THE MATHEMATICAL FOUNDATION OF IMAGE COMPRESSION

By

Lisa A. Soberano

A paper submitted in partial fulfillment of the requirements of the Honors Program in the Department of Mathematic and Statistics.

Approved By:

Examining Committee

_____
Faculty Supervisor

_____

_____

_____

_____
Department Chair

_____
Honors Council Representative

_____
Director of the Honors Scholars Program

The University of North Carolina at Wilmington

Wilmington, North Carolina

May 2000

The Mathematical Foundation of Image Compression

Lisa A. Soberano

May 2000

Acknowledgments

In eigth grade I was privileged to have an algebra teacher, Mr. Pat Boullion, who saw beyond my average grades and recognized my potential as a math student. Not only was he a friend to me, but he challenged me to do my best and helped me to believe in myself. Without his influence in my life, my appreciation and love of mathematics would not be as great as it is now.

I have found that it is hard to come by teachers that are dedicated to their students is such a profound way as Mr. Boullion was to me. Amazingly, in my sophmore year of college at UNCW, I was fortunate enough to have Dr. Russell Herman as my Differential Equations professor. It was in this course that I learned that Dr. Herman challenges his students to understand mathematics, but he does so by giving his students as much of his time as he expects them to devote to their studies of mathematics. Since that semester, I have continually brought my questions to Dr. Herman, aware that he is eager to help me discover the world of mathematics.

It is in Dr. Herman that I have found not just a wonderful professor, but a good friend. I would like to thank Dr. Herman for his time and effort teaching and training me in mathematics. Also I would like to thank him for his advice, assistance, prompting, and encouragement throughout this yearlong honors project. Without the contributions of his Matlab programs in the Fractal Transform section of Appendix A, my honors project would be incomplete.

I would like to thank my honor's committee, including Dr. Kenneth Gurganus, Dr. Russell Herman, Dr. Gabriel Lugo, and Dr. Harry Smith for supporting me and for making this research project valuable experience for me. In addition, I extend thanks for

the countless hours that they each spent rereading and commenting on the many drafts of this honors thesis.

I would like to thank my husband, Robby, for having faith in me, uplifting me when I am down, for pushing me when I want to stop, and for teaching me how to take a break and have fun.

Most importantly, I would like to give God the glory for all of the efforts I have put into this project. If not for God's awesome creation of the universes, I would not have the zeal for mathematics that I have.

"For since the creation of the world God's invisible qualities -- his eternal power and divine nature -- have been clearly seen, being understood from what has been made, so that men are without excuse."

Romans 1:20

I. Introduction

1. Background

It is interesting to notice how our advanced technology has made us impatient beings. And what do we expect because of it … faster and more efficient technology, of course. In this paper we will discuss the search for more efficient methods of image compression. There are many forms of image compression currently being used. Some of the more familiar compressions are JPEG (Joint Photographers Expert Group) and GIF (Graphics Interchange Format); although there recently has been research seeking even more efficient compressions. The objective in image compression is to efficiently produce the smallest graphics files without compromising image quality.

Image compression is a specialized form of data compression. In fact, most forms of data compression can be applied to image compression since an image is just an array of numbers. Although a graphics interface is needed to render data as an image, the data is discrete, finite, and structured. This facilitates manipulation of the data. The longtime problem with images is storing the data. The simplest way of storing image data is pixel by pixel, but this is problematic. Storing an image that is 256×256 pixels and whose entries are in the range [0, 255] requires 8 bits per pixel, so the size of the file would be 65 KB. The larger an image is the more space it will require to be stored. For example, an image that is 512×512 pixels would require 256 KB for storage. As the image grows by a factor of 2, the space required for storage grows by a factor of 4. This may be appropriate for certain situations; however, if storage resources are limited or if the image will be transmitted through a network a better solution should be found.

Many solutions to this problem have been discovered. Some are schemes in which the image data is encoded for storage and decoded for display. These coding schemes include Huffman Coding and GIF. No data is lost in these schemes. Other algorithms cause the image to lose data, which may lessen the image quality; but, they may also result in less storage space. These algorithms include Fourier Transform, Cosine Transform, JPEG, and Fractal Image Compression, all of which will be discussed in this paper.

In the 1980's, a group called the Joint Photographers Expert Group was formed to determine the standards for image compression. Their studies resulted in the JPEG scheme, a lossy form of compression which involves many steps [9]. First the image being compressed is separated into a gray scale image (the luminance) and the color information. Each of these is compacted separately. For visual purposes, our eyes can spare color more than luminance. This is because our eyes use the gray scale edges to define boundaries but allow color to bleed across boundaries. So, the precision of the color information is usually reduced to half of the precision used for the brightness values. Then the discrete cosine transform, in which the terms in the expansion are real valued, is applied to square sub-regions in the image. The compression of the image is attained by keeping as few expansion terms as possible [9]. The fewer the number of terms kept, the greater the compression; this also means that the loss of high frequency information is greater. This process can be performed on many areas within the image. Finally more compression can be achieved by truncating the precision of each term, then using a coding scheme that stores repeated strings of numbers. With this method, it is possible to reach a compression ratio of 100:1, although ratios on the range of 10:1 to

20:1 are more typical [9][12]. After compression, loss in the sharpness and detail can be detected. Depending on the software or hardware used, the time difference in compression and decompression of an image can vary up to tens of seconds [9].

A more recent approach pioneered by Michael Barnsley is to use the similarities on different scales throughout images to assist in compression. Fractal Image Compression enables an incredible amount of data to be stored in highly compressed data files. We will explore the mathematical theory, which supports fractal image compression. One of the most important foundations for fractal image compression is the concept of iterated function systems (IFS). Through IFS we are able to systematically reproduce fractals which occur in nature. With the theory that will be presented, we will explore the development of an IFS and how one can apply IFS to obtain fractal image compression.

Michael F. Barnsley's discovery of the fractal transform in 1988 was preceded by B. Mandelbrot's development of fractal geometry. It is true that mathematicians knew about some of the basic elements of fractal geometry during the period from 1875 to 1925, but they thought that this knowledge deserved little attention [7]. Mandelbrot, a mathematician at IBM corporation, was the man who pioneered this field of mathematics in depth in the 1960's. His first publication on fractal theory was in 1975 [7].

Fractal shapes occur universally in the natural world. Mandelbrot recognized them in coastlines, lungs, landscapes, turbulent water flow, and even in the chaotic fluctuation of prices on the Chicago commodity exchange [7].

With classical Euclidean shapes only two parameters, length and position, are needed to describe the shape; whereas, fractals require three parameters: "complicated

structure on a wide range of scales, repetition of structures at different length scales (self-similarity), and a 'fractal dimension' that is not an integer [8]." Self-similarity is found in sets or shapes that have repetitive patterns on smaller scales. A line and a square are two Euclidean shapes that are self-similar. Enlarging and replicating the line can produce the square, just as reducing the square can form the line [7]. In this case, the line has dimension 1, and the square has dimension 2. It seems possible that a form between a line and a square, say a jagged line, can have a dimension between 1 and 2 .

From observing self-similar forms such as these, Felix Hausdorff and A.S. Besicovitch discovered a way to measure the dimension of a fractal [13]. To measure the fractal dimension of a bounded set $S \in \mathbf{R}^m$, one must lay a grid of *m*-dimensional boxes, of side-length ε, to cover *S* [8]. Let *N* be the number of boxes of the grid that intersect *S*. Then *S* has a box-counting dimension

$$D = \lim_{\varepsilon \to 0} \left(\frac{\log N}{\log\left(\frac{1}{\varepsilon}\right)} \right),$$

when the limit exists [8]. This definition of dimension holds true for integer dimensions as well as for fractal dimensions.

Since Mandlebrot's success in making the research of fractals and their applications popular, many people have learned to create fractal illustrations. Today these beautiful images can be generated easily on a personal computer, and have spawned a popular field of computer graphics art.

In the 1980's Michael F. Barnsley realized that the theory of fractals could be applied toward image compression. In particular, an optimal compression algorithm is sought for compressing images. This is desired in order to save storage space, as well as time. Depending on the purpose of the image, one would either want to store exactly the

same data as in the original image (this is called lossless coding) or a compressed version of the data (referred to as lossy compression). For instance, one may wish to store a medical image without loss of data. However, media conferencing requires quick "real time" transmission for images, demanding some form of compression.

Rather than storing an image pixel by pixel, the goal of fractal image compression is to find a lossy compression algorithm that takes advantage of the self-similarities in an image. Barnsley applied his knowledge of fractals and mathematics to image compression, creating an optimal forms of image compression comparable to JPEG, a form of compression which is widely used today.

2. Judging Criteria

As mentioned before, image compression is desired for storage or transmission in order to reduce the large size of most image files. There are two criteria by which image compression methods are judged [9]. One is the time needed to accomplish compression and decompression, and the second is the degree of preservation of the image. Obviously, once the image is compressed in some way, the image must be reproducible from the compressed form. We will discuss the preservation of the image.

Lossless techniques allow exact reconstruction of each individual pixel value. This method is sometimes referred to as image coding, rather than image compression. One of the early approaches of image coding is called delta compression. Rather than storing the actual pixel values, the difference in values of a pixel and its neighbor is stored. Usually there is little change in an area of a picture, so most of the difference values are close to zero. Since the magnitudes of the differences are much smaller than

the actual magnitudes of the pixel values, this calls for less storage space. Other forms of lossy compression are Huffman Coding and GIF, which will be discussed later.

Lossy schemes attain compression by discarding unimportant data. Of course it is possible to discard data that is crucial to the quality of the image, but this is not a desirable practice. Lossy methods use an algorithm to determine which part of the image data is unnecessary, and which data is essential to the clarity of the image. We have disposed if the data, it is not retrievable. This is where data is lost and compression is achieved. After discarding data, it is common to use a lossless coding method at this point to compress the existing data even more. Upon decoding and decompression, the exact data is not regenerated therefore the final image is not exactly the same as the original image, but it closely resembles it. We will cover Fourier transform, Cosine transform, JPEG compression, and the Fractal transform as examples of lossy methods.

II. Standard Types of Compression

1. Lossless Coding

Binary coding is the basis for data storage in most machines today but is not the most efficient form of coding. Although binary coding is lossless, there are other coding schemes, such as GIF and Huffman coding, which are more efficient. GIF compression is a lossless compression algorithm. This algorithm's performance is based on how many repetitions are present in an image. If the program comes across parts of an image that are the same, say some repeating sequence, it assigns that sequence a value and stores this assignment in a hash table, or a key. This hash table is then attached to the image so the decoding program can descramble it. The disadvantage to compression with a GIF is

that the amount of compression achieved is dependent on how much repetition is in the image. It is also limited to a palette of at most 256 colors [10].

Huffman coding is a widely used variable-length coding scheme [9]. This algorithm searches for the different frequencies that gray values occur throughout the image. Then it assigns a code to each value, short codes for high frequency values, and long codes for low frequency values. This process can also be applied to the difference in pixel values. In this case, more compression can be attained.

For an example, consider an image in which the pixels (or their difference values) can have one of a possible 8 brightness values [9]. This would require 3 bits per pixel $(2^3 = 8)$ for traditional representation. It is possible to produce a histogram of the image describing the frequencies for which each brightness value occurs. Huffman coding provides instantaneous codes, or codes in which no code words occur as a prefix to another [2]. This makes the decoding process efficient. The codes that are chosen can be generated by a Huffman tree, which depends on the relative probabilities of each value. A Huffman tree is formed by progressively gluing smaller trees together, until a big tree is formed. Barnsley summarizes the steps to producing a Huffman code as the following [2].

Huffman Code Steps

Step 1. List the symbols in order of probability.

Step 2. Make a tree whose branches, labeled zero and one, are the two symbols with lowest weight.

Step 3. Remove the two symbols just used from the list and add to the list a

new symbol representing the newly formed tree with probability equal to the total weight of the branches.

Step 4. Make a tree whose branches, labeled zero and one, are the two symbols with lowest weight in the new list. This tree may consist of two other symbols, or it could consist of a symbol and the tree just constructed.

Step 5. Repeat this procedure until one large tree is formed

Table 1 contains a list of possible brightness values (these are the symbols referred to in Barnsley's steps) with their probabilities of occurrence in an image. Figure 1 shows the Huffman tree constructed after following the Huffman steps. The third column in Table 1 lists the codewords for the brightness values that occur in the image. These codewords are determined by the Huffman tree. It is important to note that these codes are not unique. For each two branch tree, it is possible to interchange the zero and one assigned to each branch. This creates many possibilities for codes, but it is true that a Huffman code will minimize the number of bits needed for storage.

Brightness Value	Frequency	Huffman code
3	.47	0
5	.19	100
4	.13	110
6	.08	111
2	.07	1010
7	.03	10111
1	.02	101100
0	.01	101101

Table 1. Huffman code assigned to brightness values of an image.

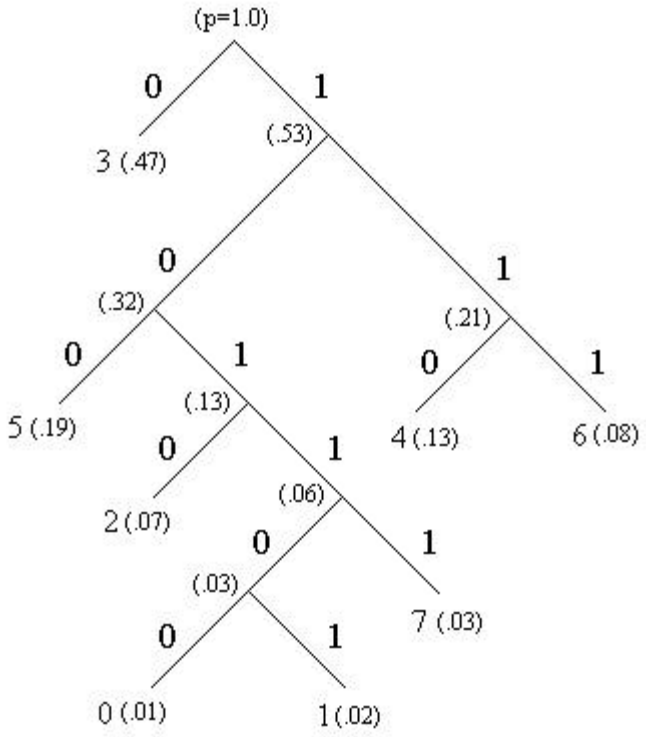

Figure 1. Huffman Tree used to generate Huffman code for Table 1.

Now that the brightness values have codes, we can analyze the effectiveness of this binary representation. Only a single bit is required for the most common pixel brightness value. The less common values have longer codes. To find the average codeword length for this code we must multiply the frequencies by their corresponding code lengths, and sum the resulting products.

$$l_{av} = 1(.47) + 3(.13 + .19 + .08) + 4(.07) + 5(.03) + 6(.01 + .02)$$

$$= 2.28 \text{ bits/pixel},$$

which is better than the 3 bits/pixel with which we started. Huffman trees are not unique, To further this study on Huffman codes see [2].

2. Delta Compression

Delta Compression was a simple early approach of reducing the number of bits per pixel. This method of data compression, along with other early lossless methods of data compression, was an attempt to transmit images of space collected from space probes [9]. Because of the low power transmitters, the communication bandwidth did not allow images to be sent unless some method was used to reduce the number of bits per pixel [9].

The basic approach to Delta Compression is to take the difference in neighboring pixel values. Given some image with a high magnitude average in pixel values, it is safe to assume that in most cases the average change in pixel values is small from one pixel to the next. This guarantees that after the Delta Compression has been applied, the magnitudes of the differences are smaller than the original pixel values. Thus, a smaller number of bits per pixel are required to store the image.

For instance, let two neighboring pixels in a particular image have the values of 167 and 165. The first pixel alone requires 8 bits to store the value, whereas, their difference value equals 2, which only requires 1 bit to be stored. Overall, if this concept is applied to the whole image, the magnitudes of each entry of the matrix representing the Delta Compression of the image will be much smaller. Thus the average number of bits per pixel needed to store the image will be smaller. Before compressing the image, some of the original data will need to be stored in order to regenerate exactly the original image when decompression takes place. There are different algorithms one could implement in order to achieve this compression.

The following is a description of the Delta Compression program 'deltacomp2d' included in Appendix A. When compressing the original image, which is represented by an $N \times M$ matrix X, the first value $X(1,1)$ should be stored as some variable, say '*first*'. In Figure 2, the first entry of X is 176, so *first* = 176. The (r,c) entries of Dx, the matrix representing the Delta Compression of an image, for $r = 1,2,\ldots N$ and $c = 1,2,\ldots M\text{-}1$ are obtained by the difference $X(r,c) - X(r,c+1)$. A special case occurs when $c = M$ (the last column in the matrix). When $c = M$, $Dx(r,\text{M}) = X(r,\text{M}) - X(r+1,1)$. In words: the last entry in a row r of Dx equals the last entry in the row r of X minus the first entry in the row $r + 1$ of X. The last entry of Dx is set equal to zero. Once all entries of Dx have been created, Dx and *first* are stored in a compressed file. It is possible to apply another lossless scheme at this point, such as Huffman Coding, which would compress this file even more.

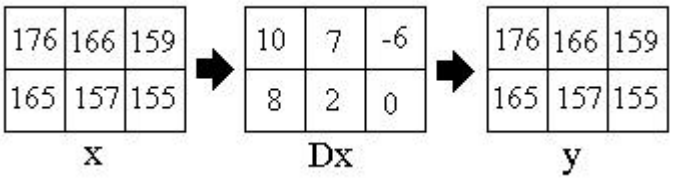

Figure 2. Example of applying Delta Compression and regenerating original data.

For the Delta Decompression program named 'deltade2d', found in Appendix A, the matrix Dx represents the compressed matrix and *first* represents the initial entry in the original image, as above. Y represents the reconstructed original image. Since this is a lossless compression method Y will be equal to X, the original image before Delta Compression is applied. Y is reconstructed, first by setting the first entry $Y(1,1) = first$. Then for $r = 1\ldots N$ and $c = 2\ldots M$, $Y(r,c) = Y(r,c\text{-}1) - Dx(r,c\text{-}1)$. But there is a special case when $c = 1$ and $r > 1$, $Y(r,1) = Y(r\text{-}1,\text{M}) - Dx(r\text{-}1,\text{M})$. When Y reaches the first entry of

a new row, $Y(r,1)$, the first entry in that row is created by subtracting the last entry of Dx ,in the r-1 row, from the last entry Y, in the r-1 row. As desired, the original image is reconstructed exactly. See Figure 2 for a numerical example.

In Appendix B, Figures 1 through 12 correspond to Delta Compression results. Figures 2, 6, and 10 are three different images to which we applied Delta Compression. The histograms of these images are to their left in Figures 1, 5, and 9. A histogram shows the distribution of data values. In this case, the pixel value, on a range of [0,255] is on the independent axis and the frequency of the pixel values within the image is on the dependent axis. It is important to notice that the pixel values in the original images are spread throughout the range of [0,255]. The histograms of the delta compressed images are in Figures 3, 7, and 11. After the images have been delta compressed, the pixel values are closer to zero than in the original image. As mentioned previously, on average, the smaller magnitudes of pixel values are what allow the data to be stored in fewer bits. The images in Figures 4, 8, and 12 represent what type of image the data in the delta compressed files would resemble. The more uniform gray color graphically displays that the pixel values are close in magnitude.

3. Fourier Transform

Our goal in using Fourier transforms is to determine and work with the spatial frequency content of an image. Images are treated as two-dimensional discrete finite signals. The Fourier transform $\hat{f}$ of a two-dimensional signal f has a matrix representation and contains the amplitudes of the fundamental frequencies that make up f. Each component of $\hat{f}$ indicates the strength of a particular frequency in f. Once the

Fourier transform is applied to an image, to achieve compression of the storage of that image, it is necessary to quantize it. Quantizing is a rounding procedure in which the high frequencies are omitted. Ideally, many entries in the matrix will be set to zero. By doing this, less storage space is needed to represent the image. The image is decompressed by application of the inverse transform. The resulting image matrix will have values close to the corresponding entries in the original image. This is how it is possible for the new image to resemble the original image. Before the Fourier Transform is discussed in detail, we will introduce some fundamental mathematics.

3.1 Periodic functions

A function $f(t)$ is called periodic if there exists a constant $T > 0$ for which $f(t+T) = f(t)$, for any t in the domain of definition of $f(t)$, where t and $t + T$ lie in this domain. The smallest such T is called the period of $f(t)$ [11]. There are many periodic functions, $\sin t$, $\cos t$, and $\tan t$ being some of the most well known periodic functions [11]. If we were to plot a periodic function on some interval $a \le t \le a+T$, we would obtain the entire graph by periodic repetition of the portion of the graph corresponding to $a \le t \le a+T$ [11]. If T is the period of $f(t)$, then any integer multiple of T, say kT, where k is any positive integer, is also a period of $f(t)$ [11].

Consider the sine wave, $t \to A\sin wt$, where $t \in \mathbf{R}$. $|A|$ represents the amplitude of the wave, $2p/w$ represents the period, and $w/2p$ represents the frequency. With this information, the sine wave can be constructed for all time t, whereas most signals we observe in practice have a finite duration. But, this does not cause a problem because any

finite length signal can be extended periodically for all time [12]. In analyzing signals more complex than the sine wave it is convenient to use the most fundamental map of an oscillation: $\theta \to e^{i\theta}$, where $\theta \in \mathbf{R}$. Using Euler's identity, $e^{i\theta} = \cos\theta + i\sin\theta$, we see that $\theta \to e^{i\theta}$ maps any interval of length 2π to the unit circle. This is the basis of all Classical Fourier Analysis [12]. If we let $\theta = \omega t$, where $\omega \in \mathbf{R}$, we obtain in the map $t \to e^{i\omega t}$, $t \in \mathbf{R}$. This map is a rotation about the unit circle that completes one revolution of (2π radians) in period $T = 2\pi/|\omega|$ [12]. An oscillation with frequency $f = \omega/2\pi$ can be described by $t \to e^{2\pi i f t}$ [12].

For each integer *n*, we set $\omega_n = 2\pi n/T = 2\pi f_n$. This map,

$$t \to e^{i\omega_n t} \to e^{2\pi i f_n t},\ t \in \mathbf{R},$$

completes *n* rotations around the unit circle (counterclockwise if *n* is positive and clockwise if *n* is negative) during the time interval $0 \le t \le T$ [12]. Each of these maps give one basis signal for each integer *n* [12]. The collection of these basis signals is the set

$$A = \{t \to e^{2\pi i f_n t} \mid n \in \mathbf{Z}\},$$

which contains the information needed to generate other period *T* signals [12].

Let *B* be the span of *A*:

$$B = \{t \to \sum_{n \in \mathbf{Z}} a_n e^{2\pi i f_n t} \mid a_n \in \mathbf{C}\}.$$

The sequence of coefficients a_n is called the Fourier transform of a signal from *B*.

3.2 One-Dimensional Fourier Transform

The Fourier transform of a signal is a vector containing the amplitudes of the fundamental frequencies that make up the signal [12]. If a continuous signal *h* is defined as

$$h(t) = \sum_{n \in \mathbf{Z}} a_n e^{2\pi i f_n t},\ 0 \le t \le T,$$

then the sequence of coefficients a_n are represented as

$$a_n = \frac{1}{T}\int_0^T h(t) e^{-2\pi i f_n t}\, dt,$$

which allows each a_n to be computed from *h* [12].

Proposition: If $h(t) = \sum_{n \in \mathbf{Z}} a_n e^{2\pi i f_n t},\ 0 \le t \le T$, then $a_n = \frac{1}{T}\int_0^T h(t) e^{-2\pi i f_n t}\, dt$.

Proof: First, we need to show that the functions $e^{2\pi i f_n t}$ are orthogonal over the interval $0 \le t \le T$, i.e.

$$\int_0^T e^{2\pi i f_n t} e^{-2\pi i f_m t}\, dt = \int_0^T e^{2\pi i t (f_n - f_m)}\, dt = \begin{cases} T & \text{if } m = n \\ 0 & \text{if } m \ne n, \end{cases}$$

where *n* and *m* are integers. There are two cases for which we need to evaluate this integral.

Case 1 $m = n$

$$\int_0^T e^{2\pi i t (f_n - f_n)}\, dt = \int_0^T dt = T$$

Case 2 $m \ne n$

$$\int_0^T e^{2\pi i t(f_n - f_n)} dt = \frac{e^{2\pi i t(f_n - f_m)}}{2\pi i(f_n - f_m)}\Bigg|_0^T = \frac{e^{2\pi i T(f_n - f_m)} - 1}{2\pi i(f_n - f_m)}$$

$$= \frac{\cos(2\pi(n-m)) + i\sin(2\pi(n-m)) - 1}{2\pi i(f_n - f_m)} = 0.$$

Now, that we know the set of basis signals is orthogonal, we can find the coefficients. Our signal *h* is defined as:

$$h(t) = \sum_{n \in \mathbf{Z}} a_n e^{2\pi i f_n t}.$$

If we multiply both sides of the above equation by $e^{-2\pi i f_m t}$, we have

$$e^{-2\pi i f_m t} h(t) = \sum_{n \in \mathbf{Z}} a_n e^{2\pi i f_n t} e^{-2\pi i f_m t}.$$

Integrating on both sides from $0 \le t \le T$,

$$\int_0^T e^{-2\pi i f_m t} h(t) dt = \int_0^T \sum_{n \in \mathbf{Z}} a_n e^{2\pi i f_n t} e^{-2\pi i f_m t} dt,$$

$$= \sum_{n \in \mathbf{Z}} a_n \int_0^T e^{2\pi i f_n t} e^{-2\pi i f_m t} dt$$

$$= T a_m,$$

by orthogonality. Therefore,

$$a_m = \frac{1}{T} \int_0^T h(t) e^{-2\pi i f_m t} dt,$$

as desired. QED.

Although knowing how to find the Fourier Coefficients of a signal *h* for all values of *t* in a continuous time interval $0 \le t \le T$ is useful in many engineering applications, our interest is in finding the Fourier coefficients of an image, a discrete signal. In fact, the image signal is not a function of time, but of position. So, in the position interval $0 \le x \le X$, where *x* denotes position and *X* represents the period of the signal, a discrete signal is sampled at discrete positions x_k, $k = 0, \ldots, N$ with $0 \le x_k \le X$ [12]. It is common and easier to keep the sampling interval constant and the period *X* as some multiple of the sampling interval [12]. The result is the sampling positions x_k are equidistant between 0 and *X*. If *N* samples are taken, the sampling interval is *X*/*N* and the sample positions are 0, *X*/*N*, …, (*N*-1)*X*/*N* [12]. If we replace the continuous signal *h*, with a step function approximation $\tilde{h}$ due to the discrete sampling intervals, then we end up with the sampling intervals $kX/N \le x \le (k+1)X/N$, $k = 0, \ldots, N\text{-}1$. So, the sampling positions are $x_k = kX/N$ and the signal can be written as

$$\tilde{h}(x_k) = h(kX/N) = \sum_{n=0}^{N-1} a_n e^{2\pi i f_n kX/N} = \sum_{n=0}^{N-1} a_n e^{2\pi i nk/N},$$

since $f_n = n/X$. The finite sum for the discrete signal can be written as

$$h_k = h(x_k) = \sum_{n=0}^{N-1} b_n e^{2\pi i nk/N}. \qquad (1)$$

Proposition: If $h_k = \sum_{n=0}^{N-1} b_n e^{2\pi i nk/N}$, $0 \le x_k \le X$, then $b_n = \sum_{k=0}^{N-1} h_k e^{-2\pi i nk/N}$.

Proof: First we need to show that the discrete functions $e^{2\pi i nk/N}$ are orthogonal over the interval $0 \le x_k \le X$, i.e.

$$\sum_{k=0}^{N-1} e^{2\pi ink/N} e^{-2\pi imk/N} = \begin{cases} N, \; if \;\; m = n, \\ 0, \;\; if \;\; m \neq n, \end{cases} \qquad (2)$$

where *m* and *n* are integers. We must consider two cases.

Case 1: $m = n$

$$\sum_{k=0}^{N-1} e^{2\pi ink/N} e^{-2\pi imk/N} = \sum_{k=0}^{N-1} [e^{2\pi i(n-m)/N}]^k = \sum_{k=0}^{N-1} 1 = N .$$

Case 2: $m \neq n$

Knowing how to sum a finite geometric series, we find

$$\sum_{k=0}^{N-1} e^{2\pi ink/N} e^{-2\pi imk/N} = \sum_{k=0}^{N-1} [e^{2\pi i(n-m)/N}]^k$$

$$= \frac{1 - [e^{2\pi i(m-n)/N}]^N}{1 - e^{2\pi i(m-n)/N}}$$

$$= \frac{1 - e^{2\pi i(m-n)}}{1 - e^{2\pi i(m-n)/N}} .$$

$$= \frac{1 - \cos(2\pi(m-n)) + i\sin(2\pi(m-n))}{1 - e^{2\pi i(m-n)/N}} = 0 .$$

Multiplying both sides of (1) by $e^{-2\pi imk/N}$, then summing from 0 to *N*-1, we have

$$\sum_{k=0}^{N-1} h_k e^{-2\pi imk/N} = \sum_{k=0}^{N-1} \left(\sum_{n=0}^{N-1} b_n e^{2\pi ink/N} \right) e^{-2\pi imk/N}$$

$$= \sum_{n=0}^{N-1} b_n \sum_{k=0}^{N-1} e^{2\pi i(n-m)k/N} = \sum_{n=0}^{N-1} b_n \delta_{n,m} N = N b_m .$$

So,

$$b_m = \frac{1}{N} \sum_{k=1}^{N-1} h_k e^{-2\pi imk/N} \qquad m = 0, \ldots, N-1,$$

where h_k are components of **h**, and b_m are components of **b**. QED.

Equation (1) defines $\mathbf{h} \to \mathbf{b}$, a map $\mathbf{C}^N \to \mathbf{C}^N$ [12]. We make the transformation symmetric by defining $\hat{\mathbf{h}} = \sqrt{N}\mathbf{b}$. This gives the discrete Fourier transform [12]

$$\begin{aligned} \hat{\mathbf{h}}_j &= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \mathbf{h}_k e^{-2\pi i k j / N}, j = 0,...,N-1, \\ \mathbf{h}_k &= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \hat{\mathbf{h}}_j e^{2\pi i k j / N}, k = 0,...,N-1. \end{aligned} \tag{3}$$

The vector $\hat{\mathbf{h}} = (\hat{h}_0, \ldots, \hat{h}_{N-1})$ is called the Fourier transform of $\mathbf{h} = (h_0, \ldots, h_{N-1})$, while each vector $\mathbf{h}$ and $\hat{\mathbf{h}}$ are the inverses of each other. So, given either $\mathbf{h}$ or $\hat{\mathbf{h}}$, we can compute the other [12].

Definition 1 A *Transformation* $T : \mathbf{C}^N \to \mathbf{C}^M$ is a rule that assigns to each vector $\mathbf{x}$ in $\mathbf{C}^N$ a vector T($\mathbf{x}$) in $\mathbf{C}^M$. For each $\mathbf{x}$ in $\mathbf{C}^N$, $T : \mathbf{x} \to A\mathbf{x}$ where A is an $m \times n$ matrix.

The defined transformations in (3) can be viewed as transformations of the vectors $\mathbf{h}, \hat{\mathbf{h}} \in \mathbf{C}^N$, so each of them have a matrix representation. If we let W be the $n \times n$ matrix whose entry in the jth row and kth column is $W_{j,k} = (1/\sqrt{N}) e^{2\pi i j k / N}$, then we have [12]

$$\begin{aligned} \hat{\mathbf{h}} &= \overline{W}\mathbf{h} \\ \mathbf{h} &= W\hat{\mathbf{h}}, \end{aligned}$$

where $\overline{W}$ stands for the matrix whose entries are the complex conjugate of the entries of W. It is clear that $\overline{W} = W^{-1}$, since $\hat{\mathbf{h}} = \overline{W}\mathbf{h} = \overline{W}W\hat{\mathbf{h}}$ for all $\hat{\mathbf{h}} \in \mathbf{C}^N$ and $\mathbf{h} = W\hat{\mathbf{h}} = W\overline{W}\mathbf{h}$ for all $\mathbf{h} \in \mathbf{C}^N$ [12]. This set of equations can be rewritten in compact form as

$$\hat{\mathbf{h}}_j = \sum_{k=0}^{N-1} \mathbf{h}_k \overline{W_k}$$
$$\mathbf{h}_k = \sum_{j=0}^{N-1} \hat{\mathbf{h}}_j W_j .$$

An example of a vector **h** and its Fourier transform $\hat{\mathbf{h}}$ are given by

$$\mathbf{h} = \begin{bmatrix} 4.5 \\ 7.8 \\ 3.9 \\ 8.3 \end{bmatrix}, \qquad \hat{\mathbf{h}} = \begin{bmatrix} 12.25 \\ .3-.25i \\ -3.85 \\ .3+.25i \end{bmatrix}.$$

See Appendix A for two functions written in Matlab to perform the Fourier transform as well as the inverse Fourier transform on a one dimensional signal.

3.3 Two-Dimensional Fourier Transform

It is possible to extend the Fourier transform to a two-dimensional signal. The two-dimensional transform takes the form

$$\hat{f}_{uv} = \frac{1}{N} \sum_{j,k=0}^{N-1} f_{jk} e^{-2\pi i(ju-kv)/N}$$
$$f_{jk} = \frac{1}{N} \sum_{u,v=0}^{N-1} \hat{f}_{uv} e^{2\pi i(ju-kv)/N}, \qquad (4)$$

defined for any pairs of integers *u* and *v* or *j* and *k* in the interval 0, …, *N* - 1 [12]. The right hand sides of (4) extend definitions of *f* and $\hat{f}$ to all of $\mathbf{Z}\times\mathbf{Z} = \mathbf{Z}^2$ and the extension is periodic with period N in both directions [12]. The two-dimensional Fourier Transform sees both *f* and $\hat{f}$ as maps from $\mathbf{Z}^2$ to **C**, with the property that $\hat{f}(u+N,v) = \hat{f}(u,v+N) = \hat{f}(u,v)$ and $f(j+N,k) = f(j,k+N) = f(j,k)$ for any (u,v)

and (*j*,*k*) in $\mathbf{Z}^2$ [12]. If we let $W_{uj} = \frac{1}{\sqrt{N}} e^{2\pi iju/N}$ then we can rewrite the two-dimensional transform in matrix notation as

$$\hat{f} = \overline{W} f W$$
$$f = W \hat{f} \overline{W}.$$

In application, it is practical to use the matrix notation. An example of a matrix representation of f and its Fourier transform $\hat{f}$:

$$f = \begin{bmatrix} 21.4 & 53.1 & 34 \\ 23 & 40.6 & 11 \\ 30.3 & 32 & 31.5 \end{bmatrix} \quad \hat{f} = \begin{bmatrix} 92.3 & -8.8+14.2i & -8.8-14.2i \\ 8.1+5.5i & 0.6+1.6i & -13.9-0.7 \\ 8.1-5.5i & -13.9+0.7 & 0.6-1.6i \end{bmatrix}.$$

See Appendix A for two functions written in Matlab to perform the Fourier transform as well as the inverse Fourier transform on two dimensional signals.

3.4 Application of Two-Dimensional Fourier Transform

Our goal in using the Fourier transform (FT) would be to apply it in some way to compress an image. But, in fact, because the FT contains complex entries, compression is difficult to reach, although possible. The approach one would take to reach compression would be to quantize the high frequencies of the FT. Quantizing is the a "rounding" procedure which reduces the magnitudes of transformed coefficients. Typically one forces a bigger reduction on the high frequency components [12]. After quantizing, the entries in the matrix are ordered from low to high frequency, trailing zeros are truncated, and the resulting string is encoded using some lossless algorithm such as a Huffman code, or a binary code. Up to this point the FT has been a lossless algorithm considering that the original finite signal can always be reproduced. But, once quantizing takes place,

the original signal can never be reproduced exactly from the quantized FT, resulting in lossy compression. Quantizing will be discussed in more detail in Section 5 of this chapter.

We run into problems when attempting to use the FT as a compression method for images. The program named 'FourierComp' in Appendix A is an example of an attempt to compress images using the FT accompanied with a quantizing algorithm as a compression method. So, each pixel value is stored in 8 bits. We apply 'FourierComp' to three grayscale images (sisters, peppers, and mandrill) holding the pixel values of [0,255]. These images are displayed in Appendix B. Figures 13, 15, and 17 are the original images, and Figures 14, 16, and 18 are the images after 'FourierComp' has been applied to the original images. The following computations correspond to the application of 'FourierComp' to the image "peppers.pgm" (256 x 256 pixels), a 65 KB image . Let the matrix of pixel values of peppers be named *f*. Let the FT of f be denoted as $\hat{f}$.

In this case the quantizing algorithm defines a high frequency by determining if the ratio of the modulus of an entry in $\hat{f}$ to the mean of the modulus of the entries in $\hat{f}$ is less than 1. In other words, $\frac{\left|\hat{f}(i,j)\right|}{mean(|\hat{f}|)} \leq 1$ for $i, j \in \mathbf{Z}$. For higher frequencies the magnitude of the entry of $\hat{f}$ is smaller. Therefore, this ratio is a good test in comparing the frequencies. The entries that pass this test are set equal to zero. It is possible to change the bound, on the right side of the inequality. If the bound is made smaller, less entries in $\hat{f}$ will be quantized, which will result in less possible compression. We will see that using a bound of 1 works well.

After quantizing, 53441 entries of 65536 are set to zero due to $\hat{f}$ having high frequencies. This new sparse matrix is called $\hat{y}$. When the inverse FT is applied to $\hat{y}$ a new image, *y*, is created. The image quality of *y* is not as good as *f*, but we are willing to sacrifice some quality if the compression is good. Now we must determine whether the compression of *f* is efficient. We know that there are 53441 zeros in $\hat{y}$, which require little or no storage space if a coding scheme is applied after quantizing. Assuming this is the case, the trailing zeros that are truncated will need no storage. If the entries in the matrix are then ordered as a string of numbers, it is possible to find the sub-strings of zeros. When these are found, the only storage necessary will be some variable *a*, denoting the number of zeros in the sub-string, followed by a special character denoting that *a* represents a substring of zeros. So, the storage space needed to store the zeros will depend on the number of substrings. In this case, $\hat{y}$ has 12095 non-zero entries. If these values were also in the range of 0 to 255, we would be in luck. But this is not so, they are complex values.

As we discussed earlier, the Fourier transform of a matrix lives in $\mathbf{C}^N$. So, not only do we need storage space for the real parts of the values, but also for the imaginary parts. Let's assume that storing positive imaginary and real parts requires twice as much space as storing only the real part. The minimum of the real parts of $\hat{y}$ is -1680.9 and the maximum of the real parts is 26600 , whereas the minimum of the imaginary parts of $\hat{y}$ is -1827.5 and the maximum is 1827.5. Therefore, assuming all 12095 non-zero entries of $\hat{y}$ have both real and imaginary parts, with a sign value plus a maximum magnitude of

26600 for the real parts and a maximum magnitude of 1827.5 for the imaginary parts, we can estimate the storage space needed if the matrix $\hat{y}$ were converted to a binary file.

Since $2^{10} = 1024$, $2^{11} = 2048$, $2^{14} = 16384$ and $2^{15} = 32768$, we know that the maximum bits per pixel needed to store the magnitude of the real components is 15 bits plus one bit for the sign, and the maximum bits per pixel needed to store the magnitude of the imaginary part is 11 bits per pixel. There are 12095 pixels, so the maximum estimate of bits needed to store this matrix $\hat{y}$ is $(12095)(15+11+2) = 338688\,\text{bits} = 41.3\,\text{KB}$ plus or minus a few bytes for the storage of the compressed strings of zeros.

So the compression ratio for "peppers" is 2:3, not too impressive, but the algorithm used here with the FT and quantizing did compress the image. Because the complex values in the Fourier transform make it so difficult to compress an image, another route should be sought. Results for the sisters image and mandrill image are along the same order.

We are familiar with the fact that $e^{i\theta} = \cos(\theta) + i\sin(\theta)$. A popular approach to more efficient compression is to use just a discrete cosine transform, dropping the imaginary components of the Fourier transform. This results in the Cosine transform (CT). JPEG compression is a form of image compression, which greatly depends on the CT and the concept of quantizing.

4. The Cosine Transform

4.1 The One Dimensional Cosine Transform

Our goal in using the Cosine transform is to avoid the imaginary numbers that are a result of the Fourier transform. To introduce the Cosine transform, we start with a one-dimensional signal **x** defined at positions $k = 0,\ldots,N\text{-}1$. Hence, **x** has period N. We know that $\cos x$ is an even function. An even function is defined as a function f such that $f(x) = f(-x)$. So, we will take the signal **x** and apply an even extension of positions to $k = N,N+1,\ldots,2N\text{-}1$ reflecting its graph across the vertical axis passing through the point $k = N\text{-}1/2$, occurring midway between $k = 0$ and $k = 2\text{N-}1$. The resulting signal is defined for $k = 0, \ldots,2\text{N-}1$ [12]. So, this extension is an even extension centered at $k = N - 1/2$, where the endpoint values match.

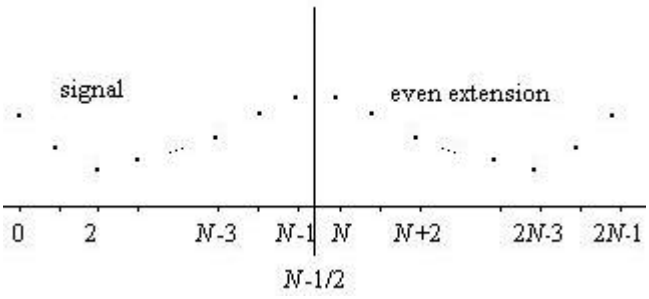

Figure 3. Even extension of a signal about $k = N\text{-}1/2$ [12].

Now, the Fourier Transform sees this signal as having a period $2N$ instead of N. If we apply the Fourier transform to this new period $2N$ signal and then use Euler's identity $2\cos\theta = e^{i\theta} + e^{-i\theta}$ repeatedly, the result is a linear combination of cosine functions instead of exponential functions [12]. The resulting pair of equations are

$$\hat{x}_v = \sum_{k=0}^{N-1} x_k C_v \cos\frac{(2k+1)v\pi}{2N},\ v = 0,\ldots,N-1,$$

$$x_k = \sum_{v=0}^{N-1} \hat{x}_v C_v \cos\frac{(2k+1)v\pi}{2N},\ k = 0,\ldots,N-1,$$

where $C_0 = \sqrt{1/N}$ and $C_k = \sqrt{2/N}$ if $k \neq 0$ [12]. See [12] for details. $\hat{\mathbf{x}}$ is usually referred to as the forward cosine transform, while **x** is referred to as the backward cosine transform [12].

If we define an $N \times N$ matrix A to be

$$A_{k,v} = C_v \cos\frac{(2k+1)v\pi}{2N}, \qquad (4)$$

with k and v defined as above, then the matrix representations of $\hat{\mathbf{x}}$ and **x** are given by

$$\hat{\mathbf{x}} = A^t\mathbf{x}, \qquad (5)$$

$$\mathbf{x} = A\hat{\mathbf{x}}. \qquad (6)$$

The columns A_v are periodic with period $2N$ and the frequency $v/2N$, which increases with the column index v. This orders $\hat{\mathbf{x}}$ with the frequencies of **x** such that $\hat{x}_0$ is the amplitude of A_0, the lowest frequency component of **x**, and $\hat{x}_{N-1}$ is the amplitude of A_{N-1}, the highest frequency in **x** [12]. The consecutive entries of $\hat{\mathbf{x}}$ from the 0th entry to the $(N\text{-}1)^{st}$ entry are the amplitudes of A_v corresponding to the frequencies in **x** from lowest to highest.

It is simple to see that with substituting (6) into (5), $\mathbf{x} = AA^t\mathbf{x}$ for each $\mathbf{x} \in \mathbf{R}^N$, since $A^{-1} = A^t$ and $AA^t = AA^{-1} = I_{N\times N}$ [12]. A matrix of real numbers with this property is known as an orthogonal matrix [12].

4.2 Two Dimensional Cosine Transform

The two dimensional CT is defined as the following

$$\begin{aligned} \hat{f}_{uv} &= \sum_{j,k=0}^{N-1} f_{jk} C_u \cos\frac{(2j+1)u\pi}{2N} C_v \cos\frac{(2k+1)v\pi}{2N} \\ f_{jk} &= \sum_{u,v=0}^{N-1} \hat{f}_{uv} C_u \cos\frac{(2j+1)u\pi}{2N} C_v \cos\frac{(2k+1)v\pi}{2N}, \end{aligned} \tag{5}$$

where f and $\hat{f}$ are extended to two dimensional periodic signals with period $2N$ defined on $\mathbf{Z}^2$. The extension given for f is even in that it extends f in both horizontal and vertical directions, with the extension usually smoother than that provided by the two-dimensional Fourier transform. One can visualize the extension of the two-dimensional signal by picturing f as an array and reflecting f over its four boundaries. Four new arrays have been created. Continue reflecting each of the new arrays created across their boundaries. Eventually $\mathbf{Z}^2$ will be tilled with these arrays, and this yields the signal that the two-dimensional CT regards as f.

If we define B_{uv} to be the $N \times N$ basis element whose entry in the jth row and kth column is

$$B_{uv}(j,k) = C_u \cos\frac{(2j+1)u\pi}{2N} C_v \cos\frac{(2k+1)v\pi}{2N},$$

then we can redefine f and $\hat{f}$ as the following:

$$\begin{aligned} \hat{f}_{uv} &= \sum_{j,k=0}^{N-1} f_{jk} B_{jk}(u,v) \\ f_{jk} &= \sum_{u,v=0}^{N-1} \hat{f}_{uv} B_{uv}(j,k). \end{aligned}$$

Furthermore, the matrix representation of the CT is

$$\hat{f} = A^t f A \tag{6}$$

$$f = A\hat{f}A^t, \tag{7}$$

where A is the same here as in the one-dimensional case. As expected, when (7) is substituted into (6) we arrive at $f = A(A^t \hat{f} A)A^t$, since $A^{-1} = A^t$.

5. JPEG Image Compression

JPEG stands for Joint Photographers Expert Group. In the 1980's this group was formed to determine standards for still-image compression including both lossless and lossy modes [12]. The acceptable solutions of this compression problem are based on the human visual system and the fact that the human eye is insensitive to certain changes in an image, and tolerant of a wide range of approximations [12]. The lossy modes are of more interest at this point. There are many forms of JPEG compression depending on what quality of compression is desired. The form of data being compressed will determine whether or not a compression mode should be lossless or lossy [12]. There are many instances in which data contains text. In most cases, the text cannot be sacrificed. But in some cases, where the data is an image, a loss of information is acceptable due to the compression that accompanies it. We will consider only ideas of lossy compression applied to gray scale images. The range of gray scale values is commonly restricted to [0, 255].

The JPEG compression takes advantage of the fact that a loss of data is acceptable. The main mathematical and physical theme of JPEG is local approximation. One step in the JPEG algorithm that reduces data is collapsing almost constant regions to their average shade of gray. One can choose a subset of an image, say a block of 8×8

pixels, average the shade in that block, and do the same for every disconnected 8×8 block in the image. For an array of 256×256 pixels, this reduces the image to a 32×32 array. Although this new array is 1/64 the size of the original array, with this algorithm too much detail is lost. It is desirable to have an algorithm that would not average blocks containing large amounts of detail. It is possible to choose a smaller block size, such as 4×4, but this may sacrifice any compression gain.

Rather than using a 4×4 block size and running the risk of losing valuable compression, JPEG uses a "detail detector", which happens to be the two-dimensional cosine transform (5) in section 4.2 [12]. In the CT, the sum has been ordered so that the "tail" contains the high frequency components of the signal. Stopping the sum at a certain point is the same as truncating high frequencies from the original block, and is equivalent to replacing the appropriate entries in the transformed matrix with zeros [12]. Retaining only the nonzero coefficients and discarding the trailing zeros corresponds to a compression method and can be considered a special case of JPEG [12].

There are basically four steps in the JPEG algorithm. First it is necessary to break the $M\times N$ image into local blocks, most popularly into 8×8 blocks, as discussed earlier. Second, these blocks need to be transformed, using the cosine transform, in order to identify the high frequency components. The cosine transform exchanges raw spatial information for information on frequency content. Then a quantizing method, or a "rounding" procedure, needs to be applied to the transformed coefficients. The high frequencies are usually reduced considering the human eye is insensitive to high frequencies. The fourth step is encoding the output of the quantizing step. The JPEG standard uses Huffman and arithmetic coding [12].

If we are dealing with a 256×256 image, call it *f*, each 8×8 block occupies only .098% of the image area. When small blocks are processed through the two-dimensional cosine transform, one at a time, it is not possible for the transform to take the entire image into account. This results in discontinuities across the block boundaries. Because there is no overlapping of the 8×8 blocks in the whole image, the edges of the transformed blocks are bound to have discontinuities from one block to the next after being decompressed. The JPEG group found the Cosine Transform to have desirable "smoothing" properties, which the other Fourier Transforms did not have.

These properties allow for a quantizer matrix, which we denote as *q*. It is at the quantizing stage that the JPEG looses information. This step, unlike the others, is not invertible. For each 8×8 block in $T(f)$, the transformed matrix, there is a corresponding 8×8 block in the *q* matrix whose entries are all positive integers, referred to as quantizers.

The matrix *q* has the same dimensions as $T(f)$, and each disconnected 8×8 block of quantizers is attached to the corresponding block in $T(f)$. Each entry in $T(f)$ is divided by its corresponding entry in *q* then the result is rounded to the nearest integer [12]. If the quantizer entries are large enough, often the result will be a sparse matrix.

JPEG uses the luminance matrix as a quantizer, in which each entry is based on a visual threshold of its corresponding basis elements [12]. Usually the smaller entries are in the upper left hand corner of *q* while the larger entries are in the lower right hand corner. In the transform matrix, the entries representing the lower frequencies are in the upper left hand corner and the entries representing the higher frequencies are in the lower

right hand corner. When dividing the entries of $T(f)$ by large magnitudes from q the high frequency entries are suppressed.

This design is typical of JPEG quantizers [12]. After quantizing, the entries are ordered from low to high frequencies, the trailing zeros are truncated, and the remaining string of numbers is encoded. The process, without encoding, can be summarized by this chart:

$$f \xrightarrow{transform} Tf \xrightarrow{quantize} QTf \xrightarrow{dequantize} \tilde{Tf} \xrightarrow{invert} \tilde{f},$$

where T is the Cosine transform, defined by $Tf = A^t fA$, and A is an 8×8 block defined by the Cosine transform matrix in equation (4).

With the tradeoff made at the quantizing stage, JPEG compression typically reaches compression ratios of 20:1 or more. Although they eye can notice blockiness in the compressed image, the image quality is not poor. It is possible to obtain better image quality, but compression will be lost. The type of compression selected is often determined by how the image will be used.

III. Fractals and Iterated Function Systems

1. Iterations

Iteration is a process, or set of rules, which one repeatedly applies to an initial state. One could even define an iteration as a repetitive task. In iterating something, there is usually a goal that one is attempting to reach, or an answer sought.

An example of a simple iteration is pressing the space bar on a keyboard. If one aims to type the date at the right hand side of a document, one must repetitively press the

space bar until the desired location has been reached. The goal of this iterative, or repetitive, task is to get to the right hand side of the document.

Applying the square root function to an initial value, and then continually taking the square root of the output, is another example of a simple iteration. Another way of explaining this iterative function is to say that one is continually composing the function with itself. In this example, we will call the initial value x_0. Shown below is the process of iterating the square root function on an initial value $x_0 \in \mathbf{R}+$, where $n = 0,1,2,\ldots$.

$$\begin{aligned}
x_1 &= F(x_0) = \sqrt{x_0} = x_0^{1/2} \\
x_2 &= F(x_1) = \sqrt{x_1} = x_1^{1/2} = \left(x_0^{1/2}\right)^{1/2} = x_0^{1/4} \\
x_3 &= F(x_2) = \sqrt{x_2} = x_0^{1/8} \\
&\vdots \\
x_n &= F(x_{n-1}) = \sqrt{x_{n-1}} = x_0^{1/2^n}.
\end{aligned}$$

The iteration of the rule $x_n = F(x_{n-1}) = \sqrt{x_{n-1}} = x_0^{1/2^n}$ produces the set of outputs, or the orbit, $\{x_0, x_1, x_2, x_3, \ldots, x_n\}$. Depending on the initial value, this orbit could have different behaviors. In this case, if $x_0 = 1$, then the orbit is always 1, which is called a fixed point. If $x_0 > 1$, then the orbit approaches 1, the same fixed point. If $0 < x_0 < 1$, then the orbit will approach zero, another stable fixed point. Lastly, if $x_0 = 0$, then the orbit always equals zero. So, there exist two fixed points of this system, namely 1 and 0.

For different rules or iterative functions, there are more possible behaviors for the orbits. It is possible that an orbit could escape off to infinity, approach or be equal to a periodic orbit, or even be chaotic.

2. The Copy Machine Algorithm

As described by Kominek [5], the metaphor of a Multiple Reduction Copying Machine is an elegant way to introduce Iterated Function Systems. The MRCM is to be understood as a regular copying machine with the exception that the lens arrangements are such that they reduce the size of the original picture, and they overlap copies of the original into the generated copy. Also, the MRCM operates with a feedback loop in which the output of the previous copy is used as the input of the next stage. It doesn't matter with what picture the user begins. What will determine the attractor, or the output of an iterated function system, will be the rules that are used in the copying, which acts as the iteration.

In the example demonstrated in Figure 4 we will produce the Sierpinski Triangle, one of the most well known Iterated Function Systems. To reach this attractor there are three rules which, when composed together, act as the lenses in the copy machine. Each rule, or lens, reduces the original seed by half the original size and translates the new image to a new location.

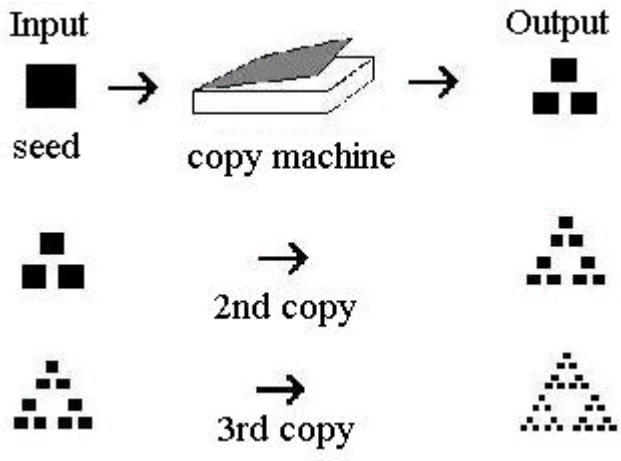

Figure 4. Multiple Reduction Copying Machine

using three rules to produce the Sierpinski Triangle.

It is a surprising fact that the attractor of an IFS does not depend on the seed. So, we could begin with a circle as the seed, and the attractor of the IFS for the Sierpinski Triangle would look the same.

3. Metric Spaces, Mappings, Transformations

To produce Iterated Function Systems, there must exist a space that supports images and on which distances can be measured. For an IFS to converge to an attractor, the mapping that defines this IFS must be a contraction mapping. Having a metric will allow us to measure distances on a given space, as well as determine which are contraction mappings and which are not. Also, having a contraction mapping is an essential ingredient in Fractal Image Compression.

We begin with some definitions.

Definition 2 A *metric space* (X, d) is a set X together with a real-valued function $d : \mathrm{X} \times \mathrm{X} \to R$, which measures the distance between pairs of points *x* and *y* in X.

d is called a *metric* on the space X when it has the following properties [2]:

i. $d(x, y) = d(y, x), \quad \forall x, y \in \mathrm{X}$

ii. $d(x, y) \geq 0, \quad \forall x, y \in \mathrm{X}$

iii. $d(x, y) = 0 \quad iff \quad x = y, \quad \forall x, y \in \mathrm{X}$

iv. $d(x, y) \leq d(x, z) + d(z, y), \quad \forall x, y, z \in \mathrm{X}.$

The Euclidean Plane, $\mathbf{R}^2$, along with the Euclidean metric,

$$d(x,y)=\sqrt{(x_1-y_1)^2+(x_2-y_2)^2},\quad \forall x,y\in \mathbf{R}^2,$$

is an example of a metric space. Another familiar metric space is the Euclidean Plane, $\mathbf{R}^2$, together with the Manhattan metric,

$$d(x,y)=|x_1-y_1|+|x_2-y_2|,\ \forall x,y\in R^2.$$

Definition 3 Let X be a space. A *transformation, map, or mapping* on X is a function $f:\mathrm{X}\to\mathrm{X}$. If $S\subset \mathrm{X}$, then $f(S)=\{f(x):x\in S\}$. The function f is *one-to-one* if $x,y\in \mathrm{X}$ with $f(x)=f(y)$ implies $x=y$. It is *onto* if $f(\mathrm{X})=\mathrm{X}$. It is called *invertible* if it is one-to-one and onto: in this case, it is possible to define a transformation $f^{-1}:\mathrm{X}\to\mathrm{X}$, called the *inverse* of *f*, by $f^{-1}(y)=x$, where $x\in\mathrm{X}$ is the unique point such that $y=f(x)$ [2].

Definition 4 *Affine transformations* on $\mathbf{R}$ are transformations $f:\mathbf{R}\to\mathbf{R}$ of the form $f(x)=ax+b,\forall x\in\mathbf{R}$, where *a* and *b* are real constants [2].

If $a<1$, then this transformation contracts the line toward the origin. If $a>1$, the line is stretched away from the origin. If $a<0$, the line is flipped $180°$ about the origin. The line is translated, or shifted, by an amount *b*. If $b>0$, then the line is shifted to the right. If $b<0$ the line is translated to the left.

We will consider affine transformations on the Euclidean plane. Let $w:\mathbf{R}^2\to\mathbf{R}^2$ be of the form

$$w(x,y)=(ax+by+e,cx+dy+f)=(x',y'),$$

where $a, b, c, d, e,$ and f are real numbers, and (x', y') is the new coordinate point. This transformation is a two-dimensional affine transformation. We can also write this same transformation with the equivalent notations:

$$w(\mathbf{x}) = w\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = A\mathbf{x} + T,$$

where A is a 2×2 real matrix and $T = \begin{pmatrix} e \\ f \end{pmatrix}$ represents translations [2].

The matrix A can always be written in the form of

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} r_1 \cos\theta_1 & -r_2 \sin\theta_2 \\ r_1 \sin\theta_1 & r_2 \cos\theta_2 \end{pmatrix},$$

where (r_1, θ_1) are the polar coordinates of the point (a, c) and $(r_2, (\theta_2 + \frac{\pi}{2}))$ are the polar coordinates of the point (*b, d*) [2]. This means that

$$r_1 = \sqrt{a^2 + c^2}, \quad \tan\theta_1 = \frac{c}{a},$$

$$r_2 = \sqrt{b^2 + d^2}, \quad \tan\theta_2 = \frac{b}{d}.$$

Provided that $ab - cd \neq 0$, we can also describe $A\mathbf{x}$ as a transformation that maps a polygon of area Λ to a new polygon of area $|\det(A)| \cdot \Lambda$ [2].

The different types of transformations that can be made in $\mathbf{R}^2$ are dilations, reflections, translations, rotations, similitudes, and shears.

A *dilation* on (x, y) is written in the form $w_d(x, y) = (r_1 x, r_2 y)$ or

$$w_d(\mathbf{x}) = \begin{pmatrix} r_1 & 0 \\ 0 & r_2 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix},$$

Depending on the values of r_1 and r_2, this dilation could contract or stretch $\mathbf{x}$ [2].

A *reflection* about the x axis can be written in as $w_{rx}(x,y)=(x,-y)$, while a reflection about the y axis is written as $w_{ry}(x,y)=(-x,y)$ [2]. In matrix representation, these reflections would look like

$$w_{rx}(\mathbf{x})=\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix} \text{ and } w_{ry}(\mathbf{x})=\begin{pmatrix}-1 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix}$$

respectively.

Translations can be made in the x or y direction by adding a scalar to the corresponding component of the map [2]. Translations are written in the form $w_t(x,y)=(x+e,y+f)$ or

$$w_t(\mathbf{x})=\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix}+\begin{pmatrix}e\\ f\end{pmatrix}.$$

If $e<0$, the map translates in the negative x direction. If $e>0$, the map translates in the positive x direction. If $f<0$, the map translates in the negative y direction. If $f>0$, the map translates in the positive y direction.

A *rotation* mapping has the form $w_r(x,y)=(x\cos\theta-y\sin\theta, x\sin\theta+y\cos\theta)$, also expressed as

$$w_r(\mathbf{x})=\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix},$$

for some rotation angle θ, $0\le\theta<2\pi$ [2].

A *similitude* is an affine transformation $w:\mathbf{R}^2\to\mathbf{R}^2$ of the form,

$$w_s(\mathbf{x})=\begin{pmatrix}r\cos\theta & -r\sin\theta\\ r\sin\theta & r\cos\theta\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix}+\begin{pmatrix}e\\ f\end{pmatrix},$$

$$w_s(\mathbf{x})=\begin{pmatrix}r\cos\theta & r\sin\theta\\ r\sin\theta & -r\cos\theta\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix}+\begin{pmatrix}e\\ f\end{pmatrix},$$

for some translation $(e, f) \in \mathbf{R}^2$, some real number $r \neq 0$, which is the scale factor, and some angle θ, $0 \leq \theta < 2\pi$ [2]. A similitude combines the rotation, dilation, and translation rules together.

A *shear transformation*, or a *skew transformation*, takes one of the forms,

$$w(\mathbf{x}) = \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \text{ or}$$

$$w(\mathbf{x}) = \begin{pmatrix} 1 & 0 \\ c & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix},$$

where b and c are real constants [2]. In each case, there is one coordinate, which is left unchanged. One can imagine the action of this mapping on some rectangle as if shearing a deck of cards.

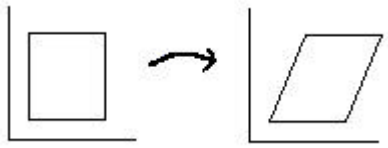

Figure 5. Shearing in the *x* direction: $w(\mathbf{x}) = (x + by, y)$.

4. Convergence and Contractions

In producing Iterated Function Systems, it is necessary to have a set of transformations that converge to a desired image. For these mappings to converge to the desired image, they must be contraction mappings. To apply IFS to Fractal Image Compression we will need the following definitions and theorems.

Definition 5 A sequence $\{x_n\}_{n=1}^{\infty}$ of points in a metric space (X, d) is called a *Cauchy sequence* if, for any given number $\varepsilon > 0$, there is an integer $N > 0$ such that $d(x_n, x_m) < \varepsilon$ for all $n, m > N$ [2].

One can interpret this to mean that as *n* and *m* grow relatively large, the difference in values of the sequence become small. This does not mean that the values of the sequence must all approach only positive, or negative values, but they may also be alternating positive and negative.

Definition 6 A sequence $\{x_n\}_{n=1}^{\infty}$ of points in a metric space (X, d) is said to *converge* to a point $x \in X$ if for any given number $\varepsilon > 0$, there is an integer $N > 0$ such that $d(x_n, x) < \varepsilon$ for all $n > N$ [2].

The point to which the system converges, $x \in X$, is called the limit of the sequence.

This definition is different than that of the Cauchy sequence because the metric, *d*, is not measuring the difference of values between consecutive numbers, but between each term in the sequence and *x*, the constant value which the sequence is approaching. So, if a sequence is converging to *x*, then as *n* approaches infinity, the *n*th term of the sequence will grow closer, less than some epsilon, in distance to *x*. Relating this terminology to an IFS, the limit $x \in X$ of the sequence $\{x_n\}_{n=1}^{\infty}$ corresponds to the attractor of an IFS, where the *n*th term in the sequence, x_n, is the *n*th level of an IFS after *n* iterations on the seed image x_0.

Theorem 1 If a sequence of points $\{x_n\}_{n=1}^{\infty}$ in a metric space (X, d) converges to a point $x \in \mathrm{X}$, then $\{x_n\}_{n=1}^{\infty}$ is a Cauchy sequence [2].

Note that the converse of this theorem is not always true. Not all Cauchy sequences converge to a limit $x \in \mathrm{X}$.

Definition 7 A metric space (X, d) is *complete* if every Cauchy sequence $\{x_n\}_{n=1}^{\infty}$ in X has a limit $x \in \mathrm{X}$ [2].

Definition 8 Let $f : \mathrm{X} \to \mathrm{X}$ be a transformation on a space. A point $x_f \in \mathrm{X}$ such that $f(x_f) = x_f$ is called a *fixed point* of the transformation [2].

A fixed point of a transformation correlates to the attractor of an IFS.

Definition 9 Let $S \subset \mathrm{X}$ be a subset of a metric space (X, d). S is *compact* if every infinite sequence $\{x_n\}_{n=1}^{\infty}$ in S contains a convergent subsequence.

Theorem 2 Let (X, d) be a complete metric space. Let $S \subset \mathrm{X}$. Then S is compact if and only if it is closed and totally bounded [2].

Definition 10 A transformation $f : \mathrm{X} \to \mathrm{X}$ on a metric space (X, d) is called *contractive,* or a *contraction mapping,* if there is a constant $0 \le s < 1$ such that

$$d(f(x), f(y)) \le (s)(d(x, y)) \forall x, y \in \mathrm{X}.$$

Any such number is called a *contractivity factor* for f [2].

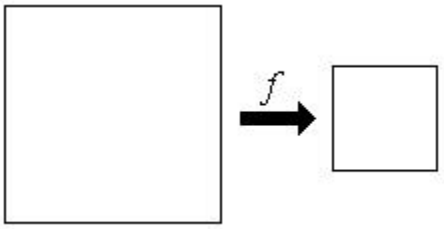

Figure 6. f is a contraction mapping action on a set of points in X.

Theorem 3 ***The Contraction Mapping Theorem***

Let $f:\mathrm{X}\to\mathrm{X}$ be a contraction mapping on a complete metric space (X,d). Then f possesses exactly one fixed point $x_f\in\mathrm{X}$, and moreover for any point $x\in\mathrm{X}$, the sequence $\left\{f^{\circ n}(x):n=0,1,2,...\right\}$ converges to x_f; that is $\lim_{n\to\infty} f^{\circ n}(x)=x_f$, for each $x\in\mathrm{X}$ [2].

5. IFS Fractals

5.1 Hausdorff Space

The Hausdorff space, $H(x)$, is a space that is convenient to use when considering real world images.

Definition 11 Let (X,d) be a complete metric space. Then $H(\mathrm{X})$, the *Hausdorff space*, denotes the space whose points are the compact subsets of X, other than the empty set [2].

These points can actually be the *n*th level of any image produced by an IFS, or a point in $H(\mathrm{X})$ can be the final attractor of an IFS. A point in the Hausdorff space any

compact set in X, including singleton points. It is important to note that the attractors, or fixed points of IFS are compact.

Definition 12 Let (X, d) be a complete metric space, Then the *Hausdorff distance between the points A and B* in $H(\mathrm{X})$ is defined by

$$h(A,B) = d(A,B) \vee d(B,A),$$

where $d(A,B) = \max\{d(x,B) : x \in A\}$, and $x \vee y$ means the maximum of x and y. We also call h the *Hausdorff metric* on H [2].

Theorem 4 Let (X, d) be a complete metric space. Then $H(\mathrm{X}\ h)$

space. Moreover, if $\{A \quad H(\ ) \quad 1, \ ,...\}$

$$A = \lim_{n\to\infty} A_n \in H(\mathrm{X})$$

can be characterized as

$$A = \{x \in \mathrm{X} : \exists \textit{ a Cauchy sequence } \{x_n \in A\} \textit{ convergent to } x\}$$ [2].

Theorem 4 allows us to declare the existence of IFS fractals. As mentioned before, it is also necessary that the mappings creating the IFS be contraction mappings.

The following lemma tells us how to determine the contractivity factor of a contraction mapping w, when this mapping is applied on the Hausdorff space.

Lemma 1 Let $w : \mathrm{X} \to \mathrm{X}$ be a contraction mapping on the metric space (X, d) with contractivity factor s. Then $w : H(\mathrm{X}) \to H(\mathrm{X})$ defined by

$$w(B) = \{w(x) : x \in B\}, \forall B \in H(\mathrm{X})$$

is a contraction mapping on $(H(\mathrm{X}),h)$ with contractivity factor *s* [2].

5.3 Iterated Function Systems

Iterated Function Systems set the foundation for Fractal Image Compression. The basic idea of an Iterated Function System is to create a finite set of contraction mappings, written as affine transformations, based on what image one desires to create. If these mappings are contractive, applying the IFS to a seed image will eventually produce an attractor of that map. It does not matter what the seed image is for the mappings, the same fixed point will be produced regardless.

Definition 13 An (hyperbolic) iterated function system consists of a complete metric space (X,d) together with a finite set of contraction mappings $w_n:\mathrm{X}\to\mathrm{X}$, with respective contractivity factors s_n, for n = 1,2,…,*N*. The abbreviation "IFS" is used for "iterated function system". The notation for this IFS is $\{\mathrm{X};w_n,n=1,2,...,N\}$ and its contractivity factor is $s=\max\{s_n:n=1,2,...N\}$ [2].

One can understand these affine transformations as a set of rules, that tell the seed, or the initial image where to “go”, or what to do, in order to converge to the attractor, or desired image. In the example of the Copy Machine Algorithm described previously, three rules are applied to the seed image. After each rule is applied, the resulting images are collaged together to produce the first level of the system. Applying the three rules again and collaging the second level of the system is created. Repeating an infinite number of times, the attractor is approached. Obviously, in practice, it is not

possible to iterate the rules an infinite number of times. Depending on the system, and the rules, it may only take a few iterations to visually notice what attractor the system is approaching. As in the Copy Machine Algorithm, what the attractor will approximately look like is noticeable to the viewer at the third level. In some of our examples, in which we iterate mappings to produce an image from a seed, the attractor is noticeable after one iteration (see Figures 21 to 40 in Appendix B).

As mentioned above, a crucial step in applying an IFS is to collage all of the smaller images (or points in the Hausdorff space) produced by each rule in order to reach the image at the next level.

Theorem 5 ***The Collage Theorem*** Let (X, d) be a complete metric space. Let $T \in H(\mathrm{X})$ be given, and let $\varepsilon \geq 0$ be given. Choose an IFS $\{\mathrm{X}; (w_0), w_1, w_2, ..., w_N\}$ with contractivity factor $0 \leq s < 1$ so that

$$h\left(T, \bigcup_{n=1}^{N} w_n(T)\right) \leq \varepsilon,$$

where $h(d)$ is the Hausdorff metric. Then

$$h(T, A) \leq \frac{\varepsilon}{1-s},$$

where A is the attractor of the IFS. Equivalently,

$$h(T, A) \leq (1-s)^{-1} h\left(T, \bigcup_{n=1}^{N} w_n(T)\right) \text{ for all } T \in H(\mathrm{X}) \text{ [2].}$$

The Collage Theorem tells us that in order to find an IFS whose attractor looks like a given set, we must find a set of contractive transformations on a suitable space, in

which the given set lies, such that the distance between the given set and the union of the transformations is small. In other words, the union of the transformations is close to, or looks like, the given set. The IFS which satisfies this may be a good candidate for reproducing the given set, or image, by the attractor of the IFS. Thus this image can be stored using much less space.

5.4 The Random IFS Approach

In the random approach, the method of applying the *n* affine transformations, which act as rules for the IFS, is different than in the deterministic approach described above. Rather than starting with any image, applying each rule and then collaging the produced images to create the first level, in the random approach one rule is chosen randomly and applied to an initial point, which produces another singleton point at the first level. The next transformation is chosen at random again, and applied to the singleton point from the first level. This creates a new point at the second level. This process continues for some chosen number of iterations. In this case, the points from each level are plotted, with the first few levels discarded. Depending on the number of iterations, the resulting plot is an image that may look close to the attractor of the IFS. The reason the first few points are discarded is because the initial point, and possibly the first few levels, may be points that are not in the attractor.

To apply this random approach, each rule must have a certain probability of being chosen. To choose the probability necessary for each rule, we observe the actions that each rule takes on a given area. Each transformation $w_n(\vec{x}) = A\vec{x} + T$, changes a given area by a factor of $\left|\det(A)\right|$. In order to allow the random selection process to give more

weight to the transformations with large determinants and less weight to the transformations with small determinants, "it is desirable to make the selections with probabilities that are proportional to the determinants" [3]. So, for each transformation $w_n(\vec{x}) = A\vec{x} + T$, we choose a set of weights, $p_1, p_2, \ldots, p_n$ by the formula

$$p_j = \det(A_j) / \sum_{i=1}^{n} \det(A_i), \quad j = 1,2,\ldots,n,$$

where A_i is the corresponding matrix for the affine transformation w_i, $i = 1,2,\ldots,n$ [3]. It is obvious that $p_1 + p_2 + \ldots + p_n = 1$, satisfying the conditions for a probability measure.

An example of the random IFS approach to generating a fractal can be found in Appendix A, the program named 'LisaIFS'. This program uses 16 transformations to create the IFS. The probability assigned to the *j*th transformation is determined by the above formula for p_j. The attractor of this IFS after 40,000 iterations can be seen in Appendix B, Figure 19. Another example of a random IFS is included. The program that generates this random IFS is called 'Star' and appears in Appendix A. The attractor of this IFS is displayed after 20,000 iterations and is in Appendix B, Figure 20.

IV. Fractal Image Compression

"The central goal of fractal image compression is to find resolution independent models, defined by finite length (and hopefully short) strings of zeros and ones, for real world images."

-Michael F. Barnsley

1. Using IFS fractals for Fractal Image Compression

The IFS compression algorithm starts with some target image *T* which lies in a subset $S \subset \mathbf{R}^2$. The target image *T* is rendered on a computer graphics monitor. In order to begin fractal image compression, an affine transformation,

$$w_1(\mathbf{x}) = w_1\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

is introduced with coefficients that produce a new image, $w_1(T)$, with dimensions smaller than that of *T*. This ensures a contraction mapping.

The user adjusts the coefficients *a*, *b*, *c*, *d*, *e*, *f* in order to shrink, translate, rotate, and shear the new image, $w_1(T)$, on the screen so that it lies over a part of *T*. Once $w_1(T)$ is in place, it is fixed, the coefficients are recorded, and a new affine transformation $w_2(x)$ is introduced along with its sub-copy of T, $w_2(T)$. The same process is carried out with this new image as was done with $w_1(T)$. Whenever possible, overlaps between $w_1(T)$ and $w_2(T)$ should be avoided. Overlaps only complicate the situation, although there exist compression methods, such as wavelets, which confront this issue. In this manner, a set of affine transformations $w_1, w_2, w_3 \ldots, w_n$ is obtained such that

$$\tilde{T} = \bigcup_{n=1}^{N} w_n(T),$$

where *N* is as small as possible.

The Collage Theorem assures us that the attractor *A* of this IFS will be visually close to *T*. Moreover, if $\tilde{T} = T$, then $A = T$. As desired, *A* provides an image which is visually close to *T* and is resolution independent using a finite string of ones and zeros. By

adjusting the parameters in the transformations we can continuously control the attractor of the IFS. This is what is done in fractal image compression.

Complex images can be built up using fractal image compression by working on subsets of the image, where each subset is represented by an IFS. This method of compression is highly optimal. If each coefficient in the affine transformations describing the IFS is represented with one byte, then an IFS of three transformations requires only 12 bytes of data to represent its image. As the number of coefficients used increases, the size of the digital file increases. Thus, it is optimal to find as few affine transformations as possible to represent an image. For an in depth study of how to optimize the storage of data in files see [2].

2. The Fractal Transform Theory

Fractal transform theory is the theory of local IFS. Although local IFS does complicate the theory of fractal image compression, in practice it simplifies the process.

A global transformation on a space X is a transformation, which is defined on all points in X; whereas, a local transformation is one whose domain is a subset of the space X and the transformation need not act on all points in X. Rather than allowing an IFS to act upon only on the whole domain, it is convenient to allow an IFS to act upon domains that are subsets of the space. This type of IFS is called a local IFS.

The idea of fractal image compression, as briefly mentioned above, is to find subspaces (or sub-images) of the original image space, which can be regenerated using an IFS. Where possible, if one IFS can be used in place of several IFS's which reproduce similar sub-images, it is more efficient in terms of storage space to use that one IFS. It is

more likely that an image will require more than one IFS to reproduce a compressed image, which closely resembles the original.

Definition 14 Let (X, d) be a compact metric space. Let R be a nonempty subset of X. Let $w : R \to \mathrm{X}$ and let *s* be a real number with $0 \le s < 1$. If

$$d(w(x), w(y)) \le (s)(d(x, y)) \ \forall x, y \in R,$$

then *w* is called a *local contraction mapping* on (X, d). The number *s* is a *contractivity factor* for *w* [2].

Definition 15 Let (X, d) be a compact metric space, and let $w_i : R_i \to \mathrm{X}$ be a local contraction mapping on (X, d), with a contractivity factor s_i, for *i*= 1,2,…,*N*, where *N* is a finite positive integer. Then $\{w_i : R_i \to \mathrm{X} : i = 1,2,..., N\}$ is called a *local iterated function system* (local IFS). The number $s = \max\{s_i : i = 1,2,\ldots N\}$ is called the contractivity factor of the local IFS [2].

The local IFS can be defined as follows. If we let *S* denote the set of all subsets of X, then we can define the operator $W_{local} : S \to S$ according to [2]

$$W_{local}(B) = \bigcup_{i=1}^{N} w_i(R_i \cap B), \text{ for all } B \in S.$$

Under certain restraints, W_{local} can be defined as contractive on certain subsets of the Hausdorff space. This allows us to create a fractal compression system [2]. If A is a nonempty subset of X, we call A an attractor of the local IFS if $W_{local}(A) = A$ [2]. If A and B are attractors, then so is $A \cup B$. If there is an attractor, there is a largest attractor,

which is the one that contains all the other attractors [2]. This largest attractor is referred to as the attractor of W_{local} and is found by taking the union of all the other attractors in W_{local} [2].

If we define an IFS to be $\{w_i : R_i \to \mathrm{X} : i = 1,2,\ldots,N\}$ and we suppose that the sets R_i are compact, then we can define a sequence of compact subsets of X by [2]

$$A_0 = \mathrm{X},$$

$$A_n = \bigcup_{i=1}^{N} w_i(R_i \cap A_{n-1}) \text{ for } n=1,2,3,\ldots$$

Because the IFS consists of contractive mappings, it is true that

$$A_o \supset A_1 \supset A_2 \supset A_3 \supset \ldots.$$

So, A_n is a decreasing sequence of compact sets [2]. There exists a compact set $A \subset \mathrm{X}$ so that [2]

$$\lim_{n\to\infty} A_n = A$$

and

$$A = \bigcup_{i=1}^{N} w_i(R_i \cap A) = W_{local}(A).$$

As mentioned before, if A is not empty, then A is the maximal attractor for the local IFS [2]. If one can find a compact set B such that $W_{local}(B) \supset B$, then the possibility that A is empty is ruled out. A corresponds to the attractor of an IFS in a fractal image compression scheme. The coefficients of the mappings w_i are crucial in the

determination of the compression of an image. *A* represents what the image would look like after applying the mappings to subsets of the image.

The following is an algorithm of how one would go about applying the fractal transform to an image.

Algorithm for Fractal Image Compression

1. *Input a binary image, call it M.*
2. *Cover M with square range blocks. The total set of range blocks must cover M, without overlapping.*
3. *Introduce the domain blocks D; they must intersect with M. The sides of the domain blocks are twice the sides of the range blocks.*
4. *Define a collection of local contractive affine transformations mapping domain block D to the range block* R_i.
5. *For each range block, choose a corresponding domain block and symmetry so that the domain block looks most like the part of the image in the range block.*
6. *Write out the compressed data in the form of a local IFS code.*
7. *Apply a lossless data compression algorithm to obtain a compressed IFS code.*

In practice these steps can be carried out on a digital image. The compression is attained by storing the coefficients of the transformations, rather than storing the image pixel by pixel. The following is an explanation of 'RIFSbat', a simple fractal image

compression program and 'fdec' the corresponding decompression program, both included in Appendix A.

These two programs can be run on Matlab and only compress grayscale square images that are in pgm format, although further changes can be implemented later to account for non-square images and other formats. This batch program runs through the program 10 times, allowing 10 different tolerances. Ten different mat files are saved representing 10 different compressed images. The difference in the compressed files is not the number of bits needed to store the files (this is the same as long as the range size is the same), but the time needed to produce the compressed files based on the error tolerance. The tolerances are determined by what the variable *min0* is set equal to the minimum error, denoted by the variable *minerr* is defined by the norm of the difference between the range blocks and the transformed domain blocks. 'RIFSbat' searches for the transformation with least error from domain blocks to range blocks. During the first loop, the tolerance is *min0* = 10. So, the program searches for a transformation until it finds a transformation with *minerr* < *min0*. As *min0* increases, more error is allowed. With each run, the tolerance of allowable error increases by 10.

First, the user must enter the name of the pgm image file in the first line of the program: *M* = getpgm ('imagename.pgm'). In the examples in Appendix B, we use 'sisters.pgm'. Then, the user specifies the desired range block size by setting *rsize* equal to the length of the side of the desired range block. Presently, *rsize* is set equal to 4, which allows range blocks of size 4×4. We next create the domain blocks, which are twice the size of the range blocks, in this case 8×8. In determining which mapping will need to be made from the domain blocks to the range blocks, we will need to compare the

domain blocks to the range blocks. To accurately compare these blocks, they must be the same size. So, we do some averaging over the domain blocks which allows us to shrink the domain blocks to half of its size in order to match the size of the range blocks.

Originally, each domain block is 8×8. The averaging only takes place over each distinct block of 2×2 pixels within the domain block. Then the average grayscale value in each 2×2 block of pixels is represented in one pixel in the scaled domain blocks, called M1. M1 is a 4×4 block at this point. We subtract the average of the domain block from each entry in the domain block to account for possible darkening of the decompressed image. The resulting scaled domain block is D.

Now, we save 8 different transformations of each domain block in an eight dimensional monster matrix called *bigM*. The transformations include the original domain block, a 90°, 180° and a 270° rotation, a horizontal flip, and a vertical flip, as well as the transform of the domain block and a 180° rotation of the transformed domain block. We introduce a vector **s,** which contains different specific scalings to transform the grayscale of the domain block to make a better match to a range block.

At this point, 'RIFSbat' goes through all of the range blocks, and offsets each of them by subtracting the average of the range block from each entry in the range block. Now we can equally compare the domain to the range blocks. We save the offset of the range blocks in *o*, which we will add back to the image later.

Next the program cycles through each domain block D_{ij} and tests each symmetry that is stored in *bigM*, along with the four possible gray scales for the best transformation that will map to a given range block R_{lk}. When the best map is found, the location of that domain block *i*0 and *j*0, the best symmetry *m*0 of the domain block, the best scaling *s*0,

and the offset *o* is saved in the five dimensional matrix $T(k,l,:)=[i0,\ j0,m0,s0,o]$. It is the entries of this matrix that determine the number of bytes needed to store the compressed image file. For each of the 10 cases that the program considers, the batch program saves the number of rows of the original image, the size of the range blocks, and the time the program took to achieve the compression. The time is recorded to compare the results with the amount of time the process took. 'RIFSbat' produces the two CPU charts that appear in the beginning of Section 4 of Appendix B. Once this information is saved in a file, it is possible to compress that file even more by applying a lossless coding algorithm. It is from the matrix, *T*, that the program 'fdec' can regenerate the image.

It is important to note that each transformation from the original 8×8 domain is a contraction mapping because the domain must be scaled by ½ in order to map the domain to the range. Also, the information stored in each $(k,l,:)$ entry of *T* represent the coefficients of the mappings w_i, i = 1,2,3,…N that make up the *N* local IFS mappings. The image regenerated after all the mappings in *T* are applied to some seed image, is the attractor of the local IFS.

In order to regenerate the attractor of the contractive transformations found, we must use the program 'fdec' along with the saved information from 'fcomp'. First we load the correct data using the name that we saved it under in the batch file. Then we initialize a matrix to perform the mappings on. This matrix must be the same size as the original image. As we discussed with IFS and the Sierpinski Triangle, it makes no difference what seed image is used. Although, in the program we initialize the seed image to all zeros, which is a uniformly gray image, choosing another image as the seed to the local IFS will arrive at the same result.

Depending on the block size chosen for the range blocks, one may need to vary the number of iterations applied to the seed image in order to arrive at the attractor image. As more iterations of the IFS are applied to the image, the clearer the attractor will become. After the *n*th iteration, the image produced corresponds to the A_n compact set as discussed in the local IFS theory. First, the domain blocks of the seed image must be created and rescaled to the size of the range blocks. Then using the *T* matrix, the domain blocks are transformed and mapped to the range blocks. This process is repeated for each iteration. The attractor, *M*, is then output to be displayed on the screen. Appendix B contains examples of an original image, and the consecutive images regenerated after iterating the local IFS created for that image. The quality of the attractors vary depending on the size of the range blocks used and the error allowed in finding an appropriate transformation form domain block to range block.

Our implementation of this simple method of fractal compression produced great compression ratios. Considering that each pixel requires 8 bits to store the values of 0 to 255, to store an 256×256 image pixel by pixel would require 65536 bytes (around 65KB). Using 'RIFSbat' and 'fdec' with any chosen error, to store an image of this size with a range block size of 4×4 pixels only requires 11776 bytes. The compression ratio is better than 5:1. Of course, increasing the range block size to 8×8 pixels improves the compression to only 2688 bytes, with a compression ratio of approximately 24:1. The larger range block sizes allow higher compression ratios. The time needed to produce the attractor image is based on how much error is allowable in the transformations. The larger the error, the quicker the compression. The use of the image will determine the required amount of compression and image quality.

In Appendix B, Figure 21 is the original image in this example. Figure 22, 23, and 24 are the first through third iterations of the fractal compression transformations with *minerr* = 10. Referring to Chart 1 in Appendix B we can tell that this compressed file took about 5 1/2 hours to complete compression. The attractor image that is regenerated is close to the original image, but the time needed to accomplish compression is not desirable. With a 4×4 pixel range block, a decent error is probably about 40 or 50. Although to compress an image with this error takes about 10 minutes, if the error is greater than that, the image quality becomes very low and blocky. For the 4×4 pixel range blocks, three different implementations based on a change in the allowable error of images are is Appendix B. The errors, *min0,* displayed are 10, 20, and 80.

By looking at Figure 34 and 38 we notice that the image quality is not as high as the previous case. The reason for this is because the range size in these images is 8×8 pixels. Two sets of images, one with *minerr* = 0 and *minerr* = 80 are available in Appendix B to provide a comparison between image quality and the time used to produced the compressed file, which can be found in Chart 2.

V. Conclusion

After discussing different image compression algorithms, lossless and lossy, it is only fitting to compare the algorithms. The algorithms, which are implemented and discussed in this paper, are Delta Compression, a form of Fourier Compression, and a simple form of Fractal Image Compression. The results from these compression algorithms appear in Appendix B.

Being a lossless form of compression, Delta compression can be a wonderful tool. Although it is possible exclusively use a Delta compression on image data, many compression algorithms could be optimized if a lossless code such as Delta compression is applied in addition to a lossy code. Similarly, the Huffman code can be applied to other forms of previously compressed data, which may optimize compression for that data. These lossless codes may be preferred over lossy compression methods in cases where loss of data is out of the question. The best compression may not be accomplished, but the image data and quality will remain exactly the same as the original image and quality. It is in cases where the exact data needs to be restored after compression that a lossless code is the best choice.

Although our implementation of the Fourier Compression is not efficient, when using the same quantizing approach as described in section 3.4, the Cosine Transform should generate much better compression due to the lack of imaginary values in the transform of the signal. In the future, a more thorough study and experimentation of the Cosine transform will be established. In turn, an implementation of a JPEG style compression will be possible.

The simple Fractal image compression algorithm executed is the most efficient form of compression which we implemented; although, research has revealed that JPEG is one of the better, if not the best, forms of compression available today. The question has been posed as to whether or not an optimal Fractal compression algorithm will be discovered that will outperform JPEG. Much research is being done to find faster and more efficient forms of image compression technology. The race has only begun.

VI. Appendix A

```
************************************************************************
Delta Compression - 2 dimensional
************************************************************************
% Program Name: deltacomp2D
% Purpose: The purpose of this program is to achieve
%          delta compression of a matrix representing pixel
%          values. The smaller the magnitude of the entries,
%          the less memory needed to store the data.
%          The (r,c)th entry(pixel) of comp is found by taking
%          difference of the entries (r,c) and (r,c+1) of the
%          matrix A.
% Input:   i x j matrix - O
% Output:  i x j matrix - D

function Dx=deltacomp2D(X);
 global first;
 first=X(1,1);

 global i
 global j
 [i,j]=size(X);
 %**************Delta Compression**********;
 for r=1:i
   for c=1:j
        if r==i & c==j
           Dx(r,c)=0;
        else
           if c==j
              Dx(r,c)=X(r,c)-X(r+1,1);
           else
              Dx(r,c)=X(r,c)-X(r,c+1);
           end
        end
   end
 end
************************************************************************
% Program Name: deltade2D
% Purpose: The purpose of this program is to
%          decompress the delta compression of a matrix
%          which has been compressed by deltacomp2d.m
%          This brings the compressed matrix back to original
%          form.
% Input:   i x j matrix - Dx
% Output:  i x j matrix - y

function y=deltade2D(Dx);
global i
global j
 for r=1:i
   for c=1:j
     if c==1
      if r==1
            global first
            y(r,c)=first;
```

```
        end

        if r>1
        % c=1 here.  ex. If r=2 then y(2,1)=y(1,j)-Dx(1,j)
        %                where j is end of row

           y(r,c)=y(r-1,j)-Dx(r-1,j);             end
    else
    %c is not equal to 1 here        ex. y(r,2)=y(r,1)-Dx(r,1)
    y(r,c)=y(r,c-1)-Dx(r,c-1);
    end
  end
end

**************************************************************************
Fourier Transforms
**************************************************************************
%Program Name: FT1D (Fourier transform 1 dimension)
%Purpose: The purpose of this program is transform a discrete one
%         dimensional signal x into its Fourier transform f.
%
%         The Fourier transform f of a signal x is a vector containing
%         the amplitudes of the fundamental frequencies that make up x.
%         Each component of f indicates the strength of a particular
%         frequency in x. [Hankerson]
%Input:   One dimensional signal, x
%Output:  Discrete Fourier transform of input signal, f

function[f] = FT1D(x)

[R C]=size(x);

for v = 1:R
for k = 1:R

M(k,v) = (1/sqrt(R))*exp(-2*pi*i*(k-1)*(v-1)/R);

end
end

f = M*x;
**************************************************************************
% Program Name: invFT1D (inverse Fourier transform 1 dimension)
% Purpose: The purpose of this program is return the original discrete
%          signal from a given 1 dimensional discrete Fourier
%          transform;
% Input:   Discrete Fourier transform of input signal, f
% Output:  Original 1 dimensional signal, x

function[x] = invFT1D(f)
[R C]=size(f);

for v = 1:R
for k = 1:R

M(k,v) = (1/sqrt(R))*exp(-2*pi*sqrt(-1)*(k-1)*(v-1)/R);
```

```
end
end

x = conj(M)*f;
**********************************************************************
% Program Name: FT2D (Fourier transform 2 dimensions)
% Purpose: The purpose of this program is to return the discrete
%          Fourier transform of a given discrete signal;
%          The Fourier transform f of a 2D signal x is a matrix
%          containing the amplitudes of the fundamental frequencies
%          that make up x. Each component of f indicates the strength
%          of a particular frequency in x [Hankerson].
%
% Input:   Two dimensional signal, x , a square matrix
% Output:  Two dimensional matrix representing
%         Discrete Fourier transform of input signal, f

function[f] = FT2D(x)
[R C]=size(x);

if R~=C
   display('Matrix is not square');
   return
end

for k = 1:R
   for v = 1:R
      M(k,v) = exp(2*pi*i*(k-1)*(v-1)/R);
   end
end

f = (1/R)*(conj(M))*x*M;
**********************************************************************
% Program Name: invFT2D (Inverse Fourier transform 2 dimension)
% Purpose: The purpose of this program is to return a discrete
%          inverse Fourier transform signal from a given discrete
%          Fourier transform;
% Input:   f, 2D matrix representing Fourier transform of a 2D
%          signal
% Output:  Original 2 dimensional signal, x

function[x] = invFT2D(f)

[R C]=size(f);

if R~=C
   display('Matrix is not square');
   return
end

for k = 1:R
   for v = 1:R
      M(k,v) = exp(2*pi*i*(k-1)*(v-1)/R);
   end
end
    x = (1/R)*M*f*conj(M);
```

```
**********************************************************************
% Program Name: FourierComp
% Purpose: The purpose of this program is to apply the Fourier
%          transform to an image, M, to quantize or set the high
%          frequency values equal to zero, then to apply the inverse
%          Fourier transform to the quantized image, reconstructing an
%          image, A, close to the original image. The quantized matrix,
%          A1, representing the reconstructed image is the compressed
%          form.
%
%          When quantizing, the entries of A1 whose ratio
%          abs(A1(i,j))/mean(A1) <= k , k is chosen to = 1 in this
%          program) are set equal to zero.
%
%          This program uses the following programs: FT1D, FT2D,
%          invFT1D, invFT2D
% Input:   A matrix , M, representing a signal or image(1d or 2d). The
%          user may also change the parameter k, as mentioned above,
%          which is in the quantizing section of this program.
% Output:  A signal close to that of the original represented by a
%          compressed matrix A.

function[A,A1] = FourierComp(M)

[R C]=size(M);

%*************_Check 1d_*************%
if R==1 | C==1
   choice=1;
   A1=FT1d(M);
end

%*************_Check 2d_*************%
if R~=1 | C~=1
        choice=2;
     A1=FT2d(M);
end


%*********_Variables_**************%
absA1=abs(A1);
meanA1=mean(mean(absA1));

%*********_Quantizing_*************%
counter=0;
for i=1:R
   for j=1:C
      if (absA1(i,j)/meanA1)<= 1
         A1(i,j)=0;
                  counter=counter+1;
      end
   end
end
```

```
%*********_Inverse Transform_***********%
if choice==1
   A=invFT1d(A1);
   A=real(A);
   display(counter);
   return
end
if choice==2
   A=invFT2d(A1);
   A=real(A);
   display(counter);
   return
end

**************************************************************************
Random Iterated Function System
**************************************************************************
% Program Name: LisaIFS
% Purpose: This program is an implementation of a Random IFS
%          once the graphics screen appears, the user clicks on a point
%          then waits for the IFS to randomly apply transformations.
%          The frequency at which each transformation is applied is
%          based on their assigned probability.
%          This IFS produces the name Lisa. Upon zooming in, the name
%          Lisa appears inside each of the letters.
% Input:   In the first line of the program, the user defines the
%          desired number of iterations to be applied to the initial
%          point.
% Output:  The image that the IFS creates, as the iterations are being
%          applied

numiter=40000;
axis([0 1 0 .5]);
title(['Lisa: ' num2str(numiter) ' Iterations']);
[x0,y0]=ginput(1);
hold on
for i=0:numiter
r = rand;

%1
if r >= 0 & r < .0816
   a=0; b=(1/4); c=(2/9); d=0; e=0; f=0;
elseif r >= .0816 & r < .1632
   a=0; b=(1/4); c=(2/9); d=0; e=0; f=(2/9);
elseif r >= .1632 & r < .2448
   a=0; b=(1/4); c=(2/9); d=0; e=(5/18); f=(2/9)
elseif r >= .2448 & r < .3264
   a=0; b=(1/4); c=(2/9); d=0; e=(5/18); f=0;
elseif r >= .3264 & r < .408
   a=0; b=(1/4); c=(2/9); d=0; e=(13/18); f=0;
elseif r >= .408 & r < .4896
   a=0; b=(1/4); c=(2/9); d=0; e=(13/18); f=(2/9);
elseif r >= .4896 & r < .5712
   a=0; b=(1/4); c=(2/9); d=0; e=(16/18); f=0;
elseif r >= .5712 & r < .6518
   a=0; b=(1/4); c=(2/9); d=0; e=(16/18); f=(2/9);
```

```
%2
elseif r >= .6518 & r < .7334
   a=(2/9); b=0; c=0; d=(1/4); e=(4/9); f=0;
elseif r >= .7334 & r < .815
   a=(2/9); b=0; c=0; d=(1/4); e=(4/9); f=(1/6);
elseif r >= .815 & r < .8966
   a=(2/9); b=0; c=0; d=(1/4); e=(4/9); f=(1/3);
%3
elseif r >= .8966 & r <= .917
   a=(1/9); b=0; c=0; d=(1/8); e=(1/9); f=0;
elseif r >= .917 & r <= .9374
   a=(1/9); b=0; c=0; d=(1/8); e=(4/9); f=(5/18);
elseif r >= .9374 & r < .9578
   a=(1/9); b=0; c=0; d=(1/8); e=(5/9); f=(1/9);
%4
elseif r >= .9578 & r < .9782
   a=0; b=(1/8); c=(1/9); d=0; e=(15/18); f=(1/9);
elseif r >= .9782 & r < 1
   a=0; b=(1/8); c=(1/9); d=0; e=(15/18); f=(1/3);
end

plot(x0,y0,'k');
     x1=a*x0+b*y0+e;
     y1=c*x0+d*y0+f;
     x0=x1;
     y0=y1;
end

hold off;

**********************************************************************
% Program Name: StarIFS
% Purpose: This program is an implementation of a Random IFS
%          once the graphics screen appears, the user clicks on a point
%          then waits for the IFS to randomly apply transformations.
%          The frequency at which each transformation is applied is
%          based on their assigned probability. This program is an IFS
%          that creates shapes that contain stars within the shapes.
% Input:   In the first line of the program, the user defines the
%          desired number of iterations to be applied to the initial
%          point
% Output:  The image that the IFS creates, as the iterations are being
%          applied

N=20000;
axis([-.6 1 -.5 1]);
title(['STAR ' num2str(N) ' Iterations']);
[x0,y0]=ginput(1);

hold on

for i=0:20000
r = rand;

if r >= 0 & r < .3738
   a=.538*cos(10*pi/18); b=.538*sin(10*pi/18); c=-.538*sin(10*pi/18);
   d=.538*cos(10*pi/18); e=0; f=.057;
```

```
elseif r >= .3738 & r < .5979
   a=.423*cos(1*pi/18); b=.0; c=0; d=.423*cos(1*pi/18); e=.5; f=-.220;

elseif r >= .5979 & r < 1
   a=.558*cos(10*pi/18); b=-.558*sin(10*pi/18); c=.558*sin(10*pi/18);
   d=.558*cos(10*pi/18); e=.6; f=.3;
end

plot(x0,y0,'k');
     x1=a*x0+b*y0+e;
     y1=c*x0+d*y0+f;
     x0=x1;
     y0=y1;
end

hold off;

************************************************************************
Fractal Transform
************************************************************************
%Program Name: fliph
%Purpose: This program flips the matrix M about its central row
%Input:   nxm matrix M
%Output:  nxm matrix N (M flipped on horizontal axis)

function N=fliph(M)
[nv nh]=size(M);
for k=1:nv
   for l=1:nh
      N(k,l)=M(nv-k+1,l);
   end
end
************************************************************************
%Program Name: flipv
%Purpose: This function flips the matrix M about its central
%         column
%Input:   nxm matrix M
%Output:  nxm matrix N (flipped vertically)

function N=flipv(M)
[nv nh]=size(M);
for k=1:nv
   for l=1:nh
      N(k,l)=M(k,nv-l+1);
   end
end
************************************************************************
%Program Name: rotmat
%Purpose: This program rotates a matrix by 90 degrees
%         counter clockwise
%Input:   nxm Matrix M
%Output:  nxn Matrix N (M rotated 90 deg. )

function N=rotmat(M)
[nv nh]=size(M);
for k=1:nv
   for l=1:nh
```

```
      N(k,l)=M(l,nv-k+1);
   end
end
*********************************************************************
%Program Name: RIFSBat
%Purpose: This program takes an image in M and determines an array
%         T. T lists which looks for the best transformed domains that
%         map to ranges, which each are submatrices of M. The domains
%         are gotten from subdomains of size 2nx2n which have been
%         averaged to size nxn. The data that is saved in the output
%         files are a compressed versions of the image M.  This data
%         along with the block size is needed in fdec.m to reconstruct
%         the image.
%         There are 8 possible transformations. These use the functions
%         rotmat, fliph and flipv.
%
%Input:  User must insert the pgm file desired
%Output: This batch program saves 10 different versions (based on
%        allowable error) of the compressed image (held in T) along
%        with the variables: sv, rsize, tim, cpu0  which will be used
%        by fdec to decompress the image, and to create time charts

% Get the pgm file and file size
M=getpgm('sisters.pgm');
[sv sh]=size(M);
if sv~=sh
   display('Matrix is not square');
   return
end

% Begin batch runs
for irn=1:10
   clear T;

   % Set timers
   begrun=clock;
   cpu=cputime;
   min0=10*irn;
   rsize=4;
   nd=sv/rsize/2;
   nr=sv/rsize;

   % Scale the Domain Blocks
   for i=1:rsize*nd
      for j=1:rsize*nd
         M1(i,j)=mean(mean(M((i-1)*2+1:i*2,(j-1)*2+1:j*2)));
      end
   end

   % Matrix of 4 possible scalings to transform grayscale
   s=[0.45 0.60 0.80 0.97];

   % Create monster matrix containing all possible 2D transformations
   % of the domain blocks. Store in multidimensional matrix bigM.
   for i=1:nd
      i1=(i-1)*rsize+1;
      i2=i*rsize;
```

```
   for j=1:nd
      j1=(j-1)*rsize+1;
      j2=j*rsize;
      D=M1(i1:i2,j1:j2);
      D=D-mean(mean(D));
      bigM(i1:i2,j1:j2,1)=D;
      tmp=rotmat(D);
      bigM(i1:i2,j1:j2,2)=tmp;
      tmp=rotmat(tmp);
      bigM(i1:i2,j1:j2,3)=tmp;
      tmp=rotmat(tmp);
      bigM(i1:i2,j1:j2,4)=tmp;
      bigM(i1:i2,j1:j2,5)=fliph(D);
      bigM(i1:i2,j1:j2,6)=flipv(D);
      bigM(i1:i2,j1:j2,7)=D';
      bigM(i1:i2,j1:j2,8)=rotmat(rotmat(D'));
   end
end

% Compare the range blocks and scaled domain blocks.
% k,l - used to cycle through blocks Rkl.
for k=1:nr
   k1=(k-1)*rsize+1;
   k2=k*rsize;
   for l=1:nr
      [k l]
      l1=(l-1)*rsize+1;
      l2=l*rsize;
      R=M(k1:k2,l1:l2);

      % Offset o is the average in the block Rkl
      o=mean(mean(R));
      R=R-o;

      % Initialize error to large value
      minerr=10000;
      i0=0;
      j0=0;
      m0=0;
      if minerr>min0

         % Now cycle through each Domain Dij
         for i=1:nd
            if minerr>min0
               i1=(i-1)*rsize+1;
               i2=i*rsize;
               for j=1:nd
                   if minerr>min0
                      j1=(j-1)*rsize+1;
                      j2=j*rsize;

                      % Test each transformation
                      for m=1:8
                          if minerr>min0
                             D=bigM(i1:i2,j1:j2,m);

                             % Try the four gray scalings
```

```
                                for n=1:4
                                    if norm(s(n)*D-R)<minerr
                                       minerr=norm(s(n)*D-R);
                                       i0=i;
                                       j0=j;
                                       m0=m;
                                       s0=s(n);
                                    end
                                end
                             end
                          end
                       end
                    end
                 end
              end
              T(k,l,:)=[i0 j0 m0 s0 o];
           end
        end

        % Stop the clock, store computation time in tim
        % and elapsed cpu time in cpu0.
        cpu0=cputime-cpu;
        stoprun=clock;
        tim=etime(begrun,stoprun);

        % Save data in mat file - need to change the name after each use.
        switch irn
           case 1,
              save 'sisters4_1' sv rsize T tim cpu0;
           case 2,
              save 'sisters4_2' sv rsize T tim cpu0;
           case 3,
              save 'sisters4_3' sv rsize T tim cpu0;
           case 4,
              save 'sisters4_4' sv rsize T tim cpu0;
           case 5,
              save 'sisters4_5' sv rsize T tim cpu0;
           case 6,
              save 'sisters4_6' sv rsize T tim cpu0;
           case 7,
              save 'sisters4_7' sv rsize T tim cpu0;
           case 8,
              save 'sisters4_8' sv rsize T tim cpu0;
           case 9,
              save 'sisters4_9' sv rsize T tim cpu0;
           case 10,
              save 'sisters4_10' sv rsize T tim cpu0;
        end
     end
```

```
**********************************************************************
%Program name: fdec
%Purpose: Decodes the fractal image compression data form
%         the fcomp routine. This routine needs rotmat.m, fliph.m
%         and flipv.m.
%
%         This file reads the saved information in the mat files saved
%         by fcomp. In the last runs the data saved is sv tim and T.
%         The files are called sistersD_E where D is for DxD range
%         blocks. E is the run number corresponding to the error.
%         Typically run one has been min0=10 and the others are
%         multiples of 20 from 40 to 100.
%
%Input:   The user must designate in the load statement what mat data
%         file should be loaded. This data file should have been
%         created by the previous program 'RIFSbat'.  Also, the user
%         should designate the desired number of iterations in the
%         third line of the program: for iter=1:desired number of
%         iterations
%Output:  This program outputs the attractor image after the specified
%         number of iterations

% Read in mat data file
load 'sisters4_7'

% Initialize matrix
M=100*ones(sv);

% Start Iteration
for iter=1:5

   % Enter range block size used in fcomp
   rsize=4;
   nd=sv/rsize/2;
   nr=sv/rsize;

   % Rescale Domain Blocks
   for i=1:rsize*nd
      for j=1:rsize*nd
         M1(i,j)=mean(mean(M((i-1)*2+1:i*2,(j-1)*2+1:j*2)));
      end
   end

   % Transform Domain Block Using T matrix
   for k=1:nr
      k1=(k-1)*rsize+1;
      k2=k*rsize;
      for l=1:nr
         l1=(l-1)*rsize+1;
         l2=l*rsize;
         i0 = T(k,l,1);
         j0 = T(k,l,2);
         m0 = T(k,l,3);
         s0 = T(k,l,4);
          o = T(k,l,5);
         i1 = (i0-1)*rsize+1;
         i2 = i0*rsize;
```

```
         j1 = (j0-1)*rsize+1;
         j2 = j0*rsize;
          D = M1(i1:i2,j1:j2);
          D = D-mean(mean(D));
         if m0==2
            D=rotmat(D);
         elseif m0==3
            D=rotmat(rotmat(D));
         elseif m0==4
            D=rotmat(rotmat(rotmat(D)));
         elseif m0==5
            D=fliph(D);
         elseif m0==6
            D=flipv(D);
         elseif m0==7
            D=D';
         elseif m0==8
            D=rotmat(rotmat(D'));
         end
         R=s0*D+o*ones(size(D));
         MM(k1:k2,l1:l2)=R;
      end
   end
   M=MM;
end

% Output Image which is in M
imagesc(M)
colormap(gray);
```

VII. Appendix B

1. Delta Compression Results

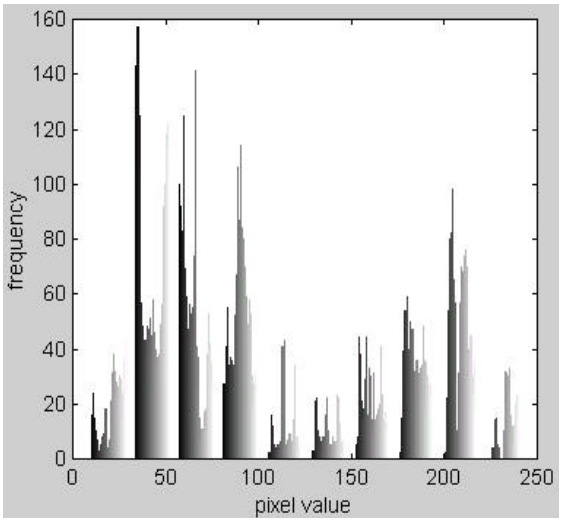

Figure 1. Histogram of Pixel Values in Original Sisters Image

Figure 2. Original Sisters Image

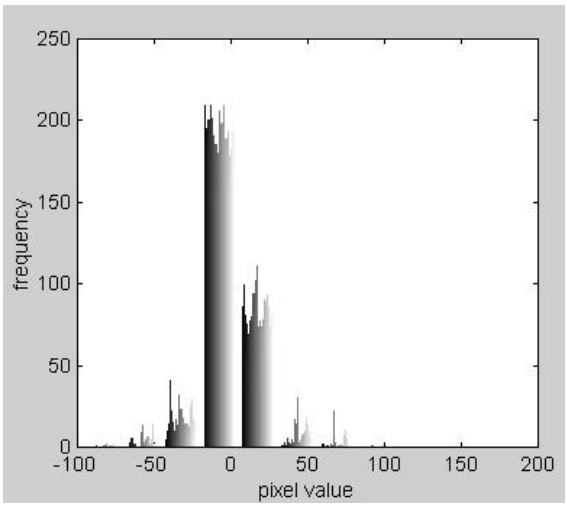

Figure 3. Histogram of Pixel Values in Sisters Image After Delta Compression

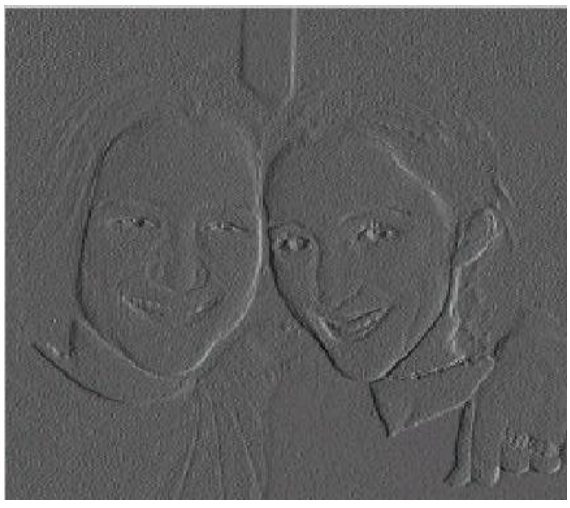

Figure 4. Sister Image After Delta Compression

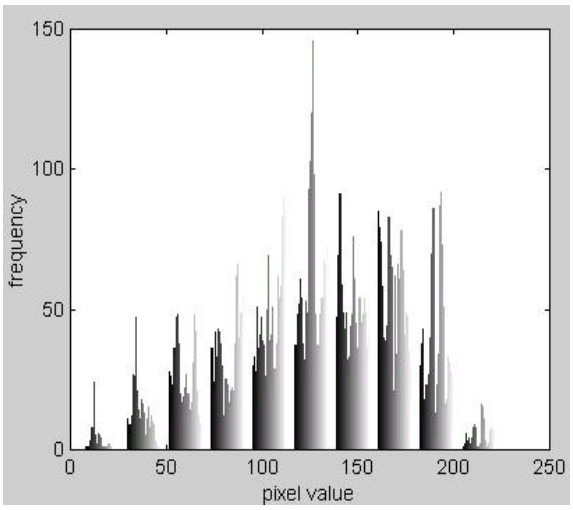

Figure 5. Histogram of Pixel Values in Original Mandrill Image

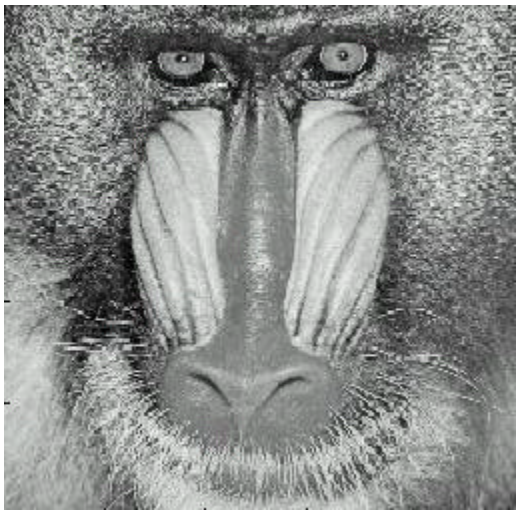

Figure 6.Original Mandrill Image

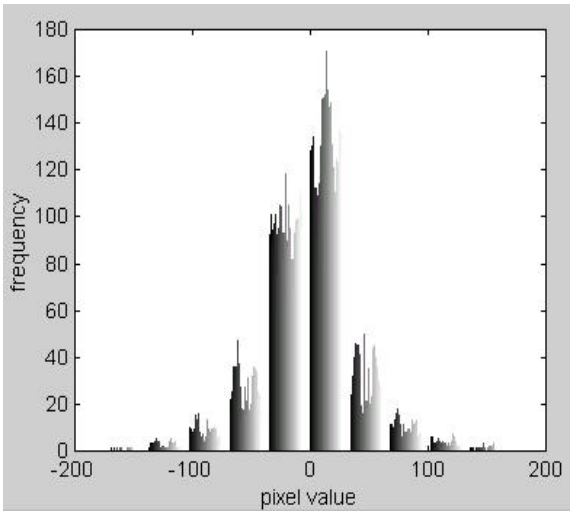

Figure 7. Histogram of Pixel Values in Original Mandrill Image

Figure 8. Mandrill Image After Delta Compression

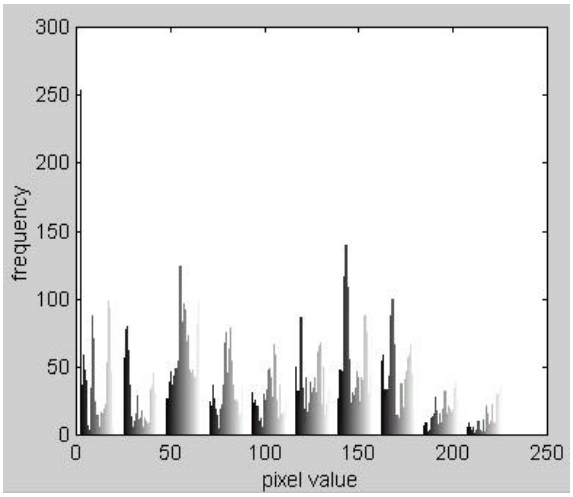

Figure 9. Histogram of Pixel Values in Original Peppers Image

Figure 10. Original Peppers Image

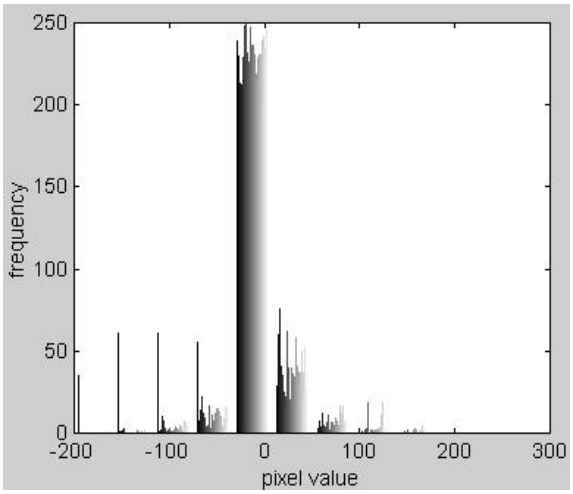

Figure 11. Histogram of Pixel Values in Peppers Image After Delta Compression

Figure 12. Peppers Image After Delta Compression

2. Fourier Compression Results

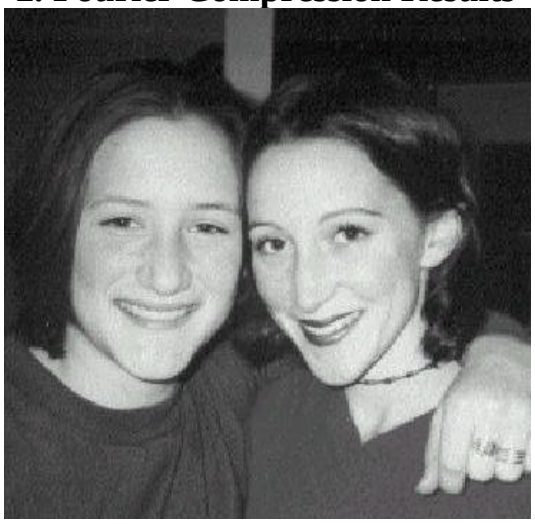

Figure 13. Original Sisters Image

Figure 14. Sisters Image after Fourier Compression

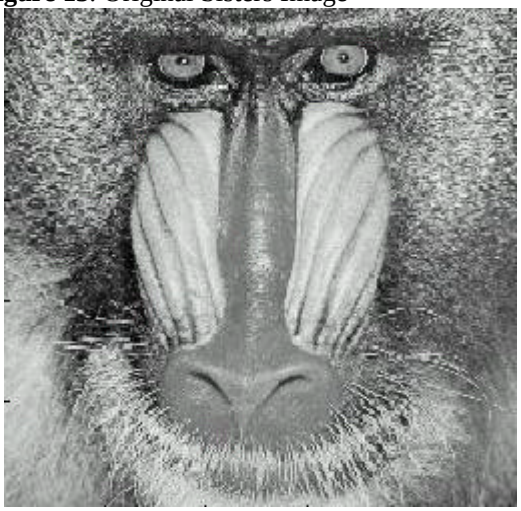

Figure 15. Original Mandrill Image

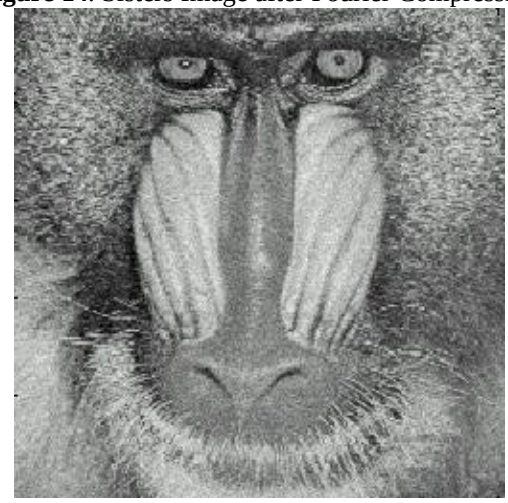

Figure 16. Mandrill Image after Fourier Compression

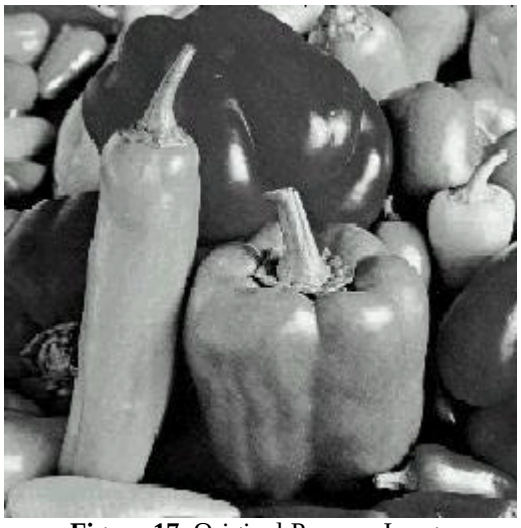

Figure 17. Original Peppers Image

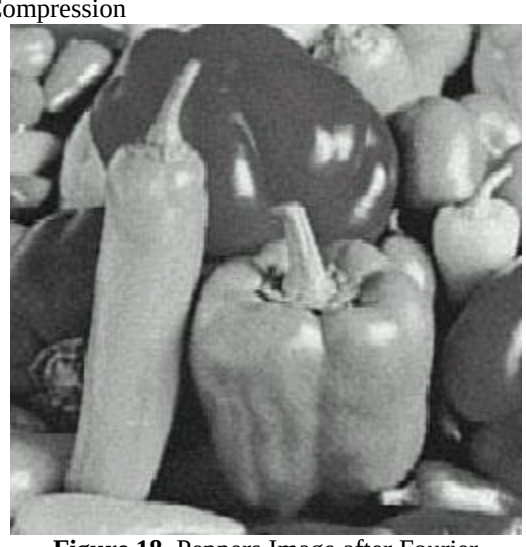

Figure 18. Peppers Image after Fourier

3. Random Iterated Function Systems

Figure 19. Random IFS of a fractal. The attractor is the name 'LISA'.

Figure 20. Random IFS with 3 transformations that produces a fractal shape.

4. Fractal Image Compression Results

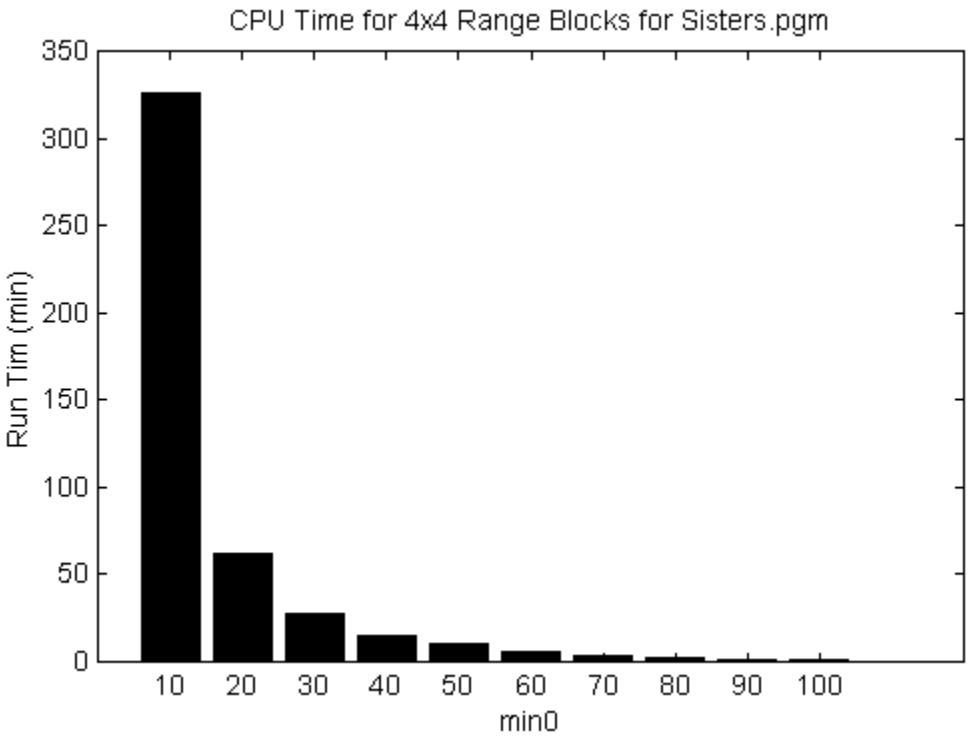

Chart 1.

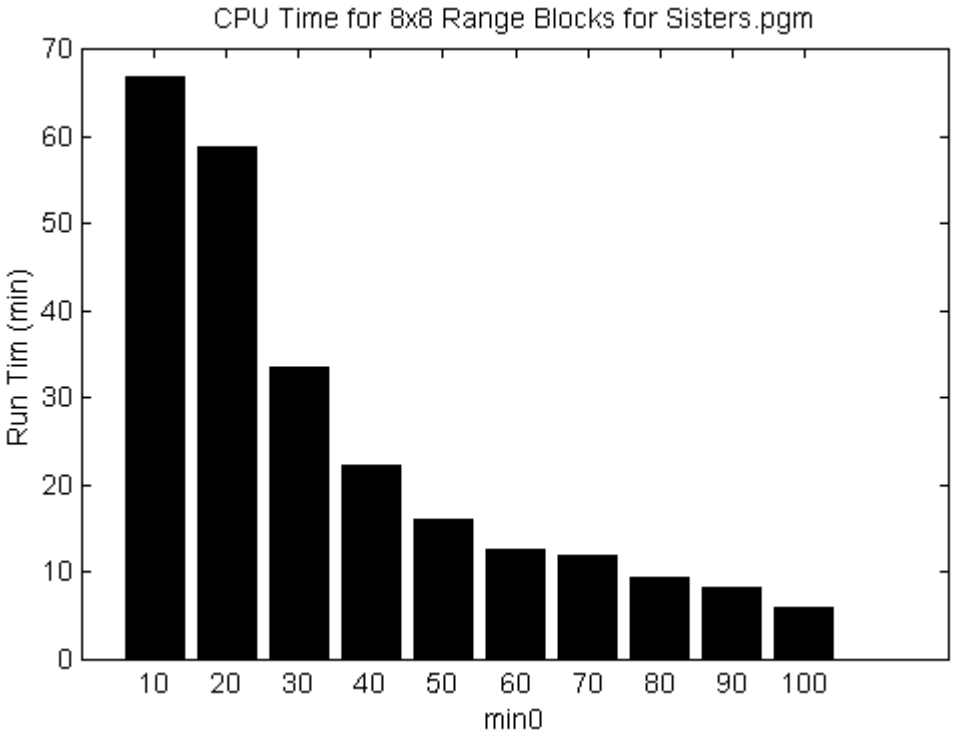

Chart 2.

Figure 21. Original Sisters Image

Figure 22. Sisters 4x4 range blocks Min0=10 Iteration 1

Figure 23. Sisters 4x4 Range blocks Min0=10 Iteration 2

Figure 24. Sisters 4x4 Range blocks Min0=10 Iteration 3

Figure 25. Original Sisters Image

Figure 26. Sisters 4x4 range blocks Min0=20 Iteration 1

Figure 27. Sisters 4x4 Range blocks Min0=20 Iteration 2

Figure 28. Sisters 4x4 Range blocks Min0=20 Iteration 3

Figure 29. Original Sisters Image

Figure 30. Sisters 4x4 range blocks Min0=80 Iteration 1

Figure 31. Sisters 4x4 Range blocks Min0=80 Iteration 2

Figure 32. Sisters 4x4 Range blocks Min0=80 Iteration 3

Figure 33. Original Sisters Image

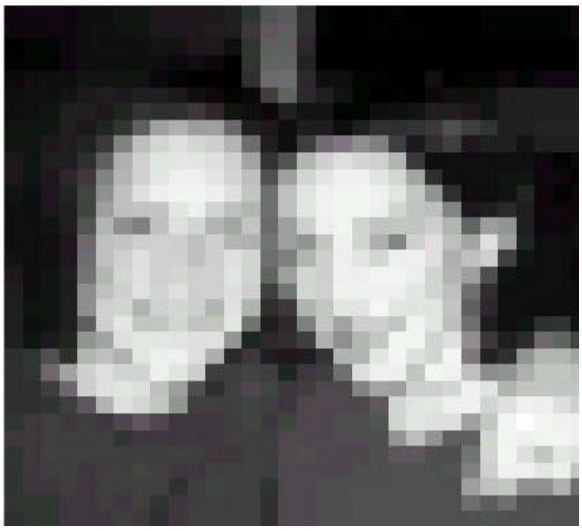

Figure 34. Sisters 8x8 range blocks Min0=10 Iteration 1

Figure 35. Sisters 8x8 Range blocks Min0=10 Iteration 2

Figure 36. Sisters 8x8 Range blocks Min0=10 Iteration 3

Figure 37. Original Sisters Image

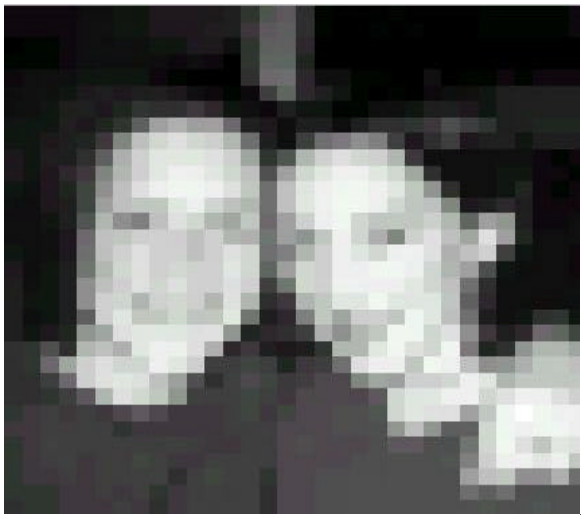

Figure 38. Sisters 8x8 range blocks Min0=80 Iteration 1

Figure 39. Sisters 8x8 Range blocks Min0=80 Iteration 2

Figure 40. Sisters 8x8 Range blocks Min0=80 Iteration 3

VIII. References

1. Barnsley, Michael; Fractals Everywhere, Academic Press,1988.
2. Barnsley, Michael and Lyman P. Hurd; Fractal Image Compression, AK Peters, Ltd., 1993.
3. Crownover, Richard M.; Introduction to Fractals and Chaos; Jones and Bartlett Publishers, 1995.
4. Devaney, Robert L.; A First Course in Chaotic Dynamical Systems: Theory and Experiment, Addison-Wesley Publishing Company, Inc., 1992.
5. Kominek, John; "Advances in Fractal Compression for Multimedia Applications", University of Waterloo.
6. Peak, David and Michael Frame; Chaos Under Control: The Art and Science of Complexity, W.H. Freeman and Company, 1994.
7. Solomon Garfunkel; For all Practical Purposes: Introduction to Contemporary Mathematics, W.H. Freeman and Company,1988.
8. Kathleen T. Alligood, Tim D. Sauer, and James A. Yorke; Chaos: An Introduction to Dynamical Systems, Springer, 1996.
9. Russ, John C. ; The Image Processing Handbook, CRC Press, 1994.
10. http://www.widearea.co.uk/designer/compress.html
11. Tolstov, Georgi P.; Fourier Series, Dover Publications, 1976.
12. Darrel Hankerson, Greg A. Harris, Peter D. Johnson, Jr.; Introduction to Information Theory and Data Compression, CRC Press.
13. Stewart, Ian; Does God Play Dice? The Mathematics of Chaos, Blackwell Publishers, 1989.