FRACTAL IMAGE COMPRESSION

[illegible]

Dissertation Submitted to
JAWAHARLAL NEHRU UNIVERSITY
in partial fulfilment of requirements
for the award of the degree of
Master of Technology
in
Computer Science

by
J. Vijaya Kumar

Jawaharlal Nehru University

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067
January 1997

CERTIFICATE

This is to certify that the dissertation entitled

FRACTAL IMAGE COMPRESSION

Which is being submitted by **Mr. J. Vijaya Kumar** to the **School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi** for the award of **Master of Technology** in **Computer Science**, is a record of bonafide work carried out by him under the supervision and guidance of **Dr. S. Balasundram**.

This work is original and has not been submitted in part or full to any University or Institution for the award of any degree.

Vijay.
(J. VIJAYA KUMAR)

[illegible]
6/1/97

[illegible]

Prof. G.V. Singh
(Dean SC & SS)

S. Balasundram
(Supervisor)

ACKNOWLEDGEMENT

I wish to convey my heartfelt gratitude and sincere acknowledgements to my guide **Dr. S. Balasundram,** School of Computer & System Sciences for his whole hearted, tireless and relentless effort in helping me for the successful completion of the project.

I would like to record my sincere thanks to my Dean, **Prof. G.V.Singh,** School of Computer & System Sciences for providing the necessary facilities in the centre for the successful completion of the project.

I take this opportunity to thank all my faculty members and friends for their critical comments during the course of the project.

J. Vijaya Kumar

CONTENTS

1.*INTRODUCTION*

1.1 What is Image Compression:

Image compression maps an original image into a bit stream suitable for communication over or storage in a digital medium. The number of bits required to represent the coded image should be smaller than that required for the original image so that one can use less storage space or communication time. Image compression reduces redundancy in image data in order to store or transmit only a minimal number of bits per sample from which a good approximation of the original image can be constructed in accordance with human visual perception.

A single 800 by 600 pixel true color image requires 1.44 MB of storage space and an uncompressed 10-sec video clip with 30 frames per second of 320 by 200 pixels in true color requires an enormous 57.6 MB of disk space. Now a days multimedia applications and video conferencing are most common things in daily life. With these applications we must be able to store many images and videos. So there is an increasing need for a good compression method for storing images.

1.2 Image Compression Methods:

There are two basic types of compression techniques. *Lossless* compression, which is also called noiseless coding, data compression, entropy coding, or invertible coding, refers to algorithms that allow the original pixel intensities to be perfectly recovered from the compressed representation. The most common *lossless* compression method is PKWare's PKZip. These type of compression techniques are not much popular, since they can't achieve high compression ratios. The other type of compression, *Lossy* compression techniques are most popular because they can achieve more compression ratios. They basically tries to reduce redundancy in image data in order to store only a minimal number of bits per sample from which a good approximation of the original image can be constructed in accordance with human visual perception.

There are many lossy compression techniques which can achieve good compression ratios. One usually distinguish between different coding schemes: transform coding[2]; multi resolution coding; vector quantization[1]; predictive methods; and other more recent schemes such as fractal image coding. Generally, bit rates around 0.8-0.4 b/pixel are reported for still monochrome images without a great loss of visual quality. Higher ratios can be obtained with hybrid coders incorporating different techniques with respect to local image properties.

A general system for digital image compression is shown in figure 1(a). It consists of one or more of the following operations, which may be combined each other or with additional signal processing[1].

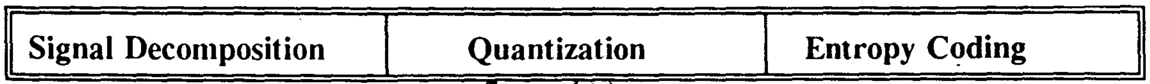


figure 1(a)

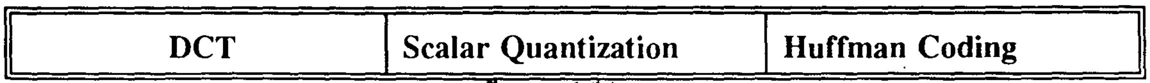


figure 1(b)

Subband Coding	**Vector Quantization**	**Lossless Coding**

figure 1(c)

- *Signal Decomposition:* The image is decomposed into several images for separate processing, typically by linear transformation by a Fourier or discrete cosine transform or by filtering with a subband or wavelet filter bank. The goal to concentrate energy in a few coefficients, to reduce correlation, or to provide a useful data structure.
- *Quantization:* High-rate digital pixel intensities are mapped into a relatively small number of symbols. This operation is nonlinear and noninvertible; it is "lossy". The conversion can operate on individual pixels (scalar quantization (SQ)) or groups of pixels (Vector quantization (VQ)) quantization can include throwing away some of the components of the signal decomposition step.
- *Lossless Compression*: Further compression is achieved by an invertible (Lossless, entropy code) such as Huffman, Lempul-Ziv, or arithmetic code. The idea here is to assign code words with a few bits to likely symbols and codewords with more bits to unlikely symbols so that the average number of bits is minimized.

A bewildering variety of image compression systems have proposed, which involves various choices for each of the true basic components. The JPEG[25] still-image compression, for example, uses a discrete cosine transform for the first step. SQ with different quantizer step sizes for the different transform coefficients in the second step, and run-length coding combined with Huffman coding for the third step, as shown in figure 1(b).

DCT breaks an image into 8-by-8-pixel blocks and then uses mathematical tricks to decide what image information can be thrown away without damaging the appearance of image too much. DCT transforms the image data in the 8-by-8 block mathematically from x,y space into frequency space. DCT views the data as a varying signal that can be approximated by a collection of 64 cosine functions with appropriate amplitude. Each cosine that DCT uses as a basis function is associated with a value called its DCT coefficient, which determines each cosine function's amplitude.

Most of the important visual information for typical continuous-tone images is concentrated in the cosine function with lower frequencies. Thus, by giving less weight to higher-frequency cosines and approximating small DCT coefficients zeros, compression can be achieved without too much image degradation. Further space saving are possible if we quantize the remaining DCT coefficients to a predefined set of values. With JPEG/DCT, the algorithm is symmetrical; compression and decompression takes roughly the same amount of time.

The other types of compression systems use a subband or wavelet decomposition for the first step, some form of VQ for the second step, and any lossless coding for the third step, as shown in figure 1(c).

With subband coding, the image is decomposed into a finite set of subimages, which are usually encoded separately using different codebooks at different rates [2]. These codes are thus recombined to restore the whole image. This class of methods regroups subband coding where each sub image lies in a specific frequency hand. In vector quantization (VQ), sequences of pixels are encoded rather than each pixel separately. According to Shannan's rate-distortion theory, VQ should perform better than scalar quantization because it takes the samples correlations different types of samples including DCT or wavelet coefficient [2], [3].

The most popular compression technique from past one decade, Fractal Image compression is based on concepts of iterated function systems (IFS) [4]. It can be seen as a kind of vector quantization and multiresolution coding. The method consists of partitioning blocks (vectors) and approximating each vector by a transformed code book block derived from the image itself [5]. Each transform, described by a linear term and a translation term, maps a block onto another block with a different resolution and composes the coded information.

1.3 Fractal Image Coding:

A number of papers on fractal image compression have been proposed by researchers since the original idea of Barnsley and Sloan in 1988 [4], implemented in a fully automated algorithm by Jacquin in 1989 [5]. Barnsley showed that it is possible to model a self-similar image as attractor of an IFS, with the help of the collage theorem[4]. Such as image is encoded with the very limited number of coefficients of the contractive mappings defining the IFS. The nonautomatic encoding algorithm he proposed consists of segmenting the image into self-similar objects. Each object, defining a part of the image, is covered by a set of contractive affine transformation of itself, with the property that the resulting union approximates the object. Each part of the image is coded by the coefficients of its associated set of transformation. The decoded image is formed by the association of the attractor of each IFS.

The encoding algorithm of Jacquin is fully automated because it does not require an "intelligent" segmentation of the image into distinct objects. The method is based on the observation that a real-world image is "affine-redundant". If exploits the partial self-similarity of the image.

The first step of this algorithm is to partition the image support into nonoverlapping square domain blocks and larger square range blocks. Given a domain block, a range block is searched such that it provides the best affine mapping to the domain block in the root mean squared error (MSE) sense. The encoder attempts to find, for a given image, the set of transformation under which the distortion between the original image and the union of the transformed range blocks (mapped into the domain blocks) is minimal. In this paper[6], Jacquin proposed improvements by the use of a classification scheme and a split of range blocks to adopt the mappings to the local properties of the image. Since the publication of this original work, a number of research areas have been investigated. They are summarized [7][8] and [9] as follows:

- Construction of the range and domain block partition (different block shapes, different mappings)
- Quantization of affine transformation parameters.
- Improvement of the encoding step - complexity reduction, introduction of fixed basic blocks, orthogonalization of the blocks.
- Decoding step - standard iteration, noniterative decoding, hierarchical decoding.
- Theoretical studies on the convergence to the attractor image and on the college theorem.

2. *HISTORY OF FRACTAL IMAGE COMPRESSION*

Fractal image compression is based on the concepts and mathematical results of iterated function systems (IFS). The roots of this theory are atleast 10-20 years old. In the mid 1980's , IFS's became very popular. It was Barnsley and his coworkers at Georgia Institute of Technology who first noticed the potential of IFS for applications in computer graphics. Initially, around 1985, their research focused on modeling natural shapes such as leaves and clouds but then Barnsley and Sloan advertised in popular science magazines the incredible power of IFS for compressing color images at rates of our 10,000 to 1 [10], [11]. Company, Iterated System, Inc., devoted to application of iterated systems, especially fractal image compression. Their algorithm is basically image dependent partitioning of the image into self-similar parts and IFS coding of each part separately. Several researchers have taken up the challenge to design an automated algorithm to solve the inverse (i.e the encoding) problem using the basic IFS method and its generalizations (recurrent iterated function systems, RIFS).

In 1989 Jacquin proposed the first fully automated algorithm for fractal image compression. It was based on affine transformations acting locally rather than globally. This new approach first appeared in his Ph.D thesis [5] and since then several papers [12], [13] have popularized his scheme.

A digital monochrome image is partitioned into nonoverlapping square pixel blocks (domain blocks). Larger square pixel blocks (range blocks) which may overlap are sorted into a set of categories (shade blocks, edge blocks and midrange blocks) following

a classification, well-known in image processing. For each domain block, a range block of the same category is searched (for evident complexity reduction purposes) such that its image under a local strictly contractive affine mapping minimizes its distance to the original block in the root mean squares metric. Each affine mapping is composed of a geometric part which shrinks the range block down to the size of a domain block by shuffling (8 alternatives corresponding to the isometry group of the square), scaling with quantized parameters and addition of a constant grey tone block.
These operations were called contrast scaling and luminance shift respectively. The union of the affine mappings, the Jacquin block operator, is shown to be contractive on the set of discrete images. The iteration of the block operator upon any initial image generates an approximation of the target image. This scheme is by many aspects related to vector quantization with which it shares the idea of using a codebook providing a library for the selection of the range blocks. However, the codebook in fractal compression is only a " virtual" one since the range blocks are not stored but taken from the image itself, thus exploring the redundancy of the information present in the image.

In [14], Fisher, Jacobs and Boss introduced adaptive methods in the encoding. They used quadtree, rectangular and triangular partitions of the range blocks to improve the image fidelity. They also pointed out the important fact that it is not necessary to impose strict contractivity condition on the transformations of the code since the eventual contractivity of their union is a sufficient condition to ensure the convergence of the iteration process in the decoding. Their classification scheme [15] is made with a clever design of a variable number of classes (4-12-72) taking into account not only intensity values but also intensity variance across a domain.

Fractal compression based on piecewise self-similarities has first been implemented by Jacquin for images, i.e., for digital signals in two dimensions. Of course, the same ideas are applicable for modeling one-dimensional signals. The group at the Department of Electrical Engineering at the Georgia Institute of Technology consisting of Hayes, mazel and Vines has investigated this application in a number of papers. For example, in [16] the approach using linear fractal interpolation as well as the piecewise self-affine fractal model are discussed with algorithms that are adaptive in the choice of the sizes of the ranges and domains. Some previous work on ID-coding is in [17].
an original approach to fractal coding is described in [18]. The image is partitioned into nonoverlapping rectangular blocks. Each block is split into a finite number of tiles using an IFS. Then, each tile is coded by a lease-squares approximation of the transformed block (under a contractive affine mapping). Thus, the encoding is accomplished without searching by solving a set of linear equations whose coefficients are computed in linear time with the total number of pixels. This method, called the Bath fractal transform, is generalized in [19] by including searching at different levels for which the cost/image fidelity trade-off is experimentally investigated. The results indicate that the fidelity pained by searching does not compensate the extra hits needed to specify the symmetries. It is suggested that the use of higher order contractive maps (instead of the affine ones) could be a better option [20]; in [21], Dudbridge presented a similar coding technique

with a fast non-iterative decoding algorithm. Some promising results on fractal video compression are reported in [22].

3. MATHEMATICAL FOUNDATIONS OF FRACTAL IMAGE CODING

3.1 What are Fractals:

A fractal is an infinitely magnifiable picture that can be produced with small set of instructions and data. With a fractal , the more you zoom in on an image, the more details you see. If you zoom on a bit-mapped image, however, eventually all you will see is big blocks of the same color. The word fractal was coined by Benoit Mandelbrot to mean fractured structure possessing similar-looking forms at many different sizes. For example, a tree in winter has large branches, small branches, and tiny twings, all branching off in the same way at different scales. Traditional, abstract fractals, such as the mandelbrot set, have become very popular. They tend to be harmonious, delicate, balanced, and pleasing to the eye because they have low information content, which follows from the fact that the program that produces them is finite even though the picture appears to be infinite. The following paragraph explains how fractals can be used to generate any particular image.

Suppose we want to generate pictures of man-made objects, such as bricks, wheels, roads, building and cogs, we can easily generate them by using any graphics system. But suppose we want to generate natural objects such as a sunset, a tree, a lump of mud, it is very difficult to generate, because for these we have to tell the computer the address and color attribute of each point in the cloud. To escape this difficulty, we need a richer library of geometrical shapes. These shapes need to be flexible and controllable so that they can be made to conform to clouds, mosses, feathers, leaves and faces, not to mention waving sun flowers and glaring arctic wolves. Fractal geometry provides just such a collection of shapes.

Using fractals to simulate landscapes and other natural effects is not new; it has been a primary practical application. For instance, through experimentation, you can find a fractal generates a pattern similar to tree bark. Later, when you want to render a tree, you put the tree bark fractal to work. What is new is the ability to start with an actual image, and find the fractals that will imitate it to any desired degree of accuracy. Since our method includes a compact way of representing these fractals, we end up with a highly compressed data set for reconstructing the original image.

We start with a digital image. Using image processing techniques such as color separation, edge detection, spectrum analysis, and texture-variation analysis, we break up the image into segments. A segment might be a fern, a leaf, a cloud, or a fence post. A segmenting can also be a more complex collection of pixels: A seascape, for example, may include spray, rock and mist. We then look up these segments in a library of fractals. The library doesn't contain literal fractals; that would require astronomical amount storage. Instead, out library contains relatively compact sets of numbers, called *iterated function systems*(IFS) codes, that will reproduce the corresponding fractals. Furthermore, the library's cataloging system is such that images that look alike are close together: Nearby codes correspond to the nearby fractals. This makes it feasible to setup automated procedures for searching the library to find fractals that approximate a given target image. A mathematical result known as the collage theorem quarantees that we can always find a suitable IFS code - and gives a method for doing so. Once we have looked up all the segments in our library, we can throw away the original digitized image and keep the codes, achieving our compression ratio of 10,000 to 1 - or even higher.

3.2 Affine Transformations:

The concept of affine transformation is central to fractal image compression. An *affine transformation* is a mathematical function made up from some combination of a rotation, a scaling, a skew, and a translation in n-dimensional space. A simple example in two dimensions would be

$$\mathbf{W(x,y) = (ax+by+e\ ,\ cx+dy+f)}$$

This can be written in matrix notation as the following.

$$W\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} ax+by+e \\ cx+dy+f \end{pmatrix}$$

the matrix

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

determines the rotation, skew and scaling, and

$$\begin{pmatrix} e \\ f \end{pmatrix}$$

determines the translation. This translation moves the point (0,0) to (e,f), the point (1,0) to (a+e,c+f), the point (0,1) to (b+e,d+f), and the point (1,1) to (a+b+e,c+d+f). The values a,b,c,d,e and f are the affine coefficients of this transformation.

If an affine transformation causes an image to contract in spatial dimensions, then the affine transformation is said to be *contractive*. That means , distance between to points x,y in resulted image after applying affine transformation, must be less than the distance between x,y in the original image. This type of contractive affine transformations are important to the theory and practice of fractal image compression.

Given a two-dimensional image and its corresponding image after applying affine transformation, we can solve the six simultaneous equations determined by the x,y location of the three points on the contractive image to find the values of the six coefficients (a,b,c,d,e, and f) that define the affine transformation.

Affine transformations are not restricted to two dimensional. A gray-scale image can be considered to be a 3-D entity with two spatial dimensions and one intensity dimension. If you apply a 3-D contractive affine transformation to a gray-scale image, then it will become smaller spatially, the brightness will change, and the contrast will decrease. We can represent a 3-D affine map using standard homogeneous 4×4 transformation matrix as follows.

$$\begin{pmatrix} x^1 \\ y^1 \\ z^1 \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & 0 & e \\ c & d & 0 & f \\ 0 & 0 & g & h \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1/s & 0 & 0 & 0 \\ 0 & 1/s & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

where 1/s denotes the factor that scales the original image to the sizeof nncontracted image, a,b,c,d $\in$ {-1,0,1} form an isomorphism, e,f translate the image, $g < 1$ reduces contrast, and h adjust brightness.

3.3 Iterated Function systems:

IFS theory is an extension of classical geometry. It uses affine transformations, to express relations between parts of an image. Using only these relations, it defines and conveys intricate pictures. Using IFS, we can describe a cloud as clearly as an architect can describe a house. Briefly IFS is nothing but a collection of contractive affine transformations.

An iterated function system consists of a set of affine maps $\{W_i\}^N_{i=1}$ from R^n into itself.If the maps of an IFS are contractive, each IFS include a single, compact, non empty set $A \subset R^n$, called its *attractor*, defined as the union of images of itself under the IFS maps.

$$A = \bigcup_{i=1}^{N} w_i(A)$$

The Hutchinson Operator w is a convenient shorthand notation

$$W(\bullet) = \bigcup_{i=1}^{N} W_i(\bullet)$$

that allows us to simplify the definition of an IFS attractor as $A = w(A)$.

The Inverse Problem of Iterated Transformation theory:

Let ($\mathcal{C}$, d) denote a metric space of digital images where d is a given metric-distortion measure, and let μ_{orig} be an original image that we want to encode. The *inverse problem* of iterated transformation theory is the construction of a contractive image transformation τ, defined from the space($\mathcal{C}$, d) to itself, for which μ_{orig} is an approximate *fixed point*. We denote by $\mathcal{F}$ the set of allowed transformations: a specific subset defined *a priori* of the space of all contractive transformations in ($\mathcal{C}$, d). The requirements on the transformation τ are formulated as follows:

$\exists s < 1$ such that $\forall \mu, \nu \in \mathcal{C}$, $d(\tau(\mu), \tau(\nu)) \leq s d(\mu, \nu)$, and, (contractivity) (1)

$d(\mu_{orig} \tau(\mu_{orig}))$ is as "small" as possible, (app. fixed point) (2)

The scalar s is called the *contractivity* of τ. Under these conditions, and provided that τ has a lower *complexity* than the original image, τ can be seen as a *code*-lossy in general-for μ_{orig}. By repeated application of the triangular inequality in ($\mathcal{C}$, d) and use of the contractivity of τ, it is to show that, for any image μ_0 and any positive integer n:

$$d(\mu_{orig}, \tau^n(\mu_0)) \leq \frac{1}{1-s} d(\mu_{orig}, \tau(\mu_{orig})) + s^n d(\mu_{orig}, \mu_0) \qquad (3)$$

From (3), we see that after a number of iterations, the terms of any iterated sequence of the form:

$\{\mu_n = \tau^n (\mu_0) \}_{n \geq 0}$

where μ_0 is some arbitary initial image, clustered around the original image. In a space of quantized images, the sequence converges exactly to a stable image, which as a result of its iterative construction, is said to be *fractal*[23],[24].

- the closeness of $\tau^n(\mu_0)$ to μ_{orig} is conditioned by the distortion $d(\mu_{orig}, \tau(\mu_{orig}))$.
- It is clear that if s is close to one, the error bound in (3) becomes very large. The convergence of the sequence depends fundamentally on s being strictly smaller than one.
- The constant transformation $\tau = \mu_{orig}$, which has a contractivity equal to 0 and which clearly satisfies (1) and (2), will never be in $\mathscr{T}$ because we are interested only in transformations that have a lower complexity than that of the original image.

We call a transformation τ in $\mathscr{T}$ which satisfies (1) and (2) a *fractal code* for μ_{orig}, and say that μ_{orig} is approximately *self-transformable* under τ. This terminology is used because images produced by the above procedure are the result of the iterated application of a deterministic image transformation to an initial image, a procedure which is typical of the construction of deterministic fractal objects [23], [24].

3.4 The mathematical principle behind fractal image compression

Fractal image coding exploits the piecewise self-similarity of the image. The basic idea is to construct a contractive operator W in the metric space X of digital images, for which the image to be encoded is the unique fixed point x_0.

For this, the class of affine operators is preferred because of their particularity to be "nearly linear," which makes them easy to analyze. the fixed point x_0 of such an operator W: X $\rightarrow$ X verifies

$Wx_0 = Ax_0 + b$

$= x_0.$ where A is a linear operator and b belongs to X.

Real-world images are usually not self-similar–except contrived examples–and it is impossible to find an operator that maps a whole image x onto another image x_0 such as x equals x_0.

However, real-world images present piecewise self-similarity [5]: parts of the image resemble other parts. starting from this observation, one can define an operator as the sum of piecewise mappings on X. A solution is to decompose the image into N nonoverlapping *domain* blocks D and into M different *range* blocks R and define the operator W as

$$Wx = W\left(\bigcup_{i=1}^{N} D_i\right) = \bigcup_{i=1}^{N} w_i(Ri)$$

We use the notation $D_i = x \lfloor D_i$ to denote the image x restricted to the domain part D_i, and ω_i to denote the transformation mapping the range block R_i into the domain block D_i. The blocks R_i and D_i can have different sizes and shapes. The number M of blocks R_i is less than, greater to , or equal to N and therefore the blocks R_i can be overlapping or picked only from parts of the image.

An additional constraint that must be imposed on the operator W is that it is eventually contractive. W is *contractive* if there exists a constant $s < 1$ such that $d[W\{x\}, W\{y\}] \leq s.d(x, y)$. $\forall x, y \in X$, where d is a given distance measure and s is called the contractivity of W. W is said to be *eventually contractive* (at the Kth iterate) if there exists a constant K such that the Kth iterate of W is contractive.

If this is the case, the operator W has a unique fixed point x_0 obtained with $x_0 = \lim_{k\to\infty} W^{ok}x$, for an arbitary x. The collage theorem proves that if the distance between an image x and its transformation by the eventually contractive operator W is small, then the distance between the image x and the fixed point x_0 of W will also be small. It provides a boundary given by:

$$d(x, x_0) \leq \frac{1}{1-s_k} \cdot \frac{1-s_1^K}{1-s_1} \cdot d(x, Wx)$$

where s_1 is the Lipschitz constant of W, K is the iterate number under which the operator W becomes contractive, and s_k is the contractivity of W^{0K}.

TH-6393

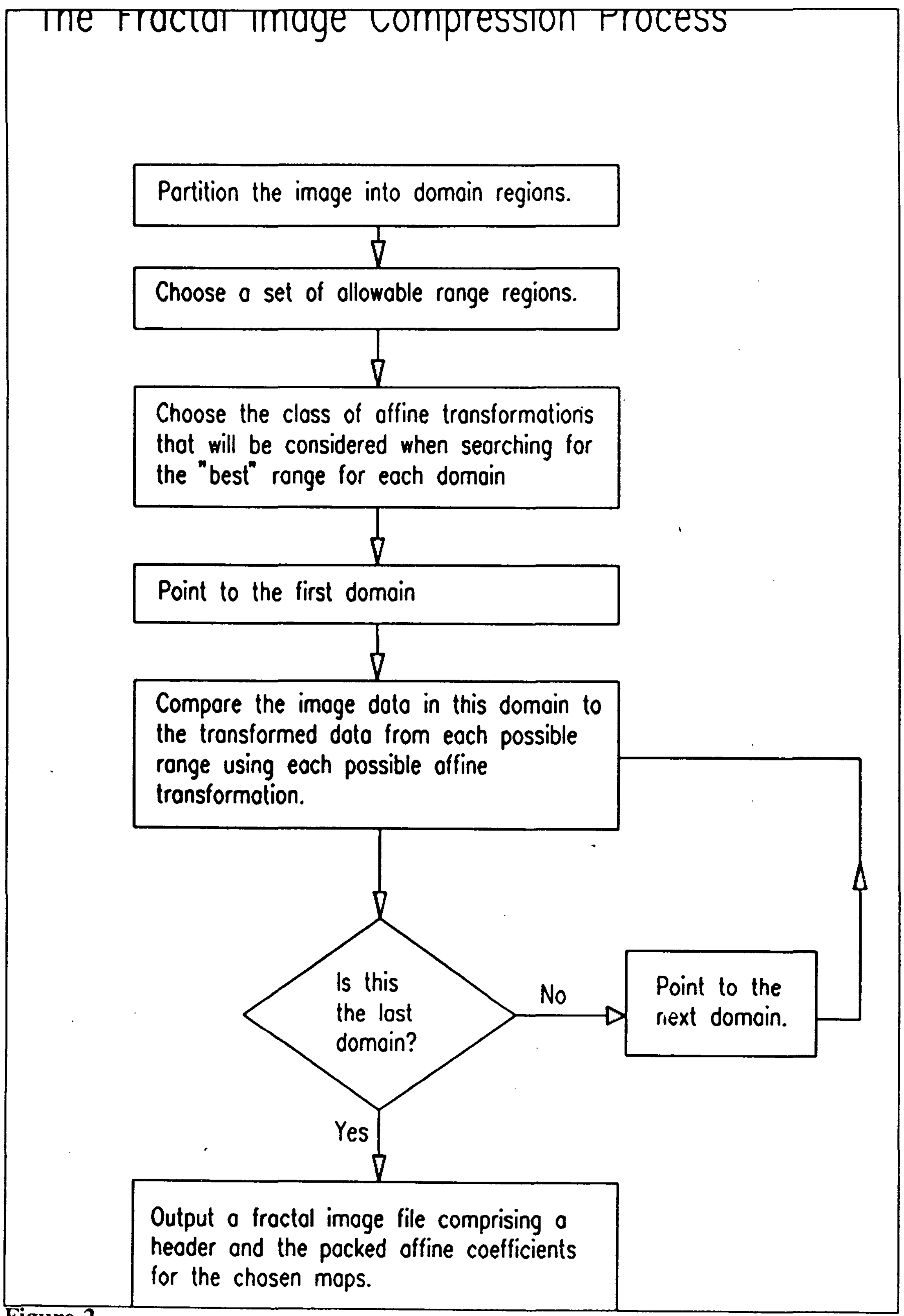

The Fractal Image Compression Process
Partition the image into domain regions.
Choose a set of allowable range regions.
Choose the class of affine transformations that will be considered when searching for the "best" range for each domain
Point to the first domain
Compare the image data in this domain to the transformed data from each possible range using each possible affine transformation.
Is this the last domain?
No
Point to the next domain.
Yes
Output a fractal image file comprising a header and the packed affine coefficients for the chosen maps.

Figure 2

4. FRACTAL IMAGE COMPRESSION PROCESS

The algorithm for Fractal Image Compression process is given in figure 2. Brief description of the algorithm is given below.

The first step in fractal image compression process is to partition the image into nonoverlapping domain regions. Taken together, the set of domain regions must cover the entire image, but they can be any size or shape. Next, the program defines a collection of possible range regions, which must be larger than the domain regions, can overlap, and need not cover the entire image.

For each domain region, the program must choose the range region that, after an appropriate 3-D affine transformation is applied, most closely matches the domain regions. The affine transformations not only shrink and deform the image within the range region, they also decrease contrast and change brightness in the intensity dimension. Each 3-D affine transformation can be described by its affine coefficients.

A FIF(Fractal Image Format) file is then written. It consists of a header with information about the specific choice of domain regions, followed by the packed list of affine coefficients chosen for each domain region. This process generates a file that is dependent of the resolution of the original image; you have found an equation for the picture. Consider a straight line: It can be represented by the equation $y = ax + b$. If you know the values of the coefficients a and b, then you can draw the line at any resolution. In an analogous way, given the affine coefficients in the FIF file, the decompression process can create a fractal replica that looks like the original at any resolution.

Commercial implementations of the fractal transform faces some complex trade-offs when choosing domain regions, range regions, and allowed transformations. The larger the domain regions, the fewer the number of transformations that are needed to model the image, and the smaller the fractal file. However, if a reasonably close match is not found between each of the domain regions and a transformed range region, the quality of the decompressed image is reduced.

The compressor considers domain regions of various sizes, finds the best range region for each in the time available and uses a mathematical procedure to assess the optimum set of domain regions for desired file size. On a region of blue sky, for example, it may be possible to use a large domain region that matches even with an even larger patch of the sky. But in another part of the picture, you might have to use smaller domain region to find good-enough range region within the available search time.

To keep compression time reasonable, practical limits must be put on the collection of possible range regions and the allowed transformations. The C language implementation of the algorithm is given in Appendix A.

4.1 Image partitions:

The square support of the original digital image μ_{orig}will be partitioned into nonoverlapping square domain cells of two different sizes, thus forming a *two-level square partition.* The larger cells-of size D $\times$ D-are referred to as (domain) *parent cells*, the smaller ones-of size D/2 $\times$ D/2-as (domain) *child cells.* A parent cell can be split into up to four nonoverlapping child cells. Decisions about the splitting of a parent cell are made during the encoding of the image block over this cell. Thus a partition constructed this way is image dependent: it allows the coder: i) to use large blocks to take the advantage of smoothly varying image ares, and ii) to use small blocks to capture detail in complex ares(rugged boundaries, fine textures).

The selection of sizes and shapes for image cells depends on several factors. Small image blocks-4 $\times$ 4 and below-are i) easy to analyze and to classify geometrically, ii) they allow a fast evaluation of interblock distances, iii) they are easy to encode accurately, and iv) they lead to a robust encoding system-one whose performance is steady, even when source images are diverse. On the other hand, large blocks-5 $\times$ 5 and above-i) allow exploitation of the redundancy in smooth image areas, and ii) lead theoretically to high compression ratios.

The maximal range pool corresponding to a domain block of size D $\times$ D can be thought of as all image blocks of size B $\times$ B (B $>$ D) located anywhere in the image to encode. It is typicallly very large, but it can be trimmed and organized in order to make the search for an optimal domain block tractable.

An initial range pool can be obtained by sliding a window of size B $\times$ B(B $=$ 2D is used) across the original image. The window is first located with its bottom left corner at (0, 0). It then moves from one position to the next by steps of either δ_n pixels horizontally to the right, or δ_v pixels vertically upwards, in such a way that it remains entirely inside the image support, at all times. The steps are typically chosen equal to D or D/2.

4.2 Discrete Image Transformations

We describe the discrete form of the B:D spatial contraction operator $\mathcal{F}$, which maps image blocks from a range cell $R_i = S(i_r, j_r, B)$, to a domain cell $D_i = S(i_d, j_d, D)$. In the simple case B = 2D, the pixel values of the contracted image on the domain block $S(i_d, j_d, D)$ are the average of four pixels in the domain block:

$(\mathcal{F}\mu)i_r+i, j_r+j = (\mu_{I(i), J(j)} + \mu_{I(i), J(j)} + \mu_{I(i), J(j)+1} + \mu_{I(i)+1, J(j)+1}) / 4$, for all $i, j \in \{0, \ldots, B-1\}$

where the index function I and J are defined by

$$I(i) = i_d + 2i, \quad \text{and } J(j) = j_d + 2j.$$

In the case where B/D is not integral, the action of the spatial contraction operator is best described in two steps. Firstly, the range block is discretized at he coarser resolution B. Secondly, the discretization at resolution B is uniformly scaled by the factor D^2/B^2. Now we find the scaling, luminance shift and the isometry for transforming the range block into the corresponding domain block in the form:

$$\mathcal{F}_j\mu\rceil_{Bj} = \tau_{ni}(\alpha_i(\mathcal{F}_j\mu\rceil_{Dj}) + \Delta g_i).$$

where α_i is the scaling factor which takes values from the set {0.5, 0.6, 0.7, 0.8, 0.9, 1.0} and Δg_i is the luminance shift and $\{\tau_n\}_{0 \leq n \leq 7}$ is the isometry.

4.3 Distortion Measure:

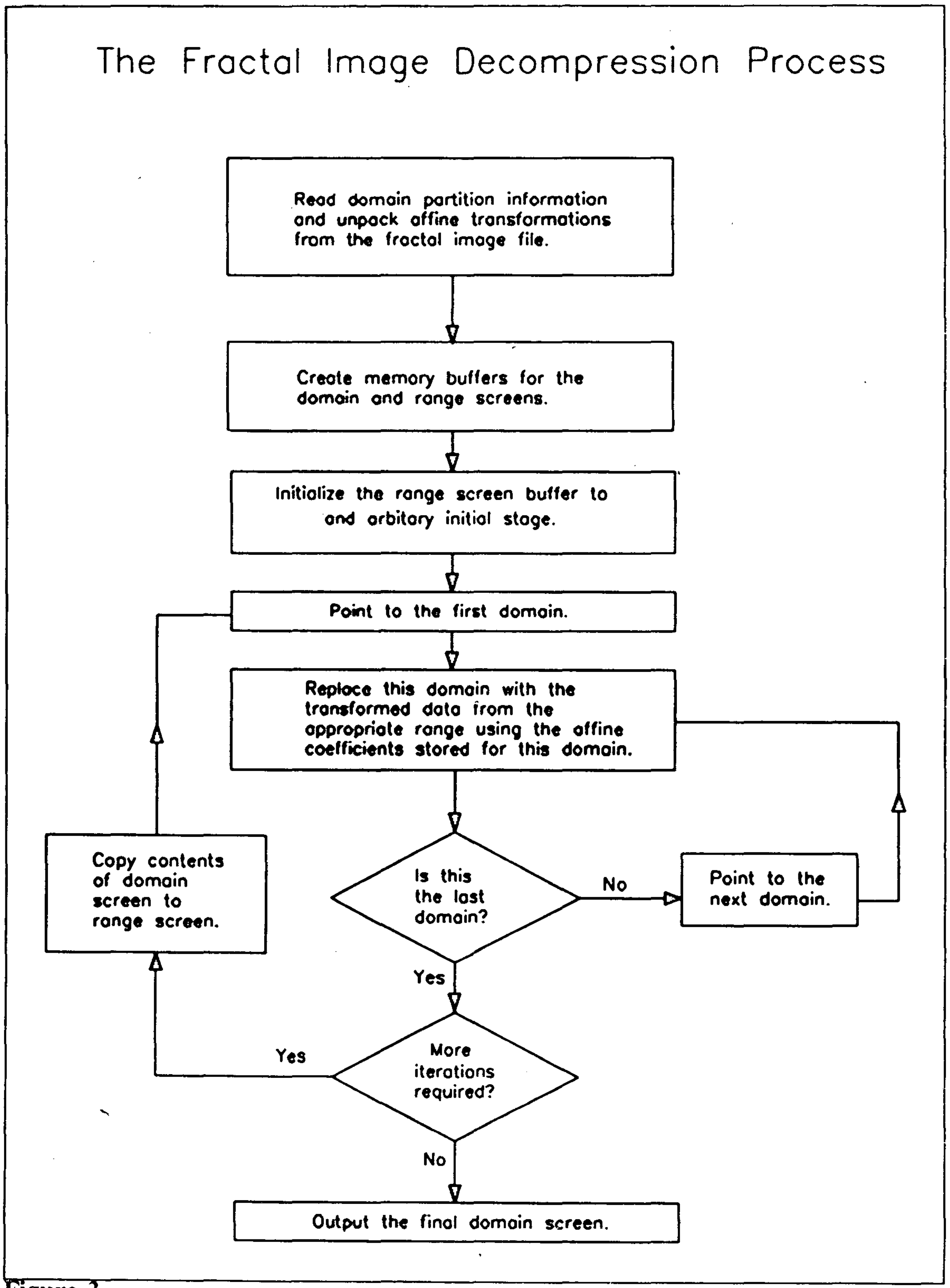
Let $S(i_o, j_o, D)$ denote the square cell of size $D \times D$, with the bottom left corner at the intersection of image row i_o and image column j_o. Let μ be an $r \times r$ image, and μ' be an approximation of μ. Let $\mu\rceil_s, \mu^1\rceil_s$ denote their restrictions to the cell $S(i_o, j_o, D)$. The L_2 or *mean squared* (MS) distortion between the image blocks $\mu\rceil_s$, and $\mu'\rceil_s$ is defined as the sum over the cell S, of the squared differences of pixel values i.e.:

$$d_{L2}(\mu_{\rceil S}, v_{\rceil S}) \sum_{0 \leq i,j < B} (\mu_{io+i,j0+j} - v_{io+i,jo+j})^2$$

We define the distortion between two images as the sum over the partition of all block distortions, and the *peak-to-peak signal-to-noise ratio* (SNR) by:

$$SNR = 10\log_{10}\left(\frac{dr(\mu)^2}{d(\mu,\mu^1)/r^2}\right)$$

where $dr(\mu)$ denotes the dynamic range of μ.


The Fractal Image Decompression Process
Read domain partition information and unpack affine transformations from the fractal image file.
Create memory buffers for the domain and range screens.
Initialize the range screen buffer to and arbitary initial stage.
Point to the first domain.
Replace this domain with the transformed data from the appropriate range using the affine coefficients stored for this domain.
Is this the last domain?
No
Point to the next domain.
Yes
More iterations required?
Yes
Copy contents of domain screen to range screen.
No
Output the final domain screen.
Figure 3

5. FRACTAL IMAGE DECOMPRESSION PROCESS

The algorithm for fractal image decompression process is given in figure 3. Brief description of the algorithm is given below.

The decompression process starts when you assigns memory for two equal-size images A and B. The size of these images can be smaller or larger than that of the original image before compression, and the initial content is unimportant. It can be data, a picture -anything.

For the first iteration of the decompression process I refer to image A as the *range image* and image B as the *domain image*. I partition the domain image into domain regions specified in the FIF file header. For each domain region in domain image, I read the affine coefficients for this domain from the FIF file, locate the range region specified by this affine transformation in the range image and map the contents of this range region from the range image to the appropriate domain region in the domain image.

Note that the transformation from the range to the domain is contractive, since I require the range regions to be larger than the domain regions. When this is done for each domain region, a new image B is created from transformed bits of image A.

For the second iteration, I make the new image B the range image and image A the domain image, and I repeat the process for each domain region. After two iterations, the arbitary starting data has been mapped from A to B and then from B to A. I repeat this process until the differences between images A and B is negligible, and I then display image A.

This simple process creates an image. How closely the compressed image matches the original depends on how accurately the chosen range regions match the domain regions during the compression process. This convergence of the initial image to the target image is guaranteed by the collage theorem as explained above. The C language implementation of the algorithm is given in Appendix A.

6. CONCLUSION

Compressing any image data using fractal techniques has been designed and implemented successfully in C language. The main features of the project are

1) Ability to take larger domain blocks in smooth parts the picture to achieve high compression ratios.

(2) Ability to take small domain blocks in parts of the image where there is a change of intensity, to get a reasonably good approximation of the original image.

(3) The Decompression time is very less compared to other methods of image compression and decompression.

6.1 Future Enhancements:

There is a lot of scope for enhancement of the project without modifying much in the original programs. The features, where there is a scope for enhancement are

(1) The compression time can be decreased a lot, by classifying domain blocks, by which searching for a suitable range block is confined to only the range blocks that are belongs to the same type as the corresponding domain block.

(2) There is a chance to consider all the range blocks that are larger than corresponding domain block without restricting them to only blocks with size B = 2D.

REFERENCES

(1) P. L. Cosman, R. M. Gray and M. Vetterli, "Vector Quantization of Subband: A Survey," ***IEEE Trans. Image Processing***, vol.5, no.2, pp.202-225, Feb 1996.

(2) M. Antonni, M. Barlaud, and I. Daubeechies, "Image Coding using Wavelet transform," ***IEEE Trans. Image Processing,*** vol.1,no.2,pp.205-220,1992.

(3) M. Barlaud, P. Sole, T. Gaidon, M. Antoni, and P. Mathieu, "Pyramidal lattice vector quantization for multiple image coding," ***IEEE trans. Image Processing,*** vol.3,no.4,pp.367-381, 1994.

(4) M. Barnsley and L. Hurd, ***Fractal Image Compression,*** Wellesley, MA;AK Peters, 1993.

(5) A. E. Jacquin, "A fractal theory of iterated markov operators with applications to digital image coding," Ph.D dissertation, Georgia Institute of Technology, Atlanta, GA, Aug 1989.

(6) Arnaud E. Jacquin, "IMage Coding Based on a Fractal Theory of Iterated Contractive Image Transformations," ***IEEE trans. image processing,*** vol.1,no.1,pp.18-30,jan 1992.

(7) Y. Fisher, Ed., ***Fractal Image Compression – Theory and Application.*** New York; Springler-Verlag, 1995.

(8) --, "Fractal Image Coding: A review," in ***proc. IEEE.,*** vol.81,no-10,pp.1451-1465, 1993.

(9) D. Saupe and R. Hamzaoui, "A review of the fractal image compression

literature," *Computer Graphics,* vol.28,no.4,pp.268-276,1994.

(10) Barnsley, M. E. Sloan., A.D., "chaotic compression", *Computer Graphics World,* Nov. 1987.

(11) Barnsley, M.F. Sloan.,A.D., "A better way to compress images", *BYTE Magazine,* Jan. 1988.

(12) Jacquin, A.E., "Fractal image coding based on a theory of iterated contractive image transformations", SPIE vol.1360, *visual communications – and Image Processing,* pp.227-239,1990.

(13) Jacquin,A.E.," A novel fractal block-coding technique for digital images", *Proc. ICA SSP4,* pp.2225-2228, 1990.

(14) Fisher,Y., Jacob,E.W., Boss,R.D., Fractal image compression using iterated transforms, in: *Image and Text Compression,* J.A.Storer(ed.),Kluwer Academic Publishers, Bostan,pp.35-61,1992.

(15) Fisher,Y., *Fractal image compression – theory and applications to Digital Images*, Springler-Verlay, New York, 1994.

(16) Mazel, d.s., Hayes, M.H., Using iterated function systems to model discrete sequences, *IEEE Trans. on Signal Processing,* vol.40,no.7,pp.1724-1734, 1992.

(17) Mantica,G., Sloan,A., "Chaotic Optimization and he construction of fractals: solution of an inverse problem," *comp. syst.* 3, pp.37-62,1989.

(18) Monro, D.M., Dudbridge,F.,Fractal block coding of images, *Electronic Letters,* vol.28, pp.1053-1055,1992.

(19) Monro, D.M., "A hybrid fractal transform", ***Proc. ICASSP 5,*** pp.169-172, 1993.

(20) Monro, D.M., Wolley, S.J., Fractal image compression without searching, ***Proc ICASSP,*** 1994.

(21) Dudbridge, F., Least-squares block coding by fractal functions, in: Fractal Image Compression - Theory and Applications to Digital images, Y. Fisher(ed.), Springler-verlag, New Work, 1994.

(22) Wilson, D.L, Nicholls, J.A., Monro. D.M., Rate buffered fractal video, ***Proc ICASSP,*** 1994.

(23) Barnsley, M., ***Fractals EveryWhere,*** Academic Press, San Diego, 1988.

(24) B. Mandelbrot, ***The Fractal Geomentry of Nature,*** San Francisco, CA; Freemah, 1982.

(25) Pannebaker, William B., and Joan Mitchell. ***JPEG still Image Compression Standard,*** New York: Van Nostrand Rein, 1993.

APPENDIX

```
/***** FILE : compress.c                                  *****/
/***** The Main Program for Compressing an input Image File *****/

#include <stdio.h>
#include <stdlib.h>
#include<alloc.h>
#include "fractal.h"

FILE *image_file,*fractal_file;
Rectangle image ,reduced_image;
unsigned long infinity;
void compress(short ,short ,short ,short ,int *);

main(int argc,char **argv)
{
   ImageHeader header;
   short domain_x,domain_y;
   int no_maps;

   if (argc != 3)
   {
     fprintf(stderr,"\nUsage: compress image_file fractal_file.\n");
     exit(1);
   }

   infinity=255*DB_SIDE*DB_SIDE;

   if (NULL == (image_file=fopen(argv[1],"rb")))
   {
     fprintf(stderr,"Unable to open file %s.\n", argv[1]);
     exit(1);
   }

   if (NULL == (fractal_file=fopen(argv[2],"wb")))
   {
     fprintf(stderr,"Unable to open file %s.\n", argv[2]);
     exit(1);
   }
   read_tga_header(image_file,&header);

   fprintf(stderr,"Width %d Height %d\n",header.width,header.height);
```

```
        /***** Input must be a Targa File *****/

        if (1!=fwrite(&header,sizeof(ImageHeader),1,fractal_file))
        {
          fprintf(stderr,"Error writing header to fractal file %s.\n", argv[2]);
          exit(1);
        }

        image.width=header.width;
        image.height=header.height;
        image.length=header.width*header.height;

        image.pixel = (Pixel *) calloc(image.length, sizeof(Pixel));

        reduced_image.width=header.width/2;
        reduced_image.height=header.height/2;
        reduced_image.length=header.width*header.height/4;

        reduced_image.pixel = (Pixel *) calloc(reduced_image.length,
sizeof(Pixel));

        if (NULL == image.pixel)
        {
          fprintf(stderr,"Unable to allocate %ld bytes for image buffer.\n",
                                image.length);
          exit(2);
        }

        if (image.length!=fread(image.pixel,
                    sizeof(Pixel),image.length,image_file))
        {
          fprintf(stderr,"Error reading header from image file %s.\n", argv[1]);
          exit(1);
        }

        fclose(image_file);

        /* rescale (contract) image in spatial and intensity directions */

        reduce_image(&image,&reduced_image);
```

```
    /* MAIN LOOP */
    no_maps=0;
    for (domain_y=0;domain_y<image.height;domain_y+=DB_SIDE)
      for (domain_x=0;domain_x<image.width;domain_x+=DB_SIDE)
      {

          /* Step 2: Get Domain Block */

          fprintf(stderr,"Dx %d Dy %d\n",domain_x,domain_y);
          compress(domain_x,domain_y,DB_SIDE,DB_SIDE,&no_maps);
      }

    fwrite(&no_maps,sizeof(int),1,fractal_file);
    fclose(fractal_file);
    free(image.pixel);
    return(0);
}

/***** Function for selecting suitable domain block for transformation *****/

void compress(short domain_x,short domain_y,short width,short height,int
                                        *no_maps)
{
    Rectangle domain_block,range_block,flipped_range_block;
    Symmetry current_symmetry;
    short current_range_x,current_range_y,current_shift;
    Pixel domain_mean;
    AffineMap best_map;
    unsigned long current_distance,minimum_distance;

    domain_block.width=range_block.width=
                flipped_range_block.width=width;
    domain_block.height=range_block.height=
                    flipped_range_block.height=height;
    domain_block.length=range_block.length=
                  flipped_range_block.length=width*height;

    domain_block.pixel = (Pixel *) calloc(width*height,sizeof(Pixel));
    range_block.pixel   =   (Pixel *) calloc(width*height,sizeof(Pixel));

    flipped_range_block.pixel = (Pixel *) calloc(width*height,sizeof(Pixel));

    minimum_distance=infinity;
```

```
copy_rectangle(&image,domain_x,domain_y,&domain_block,0,0,

                        width,height);
domain_mean=mean(&domain_block);

for (current_range_y=0;current_range_y<=
     reduced_image.height-height; current_range_y+=2)
for (current_range_x=0;current_range_x<=
          reduced_image.width-width; current_range_x+=2)
{

  copy_rectangle(&reduced_image,current_range_x,
        current_range_y,&range_block,0,0,width,height);

  current_shift = ((short) domain_mean)-((short) mean(&range_block));

  intensity_shift(&range_block,current_shift);

  for (current_symmetry=0;current_symmetry<NSYMS;
                current_symmetry++)
  {
        flip(&range_block,&flipped_range_block,
           current_symmetry);
        current_distance = l2_distance(&domain_block,
           &flipped_range_block);

        if (current_distance<minimum_distance)
        {
           minimum_distance=current_distance;
           best_map.shift=current_shift;
           best_map.symmetry=current_symmetry;
           best_map.range_x=current_range_x;
           best_map.range_y=current_range_y;
           best_map.width  = width;
        }
  }
}

free(domain_block.pixel);
free(range_block.pixel);
free(flipped_range_block.pixel);
if((minimum_distance < MAX_ERROR) || (width <= 4))
{
  fwrite(&best_map,sizeof(AffineMap),1,fractal_file);
```

```
         fprintf(stderr,"Error %d\n",minimum_distance);
         (*no_maps)++;
       }
       else
       {
        compress(domain_x,domain_y,width/2,height/2,no_maps);
        compress(domain_x+width/2,domain_y,width/2,height/2,no_maps);
        compress(domain_x,domain_y+height/2,width/2,height/2,no_maps);
compress(domain_x+width/2,domain_y+height/2,width/2,height/2,no_maps);
       }
   }
```

```
/***** FILE :  decompress.c                                   *****/
/***** The Main Program for Decompressing the coded file into an Image
File                  *****/

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <conio.h>
#include "fractal.h"

#define DEFAULT_ITERATES 16

AffineMap *affine_map_array,*map_ptr;
FILE *image_file,*fractal_file,*initial_file;
Rectangle image,reduced_image;
void decompress(short ,short ,short ,short );

main(int argc,char **argv)
{
   ImageHeader header,initial_header;
   short iterate,arg_offset=0,iterates=DEFAULT_ITERATES;
   long int i;
   int number_of_maps;
   short domain_x,domain_y;

   if ((argc < 3)||(argc>5))
   {
     fprintf(stderr,"\nUsage: decompress [num_iterates] [initial_image]
fractal_file   image_file.\n");
     exit(1);
   }

   if (argc==4)
   {
     iterates=atoi(argv[1]);
     arg_offset=1;
   }

   if (argc==5)
   {
     iterates=atoi(argv[1]);
     initial_file=fopen(argv[2],"rb");
     arg_offset=2;
   }
```

```
/* Read in affine maps and header information. */

if (NULL == (fractal_file=fopen(argv[arg_offset+1],"rb")))
{
  fprintf(stderr,"Unable to open fractal file %s.\n", argv[arg_offset+1]);
  exit(1);
}

if (NULL == (image_file=fopen(argv[arg_offset+2],"wb")))
{
  fprintf(stderr,"Unable to open image file %s.\n", argv[arg_offset+2]);
  exit(1);
}

fseek(fractal_file,-1*(int) sizeof(int),SEEK_END);
if (1!=fread(&number_of_maps,sizeof(int),1,fractal_file))
{
  fprintf(stderr,"Error reading no. of affine maps from fractal file %s.\n",
  argv[arg_offset+1]);
  exit(1);
}
printf("The number of Affine maps =  %d",number_of_maps);
fclose(fractal_file);
if (NULL == (fractal_file=fopen(argv[arg_offset+1],"rb")))
{
  fprintf(stderr,"Unable to open fractal file %s.\n", argv[arg_offset+1]);
  exit(1);
}
if (1!=fread(&header,sizeof(ImageHeader),1,fractal_file))
{
  fprintf(stderr,"Error reading header from fractal file %s.\n",
   argv[arg_offset+1]);
  exit(1);
}

write_tga_header(image_file,&header);
affine_map_array = (AffineMap  *)
     calloc(number_of_maps,sizeof(AffineMap));
```

```
          if (number_of_maps!=
fread(affine_map_array,sizeof(AffineMap),number_of_maps,fractal_file))
          {
            fprintf(stderr,"Error reading data from %s.\n",argv[arg_offset+1]);
            exit(1);
          }
          map_ptr = affine_map_array;
          fclose(fractal_file);

          image.width = header.width;
          image.height = header.height;
          image.length = header.width*header.height;
          image.pixel = (Pixel *) calloc(image.length,sizeof(Pixel));

          reduced_image.width = header.width/2;
          reduced_image.height = header.height/2;
          reduced_image.length = header.width*header.height/4;
          reduced_image.pixel = (Pixel *)
               calloc(reduced_image.length,sizeof(Pixel));


          /* Set source buffer to a predetermined starting condition. */

          if (argc<5)
          {
            for (i=0;i<image.length;i++)
                 image.pixel[i]=ARBITRARY_PIXEL_VALUE;
          }
          else
          {
            read_tga_header(initial_file,&initial_header);
            fread(image.pixel,image.length,sizeof(Pixel),initial_file);
          }

          /* Loop for a prescribed number of iterations. */

          for (iterate=0;iterate<iterates;iterate++)
          {

            reduce_image(&image,&reduced_image);

            map_ptr = affine_map_array;
```

```
      for (domain_y=0; domain_y<image.height;domain_y+=DB_SIDE)
        for (domain_x=0;domain_x<image.width;domain_x+=DB_SIDE)
        {
          decompress(domain_x,domain_y,DB_SIDE,DB_SIDE);
        }
    }



    if (image.length != fwrite(image.pixel,sizeof(Pixel),image.length,
                                image_file))
    {
      fprintf(stderr,"Error writing data to %s.\n",argv[arg_offset+2]);
      exit(1);
    }

    free(affine_map_array);
    free(image.pixel);
    free(reduced_image.pixel);
    fclose(image_file);
    return(0);
}

/***** Function to select the right domain block for decoding *****/

void decompress(short domain_x,short domain_y,short width,short height)
{
    Rectangle range_block,transformed_range_block;
    if((width > 4 )&& (map_ptr->width != width))
    {
      decompress(domain_x,domain_y,width/2,height/2);
      decompress(domain_x+width/2,domain_y,width/2,height/2);
      decompress(domain_x,domain_y+height/2,width/2,height/2);
      decompress(domain_x+width/2,domain_y+height/2,width/2,height/2);
    }
    else
    {
      range_block.width = width;
      range_block.height = height;
      range_block.length = width*height;
      range_block.pixel = (Pixel *) calloc(range_block.length,sizeof(Pixel));

      transformed_range_block.width = width;
      transformed_range_block.height = height;
```

```
                transformed_range_block.length = width*height;
                transformed_range_block.pixel =
                   (Pixel *) calloc(transformed_range_block.length,sizeof(Pixel));


copy_rectangle(&reduced_image,map_ptr->range_x,map_ptr->range_y,&rang
e_block,0, 0,width,height);
                intensity_shift(&range_block,map_ptr->shift);

                flip(&range_block,&transformed_range_block, map_ptr->symmetry);
                copy_rectangle(&transformed_range_block,0,0,&image,
                                         domain_x,domain_y,width,height);
                map_ptr++;
                free(range_block.pixel);
                free(transformed_range_block.pixel);
             }
          }
```

```
/***** FILE :  util.c                              *****/
/***** File that contains essential functions used during compression and
                                        decompression *****/

#include <process.h>
#include "fractal.h"

/*** Function to mean of pixel intensities in a given block ***/

Pixel mean(Rectangle *rectangle)
{
   int i;
   long sum=0;
   for (i=0;i<rectangle->length;i++)
        sum += rectangle->pixel[i];
   return(sum/rectangle->length);
}

/*** Function to shift intensities of all pixels by a fixed value ***/

void intensity_shift(Rectangle *rectangle,short shift)
{
   short i;
   for (i=0;i<rectangle->length;i++)
        rectangle->pixel[i]=rectangle->pixel[i]+shift;
}

/*** Function to find distance between two rectangular blocks ***/

long l2_distance(Rectangle *rect1,Rectangle *rect2)
/* rect1 and rect2 must have the same length */
{
   long d,distance=0;
   int i;
   for (i=0;i<rect1->length;i++)
   {
     d=rect1->pixel[i]-rect2->pixel[i];
     distance += d*d;
   }
   return(distance);
}
```

```
/*** Function to copy part of one rectangular image to another ***/

void copy_rectangle(Rectangle *src_rect,short src_x,short src_y,
     Rectangle *dest_rect,short dest_x,short dest_y,
         short width,short height)
{
    int i,j;
    for (j=0;j<height;j++)
      for (i=0;i<width;i++)
         dest_rect->pixel[i+dest_x+(j+dest_y)*dest_rect->width] =
                     src_rect->pixel[(src_x+i)+(src_y+j)*src_rect->width];
}

/*** Function to reduce a rectangular image into another ***/

void reduce_image(Rectangle *src_rect,Rectangle *dest_rect)
{
    int i,j;
    for (j=0;j<dest_rect->height;j++)
       for (i=0;i<dest_rect->width;i++)
       {
          /* spatial rescale by 2 */

          dest_rect->pixel[i+j*dest_rect->width] =
                        (src_rect->pixel[2*i+(2*j)*src_rect->width]+
                        src_rect->pixel[2*i+1+(2*j)*src_rect->width]+
                        src_rect->pixel[2*i+(2*j+1)*src_rect->width]+

src_rect->pixel[2*i+1+(2*j+1)*src_rect->width])/4;

          /* intensity rescale by 3/4 */

          dest_rect->pixel[i+j*dest_rect->width] =
                        (dest_rect->pixel[i+j*dest_rect->width]*3)/4;
       }
}

/*** Function to flip a rectangular image ***/

void flip(Rectangle *range_block,Rectangle *transformed_range_block,
                                              Symmetry symmetry)
{
    short i,j,x,y,t;
    for (j=0;j<range_block->height;j++)
      for (i=0;i<range_block->width;i++)
      {
          if (symmetry & FLIP_X) x=(range_block->width-1)-i;
          else x=i;
          if (symmetry & FLIP_Y) y=(range_block->height-1)-j;
          else y=j;
          if (symmetry & FLIP_DIAG) /* not allowed unless width=height */
          {
              t=y;
              y=x;
              x=t;
          }
          transformed_range_block->pixel[x+y*range_block->width] =
                range_block->pixel[i+j*range_block->width];
      }
}


/*** Function to read a targa file header ***/

void read_tga_header(FILE *image_file, ImageHeader *header)
{
    struct tga_hdr tgaheader;
    if (1 != fread(&tgaheader,sizeof(struct tga_hdr),1,image_file))
    {
      fprintf(stderr,"Error reading Targa header.\n");
      exit(1);
    }
    if ((tgaheader.imtype != TGA_GRAYSCALE)||(tgaheader.depth!=8))
    {
      fprintf(stderr,"Invalid Targa file.\n");
      exit(1);
    }
    header->width=tgaheader.width;
    header->height=tgaheader.height;
}


/*** Function to write targa file header to a file ***/

void write_tga_header(FILE *image_file, ImageHeader *header)
{
    static struct tga_hdr
```

```
        tgaheader={0,0,TGA_GRAYSCALE,0,0,0,0,0,0,0,0,0,8,0};
    tgaheader..width=header->width;
    tgaheader.height=header->height;
    if (1 != fwrite(&tgaheader,sizeof(struct tga_hdr),1,image_file))
   {
      fprintf(stderr,"Error writing Targa header.\n");
      exit(1);
    }
}
```

```
        /*** FILE :  fractal.h                        ***/
        /*** File that contains all declarations used during compression and
decompression        ***/

        #include<stdio.h>
        #include "tga.h"

        #define DB_SIDE   16
        #define MAX_PIXEL_VALUE 255
        #define NSYMS     8
        #define NUM_ITS  16
        #define FLIP_X    1
        #define FLIP_Y    2
        #define FLIP_DIAG 4
        #define ARBITRARY_PIXEL_VALUE 128
        #define MAX_ERROR 500


        typedef unsigned char Pixel;
        typedef unsigned char Symmetry;

        typedef struct rectangle {
            unsigned short width,height;
            unsigned long length;
            Pixel *pixel;
        } Rectangle;

        typedef struct affinemap {
            unsigned char range_x,range_y;
            short shift;
            Symmetry symmetry;
            short width;
        } AffineMap;

        typedef struct imageheader {
            unsigned short width,height;
        } ImageHeader;

        extern Pixel mean(Rectangle *rectangle);
        extern long l2_distance(Rectangle *rect1,Rectangle *rect2);
        extern void copy_rectangle(Rectangle *src_rect,short src_x,
                    short src_y,Rectangle *dest_rect,short dest_x,
                    short dest_y,short width,short height);
```

```
extern void reduce_image(Rectangle *src_rect,Rectangle *dest_rect);

extern void flip(Rectangle *range_block,Rectangle *transformed_range_block,
                                        Symmetry symmetry);

extern void intensity_shift(Rectangle *rectangle,short shift);

extern long swap_bytes(long qbyte);

extern void read_tga_header(FILE *image_file, ImageHeader *header);

extern void write_tga_header(FILE *image_file, ImageHeader *header);
```

```
/****** FILE :   tga.h                                   ******/
/****** File that contain declaration of a targa file header ******/

#define TGA_GRAYSCALE 3

struct tga_hdr {
      unsigned char id,cmaptype,imtype,col1,col2,col3,col4,col5;
      short xorigin,yorigin,width,height;
      unsigned char depth,descriptor;
};
```