conference on Communications (ICC '78), Vol. 1, June 4-7, 1978.

17. Murakami, S., Mitsuya, E., Mori, K., Kishimoto, T. and Kamae, T., "One Bit/Pel Coding of Still Pictures", Proc. ICC '79, (Boston), 23.1.1-23.1.5, 1979.

18. Pratt, W. K., Kane, J. and Andrews, H. C., " Hadamard Transform Image Coding", Proc. IEEE, Vol. 57, pp. 58-68, Jan 1969.

19. Pratt, W. K., Chen, W-H. and Welch, L. R., "Slant Transform Image Coding", IEEE Trans. on Communications, Vol. COM-22, pp. 1075-1093.

20. Pirenne, M. H., Vision and the Eye, Science Paperbacks, Chapman and Hall Ltd., London, 1948, 1967.

21. Robson, L. G., "Receptive Fields: Neural Representation of the Spatial and Intensive Attributes of the Visual Image", Handbook of Perception, Vol. 5.

22. Sakrison, D. J., "On the Role of the Observer and a Distortion Measure in Image Transmission", IEEE Transactions on Communications, Vol. COM-25, No. 11, pp. 1251-1267, November 1977.

23. Secord, N. and Wong, K. M., "Median Filtering for Improved Image Compression", IEEE Electronicom '85, Conference Proceedings, Vol. 1, pp. 190-193, 1985.

24. Stevens, S. S., "The Psychophysics of Sensory Functions", American Scientist, Vol. 48, pp. 226-235, 1960.

25. Wintz, P. A., "Transform Picture Coding", Proceedings of the IEEE, Vol. 60, No. 7, pp. 809-820, July 1972.

Appendix A

Listing of program: mbtc.c

```
/*
   Program: mbtc.c
   Date: October 10, 1988
   By: H. Pham
   Description:
   This program performs the modified Block Truncation
   Coding of an image.
   The program expects 3 input files:
          a. Original image file
          b. Compressed image file
          c. Reconstructed image file
   First, it asks the user to input the
   image size and two threshold values.  The threshold
   values represent the percent distortion.  The first
   threshold value determines the block size selection,
   i.e should the BTC algorithm be applied on a larger
   block.  The second threshold value determines if the
   bit plane of a particular block be transmitted. This
   value = 0 means that all the bit planes will be
   transmitted.
   Only four different block sizes are implemented:
   16 X 8, 8 X 8, 8 X 4, and 4 X 4.
   The block average value occupies one byte, 8 bits.
   The block alpha value occupies 6 bits.  The remaining
   2 bits are used to decode the block size.
   The program puts the compressed image in a temporary
   file, fp1a, and computes the compression ratio as well
   as block usage statistics.  Then, it will generate the
   reconstructed image and put in file fp2.
*/

#include <math.h>
#include <stdio.h>
#define mask1 0x01
#define mask2 0x80
#define mask3 0x03

struct block_info
     {
             int i;      /* index in the x, or horizontal,
                            direction  */
             int j;      /* index in the y, or vertical
                            direction  */
             int threshold1;    /* block size threshold  */
             int threshold2;    /* bit plane transmitting
                                   threshold  */
             int ratio;
                 /* block value which is used to compared
                    against threshold values  */
             float count168;    /* number of 16 X 8 blocks
                                   used */
             float count88;     /* number of  8 X 8 blocks
                                   used */
```

```
            float count84;     /* number of  8 X 4 blocks
                                  used */
            float count44;     /* number of  4 X 4 blocks
                                  used */

            unsigned char average;      /* block average */
            unsigned char alpha_flag;
              /* value of the alpha and block size flag  */
            unsigned char notdone;     /* notdone flag  */
            unsigned char flag;        /* block size flag */
            unsigned char buffer[9][259];
                         /* working buffer, equalto 9
                            lines of image  */
      };


main(argc, argv)
int argc;
char *argv[];
{


      FILE *fp1, *fp1a, *fp3, *fopen();

      unsigned char header, alpha;

      int i, j, k, m, n, num_line, xblocksize, yblocksize,
          nlines, xwidth, x_cor,  xnum168, ynum168 ;

      float data_in, data_out, compress_ratio, temp_real ;

      struct block_info info, process_block(),
                        expand_block();


fp1 = fopen(*++argv, "rb");
    /* open image file */
printf("The image file name is : %s \n",*argv);
fp1a = fopen(*++argv, "w+b");
printf("The output compressed file is : %s \n",*argv);
fp3 = fopen(*++argv, "wb");
printf("The output image file is : %s \n",*argv);

printf("Please specify the width of the input image \n");
scanf("%d", &xwidth);

printf("Please input the number of lines to be processed
\n");
scanf("%d", &num_line);

printf("Please input the desired first threshold \n");
scanf("%d", &info.threshold1);

printf("Please input the desired second threshold \n");
```

```
scanf("%d", &info.threshold2);

xblocksize = 16;
yblocksize = 8;
xnum168 = (xwidth / xblocksize) - 1;
ynum168 = num_line / yblocksize;


/* get file header of the image file , which is 18 bytes
long, and put at the beginning of the output image file  */
for (j = 1; j <= 18; j++)
     {
     header = getc(fp1);
     putc(header, fp3);
     }


/*********************************************/
/*                                           */
/*       Compress the image                  */
/*                                           */
/*********************************************/


printf("\n");
printf("The image is being compressed.  Please wait ...
\n");

info.count168 = info.count88 = info.count84 = info.count44 =
0;

for (k = 1; k <= ynum168; k++)
    {
/* read 8 lines of data, and store them at buffer array  */
    for (j = 1; j <= yblocksize; j++)
          for (i = 1; i <= xwidth; i++)
               info.buffer[j][i] = getc(fp1);

     for (n = 0; n <= xnum168; n++)
          {
          x_cor = n * xblocksize;
          info.i = 1;
          info.j = 1;
          info.flag = 3;
          info.notdone = 1;

          while ( info.notdone)
               info = process_block(x_cor, info, fp1a) ;
          }
     nlines = yblocksize * k;
     printf("%d lines have been compressed \r", nlines);
     }
```

```
 /*  calculate the compression ratio  */

temp_real = xwidth;
data_in = nlines * temp_real;
data_out = info.count44 * 4.0 + info.count84 * 6.0 +
             info.count88 * 10.0 + info.count168 * 18.0;
compress_ratio = data_in / data_out;

printf("\n");
printf("count168 = %f \n", info.count168);
printf("count88 = %f \n", info.count88);
printf("count84 = %f \n", info.count84);
printf("count44 = %f \n", info.count44);

printf("data_in = %f, data_out = %f, compress_ratio = %f
\n",
          data_in, data_out, compress_ratio);

rewind(fp1a);


/*****************************************************/
/*                                                   */
/*          Decompress the image                     */
/*                                                   */
/*****************************************************/

for (k = 1; k <= ynum168; k++)
     {
     for (n = 0; n <= xnum168; n++)
          {
          x_cor = n * xblocksize;
          info.notdone = 1;
          info.i = 1;
          info.j = 1;
          while (info.notdone)
               {
               info.average = getc(fp1a);
               info.alpha_flag = getc(fp1a);
               info.flag = info.alpha_flag & mask3;
               alpha = info.alpha_flag >> 2;

               info = expand_block(alpha, info, x_cor,
                                   fp1a);
               }
          }

     for (j = 1; j<= yblocksize; j++)
          for (i = 1; i <= xwidth; i++)
               putc(info.buffer[j][i], fp3);
     nlines = yblocksize * k;
     printf("%d lines have been decompressed \r", nlines);
     }
```

```
fclose(fp1);
fclose(fp1a);
fclose(fp3);

} /* end main */


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

int bitcount(n)      /* count 1 bits in n  */

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

/* Count number of 1's in n */

unsigned char n;
{
     int b;

    for (b = 0; n!= 0; n>>= 1)
     if (n & 01)
        b++;

    return(b);

}



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

struct block_info process_block(x_cor, info, fp1a)

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

/* This routine processes an image block.  It calculates
   the block average and alpha values, by calling the
   calc_block routine.  It determines if the block is ready
   for sending by comparing the block ratio against the
   value of threshold1.  If it is, it:
     - updates the coordinate of the next block to be
       processed by calling inc_index routine
     - sends the compressed data with or without the bit
       plane (depend on the value of the second threshold)
       by calling send_pixel routine
     - updates the block usage statistics
   If it is not, the block size flag is revised to reflect
   the next smaller block size
*/

int x_cor;
struct block_info info;
FILE *fp1a;
```

```
{
     int xsize, ysize,
          ratio, send_pixel();
     struct block_info calc_block(), inc_index();

     switch (info.flag)
     {
     case (3):
          xsize = 16;
          ysize = 8;
          info = calc_block(info, x_cor, xsize, ysize);
          if (info.ratio < info.threshold1)
             {
             send_pixel(info, xsize, ysize, x_cor, fp1a);
             info = inc_index(info, xsize, ysize);
             if (info.ratio < info.threshold2)
                 info.count168 = info.count168 + 0.111;
             else
                 info.count168 = info.count168 + 1;
             }
          else
             info.flag = 2;
        break;

     case (2):
          xsize = 8;
          ysize = 8;
          info = calc_block(info, x_cor, xsize, ysize);
          if (info.ratio < info.threshold1)
             {
             send_pixel(info, xsize, ysize, x_cor, fp1a);
             info = inc_index(info, xsize, ysize);
             if (info.ratio < info.threshold2)
                 info.count88 = info.count88 + 0.2;
             else
                 info.count88 = info.count88 + 1;
             }
          else
             info.flag = 1;
          break;

     case (1):
          xsize = 8;
          ysize = 4;
          info = calc_block(info, x_cor, xsize, ysize);
          if (info.ratio < info.threshold1)
             {
             send_pixel(info, xsize, ysize, x_cor, fp1a);
             info = inc_index(info, xsize, ysize);
             if (info.ratio < info.threshold2)
                 info.count84 = info.count84 + 0.3333;
             else
```

```
                info.count84 = info.count84 + 1;
            if (info.j == 1)
                info.flag = 2;
            }
        else
            info.flag = 0;
        break;

    case (0):
        xsize = 4;
        ysize = 4;
        info = calc_block(info, x_cor, xsize, ysize);
        send_pixel(info, xsize, ysize, x_cor, fp1a);
        if (info.ratio < info.threshold2)
                info.count44 = info.count44 + 0.5;
        else
            info.count44 = info.count44 + 1;
        info.i = info.i + 4;
        info = calc_block(info, x_cor, xsize, ysize);
        send_pixel(info, xsize, ysize, x_cor, fp1a);
        if (info.ratio < info.threshold2)
                info.count44 = info.count44 + 0.5;
        else
            info.count44 = info.count44 + 1;
        info = inc_index(info, xsize, ysize);
        if (info.j == 1)
            info.flag = 2;
        else
            info.flag = 1;
        break;
    }

    return (info);
}


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

struct block_info calc_block(info, x_cor, xsize, ysize)

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


/* This routine calculates the block average and
   alpha values.  It also calculates the ratio
   of the alpha over average (expressed in per cent)
*/

int x_cor, xsize, ysize ;
struct block_info info;

{
    int i, j, numpix, alpha1 ;
    float sum, val, temp, temp_float, alpha2;
```

```
     numpix = xsize * ysize;
     sum = 0;
     alpha1 = alpha2 = 0;

     for (j = info.j; j < (info.j + ysize); j++)
          for (i = info.i; i < (info.i + xsize); i++)
               sum = sum + info.buffer[j][i + x_cor];
     temp = sum / numpix;
     info.average = temp;
     for (j = info.j; j < (info.j + ysize); j++)
          for (i = info.i; i < (info.i + xsize); i++)
               {
               val = info.buffer[j][i + x_cor];
               if (val >= temp)
                    {
                    alpha1 = alpha1 + 1;
                    alpha2 = alpha2 + val;
                    }
               }
     temp_float = 2 * (alpha2 - alpha1 * temp) / numpix;
     info.alpha_flag = temp_float;
     if (info.average == 0)
        info.ratio = 0;
     else
        info.ratio = (info.alpha_flag * 100) / info.average;

     return(info);

}


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

struct block_info inc_index(info, xsize, ysize)

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

/* This routine calculates the coordinate of the next
   block to be processed based on the current block
   location and size
*/

struct block_info info;
int xsize, ysize;

{
if (xsize == 4)
   {
   info.i = info.i - 4;
   xsize = 8;
   }
info.j = info.j + ysize;
if (info.j > 8)
```

```
    {
    info.i = info.i + xsize;
    if (info.i > 16)
        info.notdone = 0;
    else
        info.j = info.j - 8;
    }
return(info);
}


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

int send_pixel(info, xsize, ysize, x_cor, fp1a)

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

/* This routine sends the block average and alpha values
   and the block size indicator (in two bytes).
   It will send the bit plane as well if the block ratio
   is >= the second threshold.
*/

struct block_info info;
int xsize, ysize, x_cor;
FILE *fp1a;
{
     int i, j, m, numpix, num_patt, index;
     unsigned char pattern[17];


     putc(info.average, fp1a);

     if (info.alpha_flag > 63)
        info.alpha_flag = 63;
     info.alpha_flag = (info.alpha_flag << 2) | info.flag ;

     putc(info.alpha_flag, fp1a);

     if (info.ratio >= info.threshold2)
        {
        numpix = xsize * ysize;
        num_patt = numpix >> 3;
        index = 0;
        for (j = info.j; j < (info.j + ysize); j++)
             for (i = info.i; i < (info.i + xsize); i++)
                  {
                  m = index >> 3;
                  index = index + 1;
                  pattern[m] = pattern[m] << 1;
                  if (info.buffer[j][i + x_cor] >=
info.average)
                       pattern[m] = pattern[m] | mask1;
                  }
```

```
        for (m = 0; m < num_patt; m++)
             putc(pattern[m], fp1a);
        }

}


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

struct block_info expand_block(alpha, info, x_cor, fp1a)

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

/* This routine reconstruct a block based on the average
   and alpha values as well as the block size indicator.
   If the caculated block ratio is >= the second threshold,
   it will read in the next block of data as the block bit
   plane by calling the get_pixel routine.  It also updates
   the index for the next to be decompressed block.
*/

unsigned char  alpha;
int x_cor;
struct block_info info;
FILE *fp1a;

{
     int xsize, ysize;
     struct block_info inc_index(), get_pixel();

     switch (info.flag)
     {
     case (3):
          xsize = 16;
          ysize = 8;
          info = get_pixel(alpha, info, x_cor, xsize, ysize,
                           fp1a);
          info = inc_index(info, xsize, ysize);
          break;

     case (2):
          xsize = 8;
          ysize = 8;
          info = get_pixel(alpha, info, x_cor, xsize, ysize,
                           fp1a);
          info = inc_index(info, xsize, ysize);
          break;

     case (1):
          xsize = 8;
          ysize = 4;
          info = get_pixel(alpha, info, x_cor, xsize, ysize,
                           fp1a);
          info = inc_index(info, xsize, ysize);
```

```
            break;

        case (0):
            xsize = 4;
            ysize = 4;
            info = get_pixel(alpha, info, x_cor, xsize, ysize,
                             fp1a);
            info.i = info.i + 4;
            info.average = getc(fp1a);
            info.alpha_flag = getc(fp1a);
            alpha = info.alpha_flag >> 2;
            info = get_pixel(alpha, info, x_cor, xsize, ysize,
                             fp1a);
            info = inc_index(info, xsize, ysize);
            break;
        }
        return(info);

}


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

struct block_info get_pixel(alpha, info, x_cor, xsize,
                            ysize, fp1a)

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

/* This routine does the actual reading of block data
   and reconstructs the block
*/

unsigned char  alpha;
int x_cor, xsize, ysize;
struct block_info info;
FILE *fp1a;

{
        int i, j, k, m, n, index, num_patt, ratio;
        unsigned char pattern[17], a, b, value;
        float q, numpix, temp2, temp3, temp4, temp5;

        ratio = (alpha * 100) / info.average;
        numpix = xsize * ysize;
        num_patt = numpix / 8 ;
        q = 0;
        for (m = 0; m < num_patt; m++)
             {
             if (ratio < info.threshold2)
                pattern[m] = 0;
             else
                {
                pattern[m] = getc(fp1a);
                q = bitcount(pattern[m]) + q;
```

```
            }
        }
    if ( (q == numpix) || (q == 0) || (alpha == 0) )
        a = b = info.average;
    else
        {
        temp5 = alpha;
        temp2 = (numpix * temp5) / ( 2 * (numpix - q));
        temp3 = info.average - temp2;
        if (temp3 < 0)
           temp3 = 0 - temp3;
        temp4 = info.average + (temp5 * numpix / (2 * q));
        a = temp3;
        b = temp4;
        }
        index = 0;
        for (j = info.j; j < (info.j + ysize); j++)
             for (i = info.i; i < (info.i + xsize); i++)
                  {
                  m = index >> 3;
                  index = index + 1;
                  value = pattern[m] & mask2;
                  pattern[m] = pattern[m] << 1;
                  if (value > 0)
                     info.buffer[j][i + x_cor] = b;
                  else
                     info.buffer[j][i + x_cor] = a;
                  }

    return(info);
}
```

Appendix B

Listing of program: btc.c

```
/* Program: btc.c
   Date: July 2, 1988
   By: H. Pham
   Description:
   This program performs the Block Truncation Coding (BTC)
   of an image.
   Program usage :
   btc <original image> <compress image file>
                                    <decompressed image>

   The user will be asked to input the size of the original
   image, and the block size.
*/

#include <math.h>
#include <stdio.h>
#define mask1  0X01
#define mask2  0X80
#define CR '\0x0d'

main(argc, argv)
int argc;
char *argv[];
{
     FILE *fp1, *fp1a, *fp3, *fopen();
     unsigned char header;
     unsigned char buffer[9][512], average, alpha,
                   pattern[17],value, a, b ;
     int i, j, k, m, n, o, num_line, xblocksize, yblocksize,
        nline, xwidth, numpix, xnum, ynum, num_patt, alpha1;
     float q, sum, temp, temp1, temp2, temp3 , temp4, temp5,
           val, alpha2, temp_float;

fp1 = fopen(*++argv, "rb");
    /* open image file */
printf("The image file name is : %s \n",*argv);
fp1a = fopen(*++argv, "w+b");
printf("The output compressed file is : %s \n",*argv);
fp3 = fopen(*++argv, "wb");
printf("The output image file is : %s \n",*argv);

printf("Please specify the width of the input image \n");
scanf("%d", &xwidth);
printf("Please input the number of lines to be processed
\n");
scanf("%d", &num_line);

printf("Please input the blocksize (X, Y) \n");
scanf("%d %d", &xblocksize, &yblocksize);

xnum = (xwidth / xblocksize) - 1 ;
ynum = num_line / yblocksize;
numpix = xblocksize * yblocksize;
num_patt = (numpix / 8) - 1;
```

```
/* get file header of the image file , which is 18 bytes
long, and put at the beginning of the output image file  */
for (j = 1; j <= 18; j++)
     {
     header = getc(fp1);
     putc(header, fp3);
     }

/*********************************************/
/*                                           */
/*       compress the image                  */
/*                                           */
/*********************************************/

printf("\n");
printf("The image is being compressed.  Please wait ...
\n");

for (k = 1; k <= ynum; k++)
    {

    for (j = 1; j <= yblocksize; j++)
     for (i = 1; i <= xwidth; i++)
         buffer[j][i] = getc(fp1);

    for (n = 0; n <= xnum; n ++)
        {
          sum = 0;
          alpha1 = 0;
          alpha2 = 0;
          for (m = 0; m <= num_patt; m++)
               pattern[m] =  0;

          for (j = 1; j <= yblocksize; j++)
               for (i = 1; i <= xblocksize; i++)
                    sum = buffer[j][n * xblocksize + i] +
                          sum;

          temp = sum / numpix;
          average = temp;
          o = 0;
          for (j = 1; j <= yblocksize; j++)
               for (i = 1; i <= xblocksize; i++)
                    {
                    m = o / 8;
                    o = o + 1;
                    pattern[m] = pattern[m] << 1;
                    val = buffer[j][n*xblocksize+i];
                    if ( val >= temp )
                         {
                         pattern[m] = pattern[m] | mask1;
                         alpha1 = alpha1 + 1;
                         alpha2 = alpha2 + val;
```

```
                        }
                    }
            temp_float = 2 * (alpha2 - alpha1 * temp)/numpix;
            alpha = temp_float;
            for (m = 0; m <= num_patt; m ++)
            putc(pattern[m], fp1a);

            putc(average, fp1a);
            putc(alpha, fp1a);
            }
        nline = yblocksize * k ;
        printf(" %d lines have been compressed. \r", nline);
       }

rewind(fp1a);

/*******************************/
/*                             */
/* decompress the input file   */
/*                             */
/*******************************/

printf("\n");
printf("The image is now being decompressed.  Please wait
... \n");

for (k = 1; k <= ynum; k++)
   {
    for (n = 0; n <= xnum; n ++)
      {
      q = 0.0;
      for (m = 0; m <= num_patt; m ++)
           {
           pattern[m] = getc(fp1a);
           q = bitcount(pattern[m]) + q;
           }
      average = getc(fp1a);
      alpha= getc(fp1a);

      if ( ( q == numpix) || (q == 0) || (alpha ==0) )
         a = b = average;
      else
         {
         temp5 = alpha;
         temp1 = numpix ;
         temp2 = (temp1 * temp5) / (2 * (temp1 - q));
         temp3 = average - temp2 ;
```

```
            if (temp3 < 0)
                  temp3 = 0 - temp3;

            temp4 = average + (temp5 * temp1 / (2 * q));

            a = temp3;
            b = temp4;
             }
          o = 0;
          for (j = 1; j <= yblocksize; j++)
          for (i = 1; i <= xblocksize; i++)
              {
              m = o / 8;
              o = o + 1;
              value = pattern[m] & mask2;
              pattern[m] = pattern[m] << 1;
              if (value > 0)
                 buffer[j][n*xblocksize+i] = b;
              else
                 buffer[j][n*xblocksize+i] = a;
              }
          }
          for (j = 1; j <= yblocksize; j++)
               for (i= 1; i <= xwidth; i++)
                    putc(buffer[j][i], fp3);
          nline = yblocksize * k ;
          printf(" %d lines have been decompressed. \r", nline);
      }

fclose(fp1);
fclose(fp1a);
fclose(fp3);
}


/* This routine counts the number of 1's in n  */

int bitcount(n)      /* count 1 bits in n  */

unsigned char n;
{
     int b;

    for (b = 0; n!= 0; n>>= 1)
     if (n & 01)
        b++;

    return(b);

}
```

Appendix C

Listing of program: convert.c

```
/* Program: convert.c
   Date: August 21, 1988
   Description:
   This program converts the image data, which is in the
   reflectance domain, into the lightness domain.
   At first, a conversion table for 256 possible
   pixels values is calculated.
   Then, the image is read in, its header is discarded,
   and the pixel value, one by one, will be convert by
   using the pre-constructed look up table.
   The outputs are written to a file with the MatLab format
*/

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

typedef struct {
      long type;   /* type */
      long mrows;  /* row dimension */
      long ncols;  /* column dimension */
      long imagf;  /* flag indicating imag part */
      long namlen; /* name length (including NULL) */
} Fmatrix;

main(argc, argv)
int argc;
char *argv[];
{
     FILE *fp1, *fp2, *fopen();
     unsigned char value;
     int i, j, xwidth, ylength, num,
          datatype, imagf, datasize, sign;
     float  A, B, alpha, temp;
     double imagedata[513], L[256] ;


fp1 = fopen(*++argv, "rb");
    /* open image file */
printf("The image file name is : %s \n",*argv);

fp2 = fopen(*++argv, "wb");
     /* open the translated lightness file */
printf("The file name of the converted image is : %s
\n",*argv);

printf("Please specify the width of the image \n");
scanf("%d", &xwidth);

printf("Please input the number of lines to be processed
\n");
scanf("%d", &ylength);

imagf = 0; /* no imaginary data */
datatype = 0; /*IBM PC data */
```

```
savemathd(fp2, datatype, "image", xwidth, ylength,
          imagf, imagedata, (double *)0);
  /* image is the name of the matrix inside MatLab  */

/* generate the translation table based on the lightness
equation:
            L = A(100R + B)^alpha - 16
*/
A = 24.98;
B = 0.2631;
alpha = 0.333333;

/* generate the look up table  */
for (i=0; i <=255; i++)
     {
     temp = i / 255.0;
     L[i] = ( A * pow( (100.0 * temp + B), alpha) ) - 16;
     }

/* get file header, which is 18 bytes long, and discard it
*/
for (j = 1; j <= 18; j++)
     {
     value = getc(fp1);
     }
imagf = 0;
datasize = sizeof(double);

/* convert the pixel value and write to an output file
   using file MatLab format  */
for (j = 0; j < ylength; j++)
     {
     sign = pow( (-1), j);
     for (i = 0; i < xwidth; i++)
         {
          imagedata[i] = L[getc(fp1)] * sign;
          sign = - sign;
         }
     fwrite(imagedata, datasize, xwidth, fp2);
     }


fclose(fp1);
fclose(fp2);

}
```

```
/*********************************************************/

/* This subroutine saves the header of MatLab data file.
   This routine is a modified version of the one supplied
   by MatLab : savemat.c  */

savemathd(fp, type, pname, mrows, ncols, imagf, preal,
          pimag)

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

FILE *fp;         /* File pointer */
int type;         /* Type flag: Normally 0 for PC, 1000 for
                     Sun, Mac, and    */
                  /* Apollo, 2000 for VAX D-float, 3000 for
                     VAX G-float    */
                  /* Add 1 for text variables.  */
                  /* See LOAD in reference section of guide
                     for more info. */
int mrows;        /* row dimension */
int ncols;        /* column dimension */
int imagf;        /* imaginary flag */
char *pname;      /* pointer to matrix name */
double *preal;    /* pointer to real data */
double *pimag;    /* pointer to imag data */
{
     Fmatrix x;
     int mn;

     x.type = type;
     x.mrows = mrows;
     x.ncols = ncols;
     x.imagf = imagf;
     x.namlen = strlen(pname) + 1;
     mn = x.mrows * x.ncols;

     fwrite(&x, sizeof(Fmatrix), 1, fp);
     fwrite(pname, sizeof(char), (int)x.namlen, fp);
}
```

Appendix D

Listing of program: mtfcal.c

```
/* Program : mtfcal.c
   Date : August 22, 1988
   By : H. Pham
   Description:
   This program generates a matrix of Modulation Transfer
   Function coefficients of human visual system, based on
   the image size.  The matrix size is 256 X 256.  The
   output file format is MatLAb compatible.
*/

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

typedef struct {
      long type;   /* type */
      long mrows;  /* row dimension */
      long ncols;  /* column dimension */
      long imagf;  /* flag indicating imag part */
      long namlen; /* name length (including NULL) */
} Fmatrix;

main(argc, argv)
int argc;
char *argv[];
{
     FILE *fp1, *fopen();
     int xwidth, yheight, k, datasize, imagf, datatype,
         num_write;

     float xsize, ysize, i, j, distance, imagesize, temp,
           conver_factor, polyval();
     double mtfcoeff[513] ;


fp1 = fopen(*++argv, "wb");
    /* open image file */
printf("The MTF coefficient file name is : %s \n",*argv);

printf("Please specify the width of the image \n");
scanf("%d", &xwidth);

printf("Please specify the height of the image \n");
scanf("%d", &yheight);

printf("Please specify the size of the image, in mm \n");
scanf("%f", &imagesize);

imagf = 0;  /* no imaginary data */
datatype = 0; /* IBM PC type data  */

savemathd(fp1, datatype, "mtfcoeff", xwidth, yheight,
          imagf, mtfcoeff, (double *)0);
  /* mtfcoeff is the name of the matrix inside MatLab  */
```

```
/* the unit of conver_factor is cycle per mm  */
conver_factor = xwidth / (2.0 * imagesize);

xsize = (xwidth - 1) / 2.0 ;
ysize = (yheight - 1) / 2.0 ;

datasize = sizeof(double);

for (j = -ysize; j <= ysize; j=j+1.0)
     {
     k = 0;
     for(i = -xsize; i <= xsize; i=i+1.0)
        {
        distance = (sqrt(i*i + j*j)) * conver_factor;
        temp = log10(distance);
        mtfcoeff[k] = polyval(temp);
        k++;
        }
     fwrite(mtfcoeff, datasize, xwidth, fp1);

     }
fclose(fp1);
}

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

float polyval(value)

/* This routine calculates the MTF response of a pixel based
   on its location with respect to the center of the image.
   The MTF response is a 10th order polynomial, fitted from
   the published curve.  */

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

float value;
{
     float result;

     result = 0.255508076 - (1.97074731 * value)
                + (29.81228475 * value * value)
                - (97.71690434 * pow(value,3.0))
                + (163.80902908 * pow(value,4.0))
                - (164.49711170 * pow(value,5.0))
                + (104.27738152 * pow(value,6.0))
                - (42.09313230 * pow(value,7.0))
                + (10.49801802 * pow(value,8.0))
                - (1.47481432 * pow(value,9.0))
                + (0.08926088 * pow(value,10.0));

     return(result);

}
```

```
/*************************************************************/

savemathd(fp, type, pname, mrows, ncols, imagf, preal,
          pimag)

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

/*
 savemat - C language routine to save a matrix in a
           MATLAB file.
*/

FILE *fp;         /* File pointer */
int type;         /* Type flag: Normally 0 for PC, 1000 for
                     Sun, Mac, and    */
                  /* Apollo, 2000 for VAX D-float, 3000 for
                     VAX G-float     */
                  /* Add 1 for text variables.  */
                  /* See LOAD in reference section of guide
                     for more info. */
int mrows;        /* row dimension */
int ncols;        /* column dimension */
int imagf;        /* imaginary flag */
char *pname;      /* pointer to matrix name */
double *preal;    /* pointer to real data */
double *pimag;    /* pointer to imag data */
{
     Fmatrix x;
     int mn;

     x.type = type;
     x.mrows = mrows;
     x.ncols = ncols;
     x.imagf = imagf;
     x.namlen = strlen(pname) + 1;
     mn = x.mrows * x.ncols;

     fwrite(&x, sizeof(Fmatrix), 1, fp);
     fwrite(pname, sizeof(char), (int)x.namlen, fp);
}
```

Appendix E

Listing of program: snr.c

```
/* Program: snr.c
   Date: October 10, 1988
   Description:
   This program calculates the signal to noise ratio, SNR,
   of a recontructed image with respect to the original
   image.  The two input files are the original image and
   the reconstructed image.
*/

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

main(argc, argv)
int argc;
char *argv[];
{
     FILE *fp1, *fp2, *fopen();
     int i, j;
     long  xwidth, num_lines ;
     float val1, val2, val3, temp_val3,
           num_pix, snr;

fp1 = fopen(*++argv, "rb");
     /* open original image file */
printf("The original image file name is : %s \n",*argv);
fp2 = fopen(*++argv, "rb");
printf("The reconstructed image file is : %s \n",*argv);

        /*
             input the size of the image
        */
printf("Please specify the width of the image\n");
scanf("%ld", &xwidth);
printf("Please specify the height of the image\n");
scanf("%ld", &num_lines);

/*
   get file header of both files, which is 18 bytes long
   and discard them.
*/
 for (j = 1; j <= 18; j++)
     {
     getc(fp1);
     getc(fp2);
     }

 /* temp_val3 stores the sum of the individual SNRs */
 temp_val3 = 0;

 for (j = 1; j <= num_lines; j++)
      for (i = 1; i <= xwidth; i++)
          {
           val1 = getc(fp1);
           val2 = getc(fp2);
```

```
            val3 =  (val1 - val2);
            if (val3 < 0)
                val3 = 0 - val3;
            if (val3 == 0)
                val3 = val1;
            else
                val3 = (val1 ) / val3;
            temp_val3 = temp_val3 + (val3 );
            }

temp_val3 = temp_val3 / (num_lines * xwidth);
snr = 10 * log10 (temp_val3) ;
printf("The SNR is : %f \n", snr);
fclose(fp1);
fclose(fp2);
}
```

Appendix F

Listing of program: stimulus.m

```
% Program stimulus.m
% Date: August 23, 1988
% Description:
% This file contains MatLab instructions to calculate the
% stimulus of an image.  This image was previous converted
% to the lightness domain and shifted so that the low
% frequently components are at the center of the image.  The
% name of the image matrix is: image.
% The image matrix is 256 x 256.
% The image matrix has to be transposed to have proper
% orientation.

image = image';

% Next, we perform the 2D-FFT on the transposed image.

image = fft2(image);

% Next, we find the magnitude of all the components.

image = abs(image);

s = size(image);
stimulus = 0;

for i = 1:s(1)
    for j = 1:s(2)
        stimulus = stimulus + image(i,j) * mtfcoeff(i,j);
    end
end

stimulus
```